Decision Procedures for Linear Arithmetic

Martin Bromberger

Decision Procedures for Linear Arithmetic

Dissertation

zur Erlangung des Grades des Doktors der Naturwissenschaften
an der Fakultät Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von Martin Bromberger
Saarbrücken, Juli, 2019

Tag des Kolloquiums:	10.12.2019
Dekan:	Prof. Dr. Sebastian Hack
Prüfungsausschuss	
Berichterstatter:	Prof. Dr. Pascal Fontaine
	Dr. Alberto Griggio
	PD Dr. Thomas Sturm
	Prof. Dr. Christoph Weidenbach
Vorsitzender:	Prof. Dr. Sven Apel
Akademischer Mitarbeiter:	Dr. Benjamin Kiesl

Abstract

In this thesis, we present new decision procedures for linear arithmetic in the context of SMT solvers and theorem provers:
1) CutSat++, a calculus for linear integer arithmetic that combines techniques from SAT solving and quantifier elimination in order to be sound, terminating, and complete.
2) The *largest cube test* and the *unit cube test*, two sound (although incomplete) tests that find integer and mixed solutions in polynomial time. The tests are especially efficient on absolutely unbounded constraint systems, which are difficult to handle for many other decision procedures.
3) Techniques for the *investigation of equalities* implied by a constraint system. Moreover, we present several applications for these techniques.
4) The *Double-Bounded reduction* and the *Mixed-Echelon-Hermite transformation*, two transformations that reduce any constraint system in polynomial time to an equisatisfiable constraint system that is bounded. The transformations are beneficial because they turn branch-and-bound into a complete and efficient decision procedure for unbounded constraint systems.

We have implemented the above decision procedures (except for CutSat++) as part of our linear arithmetic theory solver SPASS-IQ and as part of our CDCL(LA) solver SPASS-SATT. We also present various benchmark evaluations that confirm the practical efficiency of our new decision procedures.

Zusammenfassung

In dieser Arbeit präsentieren wir neue Entscheidungsprozeduren für lineare Arithmetik im Kontext von SMT-Solvern und Theorembeweisern:
1) CutSat++, ein korrekter und vollständiger Kalkül für ganzzahlige lineare Arithmetik, der Techniken zur Entscheidung von Aussagenlogik mit Techniken aus der Quantorenelimination vereint.
2) Der *Größte-Würfeltest* und der *Einheitswürfeltest*, zwei korrekte (wenn auch unvollständige) Tests, die in polynomieller Zeit (gemischt-)ganzzahlige Lösungen finden. Die Tests sind besonders effizient auf vollständig unbegrenzten Systemen, welche für viele andere Entscheidungsprozeduren schwer sind.
3) Techniken zur *Ermittlung von Gleichungen*, die von einem linearen Ungleichungssystem impliziert werden. Des Weiteren präsentieren wir mehrere Anwendungsmöglichkeiten für diese Techniken.
4) Die *Beidseitig-Begrenzte-Reduktion* und die *Gemischte-Echelon-Hermitesche-Transformation*, die ein Ungleichungssystem in polynomieller Zeit auf ein erfüllbarkeitsäquivalentes System reduzieren, das begrenzt ist. Vereint verwandeln die Transformationen Branch-and-Bound in eine vollständige und effiziente Entscheidungsprozedur für unbeschränkte Ungleichungssysteme.

Wir haben diese Techniken (ausgenommen CutSat++) in SPASS-IQ (unserem theory solver für lineare Arithmetik) und in SPASS-SATT (unserem CDCL(LA) solver) implementiert. Basierend darauf präsentieren wir Benchmark-Evaluationen, die die Effizienz unserer Entscheidungsprozeduren bestätigen.

Acknowledgments

First and foremost, I would like to thank my supervisors Christoph Weidenbach and Thomas Sturm, who not only introduced me to automated reasoning and linear arithmetic, but also supported me throughout the different stages of my research. Both gave me the freedom to follow my own research goals and the necessary guidance that enabled me to reach them.

Next, I want to thank my colleagues from the Automation of Logic group for the friendly and welcoming atmosphere, for many interesting and entertaining discussions, as well as for all their good advise. I especially want to thank Mathias Fleury, Fabian Kunze, Michaël Mera, Simon Schwarz, Dominik Wagner, and Christoph Weidenbach for helping me develop SPASS-SATT. I hope they are as proud of our joint work as I am.

I am also grateful to my friends outside of work that encouraged, advised, and distracted me whenever it was necessary. I especially want to thank Andrea Fischer, Sandy Heydrich, Ralf Jung, Jan-Oliver Kaiser, Jana Rehse, and Tim Ruffing with whom I shared this PhD journey. Our "quality friend time" made the hardships of a PhD more than bearable.

Last but not least, I thank my parents Pia and Helmut Bromberger who always support and encourage me no matter what I choose to do. They even participate willingly in some of my nerdy hobbies! I am sure that without them I would not be who I am today.

Contents

List of Figures

Chapter 1

Introduction

Linear arithmetic has been addressed and thoroughly investigated by at least two independent research lines: (i) optimization via linear programming (LP), integer programming (ILP), and mixed real integer programming (MILP) [15, 18, 25, 28, 46, 58, 71, 72, 82, 89, 93, 95, 103, 127, 128], and (ii) first-order quantifier elimination [16, 17, 44, 62, 63, 65, 67, 134, 74, 102, 117, 121, 138]. In this thesis, we are, however, interested in decision procedures for linear arithmetic for a different context: automated reasoning. To be more precise, we are interested in the satisfiability of linear arithmetic constraint systems, which we simply call *problems*, in the context of the combination of theories, as they occur, e.g., in SMT (satisfiability modulo theories) solving or theorem proving [5, 11, 13, 23, 24, 26, 39, 43, 50, 51, 52, 55, 57, 61, 68, 76, 87, 96, 97, 112, 124].

From this perspective, the above-mentioned research lines, which are based on optimization and quantifier elimination, address problems that are too general for our purposes: the former considers optimization aspects that go considerably beyond our pure satisfiability problems; the latter considers quantifier alternation, which is more complicated than the purely existentially quantified problems that we consider. It is, therefore, no surprise that the decision procedures for optimization and quantifier elimination do not fit the requirements that make SMT solving and theorem proving efficient. One of those requirements is that SMT solvers and theorem provers solve internally not just one large system of linear inequalities but a large number of incrementally connected, small systems of linear inequalities. Therefore, exploiting the incremental connection is key for making SMT solvers and theorem provers efficient [61]. To be more precise, linear arithmetic decision procedures for SMT solving and automated reasoning have to be *incrementally efficient*, i.e., gain an advantage from subsequently solving incrementally connected problems, and have to produce *conflict explanati-*

ons for unsatisfiable problems in order to instantly recognize other problems that are unsatisfiable for the same reason. Consequently, the SMT and theorem proving communities have developed several decision procedures that fit these requirements [5, 11, 26, 39, 52, 55, 57, 61, 68, 76, 87, 96, 97, 112].

In this thesis, we present our own linear arithmetic decision procedures for SMT solving and theorem proving. The focus of our decision procedures are (i) problem classes that appear in SMT solving and theorem proving but are ignored by other research lines, (ii) problem classes that were previously considered too hard to solve efficiently in practice, and (iii) problem classes on which most existing implementations diverge. Moreover, we tried to design our decision procedures (a) to produce conflict (and other) explanations, (b) to be incrementally efficient, (c) to be practically efficient, and (d) to be easy to combine with other techniques.

The remainder of the introduction is organized as follows: First, we quickly introduce the theory of linear arithmetic and its subtheories in Section 1.1. Then, we explain in Section 1.2 how linear arithmetic fits into the context of automated reasoning (e.g. via SMT solving and theorem proving). In Section 1.3, we explain *unbounded problems*, which is the class of linear arithmetic problems that most of our decision procedures target and that is difficult to handle for most existing decision procedures. Then, we explain in Section 1.4 how we measure the efficiency of our decision procedures. The last section, Section 1.5, gives an overview over our own contributions to linear arithmetic decision procedures, which we then present in more detail in the remainder of this thesis.

1.1 The Theory of Linear Arithmetic

A *first-order theory* defines a first-order language, i.e., a set of first-order formulas, and their interpretation. The *theory of linear arithmetic* is such a first-order theory and it defines the set of *linear arithmetic formulas*, i.e., the set of formulas consisting of all Boolean combinations of linear inequalities over arithmetic variables [62, 121, 129]. This means the most basic linear arithmetic formulas are *linear inequalities* $a_{i1} \cdot x_1 + \ldots + a_{in} \cdot x_n \leq b_n$, where $a_{i1}, \ldots, a_{in}, b_i \in \mathbb{Q}$ are rational constants and $x_1, \ldots, x_n$ are typed arithmetic variables. All other linear arithmetic formulas are constructed inductively through Boolean combinations, i.e., $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$, $\forall x_i.\phi$, and $\exists x_i.\phi$ are linear arithmetic formulas if ϕ and ψ are linear arithmetic formulas and if x_i and x_j are arithmetic variables typed as an integer variable and a rational variable, respectively.

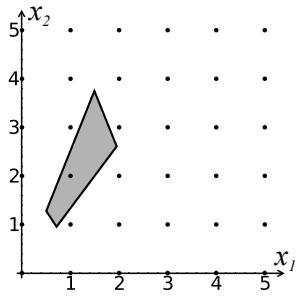


Figure 1.1: The rational solutions of our linear arithmetic example problem

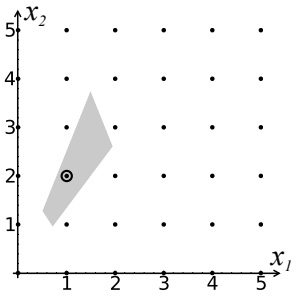


Figure 1.2: The integer solutions of our linear arithmetic example problem

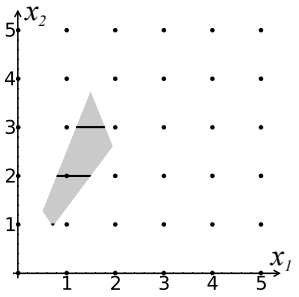


Figure 1.3: The mixed solutions of our linear arithmetic example problem

The theory of linear arithmetic also defines the interpretation of linear arithmetic formulas: all arithmetic functions, predicates, and constants (i.e., $+$, $\cdot$, $\leq$ and $a_{ij} \in \mathbb{Q}$) are interpreted by their well-known mathematical definitions and all Boolean connectives are interpreted as we are used to from first-order logic.

The majority of decision procedures that we present in this thesis handle on their own only the *ground conjunctive* subset of linear arithmetic, i.e., the subset where all Boolean combinations are conjunctions and all variables are free but treated as if they were existentially quantified. For simplicity, we call the formulas belonging to this subset *problems*. Nevertheless, our decision procedures can solve much more expressive subsets of linear arithmetic if we use them in a modular way inside an SMT solver, which is the focus of Chapter 8. Using our decision procedures in a typical SMT solver framework (i.e., CDCL(T) [11, 57, 68, 113, 114]) generates a decision procedure for linear arithmetic formulas that are ground and in CNF (clause normal form). For simplicity, we call such formulas *CNF problems*.[1]

Let us now look at an example of a problem:

$$\begin{array}{rcrcr} -5 \cdot x_1 + 2 \cdot x_2 \leq 0 & \wedge & 4 \cdot x_1 - 3 \cdot x_2 \leq 0 & \wedge \\ 5 \cdot x_1 + 2 \cdot x_2 \leq 15 & \wedge & -3 \cdot x_1 - 2 \cdot x_2 \leq -4 & \end{array}$$

This problem contains four different inequalities and two different variables. Since our problems contain only one type of Boolean connective, we also often omit the conjunction symbols $\wedge$ and denote them instead by a set of inequalities (also called a system of inequalities):

$$\begin{array}{lrr} \{ & -5 \cdot x_1 + 2 \cdot x_2 \leq 0, & 4 \cdot x_1 - 3 \cdot x_2 \leq 0, \\ & 5 \cdot x_1 + 2 \cdot x_2 \leq 15, & -3 \cdot x_1 - 2 \cdot x_2 \leq -4 \quad \} \end{array}$$

Now given such a problem, our goal is to determine whether there exists an assignment for all of our variables that satisfies each of the inequalities.

[1] Any ground linear arithmetic formula can be transformed into a CNF problem with the help of a CNF transformation (see also Chapter 8).

Such an assignment is also called a *solution* to our problem. However, the solutions which we accept and the complexity of finding them is also dependent on the types of our variables. For instance, if both of our variables x_1 and x_2 are rational variables, then all points contained in the polyhedron shown in Figure 1.1 are solutions; if both of our variables x_1 and x_2 are integer variables, then only the point $(x_1, x_2) = (1, 2)$ highlighted by the circle in Figure 1.2 is a solution; and if x_1 is a rational variable and x_2 an integer variable, then all points on the lines shown in Figure 1.3 are solutions.

The different acceptable solutions are also the reason, why we generally partition our problems into three subtheories of linear arithmetic: if our problem contains only rational variables, then it belongs to the *theory of linear rational arithmetic* (LRA), if our problem contains only integer variables, then it belongs to the *theory of linear integer arithmetic* (LIA), and if our problem contains both types of variables, then it belongs to the *theory of linear mixed arithmetic* (LIRA). Finding a solution for a LIA or LIRA problem is an NP-complete task [94, 119]. Finding a solution for an LRA problem is a much easier task and can be accomplished in polynomial time [93, 95]. This means that our three subtheories have different levels of complexity (at least if we assume that P $\neq$ NP).

1.2 Linear Arithmetic in Automated Reasoning

Automated reasoning is a research area that is dedicated to teaching computers how to reason autonomously. This is accomplished by encoding reasoning problems as logical formulas and by (dis-)proving the resulting formulas via automated reasoning programs, e.g., SMT solvers and theorem provers. Since most problems can be encoded as logical formulas (although not necessarily as first-order formulas), the potential applications for automated reasoning are infinite. The applications are, however, limited by the undecidability of some logics and the efficiency of the existing automated reasoning programs. Practically relevant example applications for automated reasoning appear in analysis, testing, verification, and synthesis of hard- and software, as well as in interactive theorem proving [13, 23, 24, 43, 50, 51, 124].

Linear arithmetic is relevant to automated reasoning because many target applications for automated reasoning use arithmetic on their own. For instance, most programs use arithmetic variables (e.g., integers) and perform arithmetic operations on those variables. Therefore, software verification has to deal with arithmetic; often in combination with other theories, e.g., for arrays and other data structures [108]. This is also the reason why there exist many approaches that combine automated reasoning programs for different theories in a modular way. Examples of such modular approaches are CDCL(T) [11, 57, 68, 113, 114] (*conflict-driven clause-learning modulo theo-*

ries) for SMT solving and SUP(T) (*superposition modulo theories*) [5, 14, 64] for theorem proving. CDCL(T), for instance, abstracts the input formula into a propostional formula and repeatedly selects propositionally satisfiable models with a SAT solver, until a (combined) theory solver verifies that one of the abstracted propositional models is also a theory satisfiable model.

Modular approaches can also be used to extend less expressive theory solvers. For instance, a *theory solver for CDCL(LA)* only has to handle problems and the CDCL(LA) framework extends it to a solver for all CNF problems. Moreover, a combined theory solver, i.e., a theory solver handling multiple theories, can be created with the Nelson-Oppen method [111].

Theory solvers also have to fulfill certain requirements or the modular approach is not efficient enough in practice [61]. To be more precise, theory solvers for SMT solving and theorem proving have to be *incrementally efficient*, i.e., gain an advantage from subsequently solving incrementally connected problems, and have to produce *conflict explanations*(*certificates of unsatisfiability*) for unsatisfiable problems in order to instantly recognize other problems that are unsatisfiable for the same reason. Also important are *certificates of satisfiability* (e.g., *solution assignments*) and *theory propagation* and *theory learning* capabilities.

We are interest in decision procedures for linear arithmetic in the context of automated reasoning because of a possible combination with superposition via the SUP(T) approach [5, 14, 64]. In the superposition context, the modular theory solver often has to handle so-called *unbounded problems*, which are particularly hard to solve linear arithmetic problems. Unbounded problems occur so often in SUP(T) because (i) SUP(T) isolates linear arithmetic inequalities to their respective clauses, i.e., inequalities belonging to different clauses interact only under very specific conditions; and (ii) the inequalities that bound an existential variable are often split over multiple clauses (especially in the ground case of SUP(T)) [5].

Unbounded problems also appear in other areas of automated reasoning, although less frequently. Either because of bad encodings, necessary but complicating transformations, e.g., slacking (see Chapter 6.5), or the sheer complexity of the verification goal. Hence, efficient techniques for handling unbounded problems are necessary for a generally reliable combined procedure and their development became the focus of this thesis.

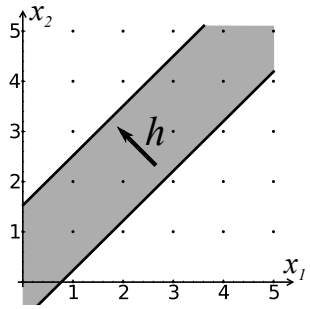


Figure 1.4: A partially bounded system; the directions $h = (-1, 1)^T$ and $-h$ are the only bounded directions in the example

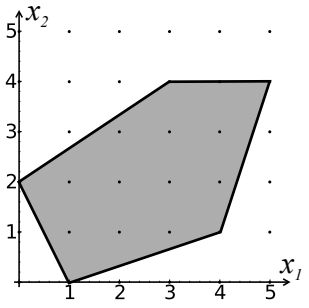


Figure 1.5: A bounded system; all directions are bounded

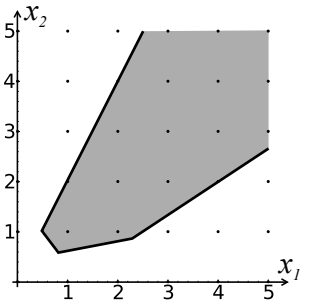


Figure 1.6: An absolutely unbounded system; all directions are unbounded

1.3 Unbounded Problems

A direction $h \in \mathbb{Q}^n$ is *bounded* in a problem if there exist two constants $l, u \in \mathbb{Q}$ such that all rational solutions $s \in \mathbb{Q}^n$ of the problem fulfill the constraints $l \leq h^T s \leq u$. A direction $h \in \mathbb{Q}^n$ is *unbounded* in a problem if no such two constants exist. (See Figures 1.4—1.6 for examples of (un)bounded directions.) Based on these definitions, a problem is *bounded* if all directions are bounded and a problem is *unbounded* if at least one direction is unbounded.

Unbounded directions become interesting when we consider termination of decision procedures for linear integer (and mixed) arithmetic. Achieving termination on a bounded problem is relatively easy because such a problem imposes bounds on (all of) its variables x_i. Therefore, there only exist finitely many potential integer solutions, which can be enumerated to find an actual integer solution. The efficiency of the decision procedure then mostly depends on the size of the variable bounds and on the order of our enumeration. Achieving termination on unbounded problems is harder because the search space is not finite. Most decision procedures, e.g., branch-and-bound [101, 104, 128], will even diverge without additional help (see also Example 2.8.3 in Chapter 2). However, the additional help described in the literature is often not valuable because it does not scale in practice. As an example, let us look at the *a priori bounds* presented by Papademitriu [119]:[2] *A priori* bounds intersect the original problem with a cube that is so large that the problem only has an integer (or a mixed) solution

[2] Unbounded directions are no obstacle for decision procedures over linear rational arithmetic because their termination strategies typically do not rely on enumerating all potential rational solutions.

outside of the cube if it also has one inside the cube. Therefore, *a priori* bounds reduce the search space for enumeration based decision procedures, e.g., branch-and-bound, to a finite search space. Hence, enumeration based decision procedures are guaranteed to terminate. However, *a priori* bounds often describe a search space that is so large that it cannot be explored in reasonable time. For instance, the *a priori* bounds for the example depicted in Figure 1.4 are in the order of billions. So *a priori* bounds are valuable in theory but not in practice.

Other termination tactics from the literature, e.g., exhaustive cutting planes [73] and quantifier elimination [44], do not scale in practice because of similar arguments. It is, therefore, no surprise that none of these tactics have been implemented in any of the state-of-the-art SMT solvers [9, 40, 41, 49, 56].

1.4 Measuring Efficiency

In this thesis, we estimate the *practical efficiency* of our decision procedures by implementing them inside a theory solver or CDCL(LA) solver and testing them on a large set of benchmarks. To be more precise, a solver is *more efficient in practice* than another solver (i) if it solves more problems from a relevant benchmark set within a reasonable time limit than the other solver; or (ii) if it solves problems from a relevant benchmark set faster than the other solver. Similarly, a decision procedure *helps in practice* if its addition to a solver makes the solver more efficient in practice. For our purposes, the relevant benchmark sets are subsets of the SMT-LIB (Satisfiability Modulo Theories Library) benchmarks [10] and our reasonable time limit is 40 minutes[3].

Note, however, that there exists no perfect method for estimating practical efficiency. Benchmark evaluations are imperfect because we can only test a finite set of benchmarks although there is an infinite number of problems. Therefore, benchmark evaluations are almost guaranteed to be biased. In some sense, this bias is even expected and necessary because our decision procedures are handling NP-hard problem classes and therefore cannot scale on all problems (unless P = NP). As a compromise, we are focusing our benchmarks and decision procedures on (types of) problems that are relevant for applications.

[3] 40 minutes is also the time limit used by the SMT-COMP 2019 (the 14th International Satisfiability Modulo Theories Competition)

1.5 Contributions

The contributions that we present in this thesis are new linear arithmetic decision procedures for SMT solving and theorem proving. We developed these decision procedures in order to efficiently handle unbounded problems. And yet, some of our decision procedures have additional applications beyond unbounded problems. For instance, one of our techniques can be used to investigate and eliminate equalities, to eliminate quantifiers, and to compute all pairs of equivalent variables inside a constraint system, which are necessary for the *Nelson-Oppen* style combination of theories [111].

The presentation of our contributions is organized as follows: First, we present in Chapter 2 some important definitions and theorems from the literature and our preferred notations and representations. Next, we present in Chapters 3—6 our new decision procedures for linear arithmetic. Then, we outline in Chapter 7 the implementation of our linear arithmetic theory solver SPASS-IQ that efficiently combines our new decision procedures with other decision procedures for linear arithmetic (e.g., with branch-and-bound and a version of the dual simplex algorithm). In Chapter 8, we then extend SPASS-IQ to a CDCL(LA) solver, which we call SPASS-SATT. Both chapters also highlight all factors that have a major impact on the efficiency of our implementations. These factors include the impact of (i) the chosen decision procedures; (ii) the optimizations that we made to the chosen decision procedures (e.g., optimized data structures and heuristics); (iii) the different interaction techniques between theory solver and SAT solver in our CDCL(LA) framework; and (iv) specialized preprocessing techniques (e.g., for if-then-else expressions and pseudo-boolean inequalities). We conclude the thesis in Chapter 9 with a summary of the presented results. Works related to the contributions of this thesis are summarized in the first section of the relevant chapters. The specific contributions that we present in this thesis are listed below.

Chapter 3: The CutSat++ Calculus

CutSat++, which we present in Chapter 3, is a calculus for linear integer arithmetic that combines techniques from SAT solving and quantifier elimination. It is an extension and refinement of the CutSat calculus by Jovanović and de Moura [87]. Compared to its predecessor, CutSat++ terminates on all problems for linear integer arithmetic even without relying on *a priori* bounds [119] for termination. With a CDCL style calculus CutSat++ efficiently handles problems over guarded variables, i.e., variables with a constant upper and lower bound. On problems with unguarded variables (e.g. unbounded problems) the CDCL style calculus alone is not guaranteed to terminate. Hence, we combine it with a lazy quantifier elimination procedure that transforms a problem containing unguarded variables

into one where feasibility depends only on guarded ones. The quantifier elimination procedure is called lazy because we only apply it to conflicts encountered by the CDCL style algorithm. This allows us to avoid certain cases of worst-case exponential behavior that we would otherwise observe by using a quantifier elimination procedure alone. Overall, CutSat++ is a sound, terminating, and complete decision procedure for linear integer arithmetic. Moreover, CutSat++ leaves enough space for model assumptions and simplification rules in order to be efficient in practice.

Chapter 3 is based on a publication with Thomas Sturm and Christoph Weidenbach as co-authors [32]. An extended version of this publication titled "A Complete and Terminating Approach to Linear Integer Solving" has been accepted for publication in the Journal of Symbolic Computation.

Chapter 4: The Largest Cube Test and the Unit Cube Test

The *largest cube test* and the *unit cube test*, which we present in Chapter 4, are two sound (although incomplete) tests that find integer and mixed solutions for linear arithmetic constraint systems in polynomial time. In contrast to many complete methods that search along the problem surface for a solution, these tests use axis-parallel hypercubes, which we simply call cubes, to explore the interior of the problem. To this end, the largest cube test finds a cube with maximum edge length contained in the rational solutions of the input problem, determines its rational valued center, and rounds it to a potential mixed/integer solution. The unit cube test determines instead whether the the rational solutions of the input problem contain a cube with edge length one, which is the minimal edge length that always guarantees that a cube contains a mixed/integer solution.

The tests are especially efficient on constraint systems with a large number of integer solutions, e.g., those that are absolutely unbounded. Inside the SMT-LIB (Satisfiability Modulo Theories Library) benchmarks [10], we have found almost one thousand problem instances that are absolutely unbounded. Experimental results confirm that our tests are superior on these instances compared to several state-of-the-art SMT solvers.

Chapter 4 is based on two publications with Christoph Weidenbach as co-author [34, 35].

Chapter 5: Techniques for the Investigation of Equalities

We present in Chapter 5 new techniques for the investigation of equalities implied by a system of linear arithmetic constraints. The main technique presented in this chapter computes a basis for all implied equalities, i.e., a finite representation of all equalities implied by the linear arithmetic constraints. The equality basis has several applications. We can use it (i) to determine whether a constraint system implies any equality, (ii) to verify

whether a constraint system implies a given equality, (iii) to eliminate the equalities from our constraint system, (iv) to eliminate quantifiers, (v) to determine the (un)bounded directions in our constraint system, and (vi) to compute all pairs of equivalent variables inside a constraint system, which are necessary for the Nelson-Oppen style combination of theories [111].

Chapter 5 is based on two publications with Christoph Weidenbach as co-author [33, 35].

Chapter 6: A New Bounding Transformation

As mentioned before, many existing approaches for linear mixed and integer arithmetic, e.g., branch-and-bound [101, 104, 128], terminate only on bounded problems. Therefore, we present in Chapter 6 a new *bounding transformation*, i.e., a transformation that reduces any problem to an equisatisfiable problem that is bounded. Our bounding transformation consists of two other transformations: the *Mixed-Echelon-Hermite transformation* by Christ and Hoenicke [39] and our own *Double-Bounded reduction*. Compared to previous bounding transformations from the literature (e.g., *a priori* bounds [119]), ours is not only valuable in theory, i.e., the transformation takes only polynomial time, but also in practice, i.e., the resulting problems are solvable in reasonable time. Our transformation is so efficient because it orients itself on the structure of the input problem instead of computing *a priori* (over-)approximations out of the available constants. Experiments provide further evidence to the efficiency of the transformations in practice. Moreover, we present a polynomial method for converting certificates of (un)satisfiability from the transformed to the original problem.

Chapter 6 is based on [30]. The paper has no coauthors.

Chapter 7: The Implementation of SPASS-IQ

SPASS-IQ is the linear arithmetic theory solver that we have implemented as part of our SPASS Workbench [3]. It efficiently combines most of our new decision procedures with existing techniques for linear arithmetic (viz., dual simplex, branch-and-bound, simple rounding, and bound refinement). In Chapter 7, we outline and evaluate SPASS-IQ's overall implementation. Since we already present our new decision procedures as stand-alone procedures in other chapters, this chapter will focus on their combination and any additional implementation factors that have a major impact on the efficiency of SPASS-IQ. The impact of these factors is measured through various benchmark evaluations on the SMT-LIB benchmarks for QF_LIA (quantifier free linear integer arithmetic) and QF_LRA (quantifier free linear rational arithmetic) [10].

Parts of Chapter 7 are summarized in [31]. Coauthors are Mathias Fleury, Simon Schwarz, and Christoph Weidenbach.

Chapter 8: The Implementation of SPASS-SATT

SPASS-SATT is an automated reasoner for ground linear arithmetic (with arbitrary boolean combinations) that we have implemented as part of our SPASS Workbench [3]. To be more precise, we combine in SPASS-SATT a CDCL(T) implementation with our theory solver SPASS-IQ to get a CDCL(LA) implementation. The CDCL(LA) implementation still handles only CNF problems (i.e., ground linear arithmetic formulae in *clause normal form*). We resolve this by adding some preprocessing techniques that turn any ground linear arithmetic formula into an equisatisfiable CNF problem.

In Chapter 8, we explain the construction of SPASS-SATT in more detail. First, we outline the interaction between the theory solver and the SAT solver in the CDCL(LA) implementation. This also includes extensions to the theory reasoning that enhance and guide the search of the SAT solver. Then, we explain the preprocessing techniques that we incorporated into SPASS-SATT. As part of our explanations, we also measure the impact of our theory reasoning extensions and preprocessing techniques through various benchmark evaluations on the SMT-LIB benchmarks for QF_LIA (quantifier free linear integer arithmetic) and QF_LRA (quantifier free linear rational arithmetic) [10].

As a consequence of our excellent benchmark results, our decision procedures have been reimplemented in several state-of-the-art SMT solvers. For instance, the unit cube test has been reimplemented in MathSAT5 [41], SMT-RAT [45], and Z3 [49]. Moreover, we participated with SPASS-SATT in the main track of the 13th International Satisfiability Modulo Theories Competition (SMT-COMP 2018) and ranked first in the category QF_LIA (quantifier free linear integer arithmetic) [1] and ranked second in the category QF_LRA (quantifier free linear rational arithmetic) [2].

Parts of Chapter 8 are summarized in [31]. Coauthors are Mathias Fleury, Simon Schwarz, and Christoph Weidenbach.

Chapter 2

Preliminaries

In this chapter, we present the foundations for this thesis. This includes some well-known theorems and definitions from the literature of linear algebra, linear arithmetic, satisfiability modulo theories, and transition systems. It also includes the notations and representations most commonly used during this thesis.

The chapter is outlined as follows: We start in Section 2.1 with our notations for basic linear algebraic constructs, e.g., vectors and matrices. We also define some general norm and distance functions in the same section. In Section 2.2, we present the syntax and semantics of general arithmetic constraint systems. This includes: several representations for arithmetic constraints; definitions for assignments, solutions, satisfiability, equivalence, and equisatisfiability of constraint systems; as well as definitions for substitutions over constraint systems.

In Section 2.3, we explain how to reduce all of our general arithmetic constraints to non-strict inequalities. We do so because the linear arithmetic literature typically supports only non-strict inequalities [28, 83, 89, 93, 103, 128]. This also allows us to define our standard (theoretical) input problems as systems of inequalities in Section 2.4. In the same section, we also define an alternative input format, the tableau representation, which we use for algorithms in the context of SMT solver implementations.

In Section 2.5, we define implied constraints and explain their relationship to combinations of linear inequalities. This relationship is also the basis for most formal proofs in linear arithmetic as well as this thesis. Moreover, they allow us to define some interesting properties of constraint systems, e.g., (un)bounded directions, which we explain in Section 2.8.

All techniques presented in this thesis are meant to be used inside SMT solvers. In Section 2.6, we explain CDCL(T), the framework most SMT solvers are based on. Moreover, we formulate most of our contributions as natural extensions of the linear arithmetic decision procedures implemented

in most SMT solvers: a version of the dual simplex algorithm for linear rational arithmetic and a branch-and-bound method for linear integer/mixed arithmetic. We present these standard procedures and some popular extensions to them in Section 2.7.

The standard arithmetic decision procedures are efficient on many problems. There does, however, exist a class of problems, where standard decision procedures for linear arithmetic do not guarantee termination or take too much time in practice. We call this class of problems *unbounded problems* and formally define it in Section 2.8. Since one of the goals of this thesis is to find complete but efficient decisions procedures for linear arithmetic, many of our contributions focus on unbounded problems.

Another way of extracting information from a system of inequalities is to view it as a geometric object. In Chapter 4, we show how to use simpler geometric objects to find integer/mixed solutions for some of our systems. The objects we use for this purpose are defined in Section 2.9.

We describe the majority of our algorithms as pseudocode written in an abstract while-language. We do so because (i) most of our contributions are designed as extensions of existing procedures that were presented in an abstract while-language; and (ii) a pseudocode description is already relatively close to an actual implementation of the respective algorithm. There is, however, one algorithm which we prefer to describe as a transition system instead of pseudocode. This algorithm is CutSat++, our terminating extension to the CutSat algorithm. We choose to present CutSat++ as a transition system because (i) the CutSat calculus was also originally presented as a transition system [87, 86], and (ii) it is a good representation for proving termination, soundness, and completeness of a complex algorithm. We present the basics of transition systems in Section 2.10.

2.1 Basics of Linear Algebra

The linear algebraic notation in this thesis follows [100] with some minor exceptions.[1] While the difference between matrices, vectors, and their components is always clear in context, we generally use upper case letters for matrices (e.g., A), lower case letters for vectors (e.g., x), and lower case letters with an index i or j (e.g., b_i, x_j) as components of the associated vector at position i or j, respectively. The only exceptions are the row vectors $a_i^T = (a_{i1}, \ldots, a_{in})$ of a matrix $A = (a_1, \ldots, a_m)^T$, which already contain an index i that indicates the row's position inside A. We also abbreviate the

[1] The exceptions are: we avoid confusion between row vectors and column vectors by simply writing row vectors as transposed column vectors; and we write 0^n for the n-dimensional origin instead of 0 to highlight the dimension of the vector.

n-dimensional origin $(0,\ldots,0)^T$ as 0^n and the n-dimensional all-ones vector $(1,\ldots,1)^T$ as 1^n. Moreover, we denote by $\text{piv}(A,j)$ the row index of the *pivot of a column* j, i.e., the smallest row index i with a non-zero entry a_{ij} or $m+j$ if there are no non-zero entries in column j.

2.1.1 Norms

Some of our techniques require the use of *d-norms* $\|.\|_d$ [58]. These d-norms are defined as functions $(\|.\|_d : \mathbb{R}^n \to \mathbb{R})$ for $d \geq 1$ such that $\|x\|_d = \left(|x_1|^d + \ldots + |x_n|^d\right)^{1/d}$. There are two d-norms that are especially relevant to this thesis: the 1-*norm* and the *maximum norm*. The 1-norm simplifies our general d-norm definition to $\|x\|_1 = (|x_1| + \ldots + |x_n|)$. The *maximum norm* is a special d-norm because it is defined by the limit of $\|.\|_d$ for $d \to \infty$: $\|x\|_\infty = \max\{|x_1|,\ldots,|x_n|\}$. We chose to focus on the 1-norm and the maximum norm because any rational vector $b \in \mathbb{Q}^n$ has a rational 1-norm $\|b\|_1 \in \mathbb{Q}$ and a rational maximum norm $\|b\|_\infty \in \mathbb{Q}$. This becomes relevant when we later use it to define our fast cube tests (Chapter 4).

2.1.2 Distance

We can also use d-norms to define *distance functions* $\text{dist}_d(x,y)$ between two points $x,y \in \mathbb{R}^n$: $\text{dist}_d(x,y) = \|x-y\|_d$ [58]. In this thesis, we are rarely interested in an actual distance between points. We use, however, distances to define other relationships between points. For instance, we define a *closest integer* for a point x as a point $x' \in \mathbb{Z}^n$ with minimal distance $\text{dist}_d(x,x')$. We also define the operators $\lceil x_j \rfloor$ and $\lceil x \rfloor$ such that they describe a *closest integer* for x_j and x, respectively [66, 116]. Formally, this means that $\lceil x \rfloor = (\lceil x_1 \rfloor, \ldots, \lceil x_n \rfloor)^T$ and

$$\lceil x_j \rfloor = \begin{cases} \lfloor x_j \rfloor & \text{if } x_j - \lfloor x_j \rfloor < 0.5\,, \\ \lceil x_j \rceil & \text{if } x_j - \lfloor x_j \rfloor \geq 0.5\,. \end{cases}$$

This definition of $\lceil x \rfloor$ is also known as *simple rounding*.

Lemma 2.1.1 (Closest Integer)**.** *$\lceil x \rfloor$ is a closest integer to x, i.e.:*

$$\forall x' \in \mathbb{Z}^n.\ dist_d(x, \lceil x \rfloor) \leq dist_d(x,x')\,.$$

Proof. We first look at the one-dimensional case, where $\|x_j\|_d$ simplifies to $|x_j|$:

$$\forall x'_j \in \mathbb{Z}.\ |x_j - \lceil x_j \rfloor\,| \leq |x_j - x'_j|\,.$$

For $\lceil x_j \rfloor, x'_j \in \mathbb{Z}$, there exists $z_j \in \mathbb{Z}$ such that $x'_j = \lceil x_j \rfloor - z_j$. Moreover, there exists a $d_j \in [-0.5, 0.5]$ for each x_j such that $d_j := x_j - \lceil x_j \rfloor$. The inequality trivially holds for $z_j = 0$:

$$|x_j - x'_j| = |x_j - \lceil x_j \rfloor + z_j| = |x_j - \lceil x_j \rfloor\,|\,,$$

Next, we use the triangle inequality to get the following relationship for the remaining $z_j \neq 0$:

$$|x_j - x'_j| = |x_j - \lceil x_j \rfloor + z_j| = |d_j + z_j| \geq |z_j| - |d_j| \,.$$

Since $z_j \neq 0$ and $d_j \in [-0.5, 0.5]$ imply $|z_j| \geq 1$ and $|d_j| \leq 0.5$, respectively, we get:

$$|x_j - x'_j| \geq |z_j| - |d_j| \geq 1 - |d_j| \geq 0.5 \geq |d_j| = |x_j - \lceil x_j \rfloor \,| \,.$$

The multidimensional case follows from the d-norms' componentwise monotonicity [58], i.e., if $|x_j - \lceil x_j \rfloor \,| \leq |x_j - x'_j|$ for all $j \in \{1, \ldots, n\}$, then $\|x - \lceil x \rfloor\|_d \leq \|x - x'\|_d$. □

A generalization of the closest integer is the *closest k-mixed point* to the point x, i.e., the point x' with minimal distance $\text{dist}_d(x, x')$ such that the last k components of x' are integer values. As with the closest integer, we can find a closest k-mixed point with the help of rounding, but we only round the last k components. We define this through the *mixed simple rounding* function $\lceil x \rfloor_k := (x_1, \ldots, x_{n-k}, \lceil x_{n-k+1} \rfloor, \ldots, \lceil x_n \rfloor)^T$.

Lemma 2.1.2 (Closest k-Mixed Point)**.** *$\lceil x \rfloor_k$ is a closest k-mixed point to x.*

Proof. Works analogously to the proof of Lemma 2.1.1. □

2.2 Basics of Linear Arithmetic

The input *problems* for linear arithmetic are *systems of constraints* C, i.e., finite sets of constraints corresponding to and sometimes used as conjunctions over their elements. The standard *constraints* for linear arithmetic are *non-strict inequalities* $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ and *strict inequalities* $a_{i1}x_1 + \ldots + a_{in}x_n < b_i$. In some cases, we also have to handle a third type of constraint, a so-called *divisibility constraint* $d_i \mid a_{i1}x_1 + \ldots + a_{in}x_n - b_i$.[2] Note, however, that divisibility constraints are introduced only during the execution of some of our algorithms (CutSat, CutSat++, and CutSat$_g$). Our actual input problems never contains divisibility constraints. Therefore, we only have to handle them explicitly in algorithms where they are introduced, which happens only in the algorithms presented in Chapter 3.

In all of the above constraints, the a_{ij} are constant rational values called *coefficients*, the b_i are constant rational values called *constraint bounds*, the d_i are constant positive integer values called *constraint divisors*, and the x_j are distinct *variables*, i.e., the variables x_j and x_i are different unless $i = j$. Each of those variables is also assigned a *type*: either *rational* or *integer*. If our problem contains only integer variables, then it belongs to the *theory of linear integer arithmetic* (LIA). If our problem contains only rational variables, then it belongs to the *theory of linear rational arithmetic* (LRA). If our problem contains both types of variables, then it belongs to the *theory of linear mixed arithmetic* (LIRA).

[2] The semantic of a divisibility constrain $d_i \mid a_{i1}x_1 + \ldots + a_{in}x_n - b_i$ can be summarized as $a_{i1}x_1 + \ldots + a_{in}x_n - b_i$ is divisible by d_i.

For easier access to the components of our constraints and problems, we also define the following functions: $\text{coeff}(I, x_j) := a_{ij}$ denotes the coefficient of variable x_j in the constraint I, where $I := a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ or $I := a_{i1}x_1 + \ldots + a_{in}x_n < b_i$ or $I := d_i \mid a_{i1}x_1 + \ldots + a_{in}x_n - b_i$; $\text{vars}(C) := \{x_1, \ldots, x_n\}$ denotes the set of all variables that appear in the problem C, i.e., that appear with a non-zero coefficient in any of its constraints; similarly, $\text{Qvars}(C)$ and $\text{Zvars}(C)$ denote the sets of rational and integer variables that appear in the problem C, respectively. Due to convenience, we assume that the first $n_1 := |\,\text{Qvars}(C)|$ variables $x_1, \ldots, x_{n_1}$ are rational variables and the remaining $n_2 := |\,\text{Qvars}(C)|$ variables $x_{n_1+1}, \ldots, x_n$ are integer variables, where $n = n_1 + n_2 := |\,\text{vars}(C)|$.

2.2.1 Constraint Representations

We previously defined constraints as either non-strict inequalities of the form

$$a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i,$$

strict inequalities of the form

$$a_{i1}x_1 + \ldots + a_{in}x_n < b_i,$$

and divisibility constraints of the form

$$d_i \mid a_{i1}x_1 + \ldots + a_{in}x_n - b_i.$$

We say that a constraint is in *standard representation* if it is in one of those three forms. However, not all linear constraints fit into this standard representation, e.g., $3 \geq 2(x_3+2(x_1-x_2)-5)$. Still, all of them can be transformed into it by using basic mathematical properties, e.g., associativity, commutativity and distributivity. For instance, $3 \geq 2(x_3 + 2(x_1 - x_2) - 5)$ can be transformed into $4x_1 - 4x_2 + 2x_3 \leq 13$. Whenever we construct a new constraint, we assume that it is transformed into an equivalent constraint in the standard representation.

Vector Representation and Focused Representation

The standard representation is a sometimes unnecessarily long representation for abstract constraints. For this reason, we additionally use two alternative representations: the *vector representation* and the *focused representation*. Both alternative representations are essentially the standard representation except for some form of abbreviation.

In the *vector representation*, we represent our coefficients and variables as vectors $a_i = (a_{i1}, \ldots, a_{in})^T$ and $x = (x_1, \ldots, x_n)^T$, respectively. This allows us to abbreviate their sum $a_{i1}x_1 + \ldots + a_{in}x_n$ as a vector product $a_i^T x$. Thus, non-strict inequalities are represented as $a_i^T x \leq b_i$, strict inequalities as $a_i^T x < b_i$, and divisibility constraints as $d_i \mid a_i^T x - b_i$. The vector representation is useful whenever we are using the same n variables and do not need to focus on one specific variable. We use it mainly in Chapters 4, 5, and 6.

In contrast, the *focused representation* is useful when the set of variables changes over time or when we focus on one specific variable. We use it mainly in Chapter 3 because the algorithms in this chapter are related to quantifier and variable elimination. For the focused representation, we select one variable x_j as our focus and separate it from the sum of variables. Then we abbreviate the sum over the remaining variables together with the constraint bound as a linear polynomial

$$p_{ij} := \left(\sum_{k=1,k\neq j}^{n} a_{ik}x_k\right) - b_i.$$

Thus, non-strict inequalities are represented as $a_{ij}x_j + p_{ij} \leq 0$, strict inequalities as $a_{ij}x_j + p_{ij} < 0$, and divisibility constraints as $d_i \mid a_{ij}x_j + p_{ij}$. We also always assume without loss of generality that the coefficient a_{ij} of the focused variable in $d_i \mid a_{ij}x_j + p_{ij}$ is non-negative. We can do so because divisibility is sign invariant, i.e., $d_i \mid a_{ij}x_j + p_{ij} \equiv d_i \mid -a_{ij}x_j - p_{ij}$. Note that the linear polynomial p_{ij} may still contain variables with negative coefficients.

Integer Coefficients and Bounds

Some of our algorithms for linear integer arithmetic (i.e., CutSat, CutSat++, and CutSat$_g$) require that all coefficients and constraint bounds are integer values instead of the more general rational values. This restriction is easy to fulfill with the help of the following equivalences: First of all, we make all coefficients $a_{ij} = \frac{c_{ij}}{q_{ij}}$ of a constraint I integral by multiplying I with $q = \text{lcm}(q_{i1}, \ldots, q_{in})$, i.e., the lowest common denominator of the coefficients. This works because of the following equivalences for every positive $q \in \mathbb{Z}$ [100, 128]: $a_i^T x \leq b_i \equiv q{\cdot}a_i^T x \leq q{\cdot}b_i$, $a_i^T x < b_i \equiv q{\cdot}a_i^T x < q{\cdot}b_i$, and $d_i \mid a_i^T x - b_i \equiv q \cdot d_i \mid q \cdot a_i^T x - q \cdot b_i$. Now that both the variables x and the coefficients a_i are integers, we can also make the constraint bounds b_i integral. For inequalities, it is enough to round the constraint bounds in the right direction, i.e., $a_i^T x \leq b_i \equiv a_i^T x \leq \lfloor b_i \rfloor$ and $a_i^T x < b_i \equiv a_i^T x \leq \lceil b_i \rceil - 1$. Divisibility constraints are simply false if the constraint bound is not integral. So we can just replace them with a trivially false constraint, e.g., $2 \mid 1$.

Highlighting Bounds and Equalities

We also use special representations when we want to *highlight* that a constraint fulfills some specific properties. For instance, if an inequality is a *variable bound* and we want to highlight this, then we represent it (depending on the variable bound) as $x_j \leq b_i$ for a *non-strict upper bound*, $x_j < b_i$ for a *strict upper bound*, $x_j \geq b_i$ for a *non-strict lower bound*, or $x_j > b_i$ for a *strict lower bound*.

Generally, our standard inequalities $a_i^T x \leq b_i$ are represented as *upper bounds*, i.e., b_i is the upper bound of $a_i^T x$. A lower bound b_i for $a_i^T x$ has to be negated in our standard representation, so written as $-a_i^T x \leq -b_i$. If we, however, want to highlight that $a_i^T x$ has a *lower bound* b_i, then we can also represent $-a_i^T x \leq -b_i$ as $b_i \leq a_i^T x$ to make this fact clearer. Similarly, we can highlight the existence of both a lower bound l_i and an upper bound u_i for $a_i^T x$ by writing $l_i \leq a_i^T x \leq u_i$.

Finally, we can highlight that $a_i^T x$ has the same upper and lower bound b_i—so C contains the two inequalities $a_i^T x \leq b_i$ and $-a_i^T x \leq -b_i$—by stating that C contains the *equality* $a_i^T x = b_i$. This also means that an equality is not really a new type of constraint but just syntactic sugar for the set of constraints $\{a_i^T x \leq b_i, -a_i^T x \leq -b_i\}$.

2.2.2 Assignments and Solutions

The variables $x = (x_1, \ldots, x_n)^T$ in a problem C can be assigned (rational) values $s = (s_1, \ldots, s_n)^T \in \mathbb{Q}^n$. This means we define assignments independent of the variable types. Under each such *assignment* s, a problem C (or a single constraint I) *evaluates* to a truth value, so either to true or false. We compute the truth value under the assignment s with the function $\text{eval}(C, s)$:

$$\begin{aligned}
\text{eval}(C, s) &:= \textstyle\bigwedge_{I \in C} \text{eval}(I, s) \\
\text{eval}(a_i^T x \leq b_i, s) &:= a_i^T s \leq b_i \\
\text{eval}(a_i^T x < b_i, s) &:= a_i^T s < b_i \\
\text{eval}(d_i \mid a_i^T x - b_i, s) &:= d_i \mid a_i^T s - b_i
\end{aligned}$$

This means that (i) the evaluation happens independent of the variable types; (ii) a constraint system evaluates to true if and only if all its constraints evaluate to true; (iii) a non-strict inequality $a_i^T x \leq b_i$ evaluates to true if and only if the constant term $a_i^T s$ simplifies to a rational value s' that is less than or equal to b_i; (iv) a strict inequality $a_i^T x < b_i$ evaluates to true if and only if the constant term $a_i^T s$ simplifies to a rational value s' that is less than b_i; and (v) a divisibility constraint $d_i \mid a_i^T x - b_i$ evaluates to true if and only if the constant term $a_i^T s - b_i$ simplifies to an integer value s' that is divisible by d_i.

If a constraint (system) evaluates to true under a given assignment s, then we also say that s *satisfies* the constraint (system). Moreover, we typically abbreviate $\text{eval}(C, s)$ by writing $C(s)$. This means that a constraint system C is not only a set of constraints and the conjunction over the same constraints but also a function from $\mathbb{Q}^n$ to $\{true, false\}$.

Based on these satisfying assignments, we define *solutions* to constraint systems as follows:

Definition 2.2.1 ((Mixed) Solutions)**.** A *(mixed) solution* is a point $s \in (\mathbb{Q}^{n_1} \times \mathbb{Z}^{n_2})$ that satisfies all constraints in C and also the types of all variables in C. Moreover, we denote by $\mathcal{M}(C) = \{s \in (\mathbb{Q}^{n_1} \times \mathbb{Z}^{n_2}) : C(s)\}$ the *set of mixed solutions* to the problem C.

Finding such a mixed solution is the main goal of our linear arithmetic decision procedures. However, there are also other satisfying assignments that are categorized as solutions by the literature [100, 128]:

Definition 2.2.2 (Rational Solutions)**.** A *rational solution* is a point $s \in \mathbb{Q}^n$ that satisfies all constraints in C, but not the types of the integer variables. Moreover, we denote by $\mathcal{Q}(C) = \{s \in \mathbb{Q}^n : C(s)\}$ the *set of rational solutions* to the problem C.

Definition 2.2.3 (Integer Solutions)**.** An *integer solution* is an integer point $s \in \mathbb{Z}^n$ that satisfies all constraints in C and assigns all variables to integer values. Moreover, we denote by $\mathcal{Z}(C) = \{s \in \mathbb{Z}^n : C(s)\}$ the *set of integer solutions* to the problem C.

The literature uses these two additional types of solutions because most algorithms that search for a mixed solution do so by solving intermediate constraint systems with relaxed or strengthened variable types.[3] As a result, it became standard to base most definitions in linear arithmetic on the relevant solution type instead of the actual variable types. Variable types are only relevant so we know which variables are supposed to be assigned to integer values in a mixed solution.

Now using these solution types we formally define the goal of linear rational/integer/mixed arithmetic independent of variable types: the goal of linear rational/integer/mixed arithmetic is to prove that a given input problem is *rational/integer/mixed satisfiable*, i.e., has a rational/integer/mixed solution, or that it is *rational/integer/mixed unsatisfiable*, i.e., the constraints build a *conflict* and have no rational/integer/mixed solution [4]. We specify at the beginning of every chapter (in the chapter specific preliminaries) the theory we are looking at and the type of solutions/satisfiability we are looking for. In this chapter, we are looking at the theory of linear mixed arithmetic and we abbreviate with C is (un)satisfiable that C is mixed (un)satisfiable.

[3] We call the version of C where all variables are relaxed to rational variables the *(rational) relaxation* of C.

[4] This means that the variables in our constraint systems are essentially existentially quantified for the standard goal of linear arithmetic.

Finding an integer or mixed solution for any constraint system is an NP-complete task [119]. Finding a rational solution for a *system of inequalities*, i.e., a constraint system without divisibility constraints, is much easier and can be done in polynomial time [93, 95]. However, finding a rational solution is again an NP-complete task if our constraint system contains divisibility constraints. The reason is that we can restrict our rational solutions to integer assignments by adding constraints $1 \mid x_j$ for all variables x_j.

2.2.3 Equivalent and Equisatisfiable

Now that we have defined the solutions to our constraints systems, we can also define when two constraint systems are *equivalent*. Two constraints systems are equivalent if they have the same solutions. Since we have three different types of solutions, this means that we also need three different types of *equivalence*.

Definition 2.2.4 (Equivalences)**.** Let C_1 and C_2 be constraint systems. C_1 is *(rationally) equivalent* to C_2 if $\mathcal{Q}(C_1) = \mathcal{Q}(C_2)$. C_1 is *mixed equivalent* to C_2 if $\mathcal{M}(C_1) = \mathcal{M}(C_2)$. C_1 is *integer equivalent* to C_2 if $\mathcal{Z}(C_1) = \mathcal{Z}(C_2)$.

Due to the subset relationship of our solution sets, we get the following relationship between our equivalences: C_1 is rationally equivalent to C_2 implies that C_1 is mixed equivalent to C_2; and C_1 is mixed equivalent to C_2 implies that C_1 is integer equivalent to C_2. The reverse is not necessarily true.

We typically talk about equivalent constraint systems when we have to transform our original system into an equivalent system. We need such transformations whenever our original system is too hard, either because it is too hard to solve or because it does not comply to some requirements of our algorithms/theorems.

Obviously, we cannot always find easier equivalent systems. A good alternative is to look for systems that are easier but only *equisatisfiable*:

Definition 2.2.5 (Equisatisfiability)**.** Let C_1 and C_2 be constraint systems. C_1 is *rationally equisatisfiable* to C_2 if the following two statements are equivalent: C_1 is rationally satisfiable and C_2 is rationally satisfiable. C_1 is *mixed equisatisfiable* to C_2 if the following two statements are equivalent: C_1 is mixed satisfiable and C_2 is mixed satisfiable. C_1 is *integer equisatisfiable* to C_2 if the following two statements are equivalent: C_1 is integer satisfiable and C_2 is integer satisfiable.

Equisatisfiable systems are especially useful if there is an efficient way to convert solutions between the two equisatisfiable systems.

2.2.4 Substitutions

Sometimes we do not want to assign all variables x to a fixed value s but just one variable x_j to another linear expression p_j, which does not contain x_j. So we want to replace all occurrences of x_j with p_j. We call this process *substituting* x_j with p_j and denote it by writing $C\{x_j \mapsto p_j\}$. We also assume without loss of generality that the substitution $C\{x_j \mapsto p_j\}$ automatically transforms all of the constraints back into the original representation format.

We are also able to substitute multiple variables x_j at once if the substituted variables do not appear in any of the replacement expressions p_j. As with the single variable substitution, we denote the substitution over multiple variables as a set

$$\sigma_{y,z}^{D,c} := \{y_i \mapsto c_i + d_i^T z : i \in \{1, \ldots, n_y\}\},$$

where $y = (y_1, \ldots, y_{n_y})^T$ and $z = (z_1, \ldots, z_{n_z})^T$ are a partition of the variables in x, $D = (d_1^T, \ldots, d_{n_y}^T)^T \in \mathbb{Q}^{n_y \times n_z}$, and $c \in \mathbb{Q}^{n_y}$. Semantically, $C\sigma_{y,z}^{D,c}$ means that each variable y_i is substituted with the term $c_i + d_i^T z$.

2.3 Reduction to Non-Strict Inequalities

The common definition of linear arithmetic in the literature supports only one type of constraint: non-strict inequality constraints [28, 83, 89, 93, 103, 128]. There are multiple reasons for this choice.

First of all, the rational solutions of system of inequalities have some very useful properties. For instance, they define a *polyhedron* in the n-dimensional vector space $\mathbb{Q}^n$, where each rational solution is equivalent to a point in the polyhedron [128]. In the case that our system C contains only non-strict inequalities, the polyhedron is even *closed convex* [128]. This entails two very useful properties: firstly, the closed convex polyhedron has a surface if it is neither empty nor encompasses the whole $\mathbb{Q}^n$; secondly, any supremum $h_{\max} = \sup\{h^T x : x \in \mathbb{Q}^n \text{ satisfies } C\}$ over a linear objective $h \in \mathbb{Q}^n$ is either $h_{\max} = -\infty$ because there exists no point satisfying our constraints, $h_{\max} = \infty$ because the supremum is unbounded, or there exists an actual maximum, i.e., there exists an $x \in \mathbb{Q}^n$ that satisfies the constraints and its cost $h^T x$ is equal to our supremum $h_{\max}$.

Adding strict inequalities to our constraint system has the effect that our rational solutions are no longer guaranteed to be a closed set. And if we add divisibility constraints, then our rational solutions describe no longer a polyhedron nor are they a convex set. However, these properties are necessary for most classical algorithms and theorems in linear arithmetic. For instance, the classical dual simplex algorithm [128] returns only rational

solutions on the surface of the polyhedron. It is, therefore, not trivial to adapt all classical algorithms and theorems to also handle strict inequalities or divisibility constraints directly. Instead, we show how to represent both divisibility constraints as well as strict inequalities via non-strict inequalities.

2.3.1 Reducing Divisibility Constraints

We can transform any divisibility constraint and negated divisibility constraint into an equality by introducing additional variables q (and r). For divisibility constraints $d_i \mid a_i^T x - b_i$, this transformation is known as the *diophantine representation* [44]:

$$\{d_i \mid a_i^T x - b_i\} \equiv \exists q \in \mathbb{Z}.\{d_i q - a_i^T x = -b_i\},$$

where q is a fresh integer variable. For negated divisibility constraints $d_i \nmid a_i^T x - b_i$, there exists a similar transformation:

$$\{d_i \nmid a_i^T x - b_i\} \equiv \exists q \in \mathbb{Z}, r \in \mathbb{Q}.\{d_i q + r - a_i^T x = -b_i \ , \ 0 < r < d\},$$

where q is a fresh integer variable and r a fresh rational variable. Both of these transformations describe the formal definition of dividing $a_i^T x - b_i$ by d_i, i.e., $a_i^T x - b_i = d_i q + r$, where q is the quotient of the division and r the remainder. Since the divisibility constraint enforces that d_i divides $a_i^T x - b_i$, the remainder r must be zero. Likewise, the negated divisibility constraint enforces that d_i does not divide $a_i^T x - b_i$. Therefore, the remainder r lies in the open interval $(0, d)$.

It is actually a big disadvantage that we have to introduce additional variables q (and r) whenever we want to get rid of a (negated) divisibility constraint because divisibility constraints are typically introduced to eliminate variables. For instance, quantifier elimination techniques for linear integer arithmetic introduce divisibility constraints in order to eliminate a quantified variable (see also Chapter 3.4 and [44]). So eliminating the newly introduced divisibility constraint by adding another quantified variable would be counter productive.

Another disadvantage is that the above reduction is only mixed equisatisfiable and not rationally equisatisfiable. For instance, $1 \mid x_j$ guarantees that x_j is assigned to an integer value in all rational solutions and the diophantine representation $q = x_j$ does not. This is due to the fact that the transformation moves the integer guarantee out of the evaluation semantics of divisibility constraints and into the type condition of the new variable q.

These disadvantages lead us to the following consequences: (i) we never eliminate divisibility constraints introduced during one of our algorithms and (ii) we are careful to differentiate between the rational relaxation of the original problem and the rational relaxation of the transformed problem.

2.3.2 Reducing Strict Inequalities

We model strict inequalities as non-strict inequalities by generalizing the field $\mathbb{Q}$ for our inequality bounds b_i and our variable assignments to $\mathbb{Q}_\delta$ [57].

Lemma 2.3.1 (δ-Rationals [57])**.** *Let $a_i \in \mathbb{Q}^n$ and $b_i \in \mathbb{Q}$. Then a set of linear arithmetic constraints C that contains strict inequalities*

$$C \supseteq C' = \{a_1^T x < b_1, \ldots, a_m^T x < b_m\}$$

is rationally satisfiable iff there exists a rational number $\delta > 0$ such that $C_{\delta'} = (C \cup C'_{\delta'}) \setminus C'$ is rationally satisfiable for all δ' with $0 < \delta' \leq \delta$, where

$$C'_{\delta'} = \{a_1^T x \leq b_1 - \delta', \ldots, a_m^T x \leq b_m - \delta'\}.$$

We express the above observation symbolically as an *infinitesimal parameter* $\delta > 0$. This leads to the ordered vector space $\mathbb{Q}_\delta = \mathbb{Q} \times \mathbb{Q}$, which we call the *$\delta$-rationals*, that has pairs of rationals as elements $(p, q) \in \mathbb{Q}_\delta$ representing $p + q\delta$ with the following operations:

$$\begin{aligned}
(p_1, q_1) + (p_2, q_2) &\equiv (p_1 + p_2, q_1 + q_2) \\
a \cdot (p_1, q_1) &\equiv (a \cdot p_1, a \cdot q_1) \\
(p_1, q_1) \leq (p_2, q_2) &\equiv (p_1 < p_2) \vee (p_1 = p_2 \wedge q_1 \leq q_2) \\
(p_1, q_1) < (p_2, q_2) &\equiv (p_1 < p_2) \vee (p_1 = p_2 \wedge q_1 < q_2) \\
d_1 \mid (p_1, q_1) &\equiv (q_1 = 0) \wedge (d_1 \mid p_1) \\
\lfloor (p_1, q_1) \rfloor &\equiv \begin{cases} \lfloor p_1 \rfloor & \text{for } p_1 \notin \mathbb{Z} \\ p_1 - 1 & \text{for } p_1 \in \mathbb{Z} \text{ and } q_1 < 0 \\ p_1 & \text{else} \end{cases} \\
\lceil (p_1, q_1) \rceil &\equiv \begin{cases} \lceil p_1 \rceil & \text{for } p_1 \notin \mathbb{Z} \\ p_1 + 1 & \text{for } p_1 \in \mathbb{Z} \text{ and } q_1 > 0 \\ p_1 & \text{else} \end{cases} \\
\lceil (p_1, q_1) \rfloor &\equiv \lfloor (p_1 + 0.5, q_1) \rfloor \\
|(p_1, q_1)| &\equiv (|p_1|, |q_1|) \\
\|p + q\delta\|_d &\equiv (\|p\|_d, \|q\|_d) \\
\mathrm{dist}_d(b, c) &\equiv \|b - c\|_d
\end{aligned}$$

where $p_i, q_i \in \mathbb{Q}$, $p = (p_1, \ldots, p_n)^T$, $q = (q_1, \ldots, q_n)^T$, $a \in \mathbb{Q}$, $d_1 \in \mathbb{Z}$ with $d_1 > 0$, $d \in \mathbb{Z} \cup \{\infty\}$ with $d > 0$, and $b, c \in \mathbb{Q}_\delta^n$ [57] [5]. Given the δ-rationals and Lemma 2.3.1, we can now represent $a_i^T x < b_i$ by $a_i^T x \leq b_i - \delta$, where $a_i \in \mathbb{Q}^n$ and $b_i \in \mathbb{Q}$. For the remainder of this thesis, we abbreviate with b_i^δ the strict version of a given bound $b_i \in \mathbb{Q}_\delta$. If the bound b_i is non-strict, i.e., $b_i = (p_i, 0)$, then the strict version is $b_i^\delta := (p_i, -1)$. Otherwise, the bound b_i is already strict, i.e., $b_i = (p_i, q_i)$ with $q_i < 0$, and we just standardize the δ-coefficient to -1, i.e, $b_i^\delta := (p_i, -1)$. Furthermore, we abbreviate with

[5] Note that Lemma 2.1.1 still works for $\mathbb{Q}_\delta$ with the above definitions for $\lceil . \rfloor$ and $\mathrm{dist}_d(.,.)$. However, the range of $\mathrm{dist}_d(.,.)$ is only $\mathbb{Q}_\delta$ for $d = 1$ and $d = \infty$ and $\mathbb{R}_\delta = \mathbb{R} \times \mathbb{R}$ otherwise.

$\bar{b_i}$ that we swap between the strict and non-strict version. If b_i is non-strict (i.e., $b_i = (p_i, 0)$), then $\bar{b_i} := b_i^\delta = (p_i, -1)$ gives us the strict version b_i. If b_i is strict (i.e., $b_i = (p_i, q_i)$ with $q_i < 0$), then $\bar{b_i} := (p_i, 0)$ gives us the non-strict version of b_i.

As mentioned before, we now also let the assignments for our variables range over $\mathbb{Q}_\delta$. This means we now also have *δ-rational solutions* and *δ-mixed solutions* and we denote their solution sets by $\mathcal{Q}_\delta(C) = \{s \in \mathbb{Q}_\delta^n : C(s)\}$ and $\mathcal{M}_\delta(C) = \{s \in (\mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_2}) : C(s)\}$. Similar to rational inequality systems, the solutions of our δ-rational inequalities describe a closed convex polyhedron in the n-dimensional $\mathbb{Q}_\delta^n$ and methods like the classical simplex algorithm are again complete.

It is also easy to extract a purely rational/mixed solution $s' \in \mathbb{Q}^n$ from a δ-rational/mixed solution $s \in \mathbb{Q}_\delta^n$. We just have to choose a small enough value $\delta' \in \mathbb{Q}$ and replace the parameter δ with this value (Lemma 2.3.1). Therefore, we call δ-rational solutions and δ-mixed solutions just rational solutions and mixed solutions, respectively. Analogously, we also say rationally/mixed satisfiable, rationally/mixed equivalent, and rationally/mixed equisatisfiable when we are actually talking about the δ-rational/δ-mixed versions of these expressions.

Representing our strict inequalities as non-strict inequalities also allows us to use the second property listed above for closed convex polyhedra, i.e., any supremum $h_{\max} = \sup\{h^T x : x \in \mathbb{Q}_\delta^n \text{ satisfies } C\}$ over a linear objective $h \in \mathbb{Q}^n$ is either $h_{\max} = -\infty$, $h_{\max} = \infty$, or there exists an actual maximum and not just a limit. This property is especially useful on techniques based on optimization like the largest cube test (see Chapter 4).

There exists an alternative and much easier elimination if all variables of a strict inequality are integer variables. In this case, the equivalence $a_i^T x < b_i \equiv a_i^T x \leq \lceil b_i \rceil - 1$ can be used to replace strict inequalities with non-strict inequalities [128]. This version is to be preferred whenever possible as it removes some (δ-)rational solutions that would violate the type of the integer variables.

2.4 Standard Input Formats

We previously gave a general definition for constraint systems. This definition not only included non-strict inequalities, but also strict inequalities and divisibility constraints. We need strict inequalities because the negation of a non-strict inequality is a strict inequality, i.e., $\neg(a_i^T x \leq b_i) \equiv a_i^T x < b_i$. This means that a theory solver for linear arithmetic, i.e., a decision procedure for conjunctions of linear arithmetic literals, has to handle strict inequalities in some ways. Furthermore, we need divisibility constraints because they are introduced during the execution of some of our algorithms (CutSat, CutSat++, and CutSat$_g$).

In the last section, we also explained that we cannot adept classical linear arithmetic algorithms and theorems directly to strict inequalities and divisibility constraints. To resolve this problem, we also presented reductions that turn our too general constraints into non-strict inequalities. This allows us to define our *standard input problems*—or for short *problems*—as systems of non-strict inequalities, which we typically abbreviate as *systems of inequalities* or *inequality systems*.[6] Thereby, we are also conforming to classical linear arithmetic [28, 83, 89, 93, 103, 128].

Later in this section, we also present an alternative input format called the *tableau representation*. We switch to this format whenever we are discussing algorithms in the context of SMT solvers because the tableau representation matches the actual representation inside most SMT implementations [9, 40, 41, 49, 56, 57].

2.4.1 Systems of Inequalities

A *system of inequalities* is a set of inequalities $\{a_1^T x \leq b_1, \ldots, a_m^T x \leq b_m\}$, which we typically abbreviate as $Ax \leq b$ [100]. The row coefficients are given by $A = (a_1, \ldots, a_m)^T \in \mathbb{Q}^{m \times n}$, the variables are given by $x = (x_1, \ldots, x_n)^T$, and the inequality bounds are given by $b = (b_1, \ldots, b_m)^T \in \mathbb{Q}_\delta^m$. Moreover, we assume that any constant rows $a_i = 0^n$ were eliminated from our system during an implicit preprocessing step. This is a trivial task and eliminates some unnecessarily complicated corner cases.

Since $Ax \leq b$ and $A'x \leq b'$ are just sets, we can write their combination as $(Ax \leq b) \cup (A'x \leq b')$. A special system of inequalities is a *system of equations* $Dx = c$, which is equivalent to the combined system of inequalities $(Dx \leq c) \cup (-Dx \leq -c)$. For such a system of equalities, the row coefficients are given by $D = (d_1, \ldots, d_m)^T \in \mathbb{Q}^{m \times n}$, the variables are given by $x = (x_1, \ldots, x_n)^T$, and the equality bounds are given by $c = (c_1, \ldots, c_m)^T \in \mathbb{Q}^m$.

The δ-coefficients q_i in the bounds $b_i = p_i + q_i\delta$ can take on any value in $\mathbb{Q}_\delta$. If $q_i = 0$, then the inequality $a_i^T x \leq b_i$ is equivalent to the non-strict inequality $a_i^T x \leq p_i$. If $q_i < 0$, then the inequality $a_i^T x \leq b_i$ is equivalent to the strict inequality $a_i^T x < p_i$. If $q_i > 0$, then we have no clear interpretation over the actual rationals (compare also Lemma 2.3.1). For instance, the two inequalities $x_1 \leq \delta$ and $-x_1 \leq -\delta$ describe a rationally satisfiable system of constraints in $\mathbb{Q}_\delta$, but there is no clear way of interpreting $x_1 \leq \delta$ in $\mathbb{Q}$. Beware also that some of our methods (e.g., the linear cube transformation) can introduce positive δ-coefficients in the bounds. But since we derive all our methods with a semantically clear construction, the semantic interpretation over the rationals is still discernible if the original system has only non-positive δ-coefficients in its inequality bounds before the transformation.

[6] *Polyhedron* is another alternative name for an inequality system. We use it mainly when we are looking at a system from a geometric perspective.

2.4.2 Tableau Representation

We defined our standard input problems as systems of inequalities $Ax \leq b$. We do so because most theorems in the literature as well as our own theorems can be proven more intuitively with inequalities. There are, however, some algorithms, e.g., the dual simplex algorithm we present in Section 2.7, for which we prefer a different representation of our input constraints: the *tableau representation* [57]. In the tableau representation, we partition our variables into two sets: the set of *non-basic variables* $z_1, \ldots, z_n \in \mathcal{N}$ and the set of *basic variables* $y_1, \ldots, y_m \in \mathcal{B}$. The constraints of the tableau representation are then defined as: the so-called *tableau* $Az = y$ and a set of bounds for the variables $\mathcal{L}(x_j) \leq x_j \leq \mathcal{U}(x_j)$ (for $x_j \in \mathcal{N} \cup \mathcal{B}$). The tableau representation also features two functions $\mathcal{L} : \mathcal{B} \cup \mathcal{N} \to \mathbb{Q}_\delta \cup \{-\infty\}$ and $\mathcal{U} : \mathcal{B} \cup \mathcal{N} \to \mathbb{Q}_\delta \cup \{\infty\}$ that map the variables $x_i \in \mathcal{B} \cup \mathcal{N}$ to their upper and lower bound values, respectively. The lower bound value $\mathcal{L}(x_j)$ is $-\infty$ for variable x_j if x_j has no (explicit) lower bound. Similarly, the upper bound value $\mathcal{U}(x_j)$ is ∞ for variable x_j if x_j has no (explicit) upper bound.

We can easily transform a system of inequalities $Ax \leq b$ into tableau representation by introducing a so-called *slack variable* s_i for every inequality in our system. The type of the slack variable is typically rational, but can be set to integer if its row coefficients a_i and all variables with non-zero coefficients are also integers. Our system is then defined by the equalities $Az = y$ (with $z := x$ and $y := s$) and the bound values $\mathcal{L}(x_j) := -\infty$ and $\mathcal{U}(x_j) := \infty$ for every original variable x_j and the bound values $\mathcal{L}(s_i) := -\infty$ and $\mathcal{U}(s_i) := b_i$ for every slack variable introduced for the inequality $a_i^T x \leq b_i$. So initially, our original variables are the non-basic variables and the slack variables are the basic variables.

We can even reduce the number of slack variables if we transform inequalities of the form $a_{ij} \cdot x_j \leq b_i$ directly into bounds for x_j. Moreover, we can use the same slack variable for multiple inequalities as long as the left side of the inequality is similar enough. For example, the inequalities $a_i^T x \leq b_i$ and $-a_i^T x \leq c_i$ can be transformed into the equality $a_i^T x = s_i$ and the bound values $\mathcal{L}(s_i) := -c_i$ and $\mathcal{U}(s_i) := b_i$.

SMT solvers typically assign the slack variables during a preprocessing step with a normalization procedure based on a variable ordering. After the normalization, all terms are represented in one directed acyclic graph (DAG) so that all equivalent terms are assigned to the same node and, thereby, to the same slack variable. For more details on these simplifications we refer to [57].

2.5 Implied Constraints and Linear Combinations

We say that a constraint system C *implies* or *entails* a (set of) constraint(s) I if I evaluates to true for all $s \in \mathcal{Q}_\delta(C)$ [128].[7] We also denote this relationship by writing $C \vdash I$. Although equalities and other highlighted constraints are not part of our standard constraints, we still say that a constraint system C implies a highlighted constraint if its equivalent standard representation evaluates to true for all $s \in \mathcal{Q}_\delta(C)$. For instance, $C \vdash h^T x = g$ if and only if $\{h^T x \leq g, -h^T x \leq -g\}$ evaluates to true for all $s \in \mathcal{Q}_\delta(C)$. A constraint I implied by C is *explicit* if it does appear in C, i.e., $I \in C$. A highlighted constraint I implied by C is also called *explicit* if its equivalent standard representation C' appears in C, i.e., $C' \subseteq C$. Otherwise, implied (highlighted) constraints are called *implicit*.

Besides our *general/rational entailments* $C \vdash I$ over the rationals, we also use *mixed entailments* and *integer entailments* denoted by $C \vdash_{\mathcal{M}} I$ and $C \vdash_{\mathcal{Z}} I$, respectively. And similarly to our general entailment definition, $C \vdash_{\mathcal{M}} I$ if I evaluates to true for all $s \in \mathcal{M}_\delta(C)$ and $C \vdash_{\mathcal{Z}} I$ if I evaluates to true for all $s \in \mathcal{Z}(C)$. Since $\mathcal{Z}(C) \subseteq \mathcal{M}_\delta(C) \subseteq \mathcal{Q}_\delta(C)$, this also means that all generally entailed constraints are also mixed entailed and all mixed entailed also integer entailed, i.e., if $C \vdash I$, then $C \vdash_{\mathcal{M}} I$ and, if $C \vdash_{\mathcal{M}} I$, then $C \vdash_{\mathcal{Z}} I$. The reverse is not true as shown by the following example: $\{2x_1 + 4x_2 \leq 7\} \vdash_{\mathcal{Z}} x_1 + 2x_2 \leq \frac{7}{2}$ and $\{2x_1 + 4x_2 \leq 7\} \vdash x_1 + 2x_2 \leq \frac{7}{2}$ because dividing a constraint by a positive constant factor does not change its solutions. However, we also know that $\{2x_1 + 4x_2 \leq 7\} \vdash_{\mathcal{Z}} x_1 + 2x_2 \leq 3$ because the term $x_1 + 2x_2$ evaluates under every integer assignment to an integer value so the fractional bound of $x_1 + 2x_2 \leq \frac{7}{2}$ can be rounded down without losing any integer solutions. In contrast, $\{2x_1 + 4x_2 \leq 7\}$ does not generally entail $x_1 + 2x_2 \leq 3$ because the assignment $(\frac{7}{2}, 0)^T$ is a rational solution for $2x_1 + 4x_2 \leq 7$ but not for $x_1 + 2x_2 \leq 3$.

There are types of constraint systems for which we can constructively define their implied/entailed constraints. For instance, any constraint system without rational solutions entails all constraints. Similarly, any constraint system without mixed solutions mixed entails all constraints and integer solutions integer entails all constraints.

Corollary 2.5.1 (Unsatisfiable Entailments)**.** *Let I be any linear arithmetic constraint. If $\mathcal{Q}_\delta(C) = \emptyset$, then $C \vdash I$. If $\mathcal{M}_\delta(C) = \emptyset$, then $C \vdash_{\mathcal{M}} I$. If $\mathcal{Z}(C) = \emptyset$, then $C \vdash_{\mathcal{Z}} I$.*

[7] Note that the general definition for implied constraints is related to the rational solutions of the problem and not the mixed solutions. This is standard in the literature [128]. Later in this section, we will also define two other types of implied constraints that are related to mixed and integer solutions.

The implied/entailed inequalities of a systems of inequalities $Ax \leq b$ can also be defined constructively. Either $Ax \leq b$ has no rational solution and implies all constraints (see Corollary 2.5.1) or all implied inequalities are so-called *linear combinations of the inequalities* in $Ax \leq b$. A linear combination of the inequalities in $Ax \leq b$ is an inequality $h^T x \leq g$ that resulted from multiplying each inequality in $Ax \leq b$ by a non-negative factor and adding them up. Formally this means that there exists a vector $y \in \mathbb{Q}^m$ with $y \geq 0^m$, $y^T A = h^T$, and $y^T b \leq g$. It is relatively easy to prove that any linear combination of $Ax \leq b$ is also implied by $Ax \leq b$:

Corollary 2.5.2 (Linear Combination Corollary [128])**.** *Let there exists a $y \in \mathbb{Q}^m$ with $y \geq 0^m$, $y^T A = h^T$, and $y^T b \leq g$, i.e., there exists a linear combination of inequalities in $Ax \leq b$ that results in the inequality $h^T x \leq g$. Then $Ax \leq b$ implies $h^T x \leq g$.*

Proof. If $s \in \mathcal{Q}_\delta(Ax \leq b)$, then all $a_i^T s \leq b_i$ are true. Since all $a_i^T s \leq b_i$ are constant, we also know that $h^T s = y^T A s \leq y^T b \leq g$ is true. Hence, every $s \in \mathcal{Q}_\delta(Ax \leq b)$ is also a solution of $h^T x \leq g$. So $Ax \leq b$ implies $h^T x \leq g$. □

Before we can finish the proof that all implications are linear combinations, we need the help of the following very important and famous lemma:

Lemma 2.5.3 (Farkas' Lemma [60, 107])**.** *$\mathcal{Q}(Ax \leq b) = \emptyset$ iff there exists a $y \in \mathbb{Q}^m$ with $y \geq 0^m$ and $y^T A = (0^n)^T$ so that $y^T b < 0$, i.e., there exists a non-negative linear combination of inequalities in $Ax \leq b$ that results in an inequality $y^T Ax \leq y^T b$ that is constant and unsatisfiable.*

Farkas' Lemma was originally formulated over the standard rationals and not the δ-rationals. So we have to prove that Farkas' Lemma still works with δ-rationals:

Lemma 2.5.4 (Farkas' Lemma for $\mathbb{Q}_\delta$)**.** *$\mathcal{Q}_\delta(Ax \leq b) = \emptyset$ iff there exists a $y \in \mathbb{Q}^m$ with $y \geq 0^m$ and $y^T A = (0^n)^T$ so that $y^T b < 0$, i.e., there exists a non-negative linear combination of inequalities in $Ax \leq b$ that results in an inequality $y^T Ax \leq y^T b$ that is constant and unsatisfiable. If such a y exists, then we call it a* certificate of unsatisfiability *or alternatively a conflict (explanation).*

Proof. Let us first consider the case where $Ax \leq b$ is rationally unsatisfiable. Dutertre and de Moura's version of the dual simplex algorithm is a complete and correct algorithm for determining the satisfiability of a linear arithmetic problem over the δ-rationals (for more details see Section 2.7 and [57]). In case the problem is rationally unsatisfiable, the algorithm returns a conflict explanation, which can be turned, together with the final simplex tableau, into the linear combination $y \in \mathbb{Q}^m$ we are looking for. Let us now consider

the case where $s \in \mathbb{Q}_\delta^n$ is a solution for $Ax \leq b$. Due to Corollary 2.5.2, we know that s is also a solution to any linear combination $y^T A \leq y^T b$ of inequalities in $Ax \leq b$. So there cannot exist a $y \in \mathbb{Q}^m$ with $y \geq 0^m$, $y^T A = (0^n)^T$, and $y^T b < 0$ because $0 = (0^n)^T s = y^T As \leq y^T b < 0$ is a contradiction. □

Now given the δ-rational version of Farkas' Lemma, we can prove that either $Ax \leq b$ has no rational solution or all implied inequalities are *linear combinations* of the inequalities in $Ax \leq b$:

Lemma 2.5.5 (Linear Implication Lemma [128][8])**.** *Let $\mathcal{Q}_\delta(Ax \leq b) \neq \emptyset$, $h \in \mathbb{Q}^n \setminus \{0^n\}$, and $g \in \mathbb{Q}_\delta$. Then $Ax \leq b$ implies $h^T x \leq g$ iff there exists a $y \in \mathbb{Q}^m$ with $y \geq 0^m$ and $y^T A = h^T$ so that $y^T b \leq g$, i.e., there exists a non-negative linear combination of inequalities in $Ax \leq b$ that results in the inequality $h^T x \leq g$.*

Proof. Let us first consider the case where $Ax \leq b$ implies the inequality $h^T x \leq g$. In this case, the system $Ax \leq b \cup \{-h^T x \leq \overline{(-g)}\}$ is rationally unsatisfiable. It follows by Lemma 2.5.4 that there exists a (i) $y \geq 0^m$ and a $y_{m+1} \geq 0$ such that (ii) $y^T A - y_{m+1} h = 0^n$ and (iii) $y^T b + y_{m+1}\overline{(-g)} < 0$. It also follows by Lemma 2.5.4 and $Ax \leq b$ being rationally satisfiable that $y_{m+1} > 0$. Now we choose the linear combination $y' = \frac{1}{y_{m+1}} y$. This linear combination is positive because of (i). Moreover, it holds that $y'^T A = h^T$ because of (ii). Finally, it holds that $y'^T b < g$ because of (iii), which implies that $y'^T b \leq g$. Hence, $y' = \frac{1}{y_{m+1}} y$ is the linear combination that gets us $h^T x \leq g$.

Let us now consider the case where there exists a $y \in \mathbb{Q}^m$ with $y \geq 0^m$ and $y^T A = h^T$ so that $y^T b \leq g$. If $s \in \mathcal{Q}_\delta(Ax \leq b)$, then s also satisfies $h^T x \leq g$ since $h^T x = y^T Ax \leq y^T b \leq g$. Hence, $Ax \leq b \vdash h^T x \leq g$. □

Linear combinations are not just useful for their relationship to implied inequalities. We can also use them to prove many more theorems over systems of linear inequalities. For instance, we can prove some properties for *minimal sets of unsatisfiable inequalities.*

2.5.1 Minimal Sets of Unsatisfiable Inequalities

We call an unsatisfiable set C of inequalities *minimal* if every proper subset $C' \subset C$ is satisfiable. Whenever a polyhedron $Ax \leq b$ is rationally unsatisfiable (i.e., $\mathcal{Q}_\delta(Ax \leq b) = \emptyset$), then there exists a minimal set C of rationally unsatisfiable inequalities so that every inequality in C appears also

[8] This lemma is also called the "affine" form of Farkas' Lemma [128].

in $Ax \leq b$ [57]. We call such a minimal set C a *minimal conflict (explanation)* for $Ax \leq b$'s rational unsatisfiability. If we are investigating a minimal set of unsatisfiable inequalities, then we can strengthen Farkas' Lemma as follows:

Lemma 2.5.6 (Farkas' Lemma for Explanations)**.** *Let the system of inequalities $C = \{a_i^T x \leq b_i : 1 \leq i \leq m\}$ be a minimal set of rationally unsatisfiable inequalities. Let $A = (a_1, \ldots, a_m)^T$ and $b = (b_1, \ldots, b_m)^T$. Then it holds for every $y \in \mathbb{Q}^m$ with $y \geq 0^m$, $y^T A = (0^n)^T$, and $y^T b < 0$ that $y_i > 0$ for all $i \in \{1, \ldots, m\}$.*

Proof. Suppose to the contrary that there exists a $y \geq 0^m$ with $y^T A = (0^n)^T$ and $y^T b < 0$ such that one component of y is zero. Without loss of generality we assume that $y_m = 0$. Let $C = \{a_i^T x \leq b_i : 1 \leq i \leq m-1\}$, $A' = (a_1, \ldots, a_{m-1})^T$, $b' = (b_1, \ldots, b_{m-1})^T$, and $y' = (y_1, \ldots, y_{m-1})^T$. Then, $y' \geq 0^{m-1}$, ${y'}^T A' = (0^n)^T$, and ${y'}^T b' < 0$. However, by Lemma 2.5.4, this implies that $(A'x \leq b') \subset C$ is rationally unsatisfiable. Therefore, C is not minimal, which contradicts our initial assumptions. □

2.6 CDCL(T): A Framework for SMT Solvers

In this thesis, we present new linear arithmetic techniques for SMT (satisfiability modulo theories) solving. To be more precise, the techniques that we present in Chapters 3–6 are meant to be integrated inside a theory solver for *CDCL(T)* [68], which is a framework used by most SMT solvers [9, 40, 41, 49, 56].[9]

CDCL(T) [68], also called *DPLL(T)*, is a very general framework that describes a set of interactions between a CDCL-based (conflict-driven-clause-learning-based) SAT solver [139] and a theory solver for a given theory T [129].[10] If a SAT solver and a theory solver are combined based on these interactions, then they become a decision procedure for ground formulas in clause normal form over the given theory T. Originally, these interactions were described through an interface for the theory solver and included four operations: (i) assert a literal, (ii) check currently asserted literals for theory satisfiability, (iii) return conflict explanation, and (iv) backtrack.

These interactions are used in CDCL(T) as follows: CDCL(T) first creates a propositional abstraction of the input formula, i.e., it replaces the theory atoms with fresh propositional variables. (A map from propositional variables to theory atoms is created as part of this abstraction.) As its se-

[9] In Chapter 7, we describe the implementation of our techniques in our theory solver SPASS-IQ. In Chapter 8, we describe the implementation of our own CDCL(LA) solver SPASS-SATT based on SPASS-IQ.

[10] A theory solver is a decision procedure for the conjunctive fragment of the theory; so systems of linear inequalities in the case of linear arithmetic.

cond step, CDCL(T) uses a CDCL-based SAT solver to find a propositional model that satisfies the abstracted formula. Next, CDCL(T) checks whether the propositional model is also theory satisfiability, i.e., CDCL(T) asserts in its theory solver the theory literals corresponding to the propositional model and checks them for theory satisfiability. If the propositional model is theory satisfiable, then the overall problem is theory satisfiable and CDCL(T) can stop the search. If the theory solver finds a conflict between the asserted literals, then it returns a conflict explanation. The SAT solver uses the conflict explanation for a conflict analysis that determines a good point for backtracking. Then the SAT solver goes back to the second step and selects a different propositional model that satisfies the abstracted formula. The problem is unsatisfiable if the SAT solver cannot find another propositional model that satisfies the abstracted formula.

There also have been several papers since CDCL(T) was first presented that extend the set of interactions [11, 57, 113, 114]. The most prominent examples are (i) *theory propagation* [129], i.e., propagating literals based on theory reasoning; (ii) *theory learning* [129], i.e., using theory reasoning to find and learn clauses implied by the input formula; and (iii) *(weakened) early pruning* [129], i.e., checking intermediate propositional models for theory satisfiability in order to find conflicts earlier in the SAT search. These additional interactions may not be necessary for a complete decision procedure, but they have a great impact on practical efficiency.

It is also very important for the practical efficiency of CDCL(T) that the theory solver fulfills certain properties: generation of minimal conflict explanations, high incremental efficiency, and efficient backtracking. We developed our linear arithmetic techniques with these properties in mind.

2.6.1 Propositional Abstraction

The input of CDCL(LA)[11] is a ground linear arithmetic formula in clause normal form, or formally:

$$F := \bigwedge_k C_k := \bigwedge_k \bigvee_i L_{ki},$$

where F is the whole formula, the C_i are the clauses in the formula, and the L_{ij} are the literals in the formula. Each literal L_{ij} is either a linear inequality $a_i^T x \leq b_i$ (with $b_i \in \mathbb{Q}$), a negated linear inequality $\neg(a_i^T x \leq b_i)$ (with $b_i \in \mathbb{Q}$), a propositional variable p_l, or a negated propositional variable $\neg(p_l)$. However, most CDCL(LA) implementations (SPASS-SATT included) use internally a different representation.

Firstly, most SMT theory solvers rely on the tableau representation, which means that they cannot handle linear inequalities directly. We resolve this (as explained in Chapter 2.4.2) through the introduction of slack variables s for all inequalities appearing in our formula F. The result is a

[11] CDCL(LA) = CDCL(T) where the theory T is set to linear arithmetic (LA)

tableau $Ax = s$ and a formula F' in clause normal form, where all literals are either propositional variables p_l, negated propositional variables $\neg(p_l)$, variable bounds $x_i \leq b_i$ or $x_i \geq b_i$ (with $b_i \in \mathbb{Q}$), or negated variable bounds $\neg(x_i \leq b_i)$ or $\neg(x_i \geq b_i)$ (with $b_i \in \mathbb{Q}$). Moreover, the combination of tableau $Ax = s$ and formula F' is equisatisfiable to the original formula F.

We can also get rid of the negated bounds by using equivalent bounds that rely on δ-rationals (see also Chapter 2.3.2):

$$\begin{aligned}
\neg(x_i \leq b_i) &\equiv x_i \geq b_i + \delta \quad \text{if } x_i \text{ is a rational variable,} \\
\neg(x_i \geq b_i) &\equiv x_i \leq b_i - \delta \quad \text{if } x_i \text{ is a rational variable,} \\
\neg(x_i \leq b_i) &\equiv x_i \geq b_i + 1 \quad \text{if } x_i \text{ is an integer variable,} \\
\neg(x_i \geq b_i) &\equiv x_i \leq b_i - 1 \quad \text{if } x_i \text{ is an integer variable.}
\end{aligned}$$

This means we now have a tableau $Ax = s$ and a formula F' in clause normal form, where all literals are either propositional variables p_l, negated propositional variables $\neg(p_l)$, or variable bounds $x_i \leq b_i$ or $x_i \geq b_i$ (with $b_i \in \mathbb{Q}_\delta$).

Next, we abstract our bounds to propositional variables. We do so by replacing each occurrence of a bound $x_i \leq b_i$ over a rational variable with a fresh propositional variable p_l and each symmetrical occurrence $x_i \geq b_i + \delta$ with the negated literal $\neg(p_l)$. Analogously, we replace each occurrence of a bound $x_i \leq b_i$ over an integer variable with a fresh propositional variable p_l and each symmetrical occurrence $x_i \geq b_i + 1$ with the negated literal $\neg(p_l)$. We maintain the connection between propositional variables and abstracted bounds by storing their relationship with the help of a map. The result is an equisatisfiable combination of a tableau $Ax = s$, a formula F'' in clause normal form, where all literals are either propositional variables p_l or negated propositional variables $\neg(p_l)$, and a function f that maps some of the propositional variables to bounds.

The SAT solver can now run on the propositional CNF formula F'' and select a model/set of literals. The corresponding bounds to the selected literals are then asserted by the theory solver with the help of the function f. The asserted bounds together with the tableau $Ax = s$ define a linear arithmetic problem in tableau representation. Naturally, all asserted bounds correspond to a (negated) inequality in the original formula F. What might be less obvious is that this set of (negated) inequalities is equisatisfiable to the set of asserted bounds and the tableau $Ax = s$. Therefore, we are also able to represent all linear arithmetic subproblems of F through our transformed formula.

2.7 Standard Arithmetic Decision Procedures for SMT

In this section, we give an overview over the most common linear arithmetic decision procedures used by the SMT community for the CDCL(T) framework. We start with a version of the dual simplex algorithm, which is a decision procedure for linear rational arithmetic. Next we explain bound refinements, a supporting technique that is often used to enhance decision procedures for linear arithmetic. We conclude this section by explaining branch-and-bound, a decision procedure for linear mixed/integer arithmetic, and some of its commonly used extensions.

2.7.1 A Simplex Version for SMT

Most SMT solvers implement the following version of the dual simplex algorithm as their linear rational arithmetic theory solver [9, 40, 41, 49, 56]. This specific version of the dual simplex algorithm was first presented by Dutertre and de Moura [57] [12]. Whenever we refer to the simplex algorithm in the remainder of this thesis, we refer to this specific version.

It has been proven that almost every variation of the simplex algorithm has an exponential worst-case runtime complexity [128].[13] For the classical simplex algorithm as presented by Dantzig [47], this worst case occurs for the KleeMinty cube [98], i.e., a unit hypercube of variable dimension whose corners have been perturbed. Since there are polynomial decision procedures for linear rational arithmetic [93, 95], this also means that there are at least theoretically faster decision procedures. But in contrast to those polynomial decision procedures, the simplex algorithm has all properties necessary for an efficient theory solver: it produces minimal conflict explanations, handles backtracking efficiently, and is highly incremental. In practice, these properties are more important than the difference between polynomial and exponential worst-case complexity [61]. It also helps that the simplex algorithm rarely reaches its worst case in practice.

The input of the simplex algorithm (Figure 2.2) is in tableau representation (Section 2.4.2). Therefore, the input consists of two sets of variables $z_1, \ldots, z_n \in \mathcal{N}$ and $y_1, \ldots, y_m \in \mathcal{B}$, the two bound value functions $\mathcal{L}$ and $\mathcal{U}$, a tableau $Az = y$, and a set of bounds $\mathcal{L}(x_j) \leq x_j \leq \mathcal{U}(x_j)$ for the variables $x_j \in \mathcal{N} \cup \mathcal{B}$. Each row in this tableau represents one basic variable $y_i \in \mathcal{B}$: $y_i = a_i^T z$. This means that the non-basic variables $z = (z_1, \ldots, z_n)^T$ define the basic variables $y = (y_1, \ldots, y_m)^T$ over the tableau $Az = y$. The

[12] The first version of the dual simplex algorithm is much older [47].

[13] It is assumed that all versions of the simplex algorithm have an exponential worst-case runtime complexity because the simplex algorithm is NP-mighty [53].

Algorithm 1: `pivot`(y_i, z_j)

Input : A basic variable y_i and a non-basic variable z_j so that a_{ij} is non-zero
Effect : Transforms the tableau so y_i becomes non-basic and z_j basic

`/* Let` $y_i = a_i^T z$ `be the row defining the basic variable` y_i. `We rewrite this row as` $z_j = \frac{1}{a_{ij}} y_i - \sum_{z_k \in \mathcal{N} \setminus \{z_j\}} \frac{a_{ik}}{a_{ij}} z_k$ `so it defines` z_j `instead */`

$A' \in \mathbb{Q}^{m \times n}$;
$y'_i := z_j$;
for $y_k \in \mathcal{B} \setminus \{y_i\}$ **do** $y'_k := y_k$;
$z'_j := y_i$;
for $z_k \in \mathcal{N} \setminus \{z_j\}$ **do** $z'_k := z_k$;
$a'_{ij} := \frac{1}{a_{ij}}$;
for $z_k \in \mathcal{N} \setminus \{z_j\}$ **do** $a'_{ik} := -\frac{a_{ik}}{a_{ij}}$;

`/* Then we substitute` z_j `in all other rows with` $\frac{1}{a_{ij}} y_i - \sum_{k \in \mathcal{N} \setminus \{z_j\}} \frac{a_{ik}}{a_{ij}} z_k$ `*/`

for $y_l \in \mathcal{B}$ **do**
  for $z_k \in \mathcal{N} \setminus \{z_j\}$ **do** $a'_{lk} := a_{lk} + a_{lj} a'_{ik}$;
  $a'_{lj} := a_{lj} a'_{ij}$;
end
$\mathcal{N} := \{z'_1, \ldots, z'_n\}$; $\quad \mathcal{B} := \{y'_1, \ldots, y'_m\}$; $\quad (Az = y) := (A'z' = y')$;

Algorithm 2: `update`(z_j, v)

Input : A non-basic variable z_j and a value $v \in \mathbb{Q}_\delta$
Effect : Sets the value $\beta(z_j)$ of z_j to v and updates the values of all basic variables

for $y_i \in \mathcal{B}$ **do** $\beta(y_i) := \beta(y_i) + a_{ij}(v - \beta(z_j))$;
$\beta(z_j) := v$;

Algorithm 3: `pivotAndUpdate`(y_i, z_j, v)

Input : A basic variable y_i, a non-basic variable z_j, and a value $v \in \mathbb{Q}_\delta$
Effect : Pivots variables y_i and z_j and updates the value $\beta(y_i)$ of y_i to v

$\theta := \frac{v - \beta(y_i)}{a_{ij}}$;
$\beta(y_i) := v$; $\quad \beta(z_j) := \beta(z_j) + \theta$;
for $y_k \in \mathcal{B} \setminus \{y_i\}$ **do** $\beta(y_k) := \beta(y_k) + a_{kj}\theta$;
`pivot`(y_i, z_j);

Figure 2.1: The pivot and update functions [57].

simplex algorithm exchanges variables from $y_i \in \mathcal{B}$ and $z_j \in \mathcal{N}$ with the `pivot` algorithm (Figure 2.1). To do so, we also have to change the tableau via substitution. All tableaux constructed in this way are equivalent to the original tableau $Az = y$.

Algorithm 4: `Check`()

Output : Returns *true* iff there exists a rationally satisfiable assignment for the tableau and the bounds u and l; otherwise, it returns (*false*,y_i), where y_i is the conflicting basic variable

```
1  while true do
2      select a basic variable y_i such that β(y_i) < L(y_i) or β(y_i) > U(y_i)
3      if there is no such y_i then return true;
4      if β(y_i) < L(y_i) then
5          select a non-basic variable z_j such that
6          (a_ij > 0 and β(z_j) < U(z_j)) or (a_ij < 0 and β(z_j) > L(z_j))
7          if there is no such z_j then return (false,y_i) ;
8          pivotAndUpdate(y_i, z_j, L(y_i))
9      end
10     if β(y_i) > U(y_i) then
11         select a non-basic variable z_j such that
12         (a_ij < 0 and β(z_j) < U(z_j)) or (a_ij > 0 and β(z_j) > L(z_j))
13         if there is no such z_j then return (false,y_i) ;
14         pivotAndUpdate(y_i, z_j,U(y_i))
15     end
16 end
```

Figure 2.2: The main function of the simplex algorithm [57]

The goal of the simplex algorithm is to find an *assignment* β that maps every variable x_i to a value $\beta(x_i) \in \mathbb{Q}_\delta$ that satisfies our inequality system, i.e., $A(\beta(z)) = \beta(y)$ and $\mathcal{L}(x_i) \leq \beta(x_i) \leq \mathcal{U}(x_i)$ for every variable x_i. The algorithm starts with an assignment β that fulfills $A(\beta(z)) = \beta(y)$ and $\mathcal{L}(z_j) \leq \beta(z_j) \leq \mathcal{U}(z_j)$ for every non-basic variable $z_j \in \mathcal{N}$. Initially, we get such an assignment through our tableau. We simply choose a value $\mathcal{L}(z_j) \leq \beta(z_j) \leq \mathcal{U}(z_j)$ for every non-basic variable $z_j \in \mathcal{N}$ and define the value of every basic variable $y_i \in \mathcal{B}$ over the tableau: $\beta(y_i) := \sum_{z_j \in \mathcal{N}} a_{ij}\beta(z_j)$. Note that this only works if we assume that $\mathcal{L}(x_j) \leq \mathcal{U}(x_j)$ for every variable $x_j \in \mathcal{N} \cup \mathcal{B}$, which is easy to guarantee through a small preprocessing step since $\mathcal{L}(x_j) > \mathcal{U}(x_j)$ already implies that the problem is unsatisfiable. As an invariant, the simplex algorithm continues to fulfill $A(\beta(z)) = \beta(y)$ and $\mathcal{L}(z_j) \leq \beta(z_j) \leq \mathcal{U}(z_j)$ for every non-basic variable $z_j \in \mathcal{N}$ and every intermediate assignment β.

The simplex algorithm finds a rationally satisfiable assignment or an explanation of rational unsatisfiability through the `Check`() algorithm (Figure 2.2). Since all non-basic variables fulfill their bounds and the tableau guarantees that $Az = y$, `Check`() only looks for a basic variable that *violates* one of its bounds. If all basic variables y_i satisfy their bounds, then β is a rationally satisfiable assignment and `Check`() returns true. If `Check`() finds a basic variable y_i that violates one of its bounds, then it looks for a non-basic

Algorithm 5: `CheckConflict`(y_i)

Input : A conflicting basic variable y_i, i.e., a variable y_i that was returned by `Check`() in a pair (false, y_i).

Output : Returns a minimal set of bounds that cause a conflict with the tableau $Az = y$

```
1  if β(y_i) < L(y_i) then
2  |  C := {y_i ≥ L(y_i)}
3  |  for z_j ∈ N do
4  |  |  if a_ij > 0 then C := C ∪ {z_j ≤ U(z_j)};
5  |  |  if a_ij < 0 then C := C ∪ {z_j ≥ L(z_j)};
6  |  end
7  end
8  if β(y_i) > U(y_i) then
9  |  C := {y_i ≤ U(y_i)}
10 |  for z_j ∈ N do
11 |  |  if a_ij < 0 then C := C ∪ {z_j ≤ U(z_j)};
12 |  |  if a_ij > 0 then C := C ∪ {z_j ≥ L(z_j)};
13 |  end
14 end
15 return C
```

Figure 2.3: A conflict extraction function for the simplex algorithm [57]

variable z_j fulfilling the conditions in lines 6 or 12 of `Check`(). If it finds a non-basic variable z_j fulfilling the conditions, then we pivot y_i with z_j and update our β assignment so $\beta(y_i)$ is set to the previously violated bound value, which satisfies our invariant once more. If it finds no non-basic variable fulfilling the conditions, then the row of y_i and all non-basic variables z_j with $a_{ij} \neq 0$ build an unresolvable *conflict*. Hence, `Check`() has found a row that explains the conflict and it can return unsatisfiable.

The simplex algorithm terminates due to a variable selection strategy called *Bland's rule* [25]. Bland's rule is based on a predetermined variable order and always selects the smallest variables fulfilling the conditions for pivoting according to a predetermined variable order.

This already concludes our general description of the simplex algorithm. However, we are still missing the properties that turn the simplex algorithm into a well-suited SMT theory solver, i.e., minimal conflict explanations, high incrementality, and efficient backtracking. We explain why the simplex algorithm fulfills these properties in the remainder of this subsection.

Conflict Extraction

In a typical SMT solver (for more details see Section 2.6), a SAT solver based on CDCL (conflict-driven clause-learning) selects and asserts a set of theory literals that satisfy the boolean model. Then the theory solver verifies that the asserted literals are consistently theory satisfiable. If the theory solver finds a conflict between the asserted literals, then it returns a conflict explanation. The SAT solver uses the conflict explanation to start a conflict analysis that determines a good point for back jumping so it can select a new set of theory literals. Naturally, a good conflict explanation greatly enhances the conflict analysis and, therefore, the remaining search.

The literals asserted in our simplex based theory solver are bounds for our variables.[14] This means that a minimal conflict explanation is a subset C of the asserted variable bounds. In order to be conflicting and minimal, the subset C has to fulfill the following properties: (i) C is, together with the tableau $Az = y$, unsatisfiable over the rationals; and (ii) any strict subset $C' \subset C$ is, together with the tableau $Az = y$, satisfiable over the rationals. Both conditions are also independent of the pivots applied to $Az = y$ because pivoting is an equivalence preserving transformation.

We can derive such a minimal conflict explanation based on the output of the `Check`() call. If the call to `Check`() exits in line 7 with (false, y_i), then the conflict explanation is

$$\begin{aligned} C_l = \{y_i \geq \mathcal{L}(y_i)\} \cup &\{z_j \leq \mathcal{U}(z_j) : z_j \in \mathcal{N} \text{ and } a_{ij} > 0\} \\ \cup &\{z_j \geq \mathcal{L}(z_j) : z_j \in \mathcal{N} \text{ and } a_{ij} < 0\}. \end{aligned}$$

If the call to `Check`() exits instead in line 13 with (false, y_i), then the conflict explanation is

$$\begin{aligned} C_u = \{y_i \geq \mathcal{U}(y_i)\} \cup &\{z_j \geq \mathcal{L}(z_j) : z_j \in \mathcal{N} \text{ and } a_{ij} > 0\} \\ \cup &\{z_j \leq \mathcal{U}(z_j) : z_j \in \mathcal{N} \text{ and } a_{ij} < 0\}. \end{aligned}$$

We can compute these sets with the function `CheckConflict`(y_i) (see Figure 2.3).

The sets are conflicting because $C_l \cup \{a_i^T z - y_i \leq 0\}$ and $C_r \cup \{y_i - a_i^T z \leq 0\}$ are minimal conflicts with regard to Farkas' Lemma (Lemma 2.5.6). Note that $a_i^T z - y_i \leq 0$ and $y_i - a_i^T z \leq 0$ are the two inequalities that define the i-th row $a_i^T z = y_i$ in the tableau $Az = y$.

Incremental Extension and Backtracking

Most CDCL(T) solvers do not just check full boolean models for theory satisfiability but also some of the intermediate partial models. This is called (weakened) early pruning [129]. If the checked model is not theory satisfiable, then the SMT solver uses the conflict (i) to either prove that the whole

[14] Actually, the literals we assert are full inequalities $a_i^T x \leq b_i$. Due to slacking, the left side of those constraints is abstracted to a slack variable s such that $s = a_i^T x$. The definition of the slack variable $s = a_i^T x$ is directly stored in the simplex tableau and only a bound $s \leq b_i$ remains as the literal for the SMT solver (see Section 2.4.2 for more details).

Algorithm 6: `AssertLower`(x_i, l_i)

Input : A variable and a new lower bound value to assert.
Output : Returns *false* if the new lower bound for x_i violates x_i's current upper bound. Returns *true* if the current assignment β satisfies the new bound. Otherwise, returns *unknown*.
Effect : Adds the current lower bound to the old bound sequence and sets $\mathcal{L}$ to l_i if $\mathcal{L}(x_i) < l_i \leq \mathcal{U}(x_i)$. If this violates the simplex invariants, then updates β accordingly.

```
1 if 𝒰(x_i) < l_i then return false;
2 if ℒ(x_i) < l_i then
3 |   𝒪 := ⟦𝒪, (x_i ≥ ℒ(x_i), d)⟧
4 |   ℒ(x_i) := l_i
5 end
6 if l_i ≤ β(x_i) then return true;
7 if x_i ∈ 𝒩 then update(x_i, l_i);
8 return unknown
```

Algorithm 7: `AssertUpper`(x_i, u_i)

Input : A variable and a new upper bound value to assert.
Output : Returns *false* if the new upper bound for x_i violates x_i's current lower bound. Returns *true* if the current assignment β satisfies the new bound. Otherwise, returns *unknown*.
Effect : Adds the current upper bound to the old bound sequence and sets $\mathcal{U}$ to u_i if $\mathcal{L}(x_i) \leq u_i < \mathcal{U}(x_i)$. If this violates the simplex invariants, then updates β accordingly.

```
1 if ℒ(x_i) > u_i then return false;
2 if 𝒰(x_i) > u_i then
3 |   𝒪 := ⟦𝒪, (x_i ≤ 𝒰(x_i), d)⟧
4 |   𝒰(x_i) := u_i
5 end
6 if u_i ≥ β(x_i) then return true;
7 if x_i ∈ 𝒩 then update(x_i, u_i);
8 return unknown
```

Figure 2.4: Incremental assert functions for the simplex algorithm [57]

problem was unsatisfiable or (ii) to find a prefix of the model that is still able to satisfy the conflict. This means that subsequent theory problems are connected as follows: either the problems are *incrementally connected*, i.e., $(Ax \leq b) \cup (Dx \leq c)$ follows $(Ax \leq b)$, or the problems are *decrementally connected*, i.e., $(Ax \leq b)$ follows $(Ax \leq b) \cup (Dx \leq c)$.

Based on this, we call the runtime advantage an algorithm gains from having already solved a subset of the problem its *incremental efficiency*. Since most problems sent to an SMT theory solver are incrementally connected, its incremental efficiency is a major factor in determining its total efficiency. Similarly, we want to reduce the overhead that our theory solver

encounters when we have to *backtrack*, i.e., remove some of the literals in reverse chronological order. The less overhead this causes the more efficient our theory solver can backtrack. The simplex algorithm is in fact able to both add literals incrementally efficient and to backtrack efficiently.

To this end, we first have to extend the simplex state to include more than just two sets of variables $\mathcal{N}$ and $\mathcal{B}$, the two bound value functions $\mathcal{L}$ and $\mathcal{U}$, the tableau $Az = y$, and the current assignment β. We additionally include a sequence of old bounds $\mathcal{O}$, the current backtrack level d, and a backup assignment ω. Initially, $\mathcal{O}$ is empty, the current backtrack level d is zero, the bound value functions $\mathcal{L}$ and $\mathcal{U}$ map all variables to $-\infty$ and ∞, respectively, the backup assignment ω and the current assignment β assign all variables to zero, and the initial tableau $Az = y$ has a slacked row $a_i^T z = y_i$ for all inequalities $a_i^T z \leq b_i$ that appear in the SMT input problem. The latter is necessary so we never have to extend the tableau.

We extend our bound value functions—and, thereby, the set of literals we consider—with the functions `AssertLower`() and `AssertUpper`() (Figure 2.4). Both functions have three possible return values: *false*, *true*, and *unknown*. They return *false* if the bound we want to add contradicts another bound of the same variable, e.g., we already asserted $x_i \leq 5$ and are now asserting $x_i \geq 6$. This is necessary to guarantee our initial assumption that $\mathcal{L}(x_i) \leq \mathcal{U}(x_i)$ for all variables $x_i \in \mathcal{N} \cup \mathcal{B}$. They return *true* if the new bound value is already satisfied by our current assignment β. And they return *unknown* otherwise. The newly asserted bound replaces the current bound value if it is stricter, e.g., if we assert $x_i \leq u_i$, then $x_i \leq u_i$ is stricter than $x_i \leq \mathcal{U}(x_i)$ whenever $u_i < \mathcal{U}(x_i)$. If we replace a bound value, then we also append the replaced bound to the end of the sequence $\mathcal{O}$ together with the current backtrack level. This is necessary for our efficient backtracking algorithm. Both functions also have to maintain the invariant $\mathcal{L}(z_j) \leq \beta(z_j) \leq \mathcal{U}(x_i)$ for every variable $z_j \in \mathcal{N}$. This means that we call `update`(z_j, v) whenever the new bound value v violates a non-basic variable $z_j \in \mathcal{N}$.

Let us now look at a series of assertions that extend our current sequence of literals.[15] First of all, we can assume without loss of generality that we start such a series of assertions from a simplex state with a rationally satisfiable assignment for the current bounds. We can stop our assertions as soon as any of the assertions in the series return *false*. In this case, we have found a conflict (i.e., the set consisting of the two contradicting bounds) and our model can no longer be satisfiable over the rationals. If all of our assertions return instead *true*, then our old assignment also satisfies all new bounds. Therefore, the extended model is still satisfiable over the rationals.

[15] For more details where these assertions come from see Section 2.6.

Algorithm 8: `AddBTPoint()`

Output : The new backtrack level.
Effect : If the current assignment is unsatisfiable, does nothing. Otherwise, increments the backtrack level and creates a backup of the current assignment.

1 **if** β *is not a satisfiable assignment* **then return** d;
2 $d := d + 1$
3 $\omega := \beta$
4 **return** d

Algorithm 9: `Backtrack`(d')

Input : The decision level to backtrack to.
Effect : Reverts the bound value functions so they map to the bounds upto decision level d'. Recovers a satisfiable assignment for the decision level d'.

1 **if** $d \leq d'$ **then return**;
2 **while** $\mathcal{O} = [\![\mathcal{O}', (\gamma, d^*)]\!]$ *with* $d^* > d'$ **do**
3 $\quad \mathcal{O} := \mathcal{O}'$
4 $\quad$ **if** $\gamma = (x_i \geq l_i)$ **then** $\mathcal{L}(x_i) := l_i$;
5 $\quad$ **if** $\gamma = (x_i \leq u_i)$ **then** $\mathcal{U}(x_i) := u_i$;
6 **end**
7 $\beta := \omega$
8 $d := d'$

Figure 2.5: A simplex backtrack function [57]

If the assertions return at least once *unknown* but never *false*, then we do not know whether the extended model is satisfiable or unsatisfiable over the rationals. However, all invariants for the simplex algorithm hold. Therefore, we can use `Check`() to determine the rational satisfiability of our model.

These assert functions are so efficient because they maintain the simplex invariants at the cost of a negligible overhead and because they never have to change the tableau. Moreover, any previous pivots to the tableau do not hurt the efficiency of the `Check`() function. In fact, the previous pivots prevent us from visiting assignments that we already eliminated as unsatisfiable. This means that `Check`() is more efficient if it previously solved a subset of the problem. Therefore, it is incrementally efficient.

Backtracking, i.e., removing bounds in reverse chronological order, can also be done efficiently with the simplex algorithm. However, we first have to define the points in our chronological order that we want to backtrack to. To this end, let us look again at the goal of a typical CDCL(T) solver.

A typical CDCL(T) solver wants to find a complete boolean model (i.e., sequence of literals) that is also theory satisfiable. Therefore, our goal is to find a sequence of bounds that is satisfiable over the rationals. Since an unsatisfiable sequence of bounds only becomes satisfiable by removing

bounds, this also means that we always want to backtrack to satisfiable prefixes of our bound sequence. An SMT solver can mark a subset of these satisfiable prefixes as its backtrack points through the function `AddBTPoint`() (Figure 2.5).

Note that we always backtrack in a linear order. Therefore, the backtrack points created by `AddBTPoint`() are actually backtrack levels d and we simply increment our current backtrack level whenever we need a new backtrack point. Apart from that, `AddBTPoint`() also makes a backup ω of the latest rationally satisfiable assignment. We need only one backup assignment for all backtrack points because ω satisfies per definition a set of bounds C if it satisfies all subsets of C. This means that ω is a rationally satisfiable assignment for all previous backtrack levels.

Thanks to this backup assignment ω and our old bounds sequence $\mathcal{O}$, we are now able to efficiently backtrack to any previous backtrack level d' with the function `Backtrack`(d') (Figure 2.5). This function is efficient because it only reverts the bounds and exchanges the assignment. So no complex calculations are necessary.

2.7.2 Bound Refinements/Propagations

Bound refinement [57, 128], also called *bound propagation*, is a technique for linear arithmetic but in itself not a decision procedure. We use bound refinement in three cases: (i) we use bound refinement as a form of theory propagation in the CDCL(T) framework (for more details see Section 2.6); (ii) we use bound refinement as an extension to the branch-and-bound approach that refines and strengthens rational relaxations (for more details see Subsection 2.7.3); and (iii) we use bound refinement as an arithmetic alternative to unit propagation in the CDCL-like CutSat++ calculus (for more details see Chapter 3).

Bound refinement takes as input an inequality $a_{ij}x_j + p_{ij} \leq 0$ and several variable bounds from our current set of constraints C. Then it tries to propagate an entailed variable bound for (at least) one of the variables x_j with non-zero coefficient a_{ij}. In the case that bound refinement succeeds, it produces a variable bound $x_j \leq u_j$ ($x_j \geq l_j$) that is not subsumed by an existing bound in C.[16] The produced bound $x_j \leq u_j$ ($x_j \geq l_j$) can then be explicitly added to C. The entailed bounds are computed as follows:

Lemma 2.7.1 (Lower Bound Refinement)**.** *Let $a_i^T x \leq b_i$ be an inequality in C. Let x_j be a variable with $a_{ij} < 0$. Let $x_k \geq l_k$ be lower bounds in C for all variables $x_k \neq x_j$ with $a_{ik} > 0$. Let $x_k \leq u_k$ be upper bounds in C for all variables $x_k \neq x_j$ with $a_{ik} < 0$. Then $C \vdash (x_j \geq l_j)$, where $l_j := \left\lceil \sum_{a_{ik}>0, k\neq j} \frac{a_{ik}}{|a_{ij}|} \cdot l_k \right\rceil + \left\lceil \sum_{a_{ik}<0, k\neq j} \frac{a_{ik}}{|a_{ij}|} \cdot u_k \right\rceil - \frac{b_i}{|a_{ij}|}$.*

[16] A bound $x_j \leq u_j$ ($x_j \geq l_j$) is subsumed if $(x_j \leq u'_j) \in C$ with $u'_j \leq u_j$ ($(x_j \geq l'_j) \in C$ with $l'_j \geq l_j$)

Proof. Follows from Lemma 2.5.5. A lower bound $x_j \geq l_j$ is written in standard notation as $-x_j \leq -l_j$. Therefore, C contains the inequalities

$$\{a_i^T x \leq b_i\} \cup \{-x_k \leq -l_k : x_k \neq x_j \text{ and } a_{ik} > 0\} \cup \{x_k \leq u_k : x_k \neq x_j \text{ and } a_{ik} < 0\} .$$

This also means that C implies $-x_j \leq -l_j$ if there exists a linear combination of those inequalities that results in $-x_j \leq -l_j$. We receive this linear combination by multiplying $a_i^T x \leq b_i$ with $y' := \frac{1}{|a_{ij}|}$ and the bounds with $y_k := \frac{|a_{ik}|}{|a_{ij}|}$ and by adding them together. This results in

$$y' \cdot a_i^T x + \left[\sum_{a_{ik}>0, k\neq i} y_k \cdot (-x_k)\right] + \left[\sum_{a_{ik}<0, k\neq i} y_k \cdot x_k\right] =$$

$$\left[\sum_{k\neq i} \frac{a_{ik}}{|a_{ij}|} \cdot x_k\right] - \left[\sum_{a_{ik}>0, k\neq i} \frac{a_{ik}}{|a_{ij}|} \cdot x_k\right] - \left[\sum_{a_{ik}<0, k\neq i} \frac{a_{ik}}{|a_{ij}|} \cdot x_k\right] - x_j = -x_j$$

on the left side of the combined inequality and

$$\frac{b_i}{|a_{ij}|} - \left[\sum_{a_{ik}>0, k\neq i} \frac{a_{ik}}{|a_{ij}|} \cdot l_k\right] - \left[\sum_{a_{ik}<0, k\neq i} \frac{a_{ik}}{|a_{ij}|} \cdot u_k\right] = -l_j$$

on the right side of the combined inequality. Thus $C \vdash (x_j \geq l_j)$. □

Lemma 2.7.2 (Upper Bound Refinement)**.** *Let $a_i^T x \leq b_i$ be an inequality in C. Let x_j be a variable with $a_{ij} > 0$. Let $x_k \geq l_k$ be lower bounds in C for all variables $x_k \neq x_j$ with $a_{ik} > 0$. Let $x_k \leq u_k$ be upper bounds in C for all variables $x_k \neq x_j$ with $a_{ik} < 0$. Then $C \vdash (x_j \leq u_j)$, where $u_j := \frac{b_i}{|a_{ij}|} - \left[\sum_{a_{ik}>0, k\neq j} \frac{a_{ik}}{|a_{ij}|} \cdot l_k\right] - \left[\sum_{a_{ik}<0, k\neq j} \frac{a_{ik}}{|a_{ij}|} \cdot u_k\right]$.*

Proof. This proof is analogous to the proof of Lemma 2.7.1. □

For both refinement methods, we call the inequality $a_i^T x \leq b_i$ and the set of bounds C' used for the refinement the *explanation* for the propagated bound. We do so because $(a_i^T x \leq b_i) \cup C'$ imply the propagated bound and every subset of $(a_i^T x \leq b_i) \cup C'$ does not.

One major disadvantage of bound refinement is that it can cause divergence if we propagate new bounds repeatedly without any restrictions:

Example 2.7.3. Let $C = \{x_1 \geq 0, x_2 \geq 0, x_1 - x_2 \leq 0, -x_1 + x_2 \leq -1\}$. Then we can propagate arbitrarily many lower bounds for x_1 and x_2 that are not subsumed by the previously propagated bounds. We get these bounds by alternately propagating for $k = 0, 1, 2, \ldots$:

- $x_1 \geq k + 1$ from $-x_1 + x_2 \leq -1$ and $x_2 \geq k$; and
- $x_2 \geq k + 1$ from $x_1 - x_2 \leq 0$ and $x_1 \geq k$.

As a consequence of these infinite propagation sequences, we always have to restrict bound refinement in some way when we use it in practice. For instance, (i) we only propagate bounds as part of theory propagation that already appear in our input formula; (ii) we set a fixed threshold on bound refinements per branching node in the branch-and-bound approach; and (iii) we use for the CutSat++ calculus a propagation strategy (see Definition 3.5.6) that prevents propagation cycles.

2.7.3 Branch-And-Bound

Branch-and-bound [101, 104, 128] is the most common approach to handle linear mixed problems [89]. For instance, most theory solvers for linear integer and mixed arithmetic are based on branch-and-bound [9, 40, 41, 49, 56]. The general idea behind branch-and-bound is to split the original problem into two or more new problems (called branches) that are easier to solve. This is done in the following way: First, we compute a rational solution for our problem C. If there exists none, then we can stop because there also exists no mixed solution. If the rational solution is a mixed solution, then we can also stop because we have found our mixed solution. If the rational solution is not a mixed solution because the assignment s_j for the integer variable x_j is not an integer value, then we split our problem into two simpler problems C_l and C_u and solve them recursively. We do so by exploiting the following trivial fact: C has a mixed solution if and only if $C_l := C \cup \{x_j \geq \lceil s_j \rceil\}$ or $C_u := C \cup \{x_j \leq \lfloor s_j \rfloor\}$ has a mixed solution. C_l and C_u are simpler than C because they have together less rational solutions than C, i.e., $\mathcal{Q}_\delta(C) := \mathcal{Q}_\delta(C_l) \uplus \mathcal{Q}_\delta(C_r) \uplus \mathcal{Q}_\delta(C \cup \{\lfloor s_j \rfloor < x_j < \lceil s_j \rceil\})$.

Now that we have discussed the general idea behind branch-and-bound, let us define branch-and-bound more formally with all of the branch-and-bound related terminology that we use in this thesis. The main data structure of branch-and-bound is a *branching tree* and each of the tree's nodes represents one system of inequalities. A branching tree and its nodes are arranged as follows: (i) each tree has exactly one *root node*; (ii) the root node represents the *original system of inequalities* C_0; (iii) except for the root node, all nodes have exactly one other node declared as their *parent*; (iv) the *ancestors* of a node are the node's parent and the parent node's ancestors; (v) the root node is an ancestor for all other nodes inside the tree; (vi) a node C is either a *leaf*, i.e., it is not declared as the parent for any node inside the tree, or it has two *child nodes* C_l and C_u, i.e., the parent for C_l and C_u is C; and (vii) if C has child nodes, then C has a mixed solution if and only if one of its children has a mixed solution. Moreover, each node is marked with exactly one of the following labels at a time: (a) a node is marked as *pruned* if branch-and-bound determines that the node has no mixed solutions; (b) a node is marked as *branched* if it is the parent of another node; and (v) a leaf is marked as *active* if it is not pruned and $\mathcal{A}$ denotes the set of *active nodes*. Branch-and-bound works because it maintains the following invariant: the original problem C_0 has a mixed solution if and only if one of the active nodes $C' \in \mathcal{A}$ has a mixed solution. The invariant holds initially because the initial tree consists only of the input problem C_0 as the root node, which is at that point also the only active node. It is maintained during the algorithm thanks to the conditions (i)–(vii) that define branching trees.

As summarized before, the branch-and-bound algorithm works as follows: In every step, one active node $C \in \mathcal{A}$ is *selected* and removed from $\mathcal{A}$. Then we use a decision procedure for linear rational arithmetic, such as the simplex algorithm, to determine if C has a rational solution. If C has no rational solution, then it is also mixed unsatisfiable and the node is marked as *pruned* and we continue to select the next active node. Otherwise, branch-and-bound has found a rational solution $\beta(x) := s \in \mathcal{Q}_\delta(C)$ for C. If it holds that all integer variables $x_j \in \text{Zvars}(C)$ are assigned to integer values $\beta(x_j) \in \mathbb{Z}$, then $\beta(x) := s$ is also a mixed solution for C and the original problem C_0. In this case, branch-and-bound stops its search and returns s. If there is an integer variable x_j not assigned to an integer value (i.e., $\beta(x_j) \notin \mathbb{Z}$), then branch-and-bound selects one such variable as the *branching variable* and makes a case distinction based on its *branching value* $\beta(x_j)$: either it holds that $x_j \leq \lfloor \beta(x_j) \rfloor$ or $x_j \geq \lceil \beta(x_j) \rceil$. (These bounds are also called *branching bounds*.)

Every such case distinction creates two new *branches* of the original problem. The two branches are $C_l := C \cup \{x_j \geq \lceil \beta(x_j) \rceil\}$ and $C_u := C \cup \{x_j \leq \lfloor \beta(x_j) \rfloor\}$ and C has a mixed solution if and only if C_l or C_u has a mixed solution. Additionally, the new branches C_l and C_u become new active nodes ($\mathcal{A} := \mathcal{A} \cup \{C_l, C_u\}$) with C as their parent. This inductively guarantees that the original problem C_0 has a mixed solution if and only if one of the active nodes $C' \in \mathcal{A}$ has a mixed solution. After adding the new branches to $\mathcal{A}$, branch-and-bound continues recursively and tests the next active node. If $\mathcal{A}$ is empty and we cannot select any more active nodes, then the original problem C_0 has no mixed solution.

2.7.4 Extensions to Branch-And-Bound

The branch-and-bound algorithm has many extensions and some of them are necessary for an implementation that is efficient in practice. Here we present the most popular extensions used by the SMT community [9, 40, 41, 49, 56].

Cutting Planes

A *cut* is any inequality that is mixed implied but not generally implied [128]. This means that adding the cut to our inequality system produces a problem with the same mixed solutions but less rational solutions.

There are two advantages of adding cuts to our nodes: (i) branch-and-bound is better informed because there are less rational solutions that might mislead it and (ii) some previously rationally satisfiable nodes become rationally unsatisfiable, which means that we can prune the node earlier. The disadvantage is that computing the rational solution becomes more expensive the more constraints we have.

There are many different ways to compute cuts and cuts are typically named after the way they are computed, e.g., gomory cuts [73]. There even exist procedures that compute finite sequences of cuts that reduce the rational solutions of a problem until the existence of a rational solution implies the existence of a mixed solution. Obviously, these procedures can be easily turned into a complete decision procedure for linear mixed arithmetic. Unfortunately, these procedures are also too slow to help in practice.

Constraint Tightening

Constraint tightening is a very basic inprocessing technique for linear mixed arithmetic [128]. Let $a_i^T x \leq b_i$ be an inequality so that $a_i \in \mathbb{Z}^n$, all $a_{ij} \neq 0$ belong to integer variables (i.e., $j > n_1$), and $g = \gcd\{a_{ij} : \text{for } j = 1, \ldots, n\}$.[17] Then the tightened version of $a_i^T x \leq b_i$ is defined as

$$\frac{1}{g} \cdot a_i^T x_i \leq \left\lfloor \frac{b_i}{g} \right\rfloor .$$

A tightened constraint contains all mixed solutions of the original constraint but might not contain all rational solutions. This means a constraint tightening that actually changes an inequality is a cut.

Bound Propagation

Bound propagation, which we explained in Section 2.7.2, is also useful and popular as an inprocessing technique for branch-and-bound [57, 128]. In many cases, bound propagation combined with constraint tightening is enough to produce another cut. These cuts are not very strong but they have the advantage that they always subsume another inequality/bound. Thus, bound propagation does not increase the number of constraints and, therefore, not the time for computing a rational solution. We recommend to always apply (a bounded number of) bound propagation(s) before calculating a rational solution for a new branch.

Rounding Heuristics

Rounding heuristics are an easy way to turn any rational solution $\beta(x) = s$ computed during branch-and-bound, into an assignment $s' \in \mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_2}$ that fulfills the types of all variables [66, 128]. We simply round all s_j with $j > n_1$ to an integer value s'_j and keep all other values as before. Beware that this new assignment might very well violate the node or input constraints. We, therefore, have to evaluate whether s' is a mixed solution

[17] The first condition is trivially fulfilled for every inequality if we employ the transformations described in Section 2.2.1

for our original problem C_0. If the heuristic solution s' satisfies the original problem, then we stop our branch-and-bound search and return the solution instead. Otherwise, branch-and-bound ignores s' and continues the branch-and-bound search with s.

(Rounding) heuristics are easy to determine and to verify. They typically are a shortcut to a mixed solution if branch-and-bound is already close to one. As with cuts, this might reduce the number of branches needed by our branch-and-bound search. The only contrast is that cuts remove mixed unsatisfiable branches and heuristic solutions mixed satisfiable branches.

In the literature, any heuristic is called a rounding heuristic that rounds the values assigned to integer variables to integer values. We do not care whether we round up $s'_j := \lceil s_j \rceil$, down $s'_j := \lfloor s_j \rfloor$, or simple $s'_j := \lceil s_j \rfloor$. In practice, most implementations use simple rounding.

2.8 (Un)Bounded and (Un)Guarded

We mentioned before that there exist algorithms that find a rational solution for a system of inequalities $Ax \leq b$ in polynomial time [93, 95]. In contrast, the theories of linear integer and linear mixed arithmetic are NP-complete [119]. So at least for now, all decision procedures for those two theories have an exponential runtime worst-case.

Some of those decision procedures (e.g., branch-and-bound) generally perform well in practice. Most of them share, however, one class of problems on which they perform extremely poorly: *unbounded problems* [128].

Definition 2.8.1 (Bounded Direction[18])**.** A *direction*/vector $h \in \mathbb{Q}^n \setminus \{0^n\}$ is *bounded* in the constraint system C if there exist $l, u \in \mathbb{Q}_\delta$ such that C (rationally) implies $h^T x \leq u$ and $-h^T x \leq -l$. Otherwise, it is called *unbounded*.

Definition 2.8.2 (Bounded System[19])**.** A constraint *system* C is *bounded* if all directions $h \in \mathbb{Q}^n \setminus \{0^n\}$ are bounded (see Figure 2.7 for an example). Otherwise, it is called *unbounded* (see Figures 2.6 and 2.8 for examples).

Unbounded problems are typically those problems, where decision procedures for linear integer/mixed arithmetic have problems with termination or at least with efficient termination. For instance, branch-and-bound without extensions diverges on most unbounded problems:

[18]Note that the definitions for (un)bounded directions/problems are related to the rational solutions (i.e. rational entailment) of the problem and not the mixed solutions (i.e. mixed entailment).

[19]In the literature, bounded systems are systems whose rational solutions fit into a finite hypercube/hyperball [128]. Our definition is equivalent but it is formulated based on (un)bounded directions. In our opinion, this alternative formulation is more useful for the structural analysis of (un)bounded systems.

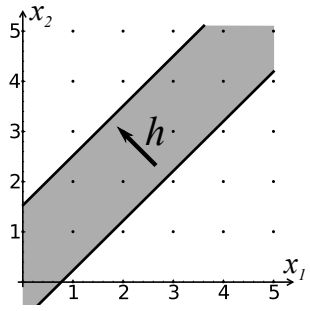


Figure 2.6: A partially bounded system; the directions $h = (-1, 1)^T$ and $-h$ are the only bounded directions in the example

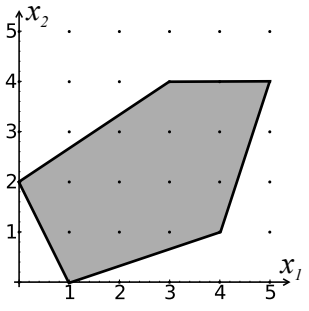


Figure 2.7: A bounded system; all directions are bounded

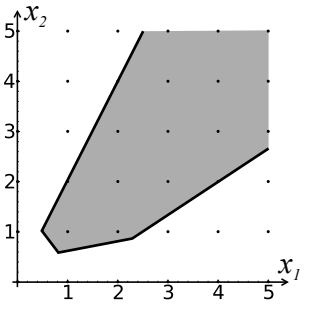


Figure 2.8: An absolutely unbounded system; all directions are unbounded

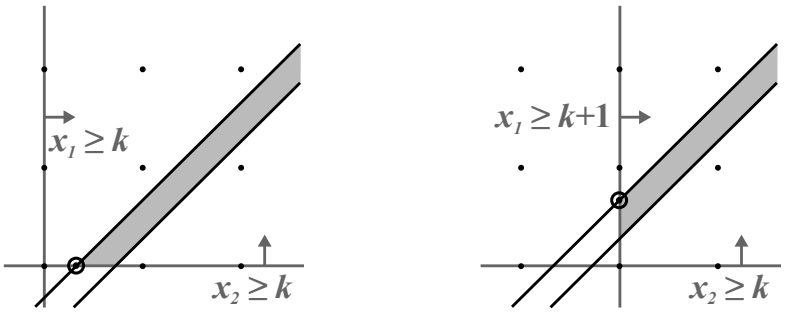


Figure 2.9: Two series of problems for which branch-and-bound diverges.

Example 2.8.3. Let us look at two series of problems (see Fig. 2.9):

$$C_k^1 := \{x_1 \geq k, x_2 \geq k, 3x_1 - 3x_2 \leq 2, -3x_1 + 3x_2 \leq -1\} \text{ and}$$
$$C_k^2 := \{x_1 \geq k + 1, x_2 \geq k, 3x_1 - 3x_2 \leq 2, -3x_1 + 3x_2 \leq -1\} .$$

In these two series, all problems have rational solutions but no integer solutions. If we perform branch-and-bound on C_k^1 to find an integer solution, then we first compute a rational solution for C_k^1, e.g., $(x_1, x_2)^T = (\frac{2k+1}{3}, k)^T$. Since x_1 is set to a fractional value, branch-and-bound has to branch on x_1. This creates the two new problems

$$C_l := C_k^1 \cup \{x_1 \leq k\} \text{ and } C_r := C_k^1 \cup \{x_1 \geq k + 1\}$$

on which we now recursively perform branch-and-bound. Since C_r is equivalent to C_k^2, one of the recursive calls is equivalent to performing branch-and-bound on C_k^2. Again, we first compute a rational solution for C_k^2, e.g., $(x_1, x_2)^T = (k + 1, \frac{2k+2}{3})^T$. Since x_2 is set to a fractional value, branch-and-bound has to branch on x_2. This creates the two new problems

$$C_l' := C_k^2 \cup \{x_2 \leq k\} \text{ and } C_r' := C_k^2 \cup \{x_2 \geq k + 1\}$$

on which we now recursively perform branch-and-bound. However, C_r'

is equivalent to C^1_{k+1}, which means one of the recursive calls is equivalent to performing branch-and-bound on C^1_k. This means we always cycle between the two cases described by the two series of problems, i.e., $C^1_k \to C^2_k \to C^1_{k+1} \to C^2_{k+1} \to \ldots$, and branch-and-bound diverges.

Similar diverging examples can be constructed for many other decision procedures.

2.8.1 A Priori Bounds

In contrast to unbounded problems, termination for bounded problems C is quite easy to achieve. Due to the implied bounds, any mixed solution $s \in \mathcal{M}_\delta(C)$ must lie between the implied variable bounds $l_j \leq x_j \leq u_j$, i.e., $s \in \mathcal{M}_\delta(C)$ implies that $l_j \leq s_j \leq u_j$ for $j = 1, \ldots, n$. This means that branch-and-bound branches at most $(u_{n_1+1} - l_{n_1+1} + 1) \cdot \ldots \cdot (u_n - l_n + 1)$ times before it terminates. Similar arguments can be made for most other decision procedures.

Termination on unbounded problems is much harder. Many decision procedures for linear arithmetic only terminate on unbounded problems because they assume an implicit preprocessing step that transforms every problem into an equisatisfiable bounded problem. And one such class of transformations is called *a priori bounds*. *A priori* bounds intersect the original problem with a cube that is so large that the problem only has a mixed solution outside of the cube if it also has one inside the cube. For example, the *a priori* bounds presented by Papadimitriou [119] guarantee that a problem has a mixed solution if and only if the problem extended by the bounds $|x_i| \leq 2n(ma)^{2m+1}$ for every variable x_i has a mixed solution. In these *a priori* bounds, n is the number of variables, m the number of inequalities, and a the largest absolute value of any integer coefficient or constant in the problem. By extending a problem with those *a priori* bounds, we reduce the search space for a branch-and-bound solver (and many other mixed arithmetic decision procedures) to a finite search space. So branch-and-bound is guaranteed to terminate.

However, *a priori* bounds typically describe a search space that is so large that it cannot be explored in reasonable time. For instance, the *a priori* bounds for Example 2.8.3 are in the order of 10 billions. So *a priori* bounds help us in theory but not in practice. As a practically efficient alternative, we present in Chapter 6 an equisatisfiable transformation that also makes branch-and-bound complete. But first we present here some additional facts on (un)bounded directions.

2.8.2 Bounded Basis

A direction h_i is *explicitly bounded* if there exist constants $l_i, u_i \in \mathbb{Q}_\delta$ so that $(l_i \leq h_i^T x \leq u_i) \in C$. They are easy to detect because it is enough to compare all constraints in C to find them. In contrast, unbounded directions and directions h that are *implicitly bounded*, i.e., bounded directions that are not explicit, are much harder to differentiate. Since there are infinitely many directions, it is not even possible to determine for each direction explicitly whether it is bounded or not. There are, however, finite representations for the bounded directions in a constraint system:

Lemma 2.8.4 (Dependent Bounded Directions)**.** *Let C be a constraint system and let $x = (x_1, \ldots, x_n)^T$ be all variables appearing in C. Let $H = (h_1, \ldots, h_k)^T$ be a matrix consisting of bounded directions $h_1, \ldots, h_k \in \mathbb{Q}^n \setminus \{0^n\}$ in C. Then any linear dependent direction h' of $h_1, \ldots, h_k$ is also bounded, i.e., any h' is also bounded for which there exists a $y \in \mathbb{Q}^k$ with $y^T H = h'^T$.*

Proof. Since the directions h_i are bounded, we know that there exist constants $l_i, u_i \in \mathbb{Q}_\delta$ such that $C \vdash l_i \leq h_i^T x \leq u_i$. If we now combine these bounds into vectors $l = (l_1^T, \ldots, l_k^T)$ and $u = (u_1^T, \ldots, u_k^T)$, then $C \vdash y^T l \leq h'^T x = y^T H x \leq y^T u$ because of Corollary 2.5.2. Thus, h' is also bounded in C. □

The above lemma gives us instructions on how to extract infinitely many bounded directions from a finite set of bounded directions. A finite representation of all bounded directions in C is then any maximum set of linear independent bounded directions in C:

Definition 2.8.5 (Bounded Basis)**.** Let C be a constraint system and let $x = (x_1, \ldots, x_n)^T$ be all variables appearing in C. Then $H = (h_1, \ldots, h_k)^T$ is a *bounded basis* for C if: (i) all $h_1, \ldots, h_k \in \mathbb{Q}^n \setminus \{0^n\}$ are linear independent bounded directions in C and (ii) there exists no linear independent direction $h' \in \mathbb{Q}^n \setminus \{0^n\}$ to H that is bounded in C.

Corollary 2.8.6 (Bounded Basis)**.** *Let C be a constraint system with n variables and $H \in \mathbb{Q}^{k \times n}$ a bounded basis of C. Then any bounded direction h' in C is a* linear combination *of H, i.e., h' is bounded iff there exists a $y \in \mathbb{Q}^k$ with $y^T H = h'^T$.*

The number of variables n in the constraint system is an upper limit for the number of linear independent bounded directions. If there are n linear independent bounded directions, then any direction can be constructed by their basis. Therefore, the whole system is bounded.

Corollary 2.8.7 (Bounded System)**.** *Let C be a constraint system with n variables. Then C is bounded if there are n linear independent bounded directions.*

The bounded systems that are easiest to recognize are so-called *guarded systems*, i.e., systems where every variable is explicitly bounded.[20] Due to their explicit boundedness, we sometimes also call them *explicitly bounded systems*. For the reverse reason, we also sometimes use *guarded variables* as an alternative name for explicitly bounded variables and *unguarded variables* for implicitly bounded and unbounded variables.

For *unguarded systems*, i.e., constraint systems that are not guarded, finding a bounded basis is more complicated. In the best case, we only need to check all explicitly bounded directions for linear independence. In the worst case, we even have to find implicitly bounded directions to complete our basis. Especially, the latter is a non-trivial task.

One of the contributions of this thesis is a method that computes a bounded basis for any system of linear inequalities (see Chapter 5.5). The method has polynomial runtime complexity and we explain how it can be implemented so it is incremental and efficient in practice. Moreover, we explain how to use the method to also derive actual bound values for our bounded basis.

With this method, we are also able to partition all systems of linear inequalities into four cases: guarded systems, unguarded but bounded systems, absolutely unbounded systems, and partially unbounded systems. We already presented the first two cases in the previous paragraphs and together they contain all bounded systems. Naturally, the other two cases contain all unbounded systems.

2.8.3 Types of Unbounded Systems

An *absolutely unbounded system* is a system where all directions are unbounded (see Figure 2.8 for an example). This also guarantees that it always has mixed/integer solutions:

Lemma 2.8.8 (Absolutely Unbounded [34])**.** *If all directions are unbounded in a system of inequalities $Ax \leq b$, then $Ax \leq b$ has an integer solution and, therefore, also a mixed solution.*

In Chapter 4, we present two cube tests that detect and solve constraint systems with infinite lattice width (another name for absolutely unbounded systems [90]) in polynomial time. The case of absolutely unbounded systems is, therefore, trivial and branch-and-bound can be easily extended so it also becomes complete for absolutely unbounded systems.

[20] Similarly, we call a *constraint guarded* if it contains only explicitly bounded variables. Otherwise, we call it an *unguarded constraint*.

The actual difficult case occurs when some directions are bounded and others unbounded. We call these systems *partially unbounded* (see Figure 2.6 for an example). They are problems for which branch-and-bound and most other algorithms diverge or become inefficient in practice. They are also the focus of the bounding transformations that we present in Chapter 6.

2.9 Other Geometric Objects

We mentioned before that the rational solutions to a system of inequalities define a geometric object that is called a *polyhedron*. Decision procedures for linear mixed arithmetic typically use these rational solutions as a guideline to find mixed solutions. These decision procedures also often have the same limitation: they consider only one single rational solution at a time.

An alternative is to consider sets with multiple rational solutions for the exploration of a polyhedron. Such sets of solutions should fulfill two properties to be useful in practice: (i) we must be able to find these sets in polynomial time (or at least an approximation); and (ii) we must be able to determine if the set contains a mixed solution in polynomial time. We actually discovered classes of sets with these properties. Here, we define these sets and later, in Chapter 4, we explain how to find and use them in inequality systems.

2.9.1 d-Norm Balls

The sets we consider in Chapter 4 belong to a class of geometric objects called *d-norm balls* (see Figure 2.10 for examples) [48, 84]. We define a d-norm ball $\mathcal{D}_r^d(z)$ with *radius* $r \in \mathbb{Q}_\delta$ ($r \geq 0$) around the *center point* z as the set of points $y \in \mathcal{D}_r^d(z)$ with at most d-norm distance r to z (i.e., $\text{dist}_d(z, y) \leq r$), or formally:

$$\mathcal{D}_r^d(z) = \{y \in \mathbb{Q}_\delta^n : \text{dist}_d(z, y) \leq r\}.$$

We also say that the points contained in $\mathcal{D}_r^d(z)$ are its rational solutions and its intersection with $\mathbb{Z}^n$ and $\mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_2}$ its integer and mixed solutions, respectively.

All d-norm balls also fulfill four additional properties that make them practical: they all describe (i) convex sets, (ii) with a center point z, (iii) they are point reflection symmetrical to z, and (iv) they are reflection symmetrical to the axis-parallel hyperplanes going through z.

The definition over the distance also guarantees that a d-norm ball $\mathcal{D}_r^d(z)$ has a mixed solution if and only if it contains the closest n_2-mixed point to its center:

Lemma 2.9.1 (d-Norm Ball Mixed Solution)**.** *A d-norm ball $\mathcal{D}_r^d(z)$ has a mixed solution if and only if it contains the closest n_2-mixed point $\lceil z \rfloor_{n_2}$ to the center z.*

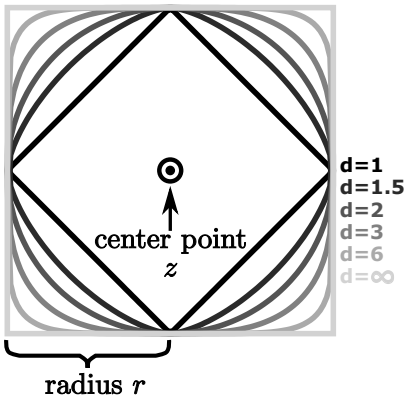


Figure 2.10: A selection of two-dimensional d-norm balls; all with the same radius and center point

Proof. The implication from left to right follows directly from Lemma 2.1.2. The implication from right to left is obvious. □

Corollary 2.9.2 (d-Norm Ball Integer Solution)**.** *A d-norm ball $\mathcal{D}_r^d(z)$ has an integer solution if and only if it contains the closest integer $\lceil z \rfloor$ to the center z.*

So $\mathcal{D}_r^d(z)$ has a mixed solution if and only if $\text{dist}_d(z, \lceil z \rfloor_{n_2}) \leq r$. The problem is that $\text{dist}_d(.,.)$ returns the dth root of a rational number, so not necessarily a rational number (with the exception of $d = 1$ and $d = \infty$). However, we do not actually have to compute $\text{dist}_d(z, y)$ to evaluate $\text{dist}_d(z, y) \leq r$. Instead we can evaluate the equivalent statement $\left(\sum_{j=1}^{n} |z_j - y_j|^d\right) \leq r^d$. Despite the exponents, we can compute and save these numbers in polynomial time and space if we encode them in binary representation. Thus, we are able to determine in polynomial time whether a given d-norm ball has a mixed solution.

2.9.2 (Axis-Parallel) Hypercubes

Most parts of Chapter 4 do not focus on general d-norm balls but only on ∞-*norm balls*. An ∞-norm ball is a hypercube that is parallel to the coordinate axes [48, 84]. For simplicity, we call these restricted hypercubes *cubes*. Cubes have two major advantages over most other d-norm balls[21]: (i) the norm they are based on contains neither roots nor exponents and (ii) they are polyhedra, which means that we can model them through systems of inequalities. The equivalent system of inequalities for a cube with *edge*

[21] The only exception are 1-norm balls.

length $e \in \mathbb{Q}_\delta$ (with $e \geq 0$) and *center* $z \in \mathbb{Q}_\delta^n$ is

$$\{z_j - \tfrac{e}{2} \leq x_j \leq z_j + \tfrac{e}{2} : j = 1, \ldots, n\}.$$

However, our methods do not actually require the constraint definition of the cubes but just the linearity it implies.

When we are talking about cubes, we are also typically using the edge length e instead of the radius $r = \frac{e}{2}$ to measure the size of the cube. For this reason, we also use a different notation for the set of points contained in a cube, i.e.,

$$\mathcal{C}_e(z) = \mathcal{D}^\infty_{e/2}(z) = \left\{y \in \mathbb{Q}_\delta^n : \forall j \in 1, \ldots, n.\ |y_j - z_j| \leq \tfrac{e}{2}\right\}$$

is the set of points contained in the cube that has edge length $e \in \mathbb{Q}_\delta$ (with $e \geq 0$) and center $z \in \mathbb{Q}_\delta^n$.

2.9.3 Flat Cubes

A disadvantage of cubes is that they extend equally in all directions. This is helpful when we focus our search on all directions equally, e.g., when we are looking for an integer solution. If we are, however, focusing our search only on some directions, e.g., when we are looking for a mixed solution, then this fact becomes bothersome. For these problems, we would prefer objects that extend only in the directions that are relevant to our search.

Our solution are cubes that expand only in the directions that correspond to integer variables and are flat in the directions that correspond to rational variables. We call these cubes *flat cubes*. As before, we formally define flat cubes by the set of points they contain, i.e., the flat cube with edge length e and center point z is defined as

$$\mathcal{F}_e(z) = \left\{y \in \mathbb{Q}_\delta^{n_1} \times \{(z_{n_1+1}, \ldots, z_{n_1+n_2})^T\} : \mathrm{dist}_d(z, y) \leq \tfrac{e}{2}\right\}.$$

Also as before, the flat cube $\mathcal{F}_e(z)$ has a mixed solution if and only if it contains the closest n_2-mixed point to its center:

Lemma 2.9.3 (Flat Cube Mixed Solution)**.** *A flat cube $\mathcal{F}_e(z)$ has a mixed solution if and only if it contains the closest n_2-mixed point $\lceil z \rfloor_{n_2}$ to the center z.*

Proof. The implication from left to right follows directly from Lemma 2.1.2. The implication from right to left is obvious. □

We could also define flat d-norm balls. We refrain from this because we use d-norm balls only for comparison with cubes and the comparison between flat cubes and flat d-norm balls would not result in any additional information.

2.10 Basics of Transition Systems

(Unlabeled) *transition systems*—which are mathematically equivalent to abstract rewriting systems—are simply state based systems that change step by step via discrete transitions [99].[22] This means that transition systems can be used as an alternative to pseudocode for the formalization of algorithms. On the one hand, algorithms written as transition systems have the advantage of explicit definitions for their states and transitions. This makes them less error prone and it makes it easier to prove invariants over them. On the other hand, the description of an algorithm as pseudocode is much closer to an actual implementation of the respective algorithm. Naturally, this makes it easier to reimplement them and to estimate their behavior as part of an actual implementation. This is why we only present the most complex algorithms as transition systems, viz., CutSat, CutSat++, and CutSat$_g$.

The formal description of a transition system must include the following information: (i) the set of *states* S over which the transition system operates; (ii) a subset of states $S_0 \subseteq S$ called the *start states* of the transition system; (iii) a subset of states $S_e \subseteq S$ called the *end states* of the transition system; and (iv) a set of *transitions* called the *transition relation* $R \subseteq (S \setminus S_e) \times S$ of the transition system. This description also implicitly defines a directed graph, where the nodes are simply the states and the directed edges the transitions.

Typically, the set of states and transitions are infinite. This means we cannot explicitly list all of them but need to formally define them. In the case of transitions, we typically do so by defining *transition rules*. A definition of a transition rule looks abstractly as follows:

Name of the Rule

$s \Longrightarrow_{\text{TS}} s'$ **if** $F(s, s')$

In this definition, $F(s, s')$ are conditions that specify the valid input states s and output states s' of the transitions defined by the rule. We also always label our transitions $\Longrightarrow_{\text{TS}}$ with an abbreviation of the name of the transition system. In our example, this label is TS.

In the terms of an algorithm, start states define the valid input of our algorithm, end states the valid output of our algorithm, and the transitions, i.e., rule applications, are the operations our algorithm can take to modify the current state. The algorithms defined by transition systems are typically non-deterministic because states can have multiple outgoing transitions.

A *run* of our algorithm/transition system is a sequence of transitions, i.e., rule applications that is: (i) starting from one of the start states (the input of the run); and (ii) either *diverging*, i.e., has infinitely many rule applications, or *terminating*, i.e., ending in a state with no outgoing transitions (the

[22] The actual computational cost of a transition plays no role for a transition system.

output of the run). A transition system is *terminating* if there exists no run from any of the start states that is diverging. The correct output of a transition system is specified by a function $f : S_0 \times S_e \to \{true, false\}$. A transition system is *sound* according to a specifying function if every terminating run for every start state $s_0 \in S_0$ ends in an end state $s_e \in S_e$ such that $f(s_0, s_e)$ evaluates to true. The *stuck states* of a transition system are all terminating states that are not also end states. A transition system is *complete* if it is terminating, sound, and has no stuck states.

Sometimes a transition system has too much non-determinism to uphold the invariants we want, e.g., termination and soundness. For this reason, we are allowed to define *strategies* for our transition system, i.e., some additional restrictions to our rule applications that are typically in the form of rule preferences. We then say that a transition system fulfills an invariant while following a given strategy if the invariant holds on all those runs that are still possible under the given strategy.

Chapter 3

CutSat++: A Complete and Terminating Approach to Linear Integer Solving

The SMT and theorem proving communities have developed several interesting approaches for their specific type of linear integer problems [11, 52, 76]. These approaches are based on a branch-and-bound strategy (see also Chapter 2.7.3), where the rational relaxation of an integer problem is used to cut off and branch on rational solutions[1]. Together with the known *a priori integer bounds* for a problem, this branch-and-bound strategy yields a terminating and complete algorithm (see Chapter 2.8 and [119]). However, the *a priori* bounds grow exponentially in the number of inequalities. Even for toy examples the bounds easily exceed standard 64-Bit integer representations of today's computers. As a result, these bounds do not cause termination in a reasonable amount of time and checking the bounds is expensive as it requires big integer representations. Hence, the *a priori* bounds are often not integrated in implementations [9, 41, 49, 56]. Furthermore, the *a priori* bounds depend only on syntactic properties of a problem, such as the largest occurring constant and the number of inequalities. Therefore, it seems that a more promising termination approach needs to explore the inner structure of a problem.

The most well known calculus of this type is the CDCL (conflict-driven clause-learning) [88, 20, 110, 130] calculus for propositional satisfiability (SAT). It has changed SAT solving from a purely academic exercise to a standard verification technique in industry. The calculus starts by generating a partial model assumption by deciding (guessing) the truth value of propositional variables and by propagating new truth values with respect

[1] For linear arithmetic problems over the rationals, such as the rational relaxation of an integer problem, the SMT and theorem proving communities already presented efficient approaches [55, 57].

to the already set values and the problem clauses. If the model assumption evolves into an overall model, i.e., a model that satisfies all problem clauses, then satisfiability is shown. If it falsifies a clause, this clause together with the partial model assumption can be explored to derive a new learned clause via resolution. The learned clause is a logical consequence of the problem clauses, hence it explores the inner structure of the problem. The learned clause also repairs the failed partial model assumption with respect to the model generating algorithm. Furthermore, the learned clause is new by construction [137] and constitutes progress on the basis of a well-founded ordering. The latter two properties both independently guarantee termination of the CDCL calculus. For a toy example, consider the clauses $P \vee Q,\ \neg P \vee \neg Q,\ \neg P \vee Q$. A run of the CDCL calculus may decide P to be true and then propagate $\neg Q$ using the clause $\neg P \vee \neg Q$. The resulting model assumption $[\![P, \neg Q]\!]$ falsifies the clause $\neg P \vee Q$. Then a resolution step between this clause and the propagating clause $\neg P \vee \neg Q$ yields the clause $\neg P$. This clause repairs the model via backtracking into $[\![\neg P]\!]$ and after one propagation into $[\![\neg P, Q]\!]$, which is an overall model of the three clauses.

The idea of building explicit (partial) model assumptions guiding inferences is actually older than CDCL and goes back to the superposition calculus [7]. It has meanwhile been transported to a variety of logics [4, 38, 120] including arithmetic theories [87, 86]. From these model guided calculi, the CutSat calculus by Jovanović and de Moura [87] is the closest to our work and constitutes an important step towards a decision procedure that is both efficient and terminating on linear integer problems. Similar to CDCL and the other above mentioned calculi, termination relies on the generation of new inequalities that were derived based on a well-founded ordering. This means termination relies no longer on *a priori* bounds but on the exploration of the inner structure of a problem. If all variables of a problem are guarded—i.e., for every variable x_i there are inequalities $l_i \leq x_i \leq u_i$, where l_i, u_i are integer constants (see also Chapter 2.8)— then CutSat terminates. If there are unguarded variables, then CutSat may diverge or get stuck (see Section 3.3). This means, however, that the termination approach of the CutSat calculus is not working on all linear integer problems.

Our contribution in this chapter is an extension and refinement of the CutSat calculus, which we call CutSat++. In contrast to CutSat, CutSat++ always terminates. The basic idea of both calculi is to reduce a problem containing unguarded integer variables to a problem containing only guarded variables. Unguarded variables are not eliminated. Instead, they are explored by learning further inequalities on smaller guarded variables to the problem. A strict total ordering on all variables, where all unguarded

variables are larger than all guarded variables, provides the scale. After adding sufficiently many inequalities, feasibility of the problem depends only on guarded variables. Then a CDCL style calculus with explicit (partial) model building tests for feasibility by employing exhaustive propagation.

The most sophisticated part is to "turn" an unguarded variable into a guarded variable. The variable elimination procedure by Cooper [44] does so by replacing an unguarded variable with a case distinction that explores lower bounds and is represented by disjunctions. The lower bounds are derived from the problems inequalities. The procedure is provably terminating at the price of potentially exponentially growing coefficients and an exponentially growing Boolean problem structure. Still, the idea behind Cooper's procedure is our starting point and we will present a variation of the procedure, which is called *weak Cooper elimination*, in Section 3.4. Weak Cooper elimination does not create a complicated boolean structure because it considers not only the lower bounds but also the upper bounds derivable from the problems inequalities. By combining all pairs of inequalities that determine a lower and an upper bound for some variable, we get a new formula that determines whether the strictest lower bound is smaller than or equal to the strictest upper bounds for that variable without complicating the boolean structure. The only additional drawback is the introduction of new guarded variables and divisibility constraints. However, guarded variables seem to be less harmful, in practice, and divisibility constraints are needed for integer quantifier elimination anyway.

Satisfiability of linear integer problems is NP-complete. All algorithms known today need exponential time on certain classes of input problems. Since Cooper elimination and weak Cooper elimination do not care about the concrete structure of a given problem, the exponential behavior is almost guaranteed. The idea of CutSat++ is, therefore, to simulate a lazy variation of weak Cooper elimination. It is lazy because it does not apply a complete variable elimination step at once, but only partially whenever false inequalities (leading to so-called *conflicting cores*) occur over unguarded variables. Since CutSat++ uses a strategy that applies the rules according to a total variable order, which is closely related to the elimination order of a quantifier elimination procedure, it is not even necessary to actually remove the unguarded variables. Instead, the conflict is blocked by learning constraints in smaller variables according to the variable order. This leaves space for repairing model assumptions and applying simplification rules in order for the calculus to adapt to the specific structure of a problem and, hence, to systematically avoid certain cases of the worst case exponential behavior observed with Cooper elimination.

In more detail, CutSat++ starts with a partial model assumption for the linear integer problem. The model assumption is build by Decide (guess a value/bound) and Propagate (propagate values/bounds through inequalities to obtain new values/bounds) rules similar to CDCL, see Figure 3.1.

However, we are dealing with inequalities instead of propositional clauses. Variables no longer have just two possible assignments but exponentially many (even infinitely many if we ignore the existence of *a priori* bounds for the variables [119]). Therefore, the propagate rule assigns bounds $x_i \leq b_i$ instead of values $x_i = b_i$ for some integer constant b_i. This also means that a variable is only fixed, i.e., assigned to a specific value, if the partial model contains two bounds $x_i \leq b_i$ and $x_i \geq b_i$.

Now bound propagation takes an inequality and uses the bounds in the model assumption to propagate additional bounds. For example, the inequality $2x_1 - x_2 \leq 0$ together with the bound $x_2 \leq 1$ propagates $2x_1 \leq 1$ or $x_1 \leq \frac{1}{2}$. Since we are looking for an integer value for x_1 the actual bound becomes $x_1 \leq 0$, which is also the bound CutSat++ would propagate. Another difference to SAT is that the number of propagation steps is no longer bound by the number of variables, but rather by the exponential *a priori* bounds for the variables. For example, let our model assumption contain the bounds $x_1 \geq 5$ and $x_2 \geq 5$ and let us look at the two inequalities $1-x_1+x_2 \leq 0$ and $x_1-x_2 \leq 0$. We can use $1-x_1+x_2 \leq 0 \equiv 1+x_2 \leq x_1$ and $x_2 \geq 5$ to propagate the bound $x_1 \geq 6$. Then we use $x_1 - x_2 \leq 0 \equiv x_1 \leq x_2$ and $x_1 \geq 6$ to propagate the bound $x_2 \geq 6$. If we continue propagation using $1 - x_1 + x_2 \leq 0$ and $x_1 - x_2 \leq 0$ alternately, we generate ever-growing bounds for x_1 and x_2 up to the exponential *a priori* bounds. Without adding the *a priori* bounds explicitly, bound propagation would even diverge in this example. This behavior is only possible because the problem contains unguarded variables. Although automatic detection of diverging bound propagations is possible in many cases, in particular in situations where subsequent propagation rule applications yield the divergence, it can also be the result of a combination of decisions and propagation rule applications. In this situation an automatic detection of the divergence gets far more complicated. To prevent ever-growing bounds, we will present a propagation strategy for CutSat++, see Definition 3.5.6, based on the already mentioned variable ordering that prevents divergence through propagation, even without *a priori* bounds.

As a result, CutSat++ generates after finitely many propagation and decision rule applications either a model for the problem, or some inequality is falsified by the partial model assumption. The analog situation in CDCL based SAT solving is the detection of a false clause. The CDCL calculus computes from the false clause via resolution steps the eventual learned clause that repairs the partial model assumption. The crucial invariant is that at any step the actual candidate clause remains false in the model assumption. The invariant guarantees that the model assumption will change eventually after backtracking parts of the model assumption and adding the learned clause. We can do something very similar for inequalities. The only difference is the resolution step, which is based on Fourier-Motzkin elimination instead of Boolean resolution. Let $-a_j x_j + p_j \leq 0$ and $b_j x_j + q_j \leq 0$ be

two constraints over the integers in the focused representation (see Chapter 2.2.1), where $a_j, b_j > 0$ are constant integer coefficients and p_j, q_j are linear polynomials without x_j. Then they—and every problem containing them—also imply the constraint $b_j \cdot (-a_j x_j + p) + a_j \cdot (b_j x_j + q) \leq 0 \equiv b_j p + a_j q \leq 0$ that does not contain the variable x_j. For example, we can combine $1 - x_1 + x_2 \leq 0$ and $x_1 - x_2 \leq 0$ through linear combination so the new constraint $1 \cdot (1 - x_1 + x_2) + 1 \cdot (x_1 - x_2) \leq 0 \equiv 1 \leq 0$ directly proves unsatisfiability of our problem. However, resolving conflicts is typically more challenging for linear integer constraints. For example, let our problem contain the inequalities $-x_2 + 1 \leq 0$, $-x_3 + 1 \leq 0$, $3x_1 - x_2 \leq 0$, and $-3x_1 + x_3 \leq 0$ and let our current model include the bounds $x_2 \geq 1$ (propagated from $-x_2 + 1 \leq 0$), $x_2 \leq 1$ (decided), $x_3 \geq 1$ (propagated from $-x_3 + 1 \leq 0$), and $x_3 \leq 1$ (decided) so x_2 and x_3 are both assigned to 1. We can use $x_2 \leq 1$ and $3x_1 - x_2 \leq 0$ to propagate the bound $x_1 \leq 0$ to our model. As a result, the inequality $-3x_1 + x_3 \leq 0$ turns into a conflict because $-3x_1 + x_3 \leq 0 \equiv x_3 \leq 3x_2$ implies, together with the bounds in our current model, that $1 \leq x_3 \leq 3x_1 \leq 0$. However, resolving $-3x_1 + x_3 \leq 0$ and $3x_1 - x_2 \leq 0$ with a linear combination results in the constraint $-x_2 + x_3 \leq 0$ which is satisfied by our model. The resolution ended prematurely because the linear combination of the constraints $-a_j x_j + p \leq 0$ and $b_j x_j + q \leq 0$ is only guaranteed to be conflict preserving if one of the coefficients a_j or b_j is 1. Hence, we cannot resolve conflicts directly with the constraints used for propagation.

In order to overcome this problem, we will compute so-called *tight justifications* $\pm x_j + r_j \leq 0$ for the propagated bounds, where r_j is a linear polynomial without x_j. Tight justifications are inequalities entailed by our set of constraints that can propagate the same bounds but have coefficients ± 1 for the propagated variables. For our example, the tight justifications are constructed as follows: The tight justifications for the bounds $x_2 \geq 1$ and $x_3 \geq 1$ are simply the propagating inequalities $-x_2 + 1 \leq 0$ and $-x_3 + 1 \leq 0$ because they already have -1 as the coefficient for the propagated variables. We generally get the tight justification for the bound $x_j \leq b_j$ propagated from $a_j x_j + p_j \leq 0$ by performing the following two steps: first we add to $a_j x_j + p_j \leq 0$ tight justifications from previously propagated bounds until the resulting inequality has the form $a_j x_j + a_j \cdot r_j + c_j \leq 0$, where r_j is a linear polynomial without x_j and c_j a constant integer; then we divide the combined inequality by the coefficient of the propagated variable x_j and we get our tight justification $x_j + r_j + \lceil \frac{c_j}{a_j} \rceil \leq 0$. This means we get the tight justification for $x_j \leq 0$ by adding two times $-x_2 + 1 \leq 0$ to $3x_1 - x_2 \leq 0$, which results in the inequality $3x_1 - 3x_2 + 2 \leq 0$. Then we divide $3x_1 - 3x_2 + 2 \leq 0$ by 3, which results in the inequality $x_1 - x_2 + 1 \leq 0$. This inequality is now a tight justification for $x_1 \leq 0$ because $x_1 - x_2 + 1 \leq 0$ can propagate $x_1 \leq 0$ from $x_2 \leq 1$ and because its coefficient for x_1 is ± 1. If we now

resolve the conflicting inequality $-3x_1 + x_3 \leq 0$ with the tight justification $x_1 - x_2 + 1 \leq 0$ instead of $3x_1 - x_2 \leq 0$, then we get the inequality $-3x_2 + x_3 + 3 \leq 0$. This inequality is still violated by our current model because $6 \leq x_3 + 3 \leq 3x_2 \leq 3$.

Conflict resolution may also cause divergence in combination with unguarded variables. The reason is that any propagated bound can also be added to the model with a combination of decisions and conflict resolution. Therefore, we can repeat the divergent example from before even without the propagation rule. As a solution to this "conflict divergence", CutSat++ introduces a second type of conflict resolution based on Cooper's quantifier elimination procedure. This new resolution is called *unguarded conflict resolution*. It uses the fact that we can transform any unguarded problem into a guarded one if we eliminate all unguarded variables with a quantifier elimination procedure. For efficiency reasons, however, the quantifier elimination rules are restricted to so-called *conflicting cores* which are condensed false inequalities generated out of the inequality falsified by the model assumption. CutSat++ enforces that all unguarded conflicts are handled by the new resolution, while all guarded conflicts are still handled solely by the conflict resolution motivated from CDCL and described above (see Section 3.2).

For the case of a conflicting core involving unguarded variables, we actually need to distinguish three subcases: (i) the conflicting core is the result of an inequality propagating a bound for an unguarded variable that then falsifies a second inequality, called an *interval conflicting core*, (ii) two inequalities have been propagating bounds for an unguarded variable falsifying a divisibility constraint, called a *divisibility conflicting core*, and (iii) a divisibility constraint is falsified although it still contains a variable that is unbounded in the model, called a *diophantine conflicting core*. For example, assume a divisibility constraint $6 \mid 4x_j + r_j$ where r_j is a linear polynomial not containing x_j and x_j is unbounded. Now assume that the partial model assumption restricts the values of the variables in r_j such that r_j evaluates to 1. Then, the divisibility constraint becomes unsatisfiable although x_j is still unbounded.

Altogether, (i) the CDCL style resolution of conflicts with purely guarded variables, (ii) the special resolution mechanisms for interval, divisibility and diophantine conflicting cores, and (iii) the exploration of an underlying well-founded ordering on the variables plus (iv) a strategy on the calculus rules yields the sound, complete and terminating calculus CutSat++. In Section 3.2, we start introducing CutSat++ formally as a transition system[2] by presenting the definitions and rules it shares with CutSat. These rules describe a CDCL style calculus and they constitute already a sound, complete, and terminating calculus for all problems that contain only guar-

[2]See Chapter 2.10 for the basic definitions necessary to understand transition systems.

ded variables. In Section 3.3, we explain the rules and strategies used by CutSat to handle unguarded variables and show that they do not suffice. Our conclusion is that CutSat lacks, in addition to some refinements, a resolution case for diophantine conflicting cores. The basis for the exploration of this conflicting core is our new variant of Cooper elimination, called *weak cooper elimination*, which we present in Section 3.4. In this section, we also prove that any procedure that is based on weak Cooper elimination needs to consider diophantine conflicting cores for completeness (Theorem 3.4.7). In Sections 3.5–3.6, we then use weak Cooper elimination to define the inference rules of CutSat++ for the elimination of unguarded variables. The result is the sound, complete, and terminating CutSat++ calculus (see Section 3.6.3, Theorem 3.6.6 and Theorem 3.6.11). Finally, we give conclusions and point at possible directions for future research.

3.1 Related Work and Preliminaries

This chapter is based on a publication with Thomas Sturm and Christoph Weidenbach as co-authors [32]. An extended version of this publication has been accepted for publication in the Journal of Symbolic Computation.

The constraints in this chapter are either formatted according to the standard representation, i.e., $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ for inequalities and $d_i \mid a_{i1}x_1 + \ldots + a_{in}x_n + b_i$ for divisibility constraints (see also Chapter 2.2.1), or formatted according to the focused representation, i.e., $a_{ij}x_j + p_{ij} \leq 0$ for inequalities and $d_i \mid a_{ij}x_j + p_{ij}$ for divisibility constraints (see also Chapter 2.2.1). Since this chapter handles a decision procedure for linear integer arithmetic, we also assume in this chapter that all variables are integer variables. For convenience, we abbreviate in this chapter integer satisfiability/equivalence/entailment with satisfiability/equivalence/entailment. Moreover, we assume that all coefficients a_{ij} and constraint bounds b_i are integer values. We are allowed to do so because of the equisatisfiability preserving transformations presented in Chapter 2.2.1.

This chapter builds on the basics of linear arithmetic (Chapter 2.2), on the concept of implied constraints (Chapter 2.5), on the definitions of (un)bounded and (un)guarded problems and variables (Chapter 2.8), and on the basics of transition systems (Chapter 2.10).

As already mentioned in the introduction to this chapter, our CutSat++ calculus is an extension and refinement of the CutSat calculus by Jovanović and de Moura [87]. The paper presenting CutSat is, therefore, the closest related work to this chapter. The similarities and differences between CutSat and CutSat++ can be summarized as follows: Both calculi can be split into two parts: (i) a CDCL style calculus, which we call CutSat$_g$; and (ii) an extension of the CDCL style calculus based on quantifier elimination techniques. The first part, CutSat$_g$, was initially presented

in [87] and is identical in both CutSat and CutSat++.[3] CutSat$_g$ is already on its own a terminating and complete calculus for all guarded problems; i.e., problems with only guarded variables. The second part (i.e., the quantifier elimination based extension) is necessary for handling problems containing unguarded variables. For this part, CutSat uses an extension called *strong conflict resolution* [87], which is not strict and exhaustive enough to guarantee termination on all unguarded problems (see Section 3.3). To resolve this, we developed the alternative extension *unguarded conflict resolution* for CutSat++ (Section 3.5) that does guarantee termination on all problems (including unguarded problems).

CutSat++ is also highly related to the IntSat calculus by Robert Nieuwenhuis [112]. IntSat is, similarly to CutSat$_g$, a CDCL style calculus for linear integer arithmetic that only terminates on guarded problems. However, IntSat is in practice faster than CutSat$_g$. The reasons are as follows: IntSat uses a weaker but faster version of conflict resolution (with respect to learning) that is still strong enough to be correct, complete, and efficient. Moreover, IntSat can add decided bounds that do not match any propagated bounds. Since IntSat and CutSat$_g$ are so similar, we should also be able to combine unguarded conflict resolution and IntSat to receive a complete and terminating calculus. We assume that the resulting combination would be more efficient in practice than CutSat++.

3.2 The Guarded Case

Our CutSat++ calculus can be separated into two parts: (i) CutSat$_g$, a CDCL style calculus, which is already on its own a terminating and complete calculus for all guarded problems; and (ii) unguarded conflict resolution, an extension to CutSat$_g$ based on quantifier elimination techniques, which makes the calculus complete even if the problem contains unguarded variables. In this section, we will be presenting CutSat$_g$ in formal detail[4].

This section is organized as follows: we define in Subsection 3.2.1 the states and models traversed by CutSat$_g$ and its extensions CutSat and CutSat++. In Subsection 3.2.2, we will explain the model construction via decisions and propagations. Besides the actual calculation of the propagated bound value via the function bound$(J, x_j, \bowtie, M)$, where $\bowtie \in \{\leq, \geq\}$, this also includes restrictions to our rules (e.g. improves$(J, x_j, \bowtie, M)$) that guarantee that our propagations always improve the model, i.e., our propagations always eliminate invalid solutions and never skip any valid solutions. We can, however, still skip valid solutions with decisions. That is why we

[3]Note that the name CutSat$_g$ does not appear in [87]. We chose it to separate the two parts of the CutSat calculus more clearly.

[4]Note that both CutSat and CutSat++ rely on this calculus as their foundation. Their only distinction is that CutSat uses strong conflict resolution and CutSat++ uses unguarded conflict resolution.

present in Subsection 3.2.3 a CDCL-like conflict resolution to undo these decisions and find actual proofs of unsatisfiability. As mentioned in the introduction of this chapter, the Fourier-Motzkin based resolution for linear integer arithmetic does not work on arbitrary constraints, but only on tight justifications.[5] For this reason, we also introduce in Subsection 3.2.3 a secondary rule system (Fig. 3.3) that refines constraints used for propagation into tight justifications. In the CutSat$_g$ rules, this secondary system is called by the functions tight(J, x_j, M) and div-derive$(D, x_j, \bowtie, M)$, where $\bowtie \in \{\leq, \geq\}$. We conclude CutSat$_g$ with the Slack-Intro rule in Subsection 3.2.6, which prevents stuck states when encountering unguarded variables.

3.2.1 States and Models

CutSat++ decides satisfiability of a problem C. It either ends in the state *unsat* or in a state $\langle \nu, \text{sat} \rangle$, where ν is a satisfiable assignment for C. In order to reach one of those two *end states*, the calculus produces *lower bounds* $x_j \geq b_j$ and *upper bounds* $x_j \leq b_j$ for the variables in C. The produced bounds are stored in a sequence $M = [\![\gamma_1, \ldots, \gamma_n]\!]$ and they describe a partial model for C. Out of convenience, we use $[\![M, \gamma]\!]$ and $[\![M_1, M_2]\!]$ to denote the concatenation of a bound γ at the end of M, and M_2 at the end of M_1, respectively. Moreover, we denote by $[\![]\!]$ the empty sequence.

By $\mathcal{L}(x_j, M) = l_j$ and $\mathcal{U}(x_j, M) = u_j$ we denote the values l_j, u_j of the greatest lower bound $x_j \geq l_j$ and least upper bound $x_j \leq u_j$ for a variable x_j in M, respectively. If there is no lower (upper) bound for x_j in M, then $\mathcal{L}(x_j, M) = -\infty$ ($\mathcal{U}(x_j, M) = \infty$). The definitions of $\mathcal{U}$ and $\mathcal{L}$ are extended to polynomials as done by [87]:

$$
\begin{array}{rcll}
\mathcal{L}(c, M) & = & c; & \\
\mathcal{L}(a \cdot x_j, M) & = & a \cdot \mathcal{L}(x_j, M), & \text{if } a > 0; \\
\mathcal{L}(a \cdot x_j, M) & = & a \cdot \mathcal{U}(x_j, M), & \text{if } a < 0; \\
\mathcal{L}(p + q, M) & = & \mathcal{L}(p, M) + \mathcal{L}(q, M); & \\
\mathcal{U}(c, M) & = & c; & \\
\mathcal{U}(a \cdot x_j, M) & = & a \cdot \mathcal{U}(x_j, M), & \text{if } a > 0; \\
\mathcal{U}(a \cdot x_j, M) & = & a \cdot \mathcal{L}(x_j, M), & \text{if } a < 0; \\
\mathcal{U}(p + q, M) & = & \mathcal{U}(p, M) + \mathcal{U}(q, M). &
\end{array}
$$

The partial model M is complete if all variables x_j are *fixed* in the sense that $\mathcal{U}(x_j, M) = \mathcal{L}(x_j, M)$. In this case, we define $\nu[M] : \text{vars}(C) \to \mathbb{Z}$ as the assignment that assigns to every variable x_j the value $\mathcal{L}(x_j, M)$. By $\text{val}(p, M) := \mathcal{L}(p, M)$, we denote the value assigned to a *fixed polynomial* p.[6] This also means that $\text{val}(p, M)$ is defined only if all variables occurring in p are fixed in M.

[5] See Subsection 3.2.3 or the introduction of this chapter for the precise definition.

[6] An expression/constraint/polynomial is *fixed* if all contained variables are fixed.

We define our CutSat++ calculus in the form of a transition system that traverses states via rules. The rules are applied in a non-deterministic way. This type of presentation supports proofs by induction on the length of rule applications, where the rules can be considered independently. A state in CutSat++ is either one of the two types of end states $\langle \nu, \text{sat} \rangle$, *unsat*, or of the form $S = \langle M, C, I \rangle$, where M is the current partial model, C is the current set of constraints, and I is either $\top$ or an inequality entailed by C ($C \vdash_{\mathbb{Z}} I$) that constitutes a conflict, i.e., it evaluates to false under the partial model M. In case I is just $\top$, we will often abbreviate $\langle M, C, \top \rangle$ as $S = \langle M, C \rangle$ and call it a *search state*. Otherwise, we call $\langle M, C, I \rangle$ a *conflict state*. The *start state* for a problem C is the search state $\langle [\![]\!], C \rangle$. Therefore, the start states of CutSat++ are all search states $\langle [\![]\!], C \rangle$, where C is a constraint system.

CutSat++ constructs all states $\langle M, C', I \rangle$ after the start state $\langle [\![]\!], C \rangle$ in such a way that (i) C' is equisatisfiable to C and that (ii) any satisfiable assignment ν for the current set of constraints C' is also a satisfiable assignment for the original set of constraints C. Moreover, any partial model M generated by CutSat++ (i) stays *consistent*, i.e., $\mathcal{L}(x_j, M) \leq \mathcal{U}(x_j, M)$ for all variables $x_j \in vars(C)$, and (ii) *improves*, i.e., $\mathcal{L}(x_j, M) > \mathcal{L}(x_j, M')$ if $M = [\![M', x_j \geq b_j]\!]$ and $\mathcal{U}(x_j, M) < \mathcal{U}(x_j, M')$ if $M = [\![M', x_j \leq b_j]\!]$. Finally, no state generated by CutSat++ is *stuck*, i.e., a state generated by CutSat++ is either an end state or a rule is applicable. All of the above properties hold also for CutSat_g and CutSat and we are going to elaborate why they hold step-by-step in the rest of this section.

3.2.2 Decisions and Propagations

We rely on decisions and propagations to construct a model (see Fig. 3.1). Via applications of the rule Decide, CutSat_g adds *decided bounds* $x_j \leq b_j$ or $x_j \geq b_j$, also called *decisions*, to the sequence M in state S [87]. A decided bound assigns a variable x_j to the lower or upper bound of x_j in M. Via applications of the propagation rules, CutSat_g adds *propagated bounds* $x_j \geq_I b_j$ or $x_j \leq_I b_j$ to the sequence M. To this end, the function $\text{bound}(J, x_j, \bowtie, M)$ defines the strictest lower bound (if $\bowtie = \geq$) or upper bound (if $\bowtie = \leq$) value b_j that can be computed for constraint J under the partial model M, and the function $\text{tight}(J, x_j, M)$ computes a corresponding justification I, i.e., a simplified version of the propagating constraint J that is needed for conflict resolution (see Subsection 3.2.3) and, therefore, annotated to the propagated bound $x \leq_I b$.[7] For an inequality J, $\text{bound}(J, x_j, \bowtie, M)$ is defined as follows:

[7] We postpone the definition of the function $\text{tight}(J, x_j, M)$ (Fig. 3.3) to Subsection 3.2.3 because justifications are first used explicitly in that part of this section.

Decide

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle [\![M, x_j \geq b_j]\!], C\rangle \quad \textbf{if} \begin{cases} \mathcal{U}(x_j, M) \neq +\infty, \\ \mathcal{L}(x_j, M) < b_j = \mathcal{U}(x_j, M) \end{cases}$$

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle [\![M, x_j \leq b_j]\!], C\rangle \quad \textbf{if} \begin{cases} \mathcal{L}(x_j, M) \neq -\infty, \\ \mathcal{L}(x_j, M) = b_j < \mathcal{U}(x_j, M) \end{cases}$$

Propagate

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle [\![M, x_j \geq_I b_j]\!], C\rangle \quad \textbf{if} \begin{cases} J \in C \text{ is an inequality}, \\ \text{improves}(J, x_j, \geq, M), \\ b_j = \text{bound}(J, x_j, \geq, M), \\ I = \text{tight}(J, x_j, M) \end{cases}$$

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle [\![M, x_j \leq_I b_j]\!], C\rangle \quad \textbf{if} \begin{cases} J \in C \text{ is an inequality}, \\ \text{improves}(J, x_j, \leq, M), \\ b_j = \text{bound}(J, x_j, \leq, M), \\ I = \text{tight}(J, x_j, M) \end{cases}$$

Propagate-Div

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle [\![M, x_j \geq_I c_j]\!], C\rangle \quad \textbf{if} \begin{cases} D = (d \mid a_j x_j + p_j) \in C, \\ p_j \text{ is fixed}, \\ \text{improves}(D, x_j, \geq, M), \\ c_j = \text{bound}(D, x_j, \geq, M), \\ I = \text{div-derive}(D, x_j, \geq, M) \end{cases}$$

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle [\![M, x_j \leq_I c_j]\!], C\rangle \quad \textbf{if} \begin{cases} D = (d \mid a_j x_j + p_j) \in C, \\ p_j \text{ is fixed}, \\ \text{improves}(D, x_j, \leq, M), \\ c_j = \text{bound}(D, x_j, \leq, M), \\ I = \text{div-derive}(D, x_j, \leq, M) \end{cases}$$

Figure 3.1: The decision and propagation rules of CutSat$_g$

$$\text{bound}(a_j x_j + p_j \leq 0, x_j, \leq, M) = \begin{cases} -\left\lceil \frac{\mathcal{L}(p_j, M)}{a_j} \right\rceil & \text{if } a_j > 0, \\ \infty & \text{if } a_j \leq 0. \end{cases}$$

$$\text{bound}(a_j x_j + p_j \leq 0, x_j, \geq, M) = \begin{cases} -\infty & \text{if } a_j \geq 0, \\ -\left\lfloor \frac{\mathcal{L}(p_j, M)}{a_j} \right\rfloor & \text{if } a_j < 0. \end{cases}$$

An inequality J only propagates finite upper bounds for x_j if $\text{coeff}(J, x_j) > 0$. Symmetrically, an inequality J only propagates finite lower bounds for x_j if $\text{coeff}(J, x_j) < 0$. For a divisibility constraint D, $\text{bound}(D, x_j, \bowtie, M)$ is defined as follows:

$$\text{bound}(d \mid a_j x_j + p_j, x_j, \bowtie, M) = \begin{cases} \left\lceil \frac{d\left\lceil \frac{a_j\, \mathcal{L}(x_j, M) + k_j}{d} \right\rceil - k_j}{a_j} \right\rceil & \text{if } \bowtie = \geq, \\ \left\lfloor \frac{d\left\lfloor \frac{a_j\, \mathcal{U}(x_j, M) + k_j}{d} \right\rfloor - k_j}{a_j} \right\rfloor & \text{if } \bowtie = \leq, \end{cases}$$

where $a_j > 0$, $d > 0$, all variables in p_j are fixed, and $\mathcal{L}(p_j) = k_j$. We defined the bound value $\text{bound}(D, x_j, \bowtie, M)$ for divisibility constraint D in such a way that CutSat$_g$ never skips a satisfiable solution for D:

Lemma 3.2.1 (No Skipping)**.** *Let $D = d \mid a_j x_j + p_j$ be a divisibility constraint with $a_j > 0$. Let $\langle M, C\rangle$ be a state where the polynomial p_j is fixed.*
(1) Let $b_j = \mathcal{L}(x_j, M)$ denote the current lower bound value for x_j and let $c_j = \text{bound}(d \mid a_j x_j + p_j, x_j, \geq, M)$ denote the lower bound value that we will propagate. Then $\text{bound}(d \mid a_j x_j + p_j, x_j, \geq, M)$ skips no satisfiable values for x_j, i.e., $d \nmid a_j v_j + \mathcal{L}(p_j, M)$ holds for all v_j such that $b_j \leq v_j < c_j$.
(2) Let $b_j = \mathcal{U}(x_j, M)$ denote the current upper bound value for x_j and let $c_j = \text{bound}(d \mid a_j x_j + p_j, x_j, \leq, M)$ denote the upper bound value that we will propagate. Then $\text{bound}(d \mid a_j x_j + p_j, x_j, \leq, M)$ skips no satisfiable values for x_j, i.e., $d \nmid a_j v_j + \mathcal{L}(p_j, M)$ holds for all v_j such that $c_j < v_j \leq b_j$.

Proof. We only prove the case where $c_j = \text{bound}(d \mid a_j x_j + p_j, x_j, \geq, M)$ denotes a lower bound. The proof for the second case is analogous. We assume for a contradiction that there exists a v_j such that $d \mid a_j v_j + \mathcal{L}(p_j, M)$ and $b_j \leq v_j < c_j$ are true. Since $d \mid a_j v_j + \mathcal{L}(p_j, M)$, it holds that

$$\left\lceil \tfrac{a_j v_j + k_j}{d} \right\rceil = \tfrac{a_j v_j + k_j}{d} \text{ and } \left\lceil \frac{d\left\lceil \frac{a_j v_j + k_j}{d} \right\rceil - k_j}{a_j} \right\rceil = \left\lceil \frac{d\frac{a_j v_j + k_j}{d} - k_j}{a_j} \right\rceil = v_j.$$

Since $\mathcal{L}(x_j, M) = b_j \leq v_j$, it holds that

$$\left\lceil \tfrac{a_j\, \mathcal{L}(x_j, M) + k_j}{d} \right\rceil \leq \left\lceil \tfrac{a_j v_j + k_j}{d} \right\rceil.$$

Hence,

$$\text{bound}(d \mid a_j x_j + p_j, x_j, \geq, M) = \left\lceil \frac{d\left\lceil \frac{a_j b_j + k_j}{d} \right\rceil - k_j}{a_j} \right\rceil \leq \left\lceil \frac{d\left\lceil \frac{a_j v_j + k_j}{d} \right\rceil - k_j}{a_j} \right\rceil = v_j.$$

Therefore, $b_j \leq v_j < c_j$ is not true, which contradicts our initial assumption. □

The rules of the CutSat$_g$ calculus are restricted in such a way that M stays *consistent*, i.e., $\mathcal{L}(x_j, M) \leq \mathcal{U}(x_j, M)$ for all variables $x_j \in vars(C)$. CutSat$_g$ also propagates only bounds that are more *strict* than the current bound for the variable x_j, e.g., if CutSat$_g$ is going to propagate the lower bound $x_j \geq b_j$, then the new lower bound b_j must be strictly greater than the current lower bound $\mathcal{L}(x_j, M)$ for x_j. This behaviour is expressed by the following predicate for constraints J:

$$\text{improves}(J, x_j, \geq, M) = \mathcal{U}(x_j, M) \geq \text{bound}(J, x_j, \geq, M) > \mathcal{L}(x_j, M)\,,$$
$$\text{improves}(J, x_j, \leq, M) = \mathcal{L}(x_j, M) \leq \text{bound}(J, x_j, \leq, M) < \mathcal{U}(x_j, M)\,.$$

3.2.3 (Guarded) Conflict Resolution

Conflict

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M, C, p_j \leq 0\rangle \qquad \textbf{if } p_j \leq 0 \in C,\ \mathcal{L}(p_j, M) > 0$$

Conflict-Div

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M, C, I\rangle \qquad \textbf{if } \begin{cases} J = (d \mid a_j x_j + p_j) \in C, \\ p_j \text{ is fixed}, \\ b_j = \mathcal{U}(x_j, M), \\ \text{bound}(J, x_j, \geq, M) > b_j, \\ I = \text{div-derive}(J, x_j, \geq, M) \end{cases}$$

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M, C, I\rangle \qquad \textbf{if } \begin{cases} J = (d \mid a_j x_j + p_j) \in C, \\ p_j \text{ is fixed}, \\ b_j = \mathcal{L}(x_j, M), \\ \text{bound}(J, x_j, \leq, M) < b_j, \\ I = \text{div-derive}(J, x_j, \leq, M) \end{cases}$$

Skip-Decision

$$\langle [\![M, \gamma]\!], C, I\rangle \Longrightarrow_{\text{CS}} \langle M, C, I\rangle \qquad \textbf{if } \begin{cases} \gamma \text{ is a decided bound}, \\ I = (p_j \leq 0), \\ \mathcal{L}(p_j, M) > 0 \end{cases}$$

Resolve

$$\langle [\![M, \gamma]\!], C, I\rangle \Longrightarrow_{\text{CS}} \langle M, C, I'\rangle \qquad \textbf{if } \begin{cases} I \neq \top, \\ \gamma \text{ is a propagated bound}, \\ I' = \text{resolve}(\gamma, I) \end{cases}$$

Backjump

$$\langle [\![M, \gamma, M']\!], C, J\rangle \Longrightarrow_{\text{CS}} \langle [\![M, \gamma']\!], C\rangle \qquad \textbf{if } \begin{cases} \gamma \text{ is a decided bound}, \\ \text{coeff}(J, x_j) < 0, \\ \text{improves}(J, x_j, M), \\ I = \text{tight}(J, x_j, M), \\ b_j = \text{bound}(J, x_j, \geq, M), \\ \gamma' = x_j \geq_I b_j \end{cases}$$

$$\langle [\![M, \gamma, M']\!], C, J\rangle \Longrightarrow_{\text{CS}} \langle [\![M, \gamma']\!], C\rangle \qquad \textbf{if } \begin{cases} \gamma \text{ is a decided bound}, \\ \text{coeff}(J, x_j) > 0, \\ \text{improves}(J, x_j, M), \\ I = \text{tight}(J, x_j, M), \\ b_j = \text{bound}(J, x_j, \leq, M), \\ \gamma' = x_j \leq_I b_j \end{cases}$$

Figure 3.2: The (guarded) conflict resolution rules of CutSat$_g$

Although CutSat$_g$ constructs only consistent models M, it is possible that a constraint J is unsatisfiable under the model M. If constraint J is unsatisfiable under the model M in the state $S = \langle M, C, I\rangle$, then we call J a *conflict*. An inequality $p_j \leq 0$ is a conflict if $\mathcal{L}(p_j, M) > 0$; a divisibility

constraint $d \mid a_j x_j + p_j$ is a conflict if all variables in p_j are fixed and $d \nmid a_j b_j + \mathcal{L}(p_j, M)$ is unsatisfiable for all integers v_j with $\mathcal{L}(x_j, M) \leq v_j \leq \mathcal{U}(x_j, M)$, i.e., there exists no integer value for x_j within its current bounds such that $d \mid a_j x_j + \mathcal{L}(p_j, M)$ is satisfiable.

Since CutSat$_g$ can use the Decide rules to introduce bounds that are not implied by the problem, encountering a conflict does not necessarily mean that the problem is unsatisfiable. It is also possible that the model became unsatisfiable because of a decided bound introduced by CutSat$_g$. To find and undo those responsible decided bounds, CutSat$_g$ uses a CDCL-like conflict resolution, which we call *guarded/standard conflict resolution* (Fig. 3.2) [114]. In CDCL, *Boolean resolution* is used to combine the current conflict $C \vee l$ with a clause used for unit propagation $C' \vee \bar{l}$ to receive a new conflict $C \vee C'$ without literals l or $\bar{l}$. For CutSat$_g$ the function resolve:

$$\text{resolve}(x_j \bowtie_I b_j, J) = \begin{cases} |a_j| q_j + |c_j| p_j \leq 0 & \text{if } a_j \cdot c_j < 0, \\ a_j x + p_j \leq 0 & \text{otherwise}, \end{cases}$$

where $I = c_j x_j + q_j \leq 0$, $J = a_j x_j + p_j \leq 0$, and $\bowtie \in \{\leq, \geq\}$, fulfills a similar purpose: If the propagated bound $x \bowtie_{cx+q \leq 0} b$ is one of the bounds responsible for $ax + p \leq 0$ being a conflict (i.e., $a \cdot c < 0$), then $\text{resolve}(x \bowtie_{cx+q \leq 0} b, ax + p \leq 0)$ combines the current conflict constraint $ax + p \leq 0$ with the propagation justification $cx + q \leq 0$ to the new conflict constraint $|a|q + |c|p \leq 0$. If the propagated bound $x \bowtie_{cx+q \leq 0} b$ is not one of the responsible bounds (i.e., $a \cdot c \geq 0$), then $\text{resolve}(x \bowtie_{cx+q \leq 0} b, ax + p \leq 0)$ ignores the propagation justification and simply returns/keeps the current conflict constraint $ax + p \leq 0$. The function resolve is also the reason why we do not annotate propagated bounds with the actual constraints used for propagation, but with specially constructed justifications. To be more precise, the justifications are constructed in such a way that resolve is *conflict preserving*, i.e., $J' = \text{resolve}(\gamma, J)$ is a conflict in state $\langle M, C, J' \rangle$ and $C \vdash_{\mathbb{Z}} J'$ if $J = a_j x_j + p_j \leq 0$ was a conflict in state $\langle [\![M, \gamma]\!], C, J \rangle$ and $C \vdash_{\mathbb{Z}} J$ [87]. In CutSat$_g$, the function resolve is conflict preserving because it requires that every *justification* I annotated to a bound $\gamma = x_j \bowtie_I b_j$ is *tight*, i.e., justification I fulfills in state $\langle M, C, J \rangle$ the following conditions: Firstly, I is an inequality and $C \vdash_{\mathbb{Z}} I$. Secondly, if $\bowtie = \leq$, then $\text{coeff}(I, x_j) = 1$; if $\bowtie = \geq$, then $\text{coeff}(I, x_j) = -1$. Finally, $\text{bound}(I, x_j, \bowtie, M) \bowtie b$, i.e., the justification I implies at least a bound as strong as $x_j \bowtie_I b_j$.

CutSat$_g$ calculates the justification I for the bound $x_j \bowtie_I b_j$ ($\bowtie \in \{\leq, \geq\}$) propagated from inequality $J = \pm a_j x_j + p_j \leq 0$ with the function $\text{tight}(J, x_j, M) = I$ defined by the set of rules in Figure 3.3. A state in this rule system is a pair

$$\langle M', \pm a_j x_j + a_j s \oplus r \rangle,$$

where $a_j > 0$, s and r are polynomials, and M' is a prefix of the initial M, i.e., $M = [\![M', M'']\!]$. The start state for $\text{tight}(\pm a_j x_j + p_j \leq 0, x_j, M)$ is $\langle M, \pm a_j x_j \oplus p_j \rangle$ and the corresponding end state is the tight linear justification for the inequality $\pm a_j x_j + p_j \leq 0$. The first goal of the tight

Consume
$\langle M, \pm a_j x_j + a_j s \oplus a_j a_i x_i + r\rangle \Longrightarrow_{\text{tight}}$
$\langle M, \pm a_j x_j + a_j s_j + a_j a_i x_i \oplus r\rangle$,
where $x_j \neq x_i$.
Resolve-Implied
$\langle [\![M, \gamma]\!], \pm a_j x_j + a_j s \oplus p\rangle \Longrightarrow_{\text{tight}}$
$\langle M, \pm a_j x_j + a_j s_j \oplus q\rangle$,
where γ is a propagated bound and $q \leq 0 = \text{resolve}(\gamma, p \leq 0)$.
Decide-Lower
$\langle [\![M, x_i \geq b_i]\!], \pm a_j x_j + a_j s \oplus a_i x_i + r\rangle \Longrightarrow_{\text{tight}}$
$\langle M, \pm a_j x_j + a_j s + a_j c_i x_i \oplus r + (a_j c_i - a_i) q\rangle$,
where $x_i \leq_I b_i$ in M, with $I = x_i + q \leq 0$, and $c_i = \left\lceil \frac{a_i}{a_j} \right\rceil$.
Decide-Lower-Neg
$\langle [\![M, x_i \geq b_i]\!], \pm a_j x_j + a_j s \oplus a_i x_i + r\rangle \Longrightarrow_{\text{tight}}$
$\langle M, \pm a_j x_j + a_j s \oplus a_i q + r\rangle$,
where $x_i \leq_I b_i$ in M, with $I = x_i - q \leq 0$, and $a_i < 0$.
Decide-Upper
$\langle [\![M, x_i \leq b_i]\!], \pm a_j x_j + a_j s \oplus a_i x_i + r\rangle \Longrightarrow_{\text{tight}}$
$\langle M, \pm a_j x_j + a_j s + a_j c_i x_i \oplus r + (a_i - a_j c_i) q\rangle$,
where $x_i \geq_I b_i$ in M, with $I = -x_i + q \leq 0$, and $c_i = \left\lfloor \frac{a_i}{a_j} \right\rfloor$.
Decide-Upper-Pos
$\langle [\![M, x_i \leq b_i]\!], \pm a_j x_j + a_j s \oplus a_i x_i + r\rangle \Longrightarrow_{\text{tight}}$
$\langle M, \pm a_j x_j + a_j s \oplus a_i q + r\rangle$,
where $x_i \geq_I b_i$ in M, with $I = -x_i + q \leq 0$, and $a_i > 0$.
Round (and terminate)
$\langle M, \pm a_j x_j + a_j s \oplus b_j\rangle \Longrightarrow_{\text{tight}}$
$\pm\, x_j + s + \left\lceil \frac{b_j}{a_j} \right\rceil \leq 0$

Figure 3.3: Rule system that derives tightly propagating inequalities [87]

rule system is to produce an inequality where all coefficients are divisible by $a_j = \text{coeff}(J, x_j)$. To this end, we apply rules (Figure 3.3) that linearly combine the constraint we want to tighten with tightened constraints from previous propagations. The process stops as soon as the polynomial on the right side of $\oplus$ becomes empty. In this case, all coefficients are divisible by $a_j = \text{coeff}(J, x_j)$ and we derive the justification I with the final rule Round. The tight-transition-system is guaranteed to derive a tight inequality because there exist appropriate bounds with tight justifications for all variables occurring in inequality J (except x_j) or CutSat$_g$ could not propagate a bound for J.

CutSat$_g$ calculates the justification I for the bound $x_j \bowtie_I b_j$ ($\bowtie \in \{\leq, \geq\}$) propagated from divisibility constraint $D = d \mid a_j x_j + p_j$ with the function div-derive$(D, x_j, \bowtie, M)$. To do so, CutSat$_g$ uses the fact that any bound propagated for D can also be propagated with an intermediate step by the two auxiliary inequalities $-dz + a_j x_j + p_j \leq 0$ and $dz - a_j x_j - p_j \leq 0$, which are implied by D. For instance, let us look at the lower bound value:

$$\text{bound}(D, x_j, \geq, M) = \left\lceil \frac{d \left\lceil \frac{a_j b_j + k_j}{d_j} \right\rceil - k_j}{a_j} \right\rceil,$$

where $D = d \mid a_j x_j + p_j$, $a_j > 0$, $d > 0$, p_j is fixed, $k_j = \mathcal{L}(p_j, M)$, and $b_j = \mathcal{L}(x_j, M)$. First we split the *diophantine representation*[8] $dz = a_j x_j + p_j$ of the divisibility constraint D into two inequalities: $-dz + a_j x_j + p_j \leq 0$ and $dz - a_j x_j - p_j \leq 0$. Then we see that the subterm $c = \left\lceil \frac{a_j b_j + k_j}{d} \right\rceil$ is equal to the bound value computable from one of the split inequalities:

$$\text{bound}(-dz + a_j x_j + p_j \leq 0, z, \geq, M) = \left\lceil \frac{a_j b_j + k_j}{d} \right\rceil.$$

The fitting justifications for the two types of subterms are abbreviated with div-part(D, x_j, M):

$$\text{div-part}(D, x_j, \geq, M) = \text{tight}(-dz + a_j x_j + p_j \leq 0, z, M)$$

is the justification for the lower bound subterm $c = \left\lceil \frac{a_j b_j + k_j}{d} \right\rceil$ and

$$\text{div-part}(D, x_j, \leq, M) = \text{tight}(\quad dz - a_j x_j - p_j \leq 0, z, M)$$

is the justification for the upper bound subterm $c = \left\lfloor \frac{a_j b_j + k_j}{d} \right\rfloor$. For div-part, we forbid tight to apply the Consume rule to the variables x_j and z. This restriction to the Consume rule guarantees that the resulting inequality $\pm z + r \leq 0 = \text{div-part}(D, x_j, M)$ does not contain the variable x_j. Given $I_2 = -z + r \leq 0 = \text{div-part}(D, x_j, M)$ and $I_1 = dz - a_j x_j - p_j \leq 0$, we use $\text{resolve}(x_j \geq_{I_2} c, I_1) = -a_j x_j + dr - p_j \leq 0 = I_3$ to receive the inequality that computes the complete lower bound:

$$\text{bound}(I_3, x_j, \geq, M) = \left\lceil \frac{d\,\mathcal{L}(r, M) - k_j}{a_j} \right\rceil \geq \left\lceil \frac{d \left\lceil \frac{a_j b_j + k_j}{d_j} \right\rceil - k_j}{a_j} \right\rceil.$$

Finally, we compute the justification for the actual divisibility constraint $D = d \mid a_j x_j + p_j$ with the function div-derive(D, x_j, M), which is defined as

$$\text{div-derive}(D, x_j, \geq, M) = \text{tight}(-a_j x_j + dr - p_j \leq 0, x_j, M),$$

where $-z + r \leq 0 = \text{div-part}(D, x_j, \geq, M)$, and

$$\text{div-derive}(D, x_j, \leq, M) = \text{tight}(\quad a_j x_j + dr + p_j \leq 0, x_j, M),$$

where $z + r \leq 0 = \text{div-part}(D, x_j, \leq, M)$.

[8] Notice that z is a variable not occurring in the problem and only introduced for the calculation of div-derive$(D, x_j, \bowtie, M)$. Before the calculation of div-derive$(D, x_j, \bowtie, M)$ ends, z is eliminated. This means that div-derive never introduces a new variable to the problem although it introduces z for the intermediate calculations.

Learn
$\langle M, C, I\rangle \Longrightarrow_{\text{CS}} \langle M, C \cup \{I\}, I\rangle$ **if** $I \notin C \cup \{\top\}$

Forget
$\langle M, C \cup \{J\}\rangle \Longrightarrow_{\text{CS}} \langle M, C\rangle$ **if** $C \vdash_{\mathbb{Z}} J$, and $J \notin C$

Figure 3.4: The Learn and Forget rules of CutSat$_g$

3.2.4 Learning

The original set of constraints C implies all constraints I derived during the standard conflict resolution, i.e., all constraints I in a conflict state $\langle M, C, I\rangle$ visited by CutSat$_g$. Therefore, we can add any such constraint I to our constraint set C and still get an equivalent set of constraints $C' = C \cup \{I\}$. CutSat$_g$ can do this too with the rule Learn (Fig. 3.4). Extending our constraint set with learned constraints is not necessary for termination or completeness. However, learning constraints is useful in practice because the learned constraints might propagate bounds that our original constraints cannot propagate. Still, we do not want to add all conflict constraints I from every conflict state $\langle M, C, I\rangle$ to our constraint set as this would also slow down CutSat$_g$. In practice, we recommend to only learn those conflicts J from which CutSat$_g$ applies the Backjump rule. Moreover, we recommend to use the rule Forget (Fig. 3.4) to remove in regular intervals those learned constraints that have not propagated enough bounds.

3.2.5 Reaching the End States

CutSat$_g$ ends a run when it has determined whether the original constraint set has an integer solution or not. Formally, CutSat$_g$ does so with the rules Sat, Unsat-Div, and Unsat (Fig. 3.5). Sat is called as soon as CutSat$_g$ has found a complete model M, i.e., a model where all variables are fixed, that also satisfies all constraints in our current constraint set C. The assignment $\nu[M]$ in the end state $\langle \upsilon[M], \text{sat}\rangle$ is then a satisfiable assignment for the current and the original constraint set.

The rules Unsat-Div and Unsat are applicable as soon as CutSat$_g$ derives a trivially unsatisfiable constraint, i.e., either a constant inequality $b_j \leq 0$ that is unsatisfiable or a divisibility constraint $d \mid a_1x_1 + \ldots + a_nx_n + c$ that can never be true because $\gcd(d, a_1, \ldots, a_n) \nmid c$. The unsatisfiability of the original constraint set is then marked by the end state unsat.

If we apply the rules according to the following strategy, then CutSat$_g$ is sound, complete, and terminates for *guarded problems*, i.e., those input problems C containing only guarded variables:

Definition 3.2.2 (Reasonable Strategy [87])**.** A strategy is *reasonable* if Propagate applied to constraints of the form $\pm x_j - b_j \leq 0$ has the highest priority over all rules and the Forget Rule is applied only finitely often.

Sat

$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle v[M], \text{sat}\rangle$ **if** $\nu[M]$ satisfies C

Unsat-Div

$\langle M, C\rangle \Longrightarrow_{\text{CS}} \text{unsat}$ **if** $\begin{cases} d \mid a_1x_1 + \ldots + a_nx_n + c \in C, \\ \gcd(d, a_1, \ldots, a_n) \nmid c \end{cases}$

Unsat

$\langle M, C, b_j \leq 0\rangle \Longrightarrow_{\text{CS}} \text{unsat}$ **if** $b_j > 0$

Figure 3.5: The rules of CutSat$_g$ that lead to end states

If the problem is *unguarded*, i.e., the input problem C contains at least one unguarded variable, then CutSat$_g$ with a reasonable strategy can diverge. In the next Section, we will discuss why CutSat$_g$ diverges when there are unguarded variables and why its extension CutSat cannot prevent all divergent behavior. But before move on to CutSat, we extend CutSat$_g$ by one final rule.

3.2.6 Slack-Intro

As mentioned before, CutSat$_g$ might diverge on unguarded problems. However, CutSat$_g$ might also get *stuck* on unguarded problems, i.e., reach a state that is not an end state but where no rule is applicable. For this reason, we are going to add the rule Slack-Intro (Fig. 3.6) to CutSat$_g$.

Until we add Slack-Intro to CutSat$_g$, stuck states are possible because CutSat$_g$ is sometimes unable to propagate and, thereby, also not decide any bounds for a variable x_j. In this case, we call the variable x_j stuck. More formally, a variable x_j is called *stuck* in state $S = \langle M, C\rangle$ if M contains no bounds for x_j and we can neither use Propagate or Propagate-Div to add a bound for x_j. Guarded variables x_j are never stuck because they have by definition constraints of the form $\pm x_j - b_j \leq 0 \in C$, which CutSat$_g$ is always able to propagate. The easiest example where stuck variables result in a stuck state is the state $\langle [\![]\!], \{x_1 - x_2 \leq 0\}\rangle$. Here both variables are stuck, so we can never construct a model.

In case all unfixed variables x_j are stuck, Slack-Intro allows us to add new constraints to the problem that make one of the variables unstuck. To this end, Slack-Intro exploits the following fact: an input problem C is only satisfiable if it is satisfiable inside an a priori fixed finite interval for all variables, i.e., if there exists an $x_S \geq 0$ such that $C \cup \{-x_S \leq x_i \leq x_S\}$ is also satisfiable. Slack-Intro simulates this fact in CutSat$_g$ by adding a new variable x_S and some constraints $\{-x_S \leq 0, x_i - x_S \leq 0, -x_i - x_S \leq 0\}$ to C such that the bounds $-x_S \leq x_i \leq x_S$ can be propagated after x_S is propagated and decided. This also means that CutSat$_g$ expands the bounds for all previously stuck variables incrementally and symmetrically from the integer constant zero towards $\pm\infty$. Slack-Intro alone actually prevents all stuck states for CutSat$_g$.

Slack-Intro

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M, C'\rangle \quad \textbf{if} \begin{cases} x_j \text{ is stuck,} \\ \text{all other unfixed variables} \\ \text{in } \langle M, C\rangle \text{ are also stuck,} \\ x_S \text{ is the slack-variable,} \\ C' = C \cup \{-x_S \leq 0, x_j - x_S \leq 0, \\ \qquad\qquad - x_j - x_S \leq 0\} \end{cases}$$

Figure 3.6: The Slack-Intro rule of CutSat$_g$

3.3 Divergence of CutSat

An easy example for divergence—already stated by [87]—relies on cyclic dependencies during propagation:

Example 3.3.1. Let $S_i = \langle M_i, C\rangle$ ($i \in \mathbb{N}$) be a series of states defined by:

$$\begin{aligned} C &:= \{\underbrace{-x_1 \leq 0}_{I_{x_1}}, \underbrace{-x_2 \leq 0}_{I_{x_2}}, \underbrace{-x_3 \leq 0}_{I_{x_3}}, \underbrace{1 - x_1 + x_2 \leq 0}_{J_1}, \underbrace{x_1 - x_2 - x_3 \leq 0}_{J_2}\} \\ M_0 &:= [\![x_1 \geq_{I_{x_1}} 0, x_2 \geq_{I_{x_2}} 0, x_3 \geq_{I_{x_3}} 0, x_3 \leq 0]\!] \\ M_{i+1} &:= [\![M_i, x_1 \geq_{J_1} i+1, x_2 \geq_{J_2} i+1]\!] \end{aligned}$$

CutSat$_g$ with a reasonable strategy has diverging runs starting in state $S'_0 = \langle [\![]\!], C\rangle$. Let CutSat$_g$ traverse the states S'_0, S_0, S_1, S_2, ... in the following fashion: CutSat$_g$ reaches state $S_0 := \langle M_0, C\rangle$ from state S'_0 after propagating the constraints I_{x_1}, I_{x_2}, I_{x_3}, and fixing x_3 with a decided bound. CutSat$_g$ reaches state S_{i+1} from state S_i after:

- applying Propagate to J_1 to propagate $\gamma_{i+1}^{x_1} := x_1 \geq_{J_1} i+1$ so that $M'_i := [\![M_i, \gamma_{i+1}^{x_1}]\!]$ and $S'_i := \langle M'_i, C\rangle$
- applying Propagate to J_2 to propagate $\gamma_{i+1}^{x_2} := x_2 \geq_{J_2} i+1$ so that $M_{i+1} := [\![M'_i, \gamma_{i+1}^{x_2}]\!]$ and $S_{i+1} := \langle M_{i+1}, C\rangle$

To summarize, this example shows how we can exploit the cyclic dependence between x_1 and x_2 to increase the lower bounds for x_1 and x_2 to an arbitrarily large value $i \in \mathbb{N}$.

To prevent this type of divergence, Jovanović and de Moura suggest to restrict the rules Propagate and Propagate-Div so they only propagate those bounds that are δ-relevant.

Definition 3.3.2 (δ-relevant [87])**.** Let $\text{nb}(x_j, M)$ be the number of bounds for x_j in M. Let $\delta > 0$ and $\text{nb}_{\max} \in \mathbb{N}$ be two parameters that are fixed at the start of the CutSat$_g$ run. Then the new lower bound $x_j \geq_J b_j$ is δ-relevant in the state $\langle M, C\rangle$ if:

- $\mathcal{L}(x_j, M) = -\infty$, or
- $\mathcal{U}(x_j, M) \neq \infty$, or
- $\mathcal{L}(x_j, M) + \delta|\,\mathcal{L}(x_j, M)| < b_j$ and $\text{nb}(x_j, M) < \text{nb}_{\max}$.

Likewise, the new upper bound $x_j \leq_J b_j$ is δ-relevant in the state $\langle M, C\rangle$ if:

- $\mathcal{U}(x_j, M) = \infty$, or
- $\mathcal{L}(x_j, M) \neq -\infty$, or
- $\mathcal{U}(x_j, M) - \delta|\mathcal{U}(x_j, M)| > b_j$ and $\mathrm{nb}(x_j, M) < \mathrm{nb}_{\max}$.

If we now restrict the rules Propagate and Propagate-Div as suggested, then δ-relevance gives us the following guarantees: The first case of δ-relevance guarantees that CutSat$_g$ can still propagate at least one bound for every variable. The second case of δ-relevance guarantees that a bounded variable (with $x_j - u_j \leq 0 \in C$ and $-x_j + l_j \leq 0 \in C$) can be propagated without restrictions because its guards $l_j \leq x_j \leq u_j$ already guarantee that there are at most $|u_j - l_j + 1|$ propagations for x_j. The last case of δ-relevance guarantees that an unbounded variable is at most propagated $\mathrm{nb}_{\max}$ times and, thus, prevents propagation divergence. The additional conditions $\mathcal{L}(x_j, M) + \delta|\mathcal{L}(x_j, M)| < b_j$ and $\mathcal{U}(x_j, M) - \delta|\mathcal{U}(x_j, M)| > b_j$ prevent CutSat$_g$ from wasting its limited amount of propagations on those that change the bounds only slightly. We also do not introduce any stuck states with our restriction to δ-relevance because any bound $x_j \geq_J b_j$ that the third case of δ-relevance prevents from being propagated can also be introduced to the bound sequence M if we first fix x_j with Decide, then apply Conflict(-Div) to J, and finally add the bound $x_j \geq_J b_j$ to M with Backjump.

However, this simulation of Propagate(-Div) via Decide and conflict resolution also means that divergence is not eliminated but only shifted divergence from propagation to conflict resolution. We can also use Example 3.3.1 for this type of divergence; but this time we simulate propagation as described above via conflict resolution. Instead of applying Propagate between the states S_i and S_{i+1}, we fix one variable at a time with a decided bound. If we fix x_1 to $\mathcal{L}(x_1, M_i)$ with a decided bound, we directly detect a conflict in J_1. We enter conflict resolution by applying Conflict to J_1, which can be exited directly with the Backjump rule. This removes the decided bound on x_1 and adds instead $x_1 \geq_{J_1} i + 1$ on top of the bound sequence. Hence, we reach the intermediate state S_i' without applying the rule Propagate. Analogously, we do the same for x_2 with J_2 and reach the state S_{i+1} without applying the rule Propagate.

The best way to handle this type of divergence is to forbid standard conflict resolution—i.e., the rules Conflict, Conflict-Div, Resolve, Backjump, Skip-Decision, Unsat, and Learn—from handling constraints containing unguarded variables. As an alternative to normal conflict resolution, Jovanović and de Moura suggested a secondary conflict analysis called *strong conflict resolution* [87]. Strong conflict resolution is also the predecessor of the unguarded conflict resolution we are going to present in Section 3.5. Strong conflict resolution is based on the fact that we can transform any unguarded problem into a guarded one if we eliminate all unguarded variables with a quantifier elimination procedure. Instead of eliminating all unguarded va-

Resolve-Cooper

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M', C \cup R\rangle \quad \textbf{if} \begin{cases} (x_j, C') \text{ is a conflicting core,} \\ x_j \text{ is unguarded,} \\ x_j \text{ is the minimal conflicting variable,} \\ (R_y, R_c) = \text{cooper}(x_j, C'), \\ R = R_y \cup R_c \end{cases}$$

Solve-Div

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M, C'\rangle \quad \textbf{if} \begin{cases} \text{divisibility constraints } I_1, I_2 \in C, \\ \{I_1', I_2'\} = \text{div-solve}(x_j, \{I_1, I_2\}), \\ C' = C \setminus \{I_1, I_2\} \cup \{I_1', I_2'\} \end{cases}$$

$$\text{div-solve}(x_j, \{d_1 \mid a_{1j}x_j + p_{1j}, d_2 \mid a_{2j}x_j + p_{2j}\}) = \{d_1 d_2 \mid dx_j + c_{1j}d_{2j}p_{1j} + c_{2j}d_{1j}p_{2j}, d \mid -a_{1j}p_{2j} + a_{2j}p_{1j}\},$$

where $d = \gcd(a_{1j}d_{2j}, a_{2j}d_{1j})$, and c_{1j} and c_{2j} are integers such that $c_{1j}a_{1j}d_{2j} + c_{2j}a_{2j}d_{1j} = d$ [44, 87].

Figure 3.7: The strong conflict resolution rules by [87]

riables before we apply CutSat$_g$, strong conflict resolution introduces two new rules (see Figure 3.7) that extend the CutSat$_g$ calculus by a second type of conflict resolution, which applies quantifier elimination in-between the applications of the original CutSat$_g$ rules. We call CutSat$_g$ with this extension CutSat. The newly added rules do not apply the complete quantifier elimination algorithm, nor do they apply it at all possible instances, but only on minimal conflicts containing unguarded variables. These minimal conflicts are also called conflicting cores.

Definition 3.3.3 (Conflicting Cores [87])**.** Let $S = \langle M, C\rangle$, $C' \subseteq C$, let x_j be a variable in C', let all other variables in C' be fixed, let $a_j, b_j > 0$, and let $l_j = \text{bound}(-a_j x_j + p_j \leq 0, x_j, \geq, M)$ and $u_j = \text{bound}(b_j x_j - q_j \leq 0, x_j, \leq, M)$. The pair (x, C') is a *conflicting core* if it is of one of the following two forms:
(1) $C' = \{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0\}$ and $l_j > u_j$, i.e., the lower bound from $-a_j x_j + p_j \leq 0$ contradicts the upper bound from $b_j x_j - q_j \leq 0$; in this case, (x_j, C') is called an *interval conflicting core* and its *strong resolvent* is $(\{-y_i \leq 0, y_i - a_j + 1 \leq 0\}, \{b_j p_j - a_j q_j + b_j y_i \leq 0, a_j \mid y_i + p_j\})$
(2) $C' = \{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0, d \mid c_j x_j + s_j\}$ and $l_j \leq u_j$, and for all $v_j \in [l_j, u_j] \cap \mathbb{Z}$ we have $d \nmid c_j v_j + \mathcal{L}(s, M)$, i.e., there exists no value for x_j within the bounds defined by the two inequalities such that the divisibility constraint becomes satisfiable; in this case, (x_j, C') is called a *divisibility conflicting core* and its *strong resolvent* is $(\{-y_i \leq 0, y_i - m \leq 0\}$, $\{b_j p_j - a_j q_j + b_j y_i \leq 0, a_j \mid y_i + p_j, a_j d \mid c_j p_j + a_j s_j + c_j y_i\})$
In both cases, y_i is a fresh variable and $m = \text{lcm}\left(a_j, \frac{a_j d}{\gcd(a_j d, c_j)}\right) - 1$.

We refer to the respective strong resolvents for a conflicting core (x_j, C') by the function cooper(x_j, C'), which returns a strong resolvent (R_y, R_c) as defined above, Definition 3.3.3. These strong resolvents (R_y, R_c) are simply the constraints that we get after we eliminate $\exists x_j \in \mathbb{Z}$ from $\exists x_j \in \mathbb{Z}.\ C'$ with Cooper's quantifier elimination procedure [44]. Note that the newly introduced variable y_i is guarded by the constraints in R_y. In Section 3.5, we will extend the definition of conflicting cores by a third type of core called a diophantine conflicting core. In the final example at the end of this section, we will also show why this core is necessary to guarantee termination.

Besides conflicting cores and strong resolvents, strong conflict resolution also relies on a total order $\prec$ over all variables such that $x_i \prec x_j$ for all guarded variables x_i and unguarded variables x_j. In relation to Cooper's quantifier elimination, the order $\prec$ describes the elimination order for the unguarded variables, viz., $x_i \prec x_j$ if x_j is eliminated before x_i. A variable x_j is called *maximal* in a constraint I if x_j is contained in I, i.e., coeff$(I, x_j) \neq 0$, and all other variables x_i in I are smaller, i.e., $x_i \prec x_j$. The maximal variable in I is also called its *top variable* $(x_j = \text{top}(I))$. If there is a conflicting core (x_j, C') in some state S, then x_j is called a *conflicting variable*. The conflicting variable is *minimal* in the state S if there exists no conflicting variable x_i in S such that $x_i \prec x_j$.

Instead of actually eliminating a conflicting variable or the conflicting core (x_j, C'), the rule Resolve-Cooper (see Figure 3.7) only adds the constraints $R_y \cup R_c$ from the strong resolvent (R_y,R_c) to the problem. By doing so, Resolve-Cooper is supposed to guarantee that the old conflicting core (x_j, C') is always ignored in favor of a "smaller" conflicting core. The following strategy guarantees that strong conflict resolution and the guarded conflict resolution interact as little as possible. To be more precise, guarded conflict resolution is supposed to be applied to *guarded conflicts*, i.e., conflict constraints containing only guarded variables, and strong conflict resolution is supposed to be applied to *unguarded conflicts*, i.e., conflict constraints containing unguarded variables. As a consequence, the strategy is also supposed to guarantee termination for the whole calculus:

Definition 3.3.4 (Two-layered Strategy)**.** We say a strategy is *two-layered* if

- it is reasonable (Definition 3.2.2);
- the Propagate & Propagate-Div rules are limited to δ-relevant bounds;
- the Forget rule is never used to eliminate resolvents introduced by Resolve-Cooper;
- it only applies the Conflict and Conflict-Div rules if Resolve-Cooper is not applicable.

However, we are now going to discuss four examples where CutSat diverges despite strong conflict resolution. The first one shows that CutSat can apply Conflict and Conflict-Div infinitely often to constraints containing unguarded variables.

Example 3.3.5. Let

$$C := \{ \underbrace{-x_1 \leq 0}_{I_1}, \underbrace{-x_2 \leq 0}_{I_2}, \underbrace{-x_3 \leq 0}_{I_3}, \underbrace{x_3 \leq 0}_{I_4}, \underbrace{x_3 + 1 \leq 0}_{I_5}, \underbrace{1 - x_1 + x_2 \leq 0}_{J_1}, \underbrace{x_1 - x_2 \leq 0}_{J_2} \}$$

be a problem. Let $S_i = \langle M_i, C \rangle$ for $i \in \mathbb{N}$ be a series of states with:

$$\begin{aligned} M_0 &:= [\![x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0]\!], \\ M_{i+1} &:= [\![M_i, x_1 \geq_{J_1} i+1, x_2 \geq_{J_2} i+1]\!]. \end{aligned}$$

Let the variable order be given by $x_3 \prec x_2 \prec x_1$. CutSat with a two-layered strategy has diverging runs starting in state $S'_0 = \langle [\![]\!], C \rangle$. Let CutSat traverse the states S'_0, S_0, S_1, S_2, ... in the following fashion: CutSat reaches state S_0 from state S'_0 after propagating the constraints I_1, I_2, I_3, and I_4. CutSat reaches state S_{i+1} from state S_i after:

- fixing x_1 to i with the decided bound $\gamma_d^{x_1} := x_1 \leq i$ so that $M_i^1 := [\![M_i, \gamma_d^{x_1}]\!]$ and $S_i^1 := \langle M_i^1, C \rangle$
- applying Conflict to the constraint J_1 because $\mathcal{L}(1 - x_1 + x_2, M_i^1) > 0$ so that $M_i^2 := M_i^1$ and $S_i^2 := \langle M_i^2, C, J_1 \rangle$
- undoing the decided bound $\gamma_d^{x_1}$ by applying Backjump; this results in the exchange of $\gamma_d^{x_1}$ with the bound $\gamma^{x_1} = x_1 \geq_{J_1} i+1$ so that $M_i^3 := [\![M_i, \gamma^{x_1}]\!]$ and $S_i^3 := \langle M_i^3, C \rangle$
- fixing x_2 to i with the decided bound $\gamma_d^{x_2} := x_2 \leq i$ so that $M_i^4 := [\![M_i^3, \gamma_d^{x_2}]\!]$ and $S_i^4 := \langle M_i^4, C \rangle$
- applying Conflict to the constraint J_2 because $\mathcal{L}(x_1 - x_2, M_i^4) > 0$ so that $M_i^5 := M_i^4$ and $S_i^5 := \langle M_i^5, C, J_1 \rangle$
- undoing the decided bound $\gamma_d^{x_2}$ by applying Backjump; this results in the exchange of $\gamma_d^{x_2}$ with the bound $\gamma^{x_2} = x_2 \geq_{J_2} i+1$ so that $M_i^6 := [\![M_i^3, \gamma^{x_2}]\!]$ and $S_{i+1} = S_i^6 := \langle M_i^6, C \rangle$

Notice that $(x_3, \{I_3, I_5\})$ is a conflicting core in the states S_i, S_i^1, and S_i^4, and, therefore, the variable x_3 is the minimal conflicting variable in those states. Since I_3 and I_4 bound x_3, the conflicting core is also guarded. Therefore, Resolve-Cooper as defined by [87] is not applicable, which in turn implies that Conflict is applicable. This means the two-layered strategy was not strict enough to prevent standard conflict resolution from being applied to unguarded conflicts.

A straightforward fix to Example 3.3.5 is to limit the application of the Conflict and Conflict-Div rules to guarded constraints. Our second example shows that CutSat can still diverge by infinitely many applications of the Solve-Div rule.

Example 3.3.6. Let d_i be the sequence with $d_0 := 2$ and $d_{k+1} := {d_k}^2$ for $k \in \mathbb{N}$, let $C_0 = \{4 \mid 2x_1+2x_2, 2 \mid x_1+x_3\}$ be a problem, and let $S_0 = \langle [\![]\!], C_0 \rangle$ be the start CutSat state. Let the variable order be given by $x_1 \prec x_2 \prec x_3$. Then CutSat has divergent runs $S_0 \Longrightarrow_{\text{CS}} S_1 \Longrightarrow_{\text{CS}} S_2 \Longrightarrow_{\text{CS}} \ldots$. For instance, let CutSat apply the Solve-Div rule whenever applicable. By an inductive argument, Solve-Div is applicable in every state $S_n = \langle [\![]\!], C_n \rangle$, and the constraint set C_n has the following form:

$$C_n = \begin{cases} \{2d_n \mid d_n x_1 + d_n x_2, d_n \mid \frac{d_n}{2} x_2 - \frac{d_n}{2} x_3\} & \text{if } n \text{ is odd,} \\ \{2d_n \mid d_n x_1 + d_n x_2, d_n \mid \frac{d_n}{2} x_1 + \frac{d_n}{2} x_3\} & \text{if } n \text{ is even.} \end{cases}$$

Therefore, CutSat applies Solve-Div infinitely often and diverges. If we were to simplify the constraints in C_n, then we would even see that the constraints C_n for an even n are always just $\{4 \mid 2x_1 + 2x_2, 2 \mid x_1 + x_3\}$ and the constraints C_n for an odd n are always just $\{4 \mid 2x_1 + 2x_2, 2 \mid x_2 - x_3\}$. This means that we are cycling between the same two sets of constraints.

A straightforward fix to Example 3.3.6 is to limit the application of Solve-Div to maximal variables in the variable order $\prec$. Our third example shows that CutSat can apply Conflict and Conflict-Div infinitely often. Example 3.3.7 differs from Example 3.3.5 in that the conflicting core contains also unguarded variables.

Example 3.3.7. Let

$$C := \{ \underbrace{-x_1 \leq 0}_{I_1}, \underbrace{-x_2 \leq 0}_{I_2}, \underbrace{-x_3 \leq 0}_{I_3}, \underbrace{x_3 \leq 0}_{I_4}, \underbrace{1 - x_1 + x_2 + x_3 \leq 0}_{J_1}, \underbrace{x_1 - x_2 - x_3 \leq 0}_{J_2} \}$$

be a problem. Let $S_i = \langle M_i, C \rangle$ for $i \in \mathbb{N}$ be a series of states with:

$$\begin{aligned} M_0 &:= [\![x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0]\!], \\ M_{i+1} &:= [\![M_i, x_1 \geq_{J_1} i+1, x_2 \geq_{J_2} i+1]\!]. \end{aligned}$$

Let the variable order be given by $x_3 \prec x_1 \prec x_2$. CutSat with a two-layered strategy has diverging runs starting in state $S'_0 = \langle [\![]\!], C \rangle$. For instance, let CutSat traverse the states S'_0, S_0, S_1, S_2, ... in the following fashion: CutSat reaches state S_0 from state S'_0 after propagating the constraints I_1, I_2, I_3 and I_4. CutSat reaches state S_{i+1} from state S_i after:

- fixing x_1 to i and x_2 to i with decided bounds $\gamma_d^{x_1} := x_1 \leq i$ and $\gamma_d^{x_2} := x_2 \leq i$ so that $M_i^1 := [\![M_i, \gamma_d^{x_1}, \gamma_d^{x_2}]\!]$ and $S_i^1 := \langle M_i^1, C \rangle$
- applying Conflict to J_1 because $\mathcal{L}(1 - x_1 + x_2 + x_3, M_i^1) > 0$ so that $M_i^2 := M_i^1$ and $S_i^2 := \langle M_i^2, C, J_1 \rangle$
- undoing the decided bounds $\gamma_d^{x_2}$ and $\gamma_d^{x_1}$ by applying Skip-Decision and then Backjump; the result is the sequence $M_i^3 := [\![M_i, \gamma^{x_1}]\!]$ and the state $S_i^3 := \langle M_i^3, C \rangle$, where $\gamma^{x_1} = x_1 \geq_{J_1} i+1$
- fixing x_2 to i and x_1 to $i+1$ with decided bounds $\gamma_d^{x_2} := x_2 \leq i$ and $\gamma_d^{x_1} := x_2 \leq i+1$ so that $M_i^4 := [\![M_i^3, \gamma_d^{x_2}, \gamma_d^{x_1}]\!]$ and $S_i^4 := \langle M_i^4, C \rangle$
- applying Conflict to J_2 because $\mathcal{L}(x_1 - x_2 - x_3, M_i^4) > 0$ so that $M_i^5 := M_i^4$ and $S_i^5 := \langle M_i^5, C, J_1 \rangle$

- undoing the decided bounds $\gamma_d^{x_1}$ and $\gamma_d^{x_2}$ by applying Skip-Decision and then Backjump; the result is the sequence $M_i^6 := [\![M_i^3, \gamma^{x_2}]\!]$ and the state $S_{i+1} = S_i^6 := \langle M_i^6, C\rangle$, where $\gamma^{x_2} = x_2 \geq_{J_2} i+1$.

Whenever we fix x_1 and x_2 with Decide to their current respective lower bounds, we turn $(x_3, \{J_1, J_2\})$ into a conflicting core. As a result, x_3 is the minimal conflicting variable in the states S_i^1 and S_i^4, which in turn prevents the application of Resolve-Cooper and admits the application of Conflict. Therefore, the two-layered strategy was again not strict enough to prevent standard conflict resolution from being applied to unguarded conflicts.

Applying the fix suggested for Example 3.3.5 to Example 3.3.7 results in a stuck state. Here, a straightforward fix is to change the definition of conflicting cores to cover only those cores where the conflicting variable is the maximal variable.[9]

We could also prevent Examples 3.3.5 and 3.3.7 if we were to fix the decision order to the variable order. However, for the following counterexample, fixing the decision order is not enough:

Example 3.3.8. Let

$$\begin{aligned} C := \{\ & \underbrace{-x_3 \leq 0}_{I_1}, \underbrace{x_3 \leq 0}_{I_2}, \underbrace{-x_4 \leq 0}_{I_3}, \underbrace{x_4 \leq 0}_{I_4}, \underbrace{-x_5 \leq 0}_{I_5}, \\ & \underbrace{-x_5 + x_1 + x_3 \leq 0}_{I_6}, \underbrace{-x_5 - x_1 - x_3 \leq 0}_{I_7}, \\ & \underbrace{-x_5 + x_2 + x_4 \leq 0}_{I_8}, \underbrace{-x_5 - x_2 - x_4 \leq 0}_{I_9}, \\ & \underbrace{2 + 3x_1 - 4x_2 + 3x_3 - 4x_4 \leq 0}_{J_1}, \underbrace{-1 + 3x_1 - 4x_2 + 3x_3 - 4x_4 \leq 0}_{J_2}, \\ & \underbrace{-1 - 3x_1 + 2x_2 - 3x_3 + 2x_4 \leq 0}_{J_3}\} \end{aligned}$$

be a problem. Let the variable order be given by $x_3 \prec x_4 \prec x_5 \prec x_1 \prec x_2$. CutSat has a divergent run starting in state $S_0' = \langle [\![], C\rangle$ if it follows the following strategy: Let x_j be the smallest unfixed variable.

- Propagate (if possible) the strictest upper bound for x_j,
- if J_1 is a conflict, then fix all unfixed variables with the rule Decide to their current upper bound, and start the standard conflict analysis with the rule Conflict,
- otherwise propagate the strictest lower bound for x_j and fix x_j with a decision to the current lower bound

[9] The restrictions to maximal variables in the definition of conflicting cores and to the Solve-Div rule were both confirmed as missing but necessary in a private communication with Jovanović.

Notice that we prevent the application of Resolve-Cooper by first fixing all variables and then shadowing x_1 and x_2 with the guarded variables x_3 and x_4, respectively. If we change the definition of conflicting cores, as proposed for Example 3.3.7, then the divergent behaviour described in Example 3.3.8 is not possible. Thus, fixing the decision order to the variable order is no real alternative to the fixes proposed before.

The fixes that we suggested for the above examples are restrictions to CutSat which have the consequence that Conflict(-Div) cannot be applied to unguarded constraints, Solve-Div is applicable only for the elimination of the maximal variable, and the conflicting variable x_1 is the maximal variable in the associated conflicting core C'. However, our next and final example shows that these restrictions are too strong and therefore lead to stuck states.

Example 3.3.9. Let CutSat include restrictions to maximal variables in the definition of conflicting cores and in the Solve-Div rule as described above. Let there be additional restrictions in CutSat to the rules Conflict and Conflict-Div such that these rules are only applicable to conflicts that contain no unguarded variable. Let

$$C := \{\underbrace{-x_1 \leq 0}_{I_1}, \underbrace{x - 1 \leq 0}_{I_2}, \underbrace{-x_2 \leq 0}_{I_3}, \underbrace{6 \mid 4x_2 + x_1}_{J}\}$$

be a problem. Let $M := [\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1, x_2 \geq_{I_3} 0, x_1 \geq 1, x_2 \leq 0]\!]$ be a bound sequence. Let the variable order be given by $x_1 \prec x_2$. CutSat has a run starting in state $S'_0 = \langle [\![]\!], C\rangle$ that ends in the stuck state $S = \langle M, C\rangle$. Let CutSat propagate I_1, I_2, I_3 and fix x_1 to 1 and x_2 to 0 with two Decisions. Through these Decisions, the constraint J is a conflict. Since x_2 is unguarded, CutSat cannot apply the rule Conflict-Div. Furthermore, Definition 3.3.3 mentions only interval or divisibility conflicting cores and the state S contains neither. Therefore, CutSat cannot apply the rule Resolve-Cooper. The remaining rules are also not applicable because all variables are fixed and there is only one divisibility constraint. Without the before introduced restriction to the rule Conflict(-Div), CutSat diverges on the example.

3.4 Weak Cooper Elimination

In order to fix the stuck state of Example 3.3.9 in the previous section, we are going to introduce in Section 3.5 a new conflicting core, which we call *diophantine conflicting core*. For understanding diophantine conflicting cores, as well as further modifications to be made, it is helpful to understand the connection between CutSat++ and a variant of Cooper's quantifier elimination procedure [44].

The original *Cooper elimination* takes a variable x_j, a problem C, and produces a disjunction of problems equivalent to $\exists x_j \in \mathbb{Z}.C$:

$$\exists x_j \in \mathbb{Z}.C \equiv \bigvee_{0 \le y_i < m} C_{-\infty}\{x_j \mapsto y_i\} \vee \bigvee_{-a_j x_j + p_j \le 0 \in C} \bigvee_{0 \le y_i < a_j \cdot m} \left[\{a_j \mid p_j + y_i\} \cup C\{x_j \mapsto \tfrac{p_j + y_i}{a_j}\}\right],$$

where $a > 0$, $m = \operatorname{lcm}\{d \in \mathbb{Z} : (d \mid a_j x_j + p_j) \in C\}$, $C_{-\infty} = \bot$ if there exists a constraint of the form $-a_j x_j + p_j \le 0 \in C$, and, otherwise, $C_{-\infty} = \{(d \mid a_j x_j + p_j) \in C\}$. Although Cooper elimination gets rid of the variable x_j, the elimination also comes at a price. One application of Cooper elimination results in a disjunction of quadratically many problems out of a single problem. Moreover, we have to somehow "remove" the fractions $\frac{p_j + y_i}{a_j}$ used to replace x_j because division is not part of the linear arithmetic language. We achieve this by multiplying each constraint with a_j. However, this "removal" of the fractions causes a worst-case quadratic increase in the absolute size of the coefficients. The number of disjunctions and the size of coefficients have even a worst-case exponential increase if we look at several iterations of Cooper elimination.

Now that we have formally defined Cooper elimination and stated its structural properties, we will explain in an intuitive way why Cooper elimination actually works. To this end, let us look at the result of applying Cooper elimination to a small example:

Example 3.4.1. Let

$$C' := \{ \underbrace{2 \mid x_1 + x_2}_{J_1}, \underbrace{2 \mid x_1 + x_3}_{J_2}, \underbrace{-3x_1 + x_2 \le 0}_{J_3}, \underbrace{-2x_1 + 2x_2 - 2 \le 0}_{J_4}, \underbrace{x_1 - 4x_3 \le 0}_{J_5}\}$$

be the initial problem. Then the result of applying Cooper elimination is:

$$[\exists x_1 \in \mathbb{Z}.C'] \equiv \left(\bigvee_{0 \le y_1 < 1} C_{-\infty}\right) \vee \left(\bigvee_{0 \le y_1 < 6} \{3 \mid y_1 + x_2\} \cup C_{J_3}\right) \vee \left(\bigvee_{0 \le y_1 < 4} \{2 \mid y_1 + 2x_2 - 2\} \cup C_{J_4}\right),$$

where

$$\begin{aligned}
C_{-\infty} &:= \bot \\
C_{J_3} := C\{x_1 \mapsto \tfrac{x_2 + y_1}{3}\} &:= \{\, 6 \mid y_1 + 4x_2, 6 \mid y_1 + x_2 + 3x_3, -y_1 \le 0, \\
&\qquad -2y_1 + 4x_2 - 6 \le 0, y_1 + x_2 - 12x_3 \le 0\}, \\
C_{J_4} := C\{x_1 \mapsto \tfrac{2x_2 - 2 + y_1}{2}\} &:= \{\, 4 \mid y_1 + 4x_2 - 2, 4 \mid y_1 + 2x_2 + 2x_3 - 2, \\
&\qquad -3y_1 - 4x_2 + 6 \le 0, -y_1 \le 0, \\
&\qquad y_1 + 2x_2 - 8x_3 - 2 \le 0\}.
\end{aligned}$$

A linear arithmetic problem has an integer solution if and only if the strictest lower bound for x_j is an integer solution for the problem. In Cooper elimination, we use this fact and do a case distinction over the strictest lower bound for x_j via a disjunction. First, we assume that there exists no lower bound

for x_j. We express this case with the subformula $C_{-\infty}$. Since both J_3 and J_4 define a lower bound for x_1 ($x_1 \geq \lceil \frac{x_2}{3} \rceil$ and $x_1 \geq \lceil \frac{2x_2-2}{2} \rceil$, respectively), $C_{-\infty}$ simplifies to $\bot$. Next, we assume that J_3 is the basis for the strictest lower bound $x_1 \geq l_1$, i.e., $l_1 := l'_1 + y_1$ (with $y_1 \geq 0$) is the smallest integer larger than or equal to $l'_1 := \lceil \frac{x_2}{3} \rceil$ such that $2 \mid l_1 + x_2$ and $2 \mid l_1 + x_3$ are true. We also know that $l'_1 + 2$ is a strict upper bound for l_1 because

$$\begin{aligned}
2 \mid l'_1 + 2y_1 + x_2 &\equiv 2 \mid l'_1 + x_2,\\
2 \mid l'_1 + 2y_1 + 1 + x_2 &\equiv 2 \mid l'_1 + 1 + x_2,\\
2 \mid l'_1 + 2y_1 + x_3 &\equiv 2 \mid l'_1 + x_3,\\
2 \mid l'_1 + 2y_1 + 1 + x_3 &\equiv 2 \mid l'_1 + 1 + x_3.
\end{aligned}$$

Therefore, l_1 is either l'_1 or $l'_1 + 1$. Since the ceiling function is not part of our syntax, we cannot assign x_1 directly to $\lceil \frac{x_2}{3} \rceil$ or $\lceil \frac{x_2}{3} \rceil + 1$. However, we know that $\lceil \frac{x_2}{3} \rceil$ is one of the three values $\frac{x_2}{3}$, $\frac{x_2+1}{3}$, or $\frac{x_2+2}{3}$ and that $\lceil \frac{x_2}{3} \rceil + 1$ is one of the three values $\frac{x_2+3}{3}$, $\frac{x_2+4}{3}$, or $\frac{x_2+5}{3}$. Hence, we do another case distinction (via a disjunction and the factor y_1) and replace x_1 with $\frac{x_2+y_1}{3}$. The result is the subformula $\bigvee_{0 \leq y_1 < 6} \{3 \mid y_1 + x_2\} \cup C_{J_3}$. Finally, we assume that J_4 is the basis of the strictest lower bound $x_1 \geq \lceil \frac{2x_2-2}{3} \rceil$. Again, we do a case distinction (via a disjunction and the factor y_1) and replace x_1 with $\frac{2x_2+y_1-2}{2}$. The result is the subformula $\bigvee_{0 \leq y_1 < 4} \{2 \mid y_1 + 2x_2 - 2\} \cup C_{J_4}$.

Our notion of weak Cooper elimination is a variant of Cooper elimination, which is very helpful for understanding problems around CutSat. The idea is, instead of building a disjunction over all potential solutions for x_j, to add additional guarded variables and constraints without x_j that guarantee the existence of a solution for x_j. We assume here that C contains exactly one divisibility constraint for x_j. If there exists no divisibility constraint for x_j, then we can simply add the trivially true divisibility constraint $1 \mid x_j$. If there exist multiple divisibility constraint for x_j, then we use the following function to transform every set of constraints C' into an equivalent set of constraints C that contains only one divisibility constraint for x_j:

$$\begin{aligned}
&\text{div-solve}(x_j, \{d_1 \mid a_{1j}x_j + p_{1j}, d_2 \mid a_{2j}x_j + p_{2j}\}) =\\
&\{d_1 d_2 \mid dx_j + c_{1j}d_{2j}p_{1j} + c_{2j}d_{1j}p_{2j}, d \mid -a_{1j}p_{2j} + a_{2j}p_{1j}\},
\end{aligned}$$

where $d = \gcd(a_{1j}d_2, a_{2j}d_1)$, and c_{1j} and c_{2j} are integer values such that $c_{1j}a_{1j}d_2 + c_{2j}a_{2j}d_1 = d$ [44, 87]. The function div-solve$(x_j, \{D_1, D_2\})$ takes a set of two divisibility constraints $\{D_1, D_2\}$ containing x_j and returns an equivalent set of two divisibility constraints $\{D'_1, D'_2\}$, where only one constraint contains x_j [44, 87]. Therefore, we can replace every pair of divisibility constraints $\{D_1, D_2\}$ containing x_j with the pair returned from div-solve$(x_j, \{D_1, D_2\})$ to decrease the number of divisibility constraints containing x_j by one. It follows that exhaustive application of div-solve to all divisibility constraints containing x_j removes all such constraints except one.

Now *weak Cooper elimination* takes a variable x_j, a problem C with only one divisibility constraint $d \mid c_j x_j + s_j \in C$ for x_j, and produces a new and equivalent problem by replacing $\exists x_j \in \mathbb{Z}.C$ with:

$$\exists Y. \left(\{I \in C : \text{coeff}(I, x_j) = 0\} \cup \{\gcd(c_j, d) \mid s_j\} \cup \bigcup_{y_i \in Y} R_{y_i} \right)$$

where $y_i \in Y$ is a newly introduced variable for every pair of constraints $-a_j x_j + p_j \leq 0 \in C$ and $b_j x_j - q_j \leq 0 \in C$ with $a_j, b_j > 0$,

$$R_{y_i} = \{ -y_i \leq 0, y_i - m \leq 0, b_j p_j - a_j q_j + b_j y_i \leq 0, \\ a_j \mid y_i + p_j,\ a_j d \mid c_j p_j + a_j s_j + c_j y_i \}$$

is a *resolvent* for the same inequalities, where $m := \text{lcm}\left(a_j, \frac{a_j d}{\gcd(a_j d, c_j)}\right) - 1$, and $\exists Y$ abbreviates the sequence of quantified variables $\exists y_1, \ldots, y_m \in \mathbb{Z}$ contained in $Y = \{y_1, \ldots, y_m\}$. The major difference between Cooper elimination and weak Cooper elimination is that one introduces disjunctions and the other variables. For our purposes variables are more beneficial because each new variable y_i is guarded by the constraints in R_{y_i}. Hence, we can eliminate unguarded variables without changing the conjunctive structure of our problems.

Weak Cooper elimination works informally because the transformation determines whether there exists an assignment ν for all variables except x_j such that the strictest lower and upper bounds for x_j under ν still contain an integer solution. For a more detailed explanation, let ν be a satisfiable assignment for the formula after one weak Cooper elimination step on C. Then we compute a strictest lower bound $x_j \geq l_j$ and a strictest upper bound $x_j \leq u_j$ from C for the variable x_j under the assignment ν. We now argue that there is a value for x_j such that $x_j \geq l_j$, $x_j \leq u_j$, and $d \mid c_j x_j + s_j$ are all satisfied. Whenever $l_j \neq -\infty$ and $u_j \neq \infty$, the bounds $x_j \geq l_j$, $x_j \leq u_j$ are given by constraints of the form $-a_j x_j + p_j \leq 0 \in C$ and $b_j x_j - q_j \leq 0 \in C$, respectively, so that $l_j = \lceil \frac{\nu(p_j)}{a_j} \rceil$ and $u_j = \lfloor \frac{\nu(q_j)}{b_j} \rfloor$. In this case, the extension of ν with $\nu(x_j) = \frac{\nu(y_i + p_j)}{a_j}$ satisfies C because the constraint $a_j \mid y_i + p_j \in R_{y_i}$ guarantees that $\nu(x_j) \in \mathbb{Z}$, the constraint $b_j p_j - a_j q_j + b_j y_i \leq 0 \in R_{y_i}$ guarantees that $l_j \leq \nu(x_j) \leq u_j$, and the constraint $a_j d \mid c_j p_j + a_j s_j + c_j y_i \in R_{y_i}$ guarantees that ν also satisfies $d \mid c_j x_j + s_j \in C$. In all other cases, i.e., when $l_j = -\infty$ or $u_j = \infty$, we extend ν by an arbitrary small or alternatively large value for x_j that satisfies $d \mid c_j x_j + s_j \in C$. There exist arbitrarily small (large) solutions for x_j and $d \mid c_j x_j + \nu(s_j)$ because $\gcd(c_j, d) \mid s_j$ is satisfied by ν.

Now let us look at the result of applying Cooper elimination to the problem C' from Example 3.4.1:

Example 3.4.2. Let

$$C' := \{ \underbrace{2 \mid x_1 + x_2}_{J_1}, \underbrace{2 \mid x_1 + x_3}_{J_2}, \underbrace{-3x_1 + x_2 \leq 0}_{J_3}, \underbrace{-2x_1 + 2x_2 - 2 \leq 0}_{J_4}, \underbrace{x_1 - 4x_3 \leq 0}_{J_5} \}$$

be the initial problem. We first have to use div-solve to simplify C' because C' contains two divisibility constraints (J_1 and J_2) for x_1. Applying div-solve$(x_1, \{J_1, J_2\})$ returns the two divisibility constraints $4 \mid 2x_1 + 2x_2$ and $2 \mid x_2 - x_3$, which we will use to replace J_1 and J_2 in C'. The result is the new, but equivalent problem

$$C' := \{ \underbrace{4 \mid 2x_1 + 2x_2}_{J_1'}, \underbrace{2 \mid x_2 - x_3}_{J_2'}, \underbrace{-3x_1 + x_2 \leq 0}_{J_3}, \underbrace{-2x_1 + 2x_2 - 2 \leq 0}_{J_4}, \underbrace{x_1 - 4x_3 \leq 0}_{J_5} \},$$

which contains only one divisibility constraint (J_1') for x_1. Now we will split the application of weak Cooper elimination into three steps. First, we select all constraints $I \in C$ without the variable x_1:

$$\{I \in C : \text{coeff}(I, x_1) = 0\} := \{2 \mid x_2 - x_3\}.$$

Next, we take the divisibility constraint $4 \mid 2x_1 + 2x_2$ and eliminate x_1 from it :

$$(\exists x_1 \in \mathbb{Z}.\ 4 \mid 2x_1 + 2x_2) \equiv (\gcd(2, 4) \mid 2x_2) \equiv (2 \mid 2x_2).$$

Finally, we construct resolvents for every pair $-a_j x_j + p_j \leq 0 \in C$ and $b_j x_j - q_j \leq 0 \in C$:

$R_{y_1} := \{-y_1 \leq 0, y_1 - 5 \leq 0, y_1 + x_2 - 12x_3 \leq 0, 3 \mid y_1 + x_2, 12 \mid 2y_1 + 8x_2\}$ for J_3 and J_5, and

$$R_{y_2} := \{ -y_2 \leq 0, y_2 - 3 \leq 0, y_2 + 2x_2 - 8x_3 - 2 \leq 0, 2 \mid y_2 + 2x_2 - 2, \\ 8 \mid 2y_2 + 8x_2 - 4\}$$

for J_4 and J_5. The combination of these constraints is then the result of applying weak Cooper elimination:

$$[\exists x_1 \in \mathbb{Z}.C'] \equiv [\exists x_1 \in \mathbb{Z}.C] \equiv \\ [\exists y_1 \in \mathbb{Z}.\exists y_2 \in \mathbb{Z}.\ (\{J_2'\} \cup \{2 \mid 2x_2\} \cup R_{y_1} \cup R_{y_2})]\,.$$

CutSat++ performs weak Cooper elimination not in one step but subsequently adds to the states the constraints from the resolvents R_{y_i} as well as the divisibility constraint $\gcd(c_j, d) \mid s_j$ with respect to a strict ordering on the unguarded variables. The extra divisibility constraint $\gcd(c_j, d) \mid s_j$ in weak Cooper elimination is necessary whenever the problem C has no constraint of the form $-a_j x_j + p_j \leq 0 \in C$ or $b_j x_j - q_j \leq 0 \in C$. For example:

Example 3.4.3. Let $C = \{x_2 - 1 \leq 0, -x_2 + 1 \leq 0, 6 \mid 2x_1 + x_2\}$ be a problem and x_1 be the unguarded variable we want to eliminate. As there are no inequalities containing x_1, weak Cooper elimination without the extra divisibility constraint returns $C' = \{x_2 - 1 \leq 0, -x_2 + 1 \leq 0\}$. While C' has a satisfiable assignment $\nu(x_2) = 1$, C has not since $2x_1 + 1$ is never divisible by 6.

The following equivalence states the correctness of weak Cooper elimination:

$$\exists x_j \in \mathbb{Z}.C \equiv \exists Y. \left(\{I \in C : \text{coeff}(I, x_j) = 0\} \cup \{\text{gcd}(c_j, d) \mid s_j\} \cup \bigcup_{y_i \in Y} R_{y_i} \right)$$

For any R_{y_i} introduced by weak Cooper elimination, we can also show the following Lemma:

Lemma 3.4.4 (Divisibility Core Resolvent)**.** *Let y_i be a new variable. Let $a_j, b_j, c_j > 0$. Then*

$$\begin{aligned}&(\exists x_j \in \mathbb{Z}.\{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0, d \mid c_j x_j + s_j\})\\ \equiv\ &(\exists y_i \in \mathbb{Z}.\{-y_i \leq 0, y_i - m \leq 0, b_j p_j - a_j q_j + b_j y_i \leq 0,\\ &\qquad a_j \mid y_i + p_j,\ a_j d \mid c_j p_j + a_j s_j + c_j y_i\}).\end{aligned}$$

Proof. See [87] pp. 101-102 Lemma 4. □

That means satisfiability of the respective R_{y_i} guarantees a solution for the triple of constraints it is derived from. An analogous Lemma holds for the divisibility constraint $gcd(c_j, d) \mid s_j$ introduced by weak Cooper elimination:

Lemma 3.4.5 (Diophantine Core Resolvent)**.** *Let $c_j > 0$. Then*

$$(\exists x_j \in \mathbb{Z}.d \mid c_j x_j + s_j) \equiv \text{gcd}(c_j, d) \mid s_j.$$

Proof. We equivalently rewrite the divisibility constraints into diophantine equations, viz., $d \mid c_j x_j + s_j$ and $gcd(c_j, d) \mid s_j$ into $\exists z_i \in \mathbb{Z}.dz_i - c_j x_j = s_j$ and $\exists y_i \in \mathbb{Z}.\,\text{gcd}(c_j, d) y_i = s_j$, respectively. We choose $d' \in \mathbb{Z}$ and $c'_j \in \mathbb{Z}$ such that $d' \cdot \text{gcd}(c_j, d) = d$ and $c'_j \cdot \text{gcd}(c_j, d) = c_j$, respectively. Next, we differentiate between the two directions of the equivalence. Firstly, let ν be a variable assignment such that $d\nu(z_i) - c_j\nu(x_j) = \nu(s_j)$ and, therefore, also $d \mid c_j\nu(x_j) + \nu(s_j)$. Hence,

$$\nu(s_j) = d\nu(z_i) - c_j\nu(x_j) = \text{gcd}(c_j, d) \cdot (d'_j\nu(z_i) - c'_j\nu(x_j)).$$

Therefore, extending ν with $\nu(y_i) = (d'\nu(z_i) - c'_j\nu(x_j))$ leads to an assignment ν that also satisfies $\text{gcd}(c_j, d) y_i = s_j$.

Secondly, let ν be a variable assignment such that $\text{gcd}(c_j, d)\nu(y_i) = \nu(s_j)$ holds and, therefore, also $\text{gcd}(c_j, d) \mid \nu(s_j)$. By Bézout's Lemma [19], there exist $a'_j, b'_j \in \mathbb{Z}$ such that $a'_j d - b'_j c_j = \text{gcd}(c_j, d)$. Hence,

$$a'_j d\nu(y_i) - b'_j c_j \nu(y_i) = (a'_j d - b'_j c_j)\nu(y_i) = \text{gcd}(c_j, d)\nu(y_i) = \nu(s_j).$$

Therefore, extending ν with $\nu(z_i) = a'_j\nu(y_i)$ and $\nu(x_j) = b'_j\nu(y_i)$ leads to an assignment ν that also satisfies $dz_i - c_j x_j = s_j$. □

Hence, satisfiability of $\gcd(c_j, d) \mid s_j$ guarantees a solution for the divisibility constraint $d \mid c_j x_j + s_j$. The rule Resolve-Weak-Cooper (Figure 3.9) in our CutSat++ calculus exploits these properties by lazily generating the resolvents R_{y_i} and the constraint $\gcd(c_j, d) \mid s_j$ in the form of *unguarded resolvents*. Furthermore, it is not necessary for the divisibility constraints to be a priori reduced to one, as done for weak Cooper elimination. Instead, the rules Solve-Div-Left and Solve-Div-Right (Figure 3.9) perform lazy reduction.

Now let us return to the completeness proof of weak Cooper elimination. In this proof, we will use an assignment ν for all variables but x_j. Since such an assignment ν fixes all variables but x_j, we can determine the satisfiability of a problem C by looking at the extensions of ν for x_j. To this end, we will define the *solution set* $S \subseteq \mathbb{Z}$ for variable x_j, problem C, and assignment ν as the set of values $v_k \in S$ that extends ν to a satisfiable solution for C, i.e., $C\{x_j \mapsto v_k\}$ $(v_k \in S)$ is satisfied by ν. If we look, for instance, at the solution set S_d of a divisibility constraint, then we find the following useful property:

Lemma 3.4.6 (Divisibility Solution Sets)**.** *Let ν be an assignment for all variables except x_j. Let S_d be the solution set for variable x_j, assignment ν, and constraint $d \mid c_j x_j + s_j$. Then*

$$S_d = \emptyset \text{ or } S_d = \{v_0 + kv' : k \in \mathbb{Z}\} \text{ for some } v_0 \in \mathbb{Z}, v' \in \mathbb{Z} \setminus \{0\}.$$

This means that S_d is either empty or unbounded from above and below.

Proof. If $S_d \neq \emptyset$, then there exists a value $v_0 \in S_d$ such that $d \mid c_j v_0 + \nu(s_j)$. We first prove that there exists a $v' \in \mathbb{Z} \setminus \{0\}$ such that $d \mid c_j(v_0 + kv') + \nu(s_j)$ for all $k \in \mathbb{Z}$ and, therefore, $v_0 + kv' \in S_d$. We choose v' and k' such that $v' := \frac{d}{\gcd(c_j, d)}$ and $k' := \frac{c_j}{\gcd(c_j, d)}$. Then we deduce for any $k \in \mathbb{Z}$:

$$d \mid c_j(v_0 + kv') + \nu(s_j) \equiv d \mid c_j v_0 + \nu(s_j) + c_j kv' \equiv$$
$$d \mid c_j v_0 + \nu(s_j) + c_j k \tfrac{d}{\gcd(c_j, d)} \equiv d \mid c_j v_0 + \nu(s_j) + dkk' \equiv d \mid c_j v_0 + \nu(s_j)$$

It remains to show that for every $v_k \in S_d$ there exists a $k \in \mathbb{Z}$ such that $v_0 + kv' = v_k$. As S_d is the solution set, we know that both $d \mid c_j v_0 + \nu(s_j)$ and $d \mid c_j v_k + \nu(s_j)$ are true. Therefore, $d \mid c_j v_0 + \nu(s_j) - (c_j v_k + \nu(s_j)) \equiv d \mid c_j(v_0 - v_k)$. As $d = v' \gcd(c_j, d)$, the term $c_j(v_0 - v_k)$ is only divisible by d if $v_0 - v_k$ is divisible by v'. Therefore, $\exists k \in \mathbb{Z}. v_0 - v_k = kv'$. □

We need Lemma 3.4.6 in the correctness proof of weak Cooper elimination to show that we can choose an arbitrary small or large solution for x_j that satisfies $d \mid c_j x_j + \nu(s_j)$. As mentioned in the outline of the proof, the ability to choose arbitrary small and large solutions for x_j is necessary when C contains no constraints of the form $-a_j x_j + p_j \leq 0$ or $b_j x_j - q_j \leq 0$.

Theorem 3.4.7 (Weak Coooper Correctness)**.** *Let C be a problem with exactly one divisibility constraint I_d with a non-zero coefficient* $\mathrm{coeff}(I_d, x_j)$ *for x_j. Then*

$$\exists x_j \in \mathbb{Z}.C \equiv \exists Y. \left(\{I \in C : \mathrm{coeff}(I, x_j) = 0\} \cup \{\gcd(c_j, d) \mid s_j\} \cup \bigcup_{y_i \in Y} R_{y_i} \right)$$

where $y_i \in Y$ is a newly introduced variable for every pair of constraints $-a_j x_j + p_j \leq 0 \in C$ and $b_j x_j - q_j \leq 0 \in C$ with $a_j, b_j > 0$,

$$R_{y_i} = \{ -y_i \leq 0, y_i - m \leq 0, b_j p_j - a_j q_j + b_j y_i \leq 0, \\ a_j \mid y_i + p_j,\ a_j d \mid c_j p_j + a_j s_j + c_j y_i \}$$

is a resolvent *for the same inequalities, where $m := \mathrm{lcm}\left(a_j, \frac{a_j d}{\gcd(a_j d, c_j)}\right) - 1$, and $\exists Y$ abbreviates the sequence of quantified variables $\exists y_1, \ldots, y_m \in \mathbb{Z}$ contained in $Y = \{y_1, \ldots, y_m\}$.*

Proof. First, we partition the problem C as follows:

$C_l = \{-a_j x_j + p_j \leq 0 \in C : a_j > 0\}$, $C_u = \{b_j x_j - q_j \leq 0 \in C : b_j > 0\}$,
$I_d = d \mid c_j x_j + s_j \in C$, $C_r = \{I \in C : \mathrm{coeff}(I, x_j) = 0\}$.

By Lemma 3.4.4, it holds for all $-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0 \in C$ with $a_j, b_j > 0$ that:

$$(\exists x_j \in \mathbb{Z}.C) \to (\exists x_j \in \mathbb{Z}.\{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0, d \mid c_j x_j + s_j\}) \\ \to (\exists y_i \in \mathbb{Z}.R_{y_i}),$$

where

$$R_{y_i} = \{ -y_i \leq 0, y_i - m \leq 0, b_j p_j - a_j q_j + b_j y_i \leq 0, \\ a_j \mid y_i + p_j,\ a_j d \mid c_j p_j + a_j s_j + c_j y_i \}.$$

Because of Lemma 3.4.5, it also holds that:

$$(\exists x_j \in \mathbb{Z}.C) \to (\exists x_j \in \mathbb{Z}.d \mid c_j x_j + s_j) \to \gcd(c_j, d) \mid s_j.$$

Since $C_r \subseteq C$, it also holds that: $(\exists x_j \in \mathbb{Z}.C) \to C_r$. As all new variables $y_i \in Y$ appear only in one resolvent R_{y_i}, the above implications prove

$$\exists y_i \in \mathbb{Z}.C \to \exists Y. \left(\{I \in C : \mathrm{coeff}(I, x_j) = 0\} \cup \{\gcd(c_j, d) \mid s_j\} \cup \bigcup_{y_i \in Y} R_{y_i} \right).$$

Assume, vice versa, that ν is a satisfiable assignment for the formula after one step of weak Cooper elimination. Then it is easy to deduce the following facts:

- Let S_l be the solution set for x_j, ν, and $I_l = -a_j x_j + p_j \leq 0 \in C$ with $a_j > 0$. Then $S_l = \left\{ \lceil \frac{\nu(p_j)}{a_j} \rceil, \lceil \frac{\nu(p_j)}{a_j} \rceil + 1, \ldots \right\}$.
- Let S_u be the solution set for x_j, ν, and $I_u = b_j x_j - q_j \leq 0 \in C$ with $b_j > 0$. Then $S_u = \left\{ \ldots, \lfloor \frac{\nu(q_j)}{b_j} \rfloor - 1, \lfloor \frac{\nu(q_j)}{b_j} \rfloor \right\}$.
- Let S_I be the solution set for x_j, ν, and $C_l \cup C_u$. Then $S_I = \bigcap_{I_l \in C_l} S_l \cap \bigcap_{I_u \in C_u} S_u$.
- Let the set S_I be bounded from below, i.e., $S_I = \{l, l+1, \ldots\}$ or $S_I = \{l, \ldots, u\}$. Then $l = \max_{I \in C_l} \left\{ \lceil \frac{\nu(p'_j)}{a'_j} \rceil : I = -a'_j x_j + p'_j \leq 0 \right\}$.

- Let the set S_I be bounded from above, i.e., $S_I = \{\ldots, u-1, u\}$ or $S_I = \{l, \ldots, u\}$. Then $u = \min_{I \in C_u} \left\{ \lfloor \frac{\nu(q'_j)}{b'_j} \rfloor : I = b'_j x_j - q'_j \leq 0 \right\}$.
- By Lemma 3.4.5, $d \mid c_j x_j + \nu(s_j)$ is satisfiable because $\gcd(c_j, d) \mid s_j$ is contained in the result formula of weak Cooper elimination. By Lemma 3.4.6, the set of solutions for x_j, ν, and $d \mid c_j x_j + s_j$ has the form $S_d = \{v_0 + kv' : k \in \mathbb{Z}\}$.
- The solution set S for x_j, ν, and C' is $S = S_d \cap S_I$.

Next, we do a case distinction on the structure of C:

- Let $C_l = \emptyset$, then S_I is unbounded from below. We choose a small enough $v_k \in S_d$, i.e., small enough $k \in \mathbb{Z}$ such that $v_k = v_0 + kv'$. Then the assignment $x_j \mapsto v_k$ and $x_i \mapsto \nu(x_i)$ (if $x_i \neq x_j$) satisfies C'.
- Let $C_u = \emptyset$, then S_I is unbounded from above. We choose a large enough $v_k \in S_d$, i.e., large enough $k \in \mathbb{Z}$ such that $v_k = v_0 + kv'$. Then the assignment $x_j \mapsto v_k$ and $x_i \mapsto \nu(x_i)$ for all $x_i \neq x_j$ satisfies C'.
- Let $|C_l|, |C_u| > 0$. We select $I_l = -a_j x_j + p_j \leq 0$ such that
$$\lceil \tfrac{\nu(p_j)}{a_j} \rceil = \max_{I \in C_l} \left\{ \lceil \tfrac{\nu(p'_j)}{a'_j} \rceil : I = -a'_j x_j + p'_j \leq 0 \right\}$$
and $I_u = b_j x_j - q_j \leq 0$ such that
$$\lfloor \tfrac{\nu(q_j)}{b_j} \rfloor = \min_{I \in C_u} \left\{ \lfloor \tfrac{\nu(q'_j)}{b'_j} \rfloor : I = b'_j x_j - q'_j \leq 0 \right\}.$$
The resolvent for the two constraints I_l and I_u is
$$R_{y_i} = \{ \quad -y_i \leq 0, y_i - m \leq 0, b_j p_j - a_j q_j + b_j y_i \leq 0, \\ a_j \mid y_I + p_j,\ a_j d \mid c_j p_j + a_j s_j + c_j y_i \}.$$
We will now extend ν to also include $\nu(x_j) = \frac{\nu(p_j + y_i)}{a_j}$ and show that $\frac{\nu(p_j+y_i)}{a_j}$ is in the set of solutions S of C. All of the remaining deductions stem from the evaluation of the resolvent under ν. Since $a_j \mid \nu(p_j + y_i)$, $\frac{\nu(p_j+y_i)}{a_j} \in \mathbb{Z}$. Moreover, since $\frac{\nu(p_j+y_i)}{a_j} \in \mathbb{Z}$ and $\nu(b_j p_j - a_j q_j + b_j y_i) \leq 0$, $\frac{\nu(p_j+y_i)}{a_j} \in S_I = \left\{ \lceil \frac{\nu(p_j)}{a_j} \rceil, \ldots, \lfloor \frac{\nu(q_j)}{b_j} \rfloor \right\}$. Finally, since
$$a_j d \mid \nu(c_j p_j + a_j s_j + c_j y_i) = a_j d \mid a_j c_j \nu(x_j) + a_j \nu(s_j) = \\ d \mid c_j \nu(x_j) + \nu(s_j),$$
it follows that $\frac{\nu(p_j+y_i)}{a_j} \in S_d$. Hence, ν' satisfies C'.

□

We stated that weak Cooper elimination can only be applied to those problems where C contains one divisibility constraint $d \mid a_j x_j + p_j$ in x_j. To expand weak Cooper elimination to any set of constraints C', we briefly explained how to exhaustively apply div-solve to eliminate all but one constraint $d \mid a_j x_j + p_j$ for x_j. The algorithm CombDivs(x_j, C') (Figure 3.8) is a more detailed version of this procedure.

Lemma 3.4.8 (CombDivs Equivalence)**.** *Let C' be a set of LIA constraints. Let C be the set of LIA constraints we receive from* CombDivs(x_j, C')*. Then $C \equiv C'$.*

Algorithm 10: CombDivs(x_j, C')

Input : A variable x_j and a set of LIA constraints C'
Output: A set of LIA constraints C such that $C \equiv C'$ and there exists exactly one divisibility constraint $d \mid c_j x_j + s_j \in C$ such that $c_j \neq 0$

```
1  C_d := {d | c_j x_j + s_j ∈ C' : c_j > 0}
2  C := C' \ C_d ;
3  if (C_d = ∅) then
4  |  return C ∪ {1 | x_j} ;
5  while (|C_d| > 1) do
6  |  Select d_1 | a_1j x_j + p_1j, d_2 | a_2j x_j + p_2j ∈ C_d ;
7  |  C_d := C_d \ {d_1 | a_1j x_j + p_1j, d_2 | a_2j x_j + p_2j};
8  |  d = gcd(a_1j d_2j, a_2j d_1j) ;
9  |  Choose c_1j and c_2j such that c_1j a_1j d_2j + c_2j a_2j d_1j = d;
10 |  C_d := C_d ∪ {d_1 d_2 | d x_j + c_1j d_2 p_1j + c_2j d_1 p_2j} ;
11 |  C := C ∪ {d | −a_1j p_2j + a_2j p_1j};
12 end
13 return C ∪ C_d ;
```

Figure 3.8: An algorithm that combines all divisibility constraints containing x_j until only one such constraint remains

Proof. Follows directly from the proof of equivalence of the div-solve transformation [87]. □

The relationship between CutSat++ and weak Cooper elimination is as follows: On the left side of the equivalence in Theorem 3.4.7 there occur two combinations of constraints: triples consisting of two inequations with a divisibility constraint and single divisibility constraints. Each triple of constraints $\{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0, d \mid c_j x_j + s_j\} \in C$ is a potential divisibility conflicting core[10], i.e., a set of constraints that can turn into a conflicting core if p_j, q_j, s_j are fixed and the constraints are contradictory for variable x_j. Moreover, the single divisibility constraint $\{d \mid c_j x_j + s_j\} \in C$ is a potential diophantine conflicting core, i.e., a divisibility constraint that can turn into a conflicting core if it becomes contradictory for variable x_j after s_j is fixed. On the right side of the equivalence in Theorem 3.4.7, the resolvents R_{y_i} for the respective triple of constraints and the resolvent $\gcd(c_j, d) \mid s_j$ for the divisibility constraint are equivalent to the unguarded resolvent that CutSat++ introduces for the appropriate conflicting cores. However, compared to CutSat++, weak Cooper elimination introduces all

[10] Or a potential interval conflicting core if the divisibility constraint $d \mid c_j x_j + s_j$ is not part of the contradiction.

resolvents at once and, thereby, ensures that the resolvents subsume all potential conflicting cores. Therefore, we can replace all constraints containing x_j with the resolvents for conflicting cores over x_j and receive an equisatisfiable formula (see Theorem 3.4.7). This means that Theorem 3.4.7 shows that no other conflicting cores are necessary besides interval, divisibility, and diophantine conflicting cores. Moreover, Example 3.4.3 shows that we need to consider diophantine conflicting cores—and not just interval and divisibility conflicting cores—to preserve equisatisfiability.

The preprocessing step described in the CombDivs(x_j, C') algorithm can also be found in CutSat and CutSat++. However, CutSat and CutSat++ perform CombDivs(x_j, C') lazily; the former with the rule Solve-Div, the latter with the rules Solve-Div-Left and Solve-Div-Right.

3.5 Unguarded Conflict Resolution

Weak Cooper elimination is capable of creating an equisatisfiable problem where satisfiability depends only on guarded variables. We simulate it in a lazy manner by extending CutSat$_g$ with three new rules, which we call the *unguarded conflict resolution* rules (Figure 3.9). The now extended rule system is our calculus CutSat++. Instead of eliminating all unguarded variables before the application of CutSat++, the rules perform the same intermediate steps as weak Cooper elimination, viz., the combination of divisibility constraints via div-solve and the construction of resolvents, to resolve and block conflicts in unguarded constraints. As a result, CutSat++ can avoid some of the intermediate steps of weak Cooper elimination. Moreover, CutSat++ is not required to apply the intermediate steps of weak Cooper elimination one variable at a time and does not eliminate unguarded variables. This has the advantage that CutSat++ might find a satisfiable assignment or detect unsatisfiability without encountering and resolving a large number of unguarded conflicts. As a result, the number of divisibility constraint combinations and introduced resolvents might be much smaller in the lazy approach of CutSat++ than during the elimination with weak Cooper elimination. Only in the worst case, CutSat++ has to perform all of weak Cooper elimination's intermediate steps. Then the strictly-two-layered strategy (Definition 3.5.7) guarantees that CutSat++ recognizes that all unguarded conflicts have been produced.

In order to simulate weak Cooper elimination, CutSat++ uses a total order $\prec$ over all variables such that $x_k \prec x_j$ for all guarded variables x_k and unguarded variables x_j.[11] In relation to weak Cooper elimination, the order $\prec$ describes the elimination order for the unguarded variables, viz., $x_j \succ x_i$ if x_j is eliminated before x_i. This also means that the maximal

[11] While termination requires that the order is fixed from the beginning for all unguarded variables, the ordering among the guarded variables can be dynamically changed.

Resolve-Weak-Cooper

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M', C'\rangle \quad \textbf{if} \begin{cases} (x_j, C') \text{ is a conflicting core,} \\ x_j \text{ is unguarded,} \\ \text{all } x_k \prec x_j \text{ are fixed and } C' \subseteq C, \\ \text{if } J \in C \text{ is a conflict, then } \operatorname{top}(J) \not\prec x_j, \\ \text{w-cooper}(x_j, C') = (R_y, R_c), \\ x_i = \min_{I \in R_c}\{\operatorname{top}(I)\}, \\ M' = \operatorname{prefix}(M, x_i), \\ C' = C \cup R_y \cup R_c \end{cases}$$

Solve-Div-Left

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M, C'\rangle \quad \textbf{if} \begin{cases} \text{divisibility constraints } I_1, I_2 \in C, \\ x_j \text{ is unguarded and top in } I_1 \text{ and } I_2, \\ \text{all other variables in } I_1, I_2 \text{ are fixed,} \\ (I_1', I_2') = \text{div-solve}(x_j, I_1, I_2), \\ C' = (C \setminus \{I_1, I_2\}) \cup \{I_1', I_2'\}, \\ I_2' \text{ is not a conflict} \end{cases}$$

Solve-Div-Right

$$\langle M, C\rangle \Longrightarrow_{\text{CS}} \langle M', C'\rangle \quad \textbf{if} \begin{cases} \text{divisibility constraints } I_1, I_2 \in C, \\ x_j \text{ is unguarded and top in } I_1 \text{ and } I_2, \\ \text{all other variables in } I_1, I_2 \text{ are fixed,} \\ \{I_1', I_2'\} = \text{div-solve}(x_j, \{I_1, I_2\}), \\ C' = (C \setminus \{I_1, I_2\}) \cup \{I_1', I_2'\}, \\ I_2' \text{ is a conflict,} \\ x_i = \operatorname{top}(I_2'), \\ M' = \operatorname{prefix}(M, x_i) \end{cases}$$

In the above rules, $M' = \operatorname{prefix}(M, x_i)$ defines the largest prefix of M that contains only decided bounds for variables x_k with $x_k \prec x_i$.

Figure 3.9: Our unguarded conflict resolution rules

unguarded variable in a given subset of constraints C is the next variable that is eliminated from C by weak Cooper elimination. For this reason, unguarded conflict resolution also always targets the maximal unguarded variable in a given conflict set. Formally, we call a variable x_j *maximal* in a constraint I if x_j is contained in I, i.e., $\operatorname{coeff}(I, x_j) \neq 0$, and all other variables contained in I are smaller, i.e., $x_k \prec x_j$. Analogously, a variable x_j is called *maximal* in a constraint set C if x_j is contained in C, i.e., $\operatorname{coeff}(I, x_j) \neq 0$ for an $I \in C$, and all other variables contained in C are smaller, i.e., $x_k \prec x_j$. We also call the maximal variable in a constraint I (or a constraint set C') its *top variable* and denote it by $x_j = \operatorname{top}(I)$ ($x_j = \operatorname{top}(C')$).

In contrast to weak Cooper elimination, unguarded conflict resolution does not eliminate any variables but adds *(unguarded) resolvents* to the constraint set so that the satisfiability depends only on smaller variables, which essentially simulates the elimination of a variable. Moreover, unguarded conflict resolution does not add resolvents for all constraints in our problem, but only to so-called *conflicting cores*, i.e., a compact representation of the constraints that are responsible for a given conflict.

Definition 3.5.1 (Conflicting Cores)**.** Let $S = \langle M, C\rangle$ and $C' \subseteq C$. Let $x_j = \text{top}(C')$ be unguarded, let all other variables in C' be fixed, let $a, b > 0$, and let $l_j = \text{bound}(-a_j x_j + p_j \leq 0, x_j, \geq, M)$ and $u_j = \text{bound}(b_j x_j - q_j \leq 0, x_j, \leq, M)$. The pair (x, C') is a *conflicting core* if it is of one of the following three forms:
(1) $C' = \{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0\}$ and $l_j > u_j$, i.e., the lower bound from $-a_j x_j + p_j \leq 0$ contradicts the upper bound from $b_j x_j - q_j \leq 0$; in this case, (x_j, C') is called an *interval conflicting core* and its *unguarded resolvent* is $(\{-y_i \leq 0, y_i - a_j + 1 \leq 0\}, \{b_j p_j - a_j q_j + b_j y_i \leq 0, a_j \mid y_i + p_j\})$
(2) $C' = \{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0, d \mid c_j x_j + s_j\}$ and $l_j \leq u_j$, and for all $v_j \in [l_j, u_j] \cap \mathbb{Z}$ we have $d \nmid c_j v_j + \mathcal{L}(s, M)$, i.e., there exists no value for x_j within the bounds defined by the two inequalities such that the divisibility constraint becomes satisfiable; in this case, (x_j, C') is called a *divisibility conflicting core* and its *unguarded resolvent* is $(\{-y_i \leq 0, y_i - m \leq 0\},$
$\{b_j p_j - a_j q_j + b_j y_i \leq 0, a_j \mid y_i + p_j, a_j d \mid c_j p_j + a_j s_j + c_j y_i\})$
(3) $C' = \{d \mid c_j x_j + s_j\}$ and for all $v_j \in \mathbb{Z}$ we have $d \nmid cv_j + \mathcal{L}(s_j, M)$, i.e., there exists no integer value for x_j that could satisfy $d \mid c_j x_j + s_j$; in this case (x_j, C') is called a *diophantine conflicting core* and its *unguarded resolvent* is $(\emptyset, \{gcd(c_j, d) \mid s_j\})$.
In the first two cases, y_i is a fresh variable and $m = \text{lcm}\left(a_j, \frac{a_j d}{\text{gcd}(a_j d, c_j)}\right) - 1$.

We refer to the respective unguarded resolvents for a conflicting core (x_j, C') by the function w-cooper(x_j, C'), which returns a pair (R_y, R_c) as defined above. In relation to weak Cooper elimination, $R_y \cup R_c$ is the (potentially simplified) result of elimination x_j from C' with weak Cooper elimination. Note also that the newly introduced variable y_i in each resolvent is guarded by the constraints in R_y, which means y_i must be placed in the order $\prec$ so it is smaller than all unguarded variables x_k. Therefore, the resolvent $R_y \cup R_c$ implies the satisfiability of $\exists x_j \in \mathbb{Z}.C'$ although it contains only smaller variables than x_j.

Out of convenience, we are also introducing some new terminology with regard to conflicting cores: Firstly, a variable x_j is called a *conflicting variable* if there is a conflicting core (x_j, C') in some state S. Secondly, a pair (x_j, C') is a *potential conflicting core* if there exists a state S where (x_j, C') is a conflicting core.

Next, we define a generalization of unguarded resolvents. We do so because the unguarded resolvents generated out of conflicting cores will be further processed by CutSat++ and we must guarantee that the processed unguarded resolvents still imply the satisfiability of the conflicting core constraints. Moreover, this generalization guarantees that CutSat++ introduces only one unguarded resolvent for every conflicting core.

Definition 3.5.2 (Unguarded Resolvents)**.** A set of constraints R is an *unguarded resolvent* for (x_j, C') if $R \to \exists x_j \in \mathbb{Z}.C'$ and $\forall J \in R.\, \mathrm{top}(J) \prec x_j$, i.e., the satisfiability of the unguarded resolvent R implies the satisfiability of C', while using only variables that are smaller than x_j.

All unguarded resolvents of Definition 3.5.1 are also unguarded resolvents in the sense of the below definition (see also end of Section 3.4).

Lemma 3.5.3 (Unguarded Resolvent Soundness)**.** *Let C' be a subset of C ($C' \subseteq C$). Let* w-cooper$(x_j, C') = (R_y, R_c)$. *Let $R = R_y \cup R_c$. Then $\exists y_i \in \mathbb{Z}.C \cup R \equiv C$. Furthermore, R is an unguarded resolvent for (x_j, C').*

Proof. The interval conflicting core is the only new case; however, the two Lemmas 3.4.4 and 3.4.5 are still enough to prove the soundness of all three cases because w-cooper$(x_j, \{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0\})$ is equivalent to w-cooper$(x_j, \{-a_j x_j + p_j \leq 0, b_j x_j - q_j \leq 0, 1 \mid x_j\})$ (after simplifications). Therefore, $R \to \exists x_j \in \mathbb{Z}.C'$ due to Lemmas 3.4.4 and 3.4.5. Finally, $\mathrm{top}(J) \prec x_j$ holds for all $J \in R$ because y_i is guarded and therefore smaller than x_j and all other variables in R appear in C' where x_j is maximal. □

Let us now take a closer look at the rules that make up unguarded conflict resolution and their many restrictions/conditions. The first rule is Resolve-Weak-Cooper and it (i) adds an unguarded resolvent (R_y, R_c) for a given conflicting core (x_j, C') and (ii) backtracks to a prefix of the model in which (x_j, C') is no longer a conflicting core. We restrict the rule Resolve-Weak-Cooper (Figure 3.9) to unguarded constraints because we could otherwise generate infinitely many guarded variables. This poses no problem for efficiency because the standard conflict resolution is already very efficient on guarded constraints. Moreover, Resolve-Weak-Cooper requires that the conflicting variable x_j of the conflicting core (x, C') is the top variable in the constraints of C'. This simulates a setting where all variables x_i with $x_j \prec x_i$ are already eliminated. The restrictions also prevent Resolve-Weak-Cooper from being applied to the conflicting core (x_j, C') whenever our set of constraints already contains an unguarded resolvent R for this core. Since Resolve-Weak-Cooper adds an unguarded resolvent for every conflicting core it is applied to, this also means that Resolve-Weak-Cooper is never applied twice to the same conflicting core.

Lemma 3.5.4 (Blocking Resolve-Weak-Cooper)**.** *Let $S = \langle M, C\rangle$ be a* CutSat++ *state. Let $C' \subseteq C$ and x_1 be an unguarded variable. Let $R \subseteq C$ be an* unguarded resolvent *for (x_j, C'). Then Resolve-Weak-Cooper is not applicable to (x_j, C').*

Proof. Assume for contradiction that $D = (x_j, C')$ is a conflicting core, $R \in C$ is an unguarded resolvent for D in state S, and Resolve-Weak-Cooper is applicable to D in state S. Resolve-Weak-Cooper requires that all variables $x_i \prec x_j$ are fixed (Figure 3.9). This holds especially for all variables in R (Definition 3.5.2). Due to the restriction in Resolve-Weak-Cooper that every conflict $J \in C$ has $\operatorname{top}(J) \not\prec \operatorname{top}(I)$, there is no conflict in R. Furthermore, since all variables $x_i \prec x_j$ are fixed, R is satisfied by the partial assignment defined by M. By Definition 3.5.1, all conflicting cores have no satisfiable solution for x_j under partial model M. However, by Definition 3.5.2, R satisfiable implies that there exists a value for x_j such that C' is satisfiable under M. This contradicts the assumption that (x_j, C') is a conflicting core! □

If we add unguarded resolvents again and again, then CutSat++ will reach a state after which every encounter of a conflicting core guarantees a conflict in a guarded constraint. From this point forward, CutSat++ will not apply Resolve-Weak-Cooper. The remaining guarded conflicts are resolved with the rules Conflict and Conflict-Div. However, this works only because of two other facets of unguarded conflict resolution: the eager top-level propagating strategy and the two rules Solve-Div-Left and Solve-Div-Right. Both facets are needed so CutSat++ can compact any unguarded conflict into a conflicting core.

The rules Solve-Div-Left and Solve-Div-Right (Figure 3.9) combine divisibility constraints with the function div-solve$(x_j, \{I_1, I_2\})$ as done *a priori* to weak Cooper elimination. We need these rules because our conflicting cores contain at most one divisibility constraint at a time. This could be problematic (without Solve-Div-Left and Solve-Div-Right) because there are unguarded conflicts that are only conflicting because of multiple divisibility constraints:

Example 3.5.5. Let

$$C := \{\underbrace{x_1 \leq 1}_{I_1}, \underbrace{2 \mid x_2 + x_1}_{J_1}, \underbrace{2 \mid x_2}_{J_2}\}$$

be a problem, let the variable order be given by $x_1 \prec x_2$, and let our partial model be $[\![x_1 \leq_{I_1} 1, x_1 \geq 1]\!]$. Then J_1 and J_2 are alone satisfiable but together they are a conflict for x_2 because x_2 can never be even $(2 \mid x_2)$ and odd $(2 \mid x_2 + 1)$ at the same time. Moreover, we cannot resolve this problem with Resolve-Weak-Cooper because there are currently no conflicting cores.

However, if we apply div-solve$(x_2, \{J_1, J_2\}) = \{J_1', J_2'\}$ to combine J_1 and J_2 and replace $\{J_1, J_2\}$ with $\{J_1', J_2'\}$, then our constraint set transforms to:

$$C' := \{\underbrace{x_1 \leq 1}_{I_1}, \underbrace{4 \mid 2x_2 + 2x_1}_{J_1'}, \underbrace{2 \mid x_1}_{J_2'}\},$$

in which J_2' is a divisibility constraint that is a conflict on its own.

In fact, CutSat++ can always combine conflicts over multiple divisibility constraints using the rules Solve-Div-Left and Solve-Div-Right. We split this combination into two rules because div-solve$(x_j, \{I_1, I_2\}) = \{I_1', I_2'\}$ might move a conflict over the variable x_j into a conflict over a smaller variable x_i, i.e., $x_i \prec x_j$. In this case, we prefer to backtrack (with the rule Solve-Div-Right) to a prefix of our model where we can use the newly combined constraints $\{I_1', I_2'\}$ to prevent the construction of our conflicting model.

The other restrictions/conditions in Solve-Div-Left and Solve-Div-Right match restrictions of Resolve-Weak-Cooper. We again restrict the application of div-solve$(x_j, \{I_1, I_2\})$ to constraints where x_j is the top variable and where all variables x_i in I_1 and I_2 with $x_i \neq x_j$ are fixed. As before, this ordering restriction simulates the order of elimination, i.e., we apply div-solve$(x_j, \{I_1, I_2\})$ in a (simulated) setting where all variables x_i with $x_j \prec x_i$ appear to be eliminated in I_1 and I_2. Otherwise, divergence is possible (see Example 3.3.6). Requiring smaller variables x_k to be fixed prevents the accidental generation of a conflict for an unguarded variable x_k by div-solve$(x_j, \{I_1, I_2\})$.

As mentioned before, Solve-Div-Left and Solve-Div-Right alone do not yet guarantee that CutSat++ is able to compact all unguarded constraints into conflicting cores. We also need an *eager top-level propagating strategy*, as defined below:

Definition 3.5.6 (Eager Top-Level Propagating Strategy)**.** We call a strategy for CutSat++ *eager top-level propagating* if we restrict propagations and decisions for every state $\langle M, C\rangle$ in the following way:

1. Let x_j be an unguarded variable. Let $\bowtie \in \{\leq, \geq\}$. Then we only allow to propagate bounds $x_j \bowtie \text{bound}(I, x_j, \bowtie, M)$ if x_j is the top variable in I. Moreover, if I is a divisibility constraint $d \mid a_j x_j + p_j$, then we only propagate $d \mid a_j x_j + p_j$ if:
   (a) $\mathcal{L}(x_j, M) \neq -\infty$ and $\mathcal{U}(x_j, M) \neq \infty$ or
   (b) $\gcd(a_j, d) \mid \mathcal{L}(p_j, M)$ holds and $d \mid a_j x_j + p_j$ is the only divisibility constraint in C with x_j as top variable.
2. Let x_j be an unguarded variable. Let $\bowtie \in \{\leq, \geq\}$. Then we only allow decisions $\gamma = x_j \bowtie b_j$ if:
   (a) for every constraint $I \in C$ with $x_j = \text{top}(I)$ all occurring variables $x_i \neq x_j$ are fixed
   (b) there exists no $I \in C$ where $x_j = \text{top}(I)$ and I is a conflict in $[\![M, \gamma]\!]$

(c) either $\mathcal{L}(x_j, M) \neq -\infty$ and $\mathcal{U}(x_j, M) \neq \infty$ or there exists at most one divisibility constraint in C with x_j as top variable.

An eager top-level propagating strategy has two advantages. First, the strategy dictates an order of influence over the variables, i.e., a bound for unguarded variable x_j is influenced only by previously propagated bounds for variables x_i with $x_i \prec x_j$. This in itself already prevents divergence due to bound propagation. Moreover, the strategy makes decisions for unguarded variable x_j only when all constraints with $x_j = \text{top}(I)$ are fixed and satisfied by the decision. This means, any conflict $I \in C$ with $x_j = \text{top}(I)$ is impossible as long as the decision for x_j remains on the bound sequence. We need this restriction because Resolve-Weak-Cooper can only resolve conflicts between constraints (see Definition 3.5.1) and, therefore, cannot resolve conflicts based on top variable decisions. For the same purpose, i.e., avoiding conflicts I where $x_j = \text{top}(I)$ is fixed by a decision, CutSat++ backjumps in the rules Resolve-Weak-Cooper and Solve-Div-Right to a state where this is not the case. To avoid stuck states resulting from the eager top-level propagating strategy, the slack variable x_S has to be the smallest unguarded variable in $\prec$. Otherwise, the constraints $x_j - x_S \leq 0$, $-x_j - x_S \leq 0$ introduced by Slack-Intro cannot be used to propagate bounds for variable x_j and x_j would remain stuck.

Definition 3.5.7 (Strictly-Two-Layered Strategy)**.** A strategy is *strictly-two-layered* if:
(1) it is reasonable (Definition 3.2.2), (2) it is eager top-level propagating, (3) the Forget, Conflict, Conflict-Div rules only apply to guarded constraints, (4) Forget cannot be applied to a divisibility constraint or a constraint contained in an unguarded resolvent, and (5) only guarded constraints are used to propagate guarded variables.

The above *strictly-two-layered* strategy is the final restriction to CutSat++. With the condition 3.5.7-(3) it partitions conflict resolution into two layers: While every unguarded conflict is handled with the rules Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right (Figure 3.9), every guarded conflict is handled with the rules Conflict(-Div). The conditions 3.5.7-(1) and 3.5.7-(5) make the guarded variables independent from the unguarded variables. The condition 3.5.7-(2) prevents bound propagation and guarantees that all unguarded conflicts can be compacted into conflicting cores. The condition 3.5.7-(4) guarantees that no unguarded resolvent is ever removed (just transformed) and that all conflicting cores stay blocked after one application of Resolve-Weak-Cooper. We assume for the remainder of the paper that all runs of CutSat++ follow a strictly-two-layered strategy.

To better show how CutSat++ works, let us look again at the examples from Section 3.2. In contrast to CutSat, CutSat++ terminates with a correct solution on these examples:

$$
\begin{aligned}
&\langle [\![]\!], C\rangle \\
\Longrightarrow_{\text{CS}}^{\text{Propagate}} \ &\langle [\![x_1 \geq_{I_1} 0]\!], C\rangle \\
\Longrightarrow_{\text{CS}}^{\text{Propagate}} \ &\langle [\![x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0]\!], C\rangle \\
\Longrightarrow_{\text{CS}}^{\text{Propagate}} \ &\langle [\![x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, x_3 \leq_{I_3} 0]\!], C\rangle \\
\Longrightarrow_{\text{CS}}^{\text{Propagate}} \ &\langle [\![x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, x_3 \leq_{I_3} 0, x_3 \geq_{I_4} 0]\!], C\rangle \\
\Longrightarrow_{\text{CS}}^{\text{Conflict}} \ &\langle [\![x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, x_3 \leq_{I_3} 0, x_3 \geq_{I_4} 0]\!], C, x_3 + 1 \leq 0\rangle \\
\Longrightarrow_{\text{CS}}^{\text{Resolve}} \ &\langle [\![x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, x_3 \leq_{I_3} 0]\!], C, 1 \leq 0\rangle \\
\Longrightarrow_{\text{CS}}^{\text{Unsat}} \ &\mathit{unsat}
\end{aligned}
$$

Figure 3.10: A CutSat++ run for Example 3.5.8 depicted as a transition of states

Example 3.5.8. Let

$$
\begin{aligned}
C := \{ \ & \underbrace{-x_1 \leq 0}_{I_1}, \underbrace{-x_2 \leq 0}_{I_2}, \underbrace{-x_3 \leq 0}_{I_3}, \underbrace{x_3 \leq 0}_{I_4}, \underbrace{x_3 + 1 \leq 0}_{I_5}, \\
& \underbrace{1 - x_1 + x_2 \leq 0}_{J_1}, \underbrace{x_1 - x_2 \leq 0}_{J_2}\}
\end{aligned}
$$

be a problem. Let the variable order be given by $x_3 \prec x_2 \prec x_1$. Then CutSat++ would solve the problem as depicted in Figure 3.10. First of all, CutSat++ has to propagate I_1, I_2, I_3, and I_4, i.e., all constraints containing only one variable. However, this already turns I_5 into a conflicting constraint. Although we could add decided bounds and propagated bounds until $(x_1, \{J_1, J_2\})$ turns into a conflicting core, we could never apply Resolve-Weak-Cooper to $(x_1, \{J_1, J_2\})$ because $\text{top}(I_5) \prec x_1$ and I_5 is conflicting. Therefore, CutSat++ has to apply at some point Conflict to I_5. After applying Resolve to $x_3 + 1 \leq 0$ with the bound $x_3 \geq_{I_4} 0$, CutSat++ has derived the trivially unsatisfiable constraint $1 \leq 0$. Finally, CutSat++ uses $1 \leq 0$ to apply Unsat and, thereby, return *unsat*.

Example 3.5.9. Let $C_0 = \{4 \mid 2x_1 + 2x_2, 2 \mid x_1 + x_3\}$ be a problem. Let the variable order be given by $x_1 \prec x_2 \prec x_3$. Then CutSat++ would solve the problem as depicted in Figure 3.11. Since C_0 has no constraints containing only one variable, CutSat++ cannot propagate any bounds and all variables are stuck in state S_0. To resolve the stuck state, CutSat++ applies Slack-Intro to x_1. Now, CutSat++ is able to propagate the bound $x_S \geq_{I_{x_S}} 0$ for the slack variable x_S and even fix it with the decided bound $x_S \leq 0$. Since x_S is fixed, CutSat++ propagates the bounds $x_1 \leq_{I_1} 0$ and $x_1 \geq_{I_2} 0$. However, the variables x_2 and x_3 are again stuck. We resolve this by repeating what we did for x_1, first to the variable x_2 and then to the

$$
\begin{aligned}
&\langle [\![]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Slack-Intro}} \langle [\![]\!], C_1\rangle \\
&\qquad \text{where } C_1 := C_0 \cup \{\underbrace{-x_S \le 0}_{I_{x_S}}, \underbrace{x_1 - x_S \le 0}_{I_1}, \underbrace{-x_1 - x_S \le 0}_{I_2}\} \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} \langle [\![x_S \ge_{I_{x_S}} 0]\!], C_1\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Decide}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0]\!], C_1\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0]\!], C_1\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0, x_1 \ge_{I_2} 0]\!], C_1\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Slack-Intro}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0, x_1 \ge_{I_2} 0]\!], C_2\rangle \\
&\qquad \text{where } C_2 := C_1 \cup \{\underbrace{x_2 - x_S \le 0}_{I_3}, \underbrace{-x_2 - x_S \le 0}_{I_4}\} \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0, x_1 \ge_{I_2} 0, x_2 \le_{I_3} 0]\!], C_2\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0, x_1 \ge_{I_2} 0, x_2 \le_{I_3} 0, x_2 \ge_{I_4} 0]\!], C_2\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Slack-Intro}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0, x_1 \ge_{I_2} 0, x_2 \le_{I_3} 0, x_2 \ge_{I_4} 0]\!], C_3\rangle \\
&\qquad \text{where } C_3 := C_2 \cup \{\underbrace{x_3 - x_S \le 0}_{I_5}, \underbrace{-x_3 - x_S \le 0}_{I_6}\} \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0, x_1 \ge_{I_2} 0, x_2 \le_{I_3} 0, x_2 \ge_{I_4} 0, \\
&\qquad x_3 \le_{I_5} 0]\!], C_3\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} \langle [\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0, x_1 \ge_{I_2} 0, x_2 \le_{I_3} 0, x_2 \ge_{I_4} 0, \\
&\qquad x_3 \le_{I_5} 0, x_3 \ge_{I_6} 0]\!], C_3\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Sat}} \langle \nu[[\![x_S \ge_{I_{x_S}} 0, x_S \le 0, x_1 \le_{I_1} 0, x_1 \ge_{I_2} 0, x_2 \le_{I_3} 0, x_2 \ge_{I_4} 0, \\
&\qquad x_3 \le_{I_5} 0, x_3 \ge_{I_6} 0]\!]], C_3\rangle
\end{aligned}
$$

Figure 3.11: A CutSat++ run for Example 3.5.9 depicted as a transition of states

variable x_3. Finally, CutSat++ returns the satisfiable assignment with an application of the Sat rule. Note that CutSat++ could never apply the rules Solve-Div-Left or Solve-Div-Right because the top variables of the divisibility constraints are different.

Example 3.5.10. Let

$$
\begin{aligned}
C := \{ \quad & \underbrace{-x_1 \le 0}_{I_1}, \underbrace{-x_2 \le 0}_{I_2}, \underbrace{-x_3 \le 0}_{I_3}, \underbrace{x_3 \le 0}_{I_4}, \\
& \underbrace{1 - x_1 + x_2 + x_3 \le 0}_{J_1}, \underbrace{x_1 - x_2 - x_3 \le 0}_{J_2}\}
\end{aligned}
$$

be a problem. Let (R_y, R_c) be the output of w-cooper$(y_1, \{J_1, J_2\})$ so that

$$
(R_y, R_c) := (\{\underbrace{-y_1 \le 0}_{K_1}, \underbrace{y_1 \le 0}_{K_2}\}), \{\underbrace{y_1 + 1 \le 0}_{K_3}, \underbrace{1 \mid y_1 + x_2 + x_3 + 1}_{K_4}\}).
$$

Let the variable order be given by $x_3 \prec x_1 \prec x_2$. Then CutSat++ would solve the problem as depicted in Figure 3.12. First of all, CutSat++ has

$\langle [\![]\!], C_0 \rangle$

$\Longrightarrow^{\text{Propagate}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0]\!], C_0 \rangle$

$\Longrightarrow^{\text{Propagate}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0]\!], C_0 \rangle$

$\Longrightarrow^{\text{Propagate}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0, x_1 \geq_{I_1} 0]\!], C_0 \rangle$

$\Longrightarrow^{\text{Propagate}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0, x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0]\!], C_0 \rangle$

$\Longrightarrow^{\text{Decide}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0, x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, x_1 \leq 0]\!], C_0 \rangle$

$\Longrightarrow^{\text{Resolve-Weak-Cooper}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0, x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0]\!], C_1 \rangle$

where the new order is $x_3 \prec y_1 \prec x_1 \prec x_2$, and $C_1 := C_0 \cup R_y \cup R_c$

$\Longrightarrow^{\text{Propagate}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0, x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, y_1 \leq_{K_2} 0]\!], C_1 \rangle$

$\Longrightarrow^{\text{Propagate}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0, x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, y_1 \leq_{K_2} 0, y_1 \geq_{K_1} 0]\!], C_1 \rangle$

$\Longrightarrow^{\text{Conflict}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0, x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, y_1 \leq_{K_2} 0, y_1 \geq_{K_1} 0]\!], C_1, y_1 + 1 \leq 0 \rangle$

$\Longrightarrow^{\text{Resolve}}_{\text{CS}}$ $\langle [\![x_3 \geq_{I_3} 0, x_3 \leq_{I_4} 0, x_1 \geq_{I_1} 0, x_2 \geq_{I_2} 0, y_1 \leq_{K_2} 0]\!], C_1, 1 \leq 0 \rangle$

$\Longrightarrow^{\text{Unsat}}_{\text{CS}}$ *unsat*

Figure 3.12: A CutSat++ run for Example 3.5.10

to propagate I_3, I_4, I_1, and I_2, i.e., all constraints containing only one variable. Then, CutSat++ fixes x_1 with a decided bound $x_1 \leq 0$, which turns $(x_2, \{J_1, J_2\})$ into a conflicting core. Since only J_1 is a conflicting constraint and $x_2 = \text{top}(J_1)$, the rule Resolve-Weak-Cooper is applicable and CutSat++ will use it to resolve the conflicting core $(x_2, \{J_1, J_2\})$. The resulting resolvent is (R_y, R_c) for which CutSat++ introduces the fresh variable y_1 and updates the variable order to $x_3 \prec y_1 \prec x_1 \prec x_2$. Resolve-Weak-Cooper also truncates the bound sequence to remove all decided bounds for variables greater than or equal to y_1. Next, CutSat++ has to propagate the newly introduced constraints K_1 and K_2 because CutSat++ follows a reasonable strategy and K_1 and K_2 are of the form $\pm y_1 + c \leq 0$. However, this turns the constraint K_3 into a guarded conflict constraint. After applying Conflict to K_3 and resolving K_3 with $x_1 \geq_{K_1} 0$, CutSat++ has derived the trivially unsatisfiable constraint $1 \leq 0$. Finally, CutSat++ uses $1 \leq 0$ to apply Unsat and, thereby, return *unsat*.

Example 3.5.11. Let

$$
\begin{aligned}
C := \{ \quad & \underbrace{-x_3 \leq 0}_{I_1}, \underbrace{x_3 \leq 0}_{I_2}, \underbrace{-x_4 \leq 0}_{I_3}, \underbrace{x_4 \leq 0}_{I_4}, \underbrace{-x_5 \leq 0}_{I_5}, \\
& \underbrace{-x_5 - x_1 - x_3 \leq 0}_{I_6}, \underbrace{-x_5 + x_1 + x_3 \leq 0}_{I_7}, \\
& \underbrace{-x_5 - x_2 - x_4 \leq 0}_{I_8}, \underbrace{-x_5 + x_2 + x_4 \leq 0}_{I_9}, \\
& \underbrace{2 + 3x_1 - 4x_2 + 3x_3 - 4x_4 \leq 0}_{J_1}, \underbrace{-1 + 3x_1 - 4x_2 + 3x_3 - 4x_4 \leq 0}_{J_2}, \\
& \underbrace{-1 - 3x_1 + 2x_2 - 3x_3 + 2x_4 \leq 0}_{J_3} \}
\end{aligned}
$$

be a problem. Let $C_1 := C_0 \cup R_{y_1} \cup R_1$, where

$$
\begin{aligned}
(R_{y_1}, R_1) := (\quad & \{\underbrace{-y_1 \leq 0}_{K_1}, \underbrace{-3 + y_1 \leq 0}_{K_2}\}), \\
& \{\underbrace{4 \mid 2 + 3x_1 + 3x_3 + y_1}_{K_3}, \underbrace{-3x_1 - 3x_3 + y_1 \leq 0}_{K_4}\})
\end{aligned}
$$

is the output of w-cooper$(x_2, \{J_1, J_3\})$. Let $C_2 := C_1 \cup R_{y_2} \cup R_2$, where

$$
(R_{y_2}, R_2) := (\{\underbrace{-y_2 \leq 0}_{K_5}, \underbrace{-2 + y_2 \leq 0}_{K_6}\}), \{\underbrace{3 \mid y_1 + y_2}_{K_7}, \underbrace{y_1 - 3x_5 + y_2 \leq 0}_{K_8}\})
$$

is the output of w-cooper$(x_1, \{K_4, I_7\})$. Let $C_3 := C_2 \cup R_{y_3} \cup R_3$, where

$$
\begin{aligned}
(R_{y_3}, R_3) := (\quad & \{\underbrace{-y_3 \leq 0}_{K_9}, \underbrace{-11 + y_3 \leq 0}_{K_{10}}\}), \\
& \{\underbrace{3 \mid y_1 + y_3}_{K_{11}}, \underbrace{4 \mid 2 + 2y_1 + y_3}_{K_{12}}, \underbrace{y_1 - 3x_5 + y_3 \leq 0}_{K_{13}}\})
\end{aligned}
$$

is the output of w-cooper$(x_1, \{K_3, K_4, I_7\})$.[12] Let the variable order be given by $x_3 \prec x_4 \prec x_5 \prec x_1 \prec x_2$. Then CutSat++ would solve the problem as depicted in Figures 3.13, 3.14, 3.15, and 3.16.

First of all, CutSat++ has to propagate I_1, I_2, I_3, I_4, and I_5, i.e., all constraints containing only one variable. In order to propagate bounds for x_1 and x_2, CutSat++ has to fix x_5 first. It does so with the decided bound $x_5 \leq 0$. Then, CutSat++ propagates the constraints I_6, I_7, I_8, and I_9, which fixes both x_1 and x_2 to 0 and turns $(x_2, \{J_1, J_3\})$ into a conflicting core. Since only J_1 is a conflicting constraint and $x_2 = \text{top}(J_1)$, the rule Resolve-Weak-Cooper is applicable and CutSat++ will use it to resolve the conflicting core $(x_2, \{J_1, J_3\})$. The resulting resolvent is (R_{y_1}, R_1) for which CutSat++ introduces the fresh variable y_1 and updates the variable order to $x_3 \prec x_4 \prec y_1 \prec x_5 \prec x_1 \prec x_2$. Resolve-Weak-Cooper does not actually truncate the current bound sequence because the current bound sequence is already its own largest prefix that contains only decided bounds for variables x_j with $x_j \prec x_1$. Although J_1 is still a conflict in the re-

[12] For the sake of brevity, we simplified some of the constraints generated in this example. For instance, K_3 is the simplified version of $4 \mid 2+3x_1+3x_3+4x_4+y_1$. These simplifications have no influence on the overall behavior of CutSat++.

$$
\begin{aligned}
&\langle [\![]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Decide}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0, x_1 \geq_{I_6} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0, x_1 \geq_{I_6} 0, x_1 \leq_{I_7} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0, x_1 \geq_{I_6} 0, x_1 \leq_{I_7} 0, x_2 \geq_{I_8} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0, x_1 \geq_{I_6} 0, x_1 \leq_{I_7} 0, x_2 \geq_{I_8} 0, x_2 \leq_{I_9} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Resolve-Weak-Cooper}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0, x_1 \geq_{I_6} 0, x_1 \leq_{I_7} 0, x_2 \geq_{I_8} 0, x_2 \leq_{I_9} 0]\!], C_1\rangle
\end{aligned}
$$

where the new order is $x_3 \prec x_4 \prec y_1 \prec x_5 \prec x_1 \prec x_2$,
and $C_1 := C_0 \cup R_{y_1} \cup R_1$

$$
\begin{aligned}
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0, x_1 \geq_{I_6} 0, x_1 \leq_{I_7} 0, x_2 \geq_{I_8} 0, x_2 \leq_{I_9} 0, \\
& && \quad y_1 \geq_{K_1} 0]\!], C_1\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0, x_1 \geq_{I_6} 0, x_1 \leq_{I_7} 0, x_2 \geq_{I_8} 0, x_2 \leq_{I_9} 0, \\
& && \quad y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3]\!], C_1\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Decide}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, \\
& && \quad x_5 \leq 0, x_1 \geq_{I_6} 0, x_1 \leq_{I_7} 0, x_2 \geq_{I_8} 0, x_2 \leq_{I_9} 0, \\
& && \quad y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_1 \geq 3]\!], C_1\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Resolve-Weak-Cooper}} && \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0]\!], C_2\rangle
\end{aligned}
$$

where the new order is $x_3 \prec x_4 \prec y_2 \prec y_1 \prec x_5 \prec x_1 \prec x_2$,
and $C_2 := C_1 \cup R_{y_2} \cup R_2$

Figure 3.13: The first part of a CutSat++ run for Example 3.5.11. Continued in Figure 3.14

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0]\!], C_2 \rangle$$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2]\!], C_2 \rangle$$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0]\!], C_2 \rangle$$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3]\!], C_2 \rangle$$

$$\Longrightarrow_{\text{CS}}^{\text{Decide}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2]\!], C_2 \rangle$$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate-Div}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2, y_1 \leq_{K_7'} 1]\!], C_2 \rangle$$

where $S = \langle M, C_2 \rangle$ is the state before the application of Propagate-Div, and $K_7' := \text{div-derive}(K_7, y_1, \leq, M) = y_1 + y_2 - 3 \leq 0$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate-Div}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2, y_1 \leq_{K_7'} 1, y_1 \geq_{K_7''} 1]\!], C_2 \rangle$$

where $S = \langle M, C_2 \rangle$ is the state before the application of Propagate-Div, and $K_7'' := \text{div-derive}(K_7, y_1, \geq, M) = -y_1 - 2y_2 - 3 \leq 0$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2, y_1 \leq_{K_7'} 1, y_1 \geq_{K_7''} 1, x_5 \geq_{K_8'} 1]\!], C_2 \rangle$$

where $S = \langle M, C_2 \rangle$ is the state before the application of Propagate, and $K_8' := \text{tight}(K_8, x_5, M) = -x_5 + y_2 - 1 \leq 0$

$$\Longrightarrow_{\text{CS}}^{\text{Decide}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2, y_1 \leq_{K_7'} 1, y_1 \geq_{K_7''} 1, x_5 \geq_{K_8'} 1, x_5 \leq 1]\!], C_2 \rangle$$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2, y_1 \leq_{K_7'} 1, y_1 \geq_{K_7''} 1, x_5 \geq_{K_8'} 1, x_5 \leq 1, x_1 \leq_{I_7} 1]\!], C_2 \rangle$$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0, y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2, y_1 \leq_{K_7'} 1, y_1 \geq_{K_7''} 1, x_5 \geq_{K_8'} 1, x_5 \leq 1, x_1 \leq_{I_7} 1, x_1 \geq_{K_4'} 1]\!], C_2 \rangle$$

where $S = \langle M, C_2 \rangle$ is the state before the application of Propagate, and $K_4' := \text{tight}(K_4, x_1, M) = -x_1 - x_3 + y_2 - 1 \leq 0$

Figure 3.14: The second part of a CutSat++ run for Example 3.5.11. Continued in Figure 3.15

$\Longrightarrow_{\text{CS}}^{\text{Resolve-Weak-Cooper}}$ $\langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$
$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$
$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1]\!], C_3 \rangle$

where the new order is $x_3 \prec x_4 \prec y_3 \prec y_2 \prec y_1 \prec x_5 \prec x_1 \prec x_2$,
and $C_3 := C_2 \cup R_{y_3} \cup R_3$

$\Longrightarrow_{\text{CS}}^{\text{Propagate}}$ $\langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$
$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$
$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0]\!], C_3 \rangle$

$\Longrightarrow_{\text{CS}}^{\text{Propagate}}$ $\langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$
$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$
$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11]\!],$
$C_3 \rangle$

$\Longrightarrow_{\text{CS}}^{\text{Propagate-Div}}$ $\langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$
$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$
$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$
$y_3 \leq_{K'_{12}} 8]\!], C_3 \rangle$

where $S = \langle M, C_3 \rangle$ is the state before the application of Propagate-Div,
and $K'_{12} := \text{div-derive}(K_{12}, y_3, \leq, M) = 2y_1 + 4y_2 + y_3 - 18 \leq 0$

$\Longrightarrow_{\text{CS}}^{\text{Decide}}$ $\langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$
$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$
$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$
$y_3 \leq_{K'_{12}} 8, y_3 \geq 8]\!], C_3 \rangle$

$\Longrightarrow_{\text{CS}}^{\text{Propagate}}$ $\langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$
$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$
$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$
$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3]\!], C_3 \rangle$

where $S = \langle M, C_3 \rangle$ is the state before the application of Propagate,
and $K'_{13} := \text{tight}(K_{13}, x_5, M) = -x_5 + y_3 + 6y_2 - 17 \leq 0$

$\Longrightarrow_{\text{CS}}^{\text{Decide}}$ $\langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$
$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$
$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$
$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3, x_5 \leq 3]\!], C_3 \rangle$

$\Longrightarrow_{\text{CS}}^{\text{Propagate}}$ $\langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$
$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$
$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$
$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3, x_5 \leq 3, x_1 \leq_{I_7} 3]\!], C_3 \rangle$

Figure 3.15: The third part of a CutSat++ run for Example 3.5.11. Continued in Figure 3.16

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$$
$$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$$
$$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$$
$$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3, x_5 \leq 3, x_1 \leq_{I_7} 3,$$
$$x_1 \geq_{K''_4} 1]\!], C_3 \rangle$$

where $S = \langle M, C_3 \rangle$ is the state before the application of Propagate, and $K''_4 := \text{tight}(K_4, x_1, M) = -x_1 - x_3 + y_2 - 1 \leq 0$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate-Div}} \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$$
$$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$$
$$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$$
$$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3, x_5 \leq 3, x_1 \leq_{I_7} 3,$$
$$x_1 \geq_{K''_4} 1, x_1 \geq_{K'_3} 2]\!], C_3 \rangle$$

where $S = \langle M, C_3 \rangle$ is the state before the application of Propagate-Div, and $K'_3 := \text{div-derive}(K_3, x_1, \geq, M) = -x_1 - x_3 + 3y_2 - 4 \leq 0$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate-Div}} \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$$
$$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$$
$$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$$
$$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3, x_5 \leq 3, x_1 \leq_{I_7} 3,$$
$$x_1 \geq_{K''_4} 1, x \geq_{K'_3} 2, x_1 \geq_{K''_3} 3]\!], C_3 \rangle$$

where $S = \langle M, C_3 \rangle$ is the state before the application of Propagate-Div, and $K''_3 := \text{div-derive}(K_3, x_1, \geq, M) = -x_1 - x_3 + 5y_2 - 7 \leq 0$

$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$$
$$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$$
$$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$$
$$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3, x_5 \leq 3, x_1 \leq_{I_7} 3,$$
$$x_1 \geq_{K''_4} 1, x \geq_{K'_3} 2, x_1 \geq_{K''_3} 3, x_2 \leq_{I_9} 3]\!], C_3 \rangle$$
$$\Longrightarrow_{\text{CS}}^{\text{Propagate}} \quad \langle [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$$
$$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$$
$$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$$
$$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3, x_5 \leq 3, x_1 \leq_{I_7} 3,$$
$$x_1 \geq_{K''_4} 1, x \geq_{K'_3} 2, x_1 \geq_{K''_3} 3, x_2 \leq_{I_9} 3, x_2 \geq_{J'_1} 3]\!], C_3 \rangle$$

where $S = \langle M, C_3 \rangle$ is the state before the application of Propagate, and $J'_1 := \text{tight}(J_1, x_2, \geq, M) = -x_2 - x_4 + 4y_2 - 5 \leq 0$

$$\Longrightarrow_{\text{CS}}^{\text{Sat}} \quad \langle \nu [\![x_3 \geq_{I_1} 0, x_3 \leq_{I_2} 0, x_4 \geq_{I_3} 0, x_4 \leq_{I_4} 0, x_5 \geq_{I_5} 0,$$
$$y_2 \geq_{K_5} 0, y_2 \leq_{K_6} 2, y_1 \geq_{K_1} 0, y_1 \leq_{K_2} 3, y_2 \geq 2,$$
$$y_1 \leq_{K'_7} 1, y_1 \geq_{K''_7} 1, x_5 \geq_{K'_8} 1, y_3 \geq_{K_9} 0, y_3 \leq_{K_{10}} 11,$$
$$y_3 \leq_{K'_{12}} 8, y_3 \geq 8, x_5 \geq_{K'_{13}} 3, x_5 \leq 3, x_1 \leq_{I_7} 3,$$
$$x_1 \geq_{K''_4} 1, x \geq_{K'_3} 2, x_1 \geq_{K''_3} 3, x_2 \leq_{I_9} 3, x_2 \geq_{J'_1} 3]\!], C_3 \rangle$$

Figure 3.16: The fourth part of a CutSat++ run for Example 3.5.11

sulting search state, CutSat++ cannot apply Resolve-Weak-Cooper again to $(x_2, \{J_1, J_3\})$ because y_1, which is a variable smaller than x_2, is not yet fixed. Therefore, CutSat++ continues to propagate bounds for y_1 derived from the constraints K_1 and K_2. Moreover, CutSat++ has to first propagate the newly introduced constraints K_1 and K_2 because CutSat++ follows a reasonable strategy and K_1 and K_2 are of the form $\pm y_i + b_i \leq 0$. Note that CutSat++ cannot propagate bounds for y_1 from K_3 and K_4 because x_1 is the top variable in those constraints. Instead, CutSat++ fixes y_1 with a decided bound. CutSat++ is still unable to apply Resolve-Weak-Cooper to $(x_2, \{J_1, J_3\})$ because K_4 is a conflict with $\text{top}(K_4) = x_1 \prec x_2$. However, CutSat++ can and will apply Resolve-Weak-Cooper to the conflicting core $(x_1, \{K_4, I_7\})$. The resulting resolvent is (R_{y_2}, R_2) for which CutSat++ introduces the fresh variable y_2 and updates the variable order to $x_3 \prec x_4 \prec y_2 \prec y_1 \prec x_5 \prec x_1 \prec x_2$. Resolve-Weak-Cooper also truncates the bound sequence to the largest prefix that contains only decided bounds for variables x_j with $x_j \prec y_2$. Again, CutSat++ has to first propagate all constraints containing only one variable, so K_1, K_2, K_5, and K_6. Then, CutSat++ fixes y_2 with the decided bound $y_2 \geq 2$. Since y_2 is now fixed, CutSat++ can use the divisibility constraint K_7 to refine the lower and upper bound of y_1 to 1. Similarly, fixing y_1 allows CutSat++ to propagate a new lower bound for x_5 from K_8. CutSat++ cannot propagate any further bounds for x_5, so it fixes x_5 with a decided bound. Fixing x_5 allows CutSat++ to propagate an upper and a lower bound for x_1 from I_7 and K_4, respectively. However, in the resulting state, $(x, \{K_3, K_4, I_7\})$ is a conflicting core, which CutSat++ will resolve with Resolve-Weak-Cooper. The resulting resolvent is (R_{y_3}, R_3) for which CutSat++ introduces the fresh variable y_3 and updates the variable order to $x_3 \prec x_4 \prec y_3 \prec y_2 \prec y_1 \prec x_5 \prec x_1 \prec x_2$. Resolve-Weak-Cooper also truncates the bound sequence to the largest prefix that contains only decided bounds for variables x_j with $x_j \prec y_1$. Again, CutSat++ has to first propagate all constraints containing only one variable, so K_9 and K_{10}. Using the divisibility constraint K_{12}, CutSat++ is able to refine the lower bound of y_3 to 8. Since CutSat++ cannot propagate any further bounds for y_3, it fixes it with a decided bound to 8. Then, CutSat++ uses the inequality K_{13} and the fixed variables y_1 and y_3, to propagate 3 as the new lower bound for x_5. With a decided bound, CutSat++ fixes x_5 to 3. By propagating the constraints I_7, K_4, and K_3, CutSat++ fixes also x_1 to 3. Similarly, CutSat++ fixes x_2 to 3 by propagating I_9 and J_1. In the resulting bound sequence all variables are fixed and the assignment is satisfying all constraints in C_3. Therefore, CutSat++ ends with an application of Sat and returns the assignment:

$$x_3 \mapsto 0, x_4 \mapsto 0, y_3 \mapsto 8, y_2 \mapsto 2, y_1 \mapsto 1, x_5 \mapsto 3, x_1 \mapsto 3, x_2 \mapsto 3$$

$$
\begin{aligned}
&\langle [\![]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_1 \geq_{I_1} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate}} && \langle [\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1, x_2 \geq_{I_3} 0]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Decide}} && \langle [\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1, x_2 \geq_{I_3} 0, x_1 \geq 1]\!], C_0\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Resolve-Weak-Cooper}} && \langle [\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1, x_2 \geq_{I_3} 0]\!], C_1\rangle \\
& && \text{where } C_1 := C_0 \cup J_2 \\
&\Longrightarrow_{\text{CS}}^{\text{Propagate-Div}} && \langle [\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1, x_2 \geq_{I_3} 0, x_1 \leq_{J_2'} 0]\!], C_1\rangle \\
& \text{where } J_2' := \text{div-derive}(J_2, x_1, \leq, [\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1, x_2 \geq_{I_3} 0]\!]) = x_1 \leq 0 \\
&\Longrightarrow_{\text{CS}}^{\text{Decide}} && \langle [\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1, x_2 \geq_{I_3} 0, x_1 \leq_{J_2'} 0, x_2 \leq 0]\!], C_1\rangle \\
&\Longrightarrow_{\text{CS}}^{\text{Sat}} && \langle \nu[[\![x_1 \geq_{I_1} 0, x_1 \leq_{I_2} 1, x_2 \geq_{I_3} 0, x_1 \leq_{J_2'} 0, x_2 \leq 0]\!]], C_1\rangle
\end{aligned}
$$

Figure 3.17: A CutSat++ run for Example 3.5.12

Example 3.5.12. Let

$$C_0 := \{\underbrace{-x_1 \leq 0}_{I_1}, \underbrace{x_1 - 1 \leq 0}_{I_2}, \underbrace{-x_2 \leq 0}_{I_3}, \underbrace{6 \mid 4y + x_1}_{J}\}$$

be a problem. Let $(R_y, R_c) := (\emptyset, \{J_2\})$ be the output of w-cooper$(x_2, \{J\})$ where $J_2 = 2 \mid x_1$. Let the variable order be given by $x_1 \prec x_2$. Then CutSat++ would solve the problem as depicted in Figure 3.17. First of all, CutSat++ has to propagate I_1, I_2, and I_3, i.e., all constraints containing only one variable. Then CutSat++ will fix x_1 to the bound $x_1 \leq_{I_1} 1$ with the decided bound $x_1 \geq 1$, which turns $(x_2, \{J\})$ into a conflicting core. Note that CutSat++ cannot use $J_1 = 6 \mid 4x_2 + x_1$ to propagate a bound for x_2 because CutSat++ follows an eager top-level propagating strategy and $\gcd(4, 6) = 2$ does not divide 1 the value x_1 is fixed to. For the same reason CutSat++ cannot fix x_2 with a decided bound. Since only J is a conflicting constraint and $x_2 = \text{top}(J)$, the rule Resolve-Weak-Cooper is applicable and CutSat++ will use it to resolve the conflicting core $(x_2, \{J\})$. The resulting resolvent is $(\emptyset, \{J_2\})$, where $J_2 = 2 \mid x_1$. Resolve-Weak-Cooper also truncates the bound sequence so that all decided bounds for variables greater than or equal to x_1 are removed. Now CutSat++ propagates the constraint J_2 to fix x_1 to 0 instead of 1. Since $6 \mid 4x_2 + x_1$ is satisfied by the bounds for x_1 and x_2, CutSat++ is also able to fix x_2 with a decided bound. Finally, CutSat++ returns the satisfiable assignment with an application of the Sat rule.

3.6 Termination and Completeness

The CutSat++ rules are Propagate, Propagate-Div, Decide, Sat, Unsat, Unsat-Div, Conflict, Conflict-Div, Resolve, Skip-Decision, Backjump, Slack-Intro, Learn, and Forget from CutSat$_g$ [87], as well as Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right (Figure 3.9). Before we prove termination and completeness for CutSat++, we have to prove another property over unguarded resolvents. We have proven in Section 3.5 that Resolve-Weak-Cooper applied to the conflicting core (x_j, C') adds an unguarded resolvent R that blocks another application of Resolve-Weak-Cooper to (x_j, C'). However, CutSat++ is able to remove constraints from R with the rules Solve-Div-Left and Solve-Div-Right. This removes the original conflicting core R from our state. Nonetheless, CutSat++ is still unable to apply Resolve-Weak-Cooper to conflicting core (x_j, C') because the rules Solve-Div-Left and Solve-Div-Right guarantee that a new unguarded resolvent R' for conflicting core (x_j, C') is introduced:

Lemma 3.6.1 (Resolvent Stability)**.** *Let $S = \langle M, C\rangle$ be a state reachable by* CutSat++ *from the start state $\langle [\![]\!], C_0\rangle$ and let $S' = \langle M', C'\rangle$ be a state reachable by* CutSat++ *from S. Let C contain an unguarded resolvent R for (x_j, C''). Then C' contains also an unguarded resolvent R' for (x_j, C'').*

Proof. Assume for a contradiction that C contains an unguarded resolvent R for (x_j, C'') and C' contains no unguarded resolvent R' for (x_j, C''). W.l.o.g., we assume that S' is the first state after S where $R \not\subseteq C'$. By Definition 3.5.7-(4), CutSat++ with a strictly-two-layered strategy cannot remove constraints from an unguarded resolvent R except with the rules Solve-Div-Right and Solve-Div-Left. Through the equivalence proven for div-solve$(x_j, \{I_1, I_2\})$ [87], we know that there exist $I_1', I_2' \in C'$ such that $\{I_1', I_2'\} \equiv \{I_1, I_2\}$ and $R \subseteq (C' \setminus \{I_1', I_2'\}) \cup \{I_1, I_2\}$. We use this equivalence to construct $R' = (R \setminus \{I_1, I_2\}) \cup \{I_1', I_2'\}$ such that $R' \to R$. Since $R' \to R$ and $R \to \exists x_j \in \mathbb{Z}.C''$, R' is also an unguarded resolvent of (x_j, C'') such that $R' \to \exists x_j \in \mathbb{Z}.C''$. Furthermore, R' is a subset of C', which contradicts our initial assumption! □

Together with Lemma 3.5.4, this property implies that Resolve-Weak-Cooper is applied at most once to every conflicting core encountered by CutSat++. This is essential for our termination proof.

3.6.1 Termination

For the termination proof of CutSat++, we consider a (possibly infinite) sequence of rule applications $\langle [\![], C_0 \rangle = S_0 \Longrightarrow_{\text{CS}} S_1 \Longrightarrow_{\text{CS}} \dots$ on a problem C_0 following the strictly-two-layered strategy. We will show that each such sequence of rule applications can be divided into five phases—the slacking phase, the unguarded resolution phase, the guarded search phase, the unguarded extension phase, and the end phase—and that each of those phases terminates after finitely many rule applications.

First, this sequence reaches a state S_s ($s \in \mathbb{N}_0^+$), after a finite derivation of rule applications $S_0 \Longrightarrow_{\text{CS}} \dots \Longrightarrow_{\text{CS}} S_s$, such that there is no further application of the rules Slack-Intro and Forget after state S_s:

Lemma 3.6.2 (Slacking Phase)**.** *Let $\langle [\![], C_0 \rangle = S_0 \Longrightarrow_{CS} S_1 \Longrightarrow_{CS} \dots$ be a sequence of rule applications applied to a problem C_0 following the strictly-two-layered strategy. Then the sequence reaches a state S_s ($s \in \mathbb{N}_0^+$), after at most finitely many rule applications $S_0 \Longrightarrow_{CS} \dots \Longrightarrow_{CS} S_s$, such that there is no further application of the rules Slack-Intro and Forget after state S_s.*

Proof. Such a state S_s exists for two reasons: Firstly, the strictly-two-layered strategy employed by CutSat++ is also reasonable. The reasonable strategy explicitly forbids infinite applications of the rule Forget. Secondly, the Slack-Intro rule is applicable only to stuck variables and only once to each stuck variable. Only the initial set of variables can be stuck because all variables x_j introduced by one of the rule applications are introduced with at least one constraint $x_j - b_j \leq 0$ that allows at least one propagation for the variable. Therefore, the rules Slack-Intro and Forget are at most finitely often applicable. □

Next, the sequence reaches a state S_w ($w \geq s$), after a finite derivation of rule applications $S_s \Longrightarrow_{\text{CS}} \dots \Longrightarrow_{\text{CS}} S_w$, such that there is no further application of the rules Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right after state S_w:
The rules Slack-Intro, Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right are applicable only to unguarded constraints. Through the strictly-two-layered strategy, they are also the only rules producing unguarded constraints. Therefore, they form a closed loop with respect to unguarded constraints, which we use in our termination proof. We have shown in the previous paragraph that $S_s \Longrightarrow_{\text{CS}} \dots \Longrightarrow_{\text{CS}} S_w$ contains no application of the rule Slack-Intro. By Lemma 3.5.4, an application of Resolve-Weak-Cooper to the conflicting core (x_j, C') prevents any further applications of Resolve-Weak-Cooper to the same core. By Definition 3.5.1, the constraints learned through an application of Resolve-Weak-Cooper contain only variables x_i such that $x_i \prec x_j$. Therefore, an application of Resolve-Weak-Cooper blocks a conflicting core (x_j, C') and introduces potential conflicting cores

only for smaller variables than x_j. This strict decrease in the conflicting variables guarantees that we encounter only finitely many conflicting cores in unguarded variables. Therefore, Resolve-Weak-Cooper is applicable at most finitely often. An analogous argument applies to the rules Solve-Div-Left and Solve-Div-Right. Thus the rules Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right are applicable at most finitely often.

Lemma 3.6.3 (Unguarded Resolution Phase)**.** *Let $\langle [\![], C_0\rangle = S_0 \Longrightarrow_{CS} S_1 \Longrightarrow_{CS} \dots$ be a sequence of rule applications applied to a problem C_0 following the strictly-two-layered strategy. Then the sequence reaches a state S_w, after finitely many rule applications $S_0 \Longrightarrow_{CS} \dots \Longrightarrow_{CS} S_w$, such that there is no further application of the rules Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right after state S_w.*

Proof. By Lemma 3.6.2, we assume, w.l.o.g., that the sequence continues from a state S_s such that S_s is reached by the sequence after at most finitely many rule applications $S_0 \Longrightarrow_{\text{CS}} \dots \Longrightarrow_{\text{CS}} S_s$ and that there is no further application of the rules Slack-Intro and Forget after state S_s. Let $x_1 \prec \dots \prec x_n$ be the order of variables for all unguarded variables x_i. Next, we define a weight vector that strictly decreases after every call to Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right. For this weight vector, we define $\text{cores}(x_i, C)$ as the set of potential conflicting cores in the problem C with conflicting variable x_i. We also define a subset $\text{woSR}(x_i, C)$ of $\text{cores}(x_i, C)$ so it contains all potential conflicting cores within $\text{cores}(x_i, C)$ that do not have an unguarded resolvent $R \subseteq C$. It is easy to see that $|\,\text{cores}(x_i, C)| \leq 2^{|C|}$ and, therefore, both functions define finite sets. Now we define the weight vector $\mathcal{W}(S)$ for every state $S = \langle M, C, I\rangle$:
$\mathcal{W}(S) = (|\,\text{cores}(x_n, C)|, |\,\text{woSR}(x_n, C)|, \dots, |\,\text{cores}(x_1, C)|, |\,\text{woSR}(x_1, C)|)$
We order the two $\mathcal{W}$ vectors of two subsequent search-states with the well-founded lexicographic order $>_{\text{lex}}$ based on the well-founded order $>$.

By Definition 3.5.7, Conflict(-Div) is only applicable to guarded constraints and guarded variables are only propagated through guarded constraints. Therefore, the conflict I in a conflict state $\langle M, C, I\rangle$ stays always guarded—even after an application of the Resolve rule—and Learn is only applicable to guarded constraints. Therefore, Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right are the only rules potentially learning unguarded constraints and, thereby, the only rules that can increase $|\,\text{cores}(x_i, C)|$ and $|\,\text{woSR}(x_i, C)|$ between two subsequent states $S' \Longrightarrow_{\text{CS}} S$. After all other transitions $S' \Longrightarrow_{\text{CS}} S$, where $S' = \langle M', C'\rangle$ and $S = \langle M, C\rangle$ it holds that $\mathcal{W}(S') \geq_{\text{lex}} \mathcal{W}(S)$.

The reasons why the weight vector strictly decreases, i.e., $\mathcal{W}(S') >_{\text{lex}} \mathcal{W}(S)$, whenever CutSat++ applies Solve-Div-Left, Solve-Div-Right, or Resolve-Weak-Cooper are as follows:

1. By Lemma 3.5.4, an application of Resolve-Weak-Cooper to conflicting core (x_i, C^*) implies that there is no unguarded resolvent $R' \subseteq C'$ for (x_i, C^*). By Lemma 3.5.3, the new problem $C = C' \cup R$ contains an unguarded resolvent R for (x_i, C^*). Therefore, $|\operatorname{woSR}(x_i, C')| > |\operatorname{woSR}(x_i, C)|$. By Definition 3.5.2, it holds for all $x_k \in \operatorname{vars}(R)$ that $x_k \prec x_i$, which implies that $k < i$. Hence, Resolve-Weak-Cooper has not introduced new potential conflicting cores (x_j, C^{**}) with $j \geq i$ and $|\operatorname{cores}(x_j, C')| \geq |\operatorname{cores}(x_j, C)|$ for all $j \geq i$. By Lemma 3.6.1, $|\operatorname{woSR}(x_j, C')| \geq |\operatorname{woSR}(x_j, C)|$ for all $j > i$. Therefore, the weight decreases after an application of Resolve-Weak-Cooper, i.e., $\mathcal{W}(S') >_{\text{lex}} \mathcal{W}(S)$.

2. Let Solve-Div-Left (Solve-Div-Right) be applied to the pair of divisibility constraints (I_1, I_2) such that $\operatorname{top}(I_1) = x_i$ and $\text{div-solve}(x_i, \{I_1, I_2\}) = \{I_1', I_2'\}$. The new constraint set is $C = C' \setminus \{I_1, I_2\} \cup \{I_1', I_2'\}$. The number of potential conflicting cores is independent of the actual divisibility constraints in C'. Only the number of divisibility constraints with $top(I) = x_i$ is important. Therefore, removing I_1 and replacing it with I_1' doesn't increase the number of cores, i.e., $|\operatorname{cores}(x_i, C' \setminus \{I_1\} \cup \{I_1'\})| = |\operatorname{cores}(x_i, C')|$. We will, however, decrease the number of conflicting cores in x_i, i.e., $|\operatorname{cores}(x_i, C')| > |\operatorname{cores}(x_i, C)|$, because we replace I_2 with I_2' where $\operatorname{top}(I_2') \prec x_i$. Since all variables x_j in I_1' and I_2' are smaller than or equal to x_i, we do not introduce any new conflicting cores for x_k with $k > i$; thus, $|\operatorname{cores}(x_k, C')| = |\operatorname{cores}(x_k, C)|$ for $k > i$. Finally, Lemma 3.6.1 implies that $|\operatorname{woSR}(x_k, C')| \geq |\operatorname{woSR}(x_k, C)|$ for $k > i$. Therefore, $\mathcal{W}(C') >_{\text{lex}} \mathcal{W}(C)$.

This proves already that the $\mathcal{W}$ vector monotonically decreases with respect to the order $>_{\text{lex}}$ if we continue from the before mentioned state S_s. Moreover, the set of weight vectors has a minimum $(0, \ldots, 0)$ because no set can contain less than zero elements. As $>_{\text{lex}}$ is well-founded, there exists no way to decrease the weight $\mathcal{W}(C_s)$ without reaching the minimum $(0, \ldots, 0)$ after finitely many applications of the rules Solve-Div-Left, Solve-Div-Right, or Resolve-Weak-Cooper. Finally, the $\mathcal{W}$ vector cannot decrease below $(0, \ldots, 0)$; hence, CutSat++ is not able to apply Solve-Div-Left, Solve-Div-Right, or Resolve-Weak-Cooper after we reach a state S with $\mathcal{W}(S) = (0, \ldots, 0)$. We conclude that the rules Solve-Div-Left, Solve-Div-Right, and Resolve-Weak-Cooper are at most finitely often applicable. □

Next, the sequence reaches a state S_g $(g \geq w)$, after a finite derivation of rule applications $S_w \Longrightarrow_{\text{CS}} \ldots \Longrightarrow_{\text{CS}} S_g$, such that the bounds remain invariant for every guarded variable x_i, i.e., $\mathcal{L}(x_i, M_g) = \mathcal{L}(x_i, M_j)$ and $\mathcal{U}(x_i, M_g) = \mathcal{U}(x_i, M_j)$ for every state $S_j = \langle M_j, C_j, I_j \rangle$ after $S_g = \langle M_g, C_g, I_g \rangle$ $(j \geq g)$. CutSat++ reaches such a state because the strictly-two-layered strategy guarantees that unguarded variables do not influence guarded variables. Due to this independence, it is easy to extend the proof

for the completely guarded case (see [87]) so it also proves that CutSat++ has to reach a state S_g. By definition of S_g, we now also know that the sequence after S_g contains no further propagations, decisions, or conflict resolutions for the guarded variables.

Lemma 3.6.4 (Guarded Search Phase)**.** *Let $\langle [\![], C_0\rangle = S_0 \Longrightarrow_{CS} S_1 \Longrightarrow_{CS} \dots$ be a sequence of rule applications applied to a problem C_0 following the strictly-two-layered strategy. Then the sequence reaches a state S_g, after finitely many rule applications $S_0 \Longrightarrow_{CS} \dots \Longrightarrow_{CS} S_g$, such that the bounds remain invariant for every guarded variable x_i.*

Proof. This proof is based on the termination proof for CutSat on finite problems, i.e., problems without unguarded variables [87]. The proof uses a weight function that strictly decreases whenever CutSat++ changes a bound for a guarded variable and otherwise stays the same. By Lemmas 3.6.2 and 3.6.3, we assume, w.l.o.g., that the sequence continues from a state S_w such that S_w is reached by the sequence after at most finitely many rule applications $S_0 \Longrightarrow_{\text{CS}} \dots \Longrightarrow_{\text{CS}} S_w$, and there is no further application of the rules Slack-Intro, Forget, Resolve-Weak-Cooper, Solve-Div-Left, and Solve-Div-Right after state S_w. The level_B of a state $S = \langle M, C\rangle$ is the number of decisions for guarded variables in M. The maximal prefix of M containing only j decisions for guarded variables is denoted by $\text{B-subseq}_j(M) = M_j$. Since CutSat++ follows a reasonable strategy, it prefers to propagate constraints of the form $\pm x_k - b_k \leq 0$. This allows us to assume, w.l.o.g., that M_0 contains both a lower and upper bound for all guarded variables x_i. The *guarded weight* of the j-th level_B is defined by the function $w_B(M_j)$:

$$w_B(M_j) = \textstyle\sum_{x_i \text{ is guarded}} (\mathcal{U}(x_i, M_j) - \mathcal{L}(x_i, M_j))\,.$$

The *guarded weight* of a state is the vector:

$$\text{weight}_B(\langle M, C\rangle) = \langle w_B(\text{B-subseq}_0(M)), \cdots, w_B(\text{B-subseq}_n(M))\rangle,$$

where n is the number of guarded variables. We order the two weight_B-vectors of two subsequent search-states with the well-founded lexicographic order $>_{\text{lex}}$ based on the well-founded order $>$. It is easy to see that the minimum of weight_B is $(0, \dots, 0)$ and that any change to a bound of a guarded variable changes weight_B. Furthermore, by the definition of the strictly-two-layered strategy, we see that we only propagate guarded variables with guarded constraints. Thus, the strategy also implies that the conflict rules—Conflict, Conflict-Div, Backjump, Resolve, Skip-Decision, Unsat, and Learn—only handle guarded constraints. Given the proof for Theorem 2 in [87], we see that every application of Propagate, Propagate-Div, and Decide applied to a guarded variable decreases weight_B strictly. We see in the same proof that weight_B strictly decreases between one application of Conflict(-Div) and Backjump as long as the conflict rules handle only guarded constraints—as is the case for CutSat++. Since the bound sequence M is finite, the conflict rules are at most $|M|$ times applicable between one application of Conflict(-Div), and Backjump or Unsat. The remaining

rules—Propagate, Propagate-Div, and Decide—applied to unguarded variables, have no influence on weight_B or the bounds of guarded variables. Since weight_B cannot decrease below $(0, \ldots, 0)$, we conclude that CutSat++ is not able to change the bounds for guarded variables infinitely often. □

Next, the sequence reaches a state S_u $(u \geq g)$, after a finite derivation of rule applications $S_g \Longrightarrow_{\text{CS}} \ldots \Longrightarrow_{\text{CS}} S_u$, such that the bounds also remain invariant for every unguarded variable x_i, i.e., $\mathcal{L}(x_i, M_u) = \mathcal{L}(x_i, M_j)$ and $\mathcal{U}(x_i, M_u) = \mathcal{U}(x_i, M_j)$ for every state $S_j = \langle M_j, C_j, I_j \rangle$ after $S_u = \langle M_u, C_u, I_u \rangle$ $(j \geq u)$. This is true because CutSat++ propagates and decides only unguarded variables after S_g or ends with an application of Sat or Unsat(-Div). These claims are facts because CutSat++ employs a strictly-two-layered strategy, which is also an eager top-level propagating strategy. Through the top variable restriction for propagating constraints, the eager top-level propagating strategy induces a strict order of propagation over the unguarded variables. Therefore, any bound for an unguarded variable x_i is influenced only by bounds for variables $x_k \prec x_i$. This strict variable order guarantees that unguarded variables are propagated and decided only finitely often.

Lemma 3.6.5 (Unguarded Extension Phase)**.** *Let* $\langle [\![], C_0 \rangle = S_0 \Longrightarrow_{CS} S_1 \Longrightarrow_{CS} \ldots$ *be a sequence of rule applications applied to a problem* C_0 *following the strictly-two-layered strategy. Then the sequence reaches a state* S_u*, after finitely many rule applications* $S_0 \Longrightarrow_{CS} \ldots \Longrightarrow_{CS} S_u$*, such that the bounds remain invariant for every unguarded variable* x_i*.*

Proof. By Lemmas 3.6.2, 3.6.3, and 3.6.4, we assume, w.l.o.g., that the sequence continues from a state $S_g = \langle M_g, C_g, I_g \rangle$ such that S_g is reached by the sequence after at most finitely many rule applications $S_0 \Longrightarrow_{\text{CS}} \ldots \Longrightarrow_{\text{CS}} S_g$ and only the rules Sat, Unsat-Div, Propagate, Propagate-Div, and Decide are applied after S_g, but only for unguarded variables. Assume for a contradiction that there exists an infinite CutSat++ run starting in S_g. Since there is only a finite number of unguarded variables and no rule to undo a decision, the Decide rule is applied at most finitely often. Furthermore, any application of Sat or Unsat-Div ends the sequence making it finite. Thus, we assume, w.l.o.g., that there is no application to the rules Sat, Unsat-Div, and Decide in the infinite run starting in the state S_g.

Since there are at most finitely many variables in state S_g and no rule to introduce further variables after S_g, there exists a smallest unguarded variable x_i that is propagated infinitely often. We assume, w.l.o.g., that the run starting in S_g propagates only variables x_k bigger than or equal to x_i. Therefore, the bounds of all variables x_j smaller than x_i remain invariant in all subsequent states $S' = \langle M', C', I' \rangle$ of S_b, i.e., $\mathcal{L}(x_j, M_j) = \mathcal{L}(x_j, M_b)$ and $\mathcal{U}(x_j, M_j) = \mathcal{U}(x_j, M_b)$. Since there exists no applicable rule that changes the constraint set, we notice that the constraint set C_g also remains invariant

for all states after S_g. Thus, we find all constraints C^* that will be used to propagate x_i in the set C_g. Since CutSat++ is eager top-level propagating, any constraint $I \in C^*$ has x_i as their top variable. This leads us to the deduction that $\text{bound}(I, x_i, \bowtie, M') = \text{bound}(I, x_i, \bowtie, M_g)$ for all subsequent states $S' = \langle M', C' \rangle$, inequalities $I \in C^*$, and $\bowtie \in \{\leq, \geq\}$. Since the bounds defined by the inequalities in C^* remain invariant after state S_g, CutSat++ propagates x_i at most finitely often with inequalities.

Therefore, there exists an infinite CutSat++ run only if x_i is propagated infinitely often with Propagate-Div. We assume, w.l.o.g., that the run starting in S_g propagates x_i only with Propagate-Div. Next, we deduce that variable x_i stays unbounded in the remaining states of the derivation sequence. Otherwise, there exists a finite set $x_i \in \{l_i, \ldots, u_i\}$ bounding x_i and, therefore, allowing only finitely many propagations. In the case that x_i stays unbounded, the definition of the eager top-level propagating strategy states that Propagate-Div is only applicable to x_i if $I_d = d \mid a_i x_i + p_i \in C^*$ is the only divisibility constraint in C_b with x_i as their top variable. Furthermore, we know because of Definition 3.5.6 and Lemma 3.4.5 that there must exist $v_i \in \mathbb{Z}$ such that $d \mid a_i v_i + \mathcal{L}(p_i, M_g)$ is satisfied. Now we see that Propagate-Div propagates x_i at most finitely often if we consider Lemma 3.2.1. More specifically, if the lower bound of x_i is $\mathcal{L}(x, M_g) = l_i \neq -\infty$ then Propagate-Div propagates for x_i at most $v_i - l_i$ lower bounds. If the upper bound of x_i is $\mathcal{U}(x_i, M_g) = u_i \neq \infty$ then Propagate-Div propagates for x_i at most $u_i - v_i$ upper bounds. This contradicts the assumption that x_i is the smallest variable propagated infinitely often, which in turn contradicts our initial assumption! □

The final phase of our CutSat++ run is called the end phase. It is called so because the only rules applicable after state S_u are the rules Sat, Unsat, and Unsat-Div, which lead to an end state. This is also the final reason why the sequence $S_0 \Longrightarrow_{\text{CS}} S_1 \Longrightarrow_{\text{CS}} \ldots$ must be finite. We conclude that CutSat++ always terminates:

Theorem 3.6.6 (CutSat++ Termination)**.** *If* CutSat++ *starts from a start state* $\langle [\![]\!], C_0 \rangle$*, then there is no infinite derivation sequence.*

Proof. By Lemmas 3.6.2, 3.6.3, 3.6.4, and 3.6.5, CutSat++ reaches a state S_u after which only the rules Sat, Unsat, and Unsat-Div are applicable, which lead to an end state. Therefore, CutSat++ does not diverge. □

3.6.2 Stuck States

Our CutSat++ calculus not only always terminates but it also never reaches a stuck state. Let x_i be the smallest unfixed variable with respect to $\prec$. If x_i is guarded then there always exist two constraints $x_i - u_i \leq 0 \in C$ and $-x_i - l_i \leq 0 \in C$. Therefore, we can always propagate at least one upper

and one lower bound for every guarded variable x_i and fix it by introducing a decision. If we cannot propagate any bound for x_i, then x_i is unguarded and stuck and, therefore, Slack-Intro is applicable. If we cannot fix x_i by introducing a decision, then x_i is unguarded and there is a conflict (see Definition 3.5.6). Guarded conflicts are resolved via the Conflict(-Div) rules. Unguarded conflicts are resolved via the unguarded conflict resolution rules. Therefore, CutSat++ has always a rule applicable unless an end state is reached.

The proof outlined above works because all unguarded conflicts encountered by CutSat++ are either the result of multiple contradicting divisibility constraints that can by combined with the rules Solve-Div-Left and Solve-Div-Right, or the conflict is expressible via a conflicting core. Since conflicting cores are only defined over constraints and propagated bounds, we have to guarantee that CutSat++ never encounters an unguarded conflict I where $x_i = \text{top}(I)$ is fixed with a decided bound. We express this property with the following invariant fulfilled by every state visited by CutSat++:

Definition 3.6.7 (Eager Top-Level Propagated States)**.** A search/conflict state $S = \langle M, C, I\rangle$ is called *eager top-level propagated* if it holds for all unguarded variables x_i, all decided bounds $\gamma = x_i \bowtie b_i$ ($\bowtie \in \{\leq, \geq\}$) in $M = [\![M', \gamma, M'']\!]$, and all constraints $J \in C$ with $\text{top}(J) = x_i$ that: (1) all other variables contained in J are fixed in M' and (2) J is no conflict in S.

Lemma 3.6.8 (Eager Top-Level Stability)**.** *If S' is an eager top-level propagated state (Definition 3.6.7), then any successor state $S = \langle M, C, I\rangle$ reachable by* CutSat++ *is eager top-level propagated.*

Proof. Let S' be an eager top-level propagated state and S its successor, i.e., $S' \Longrightarrow_{\text{CS}} S$. We prove this Lemma with a case distinction on the rule leading to the above transition:

1. Let the applied rule be Propagate(-Div). Then $S' = \langle M', C'\rangle$ and $S = \langle [\![M', x_i \bowtie_J b_i]\!], C'\rangle$, where $\bowtie \in \{\leq, \geq\}$. Let $J' \in C'$ be the constraint used for propagation with $b_i = \text{bound}(J', x_i, \bowtie, M')$ and $J = \text{tight}(J', x_i, M')$ (or $J = \text{div-derive}(J', x_i, \bowtie, M')$). Then J' fulfills by definition of the propagation rules the property $\text{improves}(J', x_i, \bowtie, M')$. Let the unguarded variable x_j be fixed by a decided bound γ in $M' = [\![M'', \gamma, M''']\!]$. Let $I \in C'$ be a constraint with $\text{top}(I) = x_j$. Since S' is eager top-level propagated, all variables in I are fixed in M' and M''. The variable x_i is not fixed in M' because the predicate $\text{improves}(J', x_i, \bowtie, M')$ must be true for Propagate(-Div) to be applicable. Therefore, x_i is not contained in I and I is still no conflict in S. Furthermore, all variables in I are still fixed in $[\![M', x_i \bowtie_J b_i]\!]$. We conclude that S is eager top-level propagated.

2. Let the applied rule be Decide. Then $S' = \langle M', C' \rangle$ and $S = \langle [\![M', x_i \bowtie b_i]\!], C' \rangle$, where $\bowtie \in \{\leq, \geq\}$. We will use the eager top-level propagating strategy (Definition 3.5.6) to prove that S is an eager top-level propagated successor state. First, we consider all unguarded variables x_j that are already decided in S' by a decided bound γ and prove the properties for them. Since x_j is already decided in S', its decided bound γ is part of M', i.e., $M' = [\![M'', \gamma, M''']\!]$. Let $I \in C'$ be a constraint with $\text{top}(I) = x_j$. As S' is eager top-level propagated, all other variables contained in I are fixed in M'' and, therefore, also in M'. Since $\mathcal{L}(x_i, M') < \mathcal{U}(x_i, M')$ is a condition of the Decide rule, the variable x_i is not fixed in M'. Therefore, x_i is not contained in I and I is still no conflict in S. Furthermore, all variables in I are still fixed in $[\![M', x_i \bowtie b_i]\!]$. Next, we prove that S is eager top-level propagated although variable x_i is newly decided. Considering Definition 3.5.6-(2a) we see that Definition 3.6.7-(1) is fulfilled. Similarly, Definition 3.5.6-(2b) enforces Definition 3.6.7-(2). We conclude that S is eager top-level propagated.

3. Let the applied rule be Unsat(-Div) or Sat. Then the successor state S is neither a search- or conflict-state. The Lemma is thereby trivially fulfilled.

4. Let the applied rule be Forget. Then $S' = \langle M', C' \cup \{J\} \rangle$ and $S = \langle M', C' \rangle$. Therefore, any conflict $I \in C'$ and any decided bound in S is also contained in S'. We conclude that S is eager top-level propagated.

5. Let the applied rule be Slack-Intro. Then $S' = \langle M', C' \rangle$, x_i is stuck in S' and $S = \langle M', C' \cup \{-x_S \leq 0, x_i - x_S \leq 0, -x_i - x_S \leq 0\} \rangle$. Since x_i and x_S are both unguarded, we have to prove for the new constraints $I \in \{-x_S \leq 0, x_i - x_S \leq 0, -x_i - x_S \leq 0\}$ that $x_k = \text{top}(I)$ is not decided in S/S' when I is a conflict S. If the slack variable x_S is decided in S/S', then Slack-Intro was already applied to another variable and $-x_S \leq 0 \in C'$. Thus, $-x_S \leq 0$ is not a conflict in S because S' is an eager top-level propagated state, which means that $-x_S \leq 0$ is not a conflict in S'. Since x_i was stuck in S', x_i is also not fixed in S'. Moreover, x_i is the top variable in the new constraints $\{x_i - x_S \leq 0, -x_i - x_S \leq 0\}$. We conclude that S is eager top-level propagated.

6. Let the applied rule be Resolve-Weak-Cooper. Then $S' = \langle M', C' \rangle$ and $S = \langle M, C' \cup R_c \cup R_y \rangle$. Moreover, the definition of Resolve-Weak-Cooper implies that $M = \text{prefix}(M', x_j)$ with $x_j = \min_{I \in R_c}\{\text{top}(I)\}$. Therefore, M is the prefix of M' without decided bounds in variables greater or equal to x_j. Since $x_j \preceq x_i$ for all $I \in R_c$ and $x_i = \text{top}(I)$, we deduce that any $I \in R_c$ that is a conflict has no decided bound for its top variable x_i in S. Since M is a prefix of M', every conflict $I \in C'$ appearing in state S also appears in state S'. Now it is easy to see that S is eager top-level propagated because S' was eager top-level propagated.

7. Let the applied rule be Solve-Div-Right. Then $S' = \langle M', C' \cup \{I_1, I_2\}\rangle$ and $S = \langle M, C' \cup \{I_1', I_2'\}\rangle$. We notice that $M = \text{prefix}(M', x_j)$ with $x_j = \text{top}(I_2')$. Therefore, M is the prefix of M' without decided bounds in variables greater or equal to x_j, which includes especially the variable $x_i = \text{top}(I_1)$. Thus, neither the top variable of I_1' nor the top variable I_2' is fixed by a decision. Since M is a prefix of M', every conflict $I \in C'$ appearing in state S also appears in state S'. Now it is easy to see that S is eager top-level propagated because S' was eager top-level propagated.

8. Let the applied rule be Solve-Div-Left. Then $S' = \langle M', C' \cup \{I_1, I_2\}\rangle$ and $S = \langle M', C' \cup \{I_1', I_2'\}\rangle$. Since the bound sequence is the same in both states, every conflict $I \in C'$ appearing in state S also appears in state S'. By the definition of the Solve-Div-Left rule, I_2' is no conflict in state S. Note that div-solve is an equivalence preserving transformation. Thus, if I_1' were a conflict in S and $\text{top}(I_1') = x_i$ fixed by a Decision, then I_1 or I_2 is a conflict in S'. Therefore, I_1' is no conflict or $\text{top}(I_1') = x_i$ has no decided bound. Now it is easy to see that S is eager top-level propagated because S' was eager top-level propagated.

9. Let the applied rule be Conflict or Conflict-Div. Then $S' = \langle M', C'\rangle$ and $S = \langle M', C', I\rangle$, where I is a conflict. It is easy to see that S is eager top-level propagated because S' is eager top-level propagated.

10. Let the applied rule be Resolve or Skip-Decision. Then $S' = \langle [\![M, \gamma]\!], C', J'\rangle$ and $S = \langle M, C', J\rangle$, where J' and J are conflicts in S' and S, respectively. Since M is a prefix of M', every conflict $I \in C'$ appearing in state S also appears in state S'. Now it is easy to see that S is eager top-level propagated because S' was eager top-level propagated.

11. Let the applied rule be Learn. Then $S' = \langle [\![M', \gamma]\!], C', I\rangle$ and $S = \langle M', C' \cup I, I\rangle$, where I is a conflict. Since CutSat++ uses a two-layered strategy (Definition 3.5.7), I is a guarded constraint. Now it is easy to see that S is eager top-level propagated because S' was eager top-level propagated.

12. Let the applied rule be Backjump. Then $S' = \langle [\![M', \gamma, M'']\!], C', I\rangle$ and $S = \langle [\![M', \gamma']\!], C'\rangle$, where I is a conflict in S'. Since CutSat++ uses a two-layered strategy (Definition 3.5.7), I is a guarded constraint. Now it is easy to see that S is eager top-level propagated because S' was eager top-level propagated. □

Since the start state $\langle [\![]\!], C_0\rangle$ trivially fulfills the eager top-level propagated properties, it is clear that CutSat++ produces only eager top-level states; except for the end states. The eager top-level propagated property is so important because we will use it to show that CutSat++ resolves any conflict it encounters. In case the conflict is a guarded constraint, this is done with the CDCL based conflict rules. Otherwise, the conflict I is an unguarded constraint and CutSat++ simulates weak Cooper elimination with the

unguarded conflict resolution rules. First, we use Solve-Div-Left to simulate the algorithm in Figure 3.8. This either ends with a call to Solve-Div-Right resolving the conflict or CutSat++ finds a conflicting core. Then the conflicting core is resolved with the rule Resolve-Weak-Cooper.

Lemma 3.6.9 (Conflicts Progress)**.** *Let $S = \langle M, C\rangle$ be a state reachable by* CutSat++*. Let $I \in C$ be a conflict in state S. Then state S is not stuck.*

Proof. Assume for a contradiction that state S is stuck. W.l.o.g., we assume that $x_i = \text{top}(I)$ is the smallest variable in our order that is the top variable in a conflicting constraint $I' \in C$. If x_i is a guarded variable, then Conflict or Conflict-Div is applicable, which contradicts our initial assumption! Therefore, x_i is an unguarded variable. Furthermore, all variables x_j smaller than x_i are fixed. Otherwise, we deduce for the smallest unfixed variable x_j that either

- x_j is stuck and Slack-Intro is applicable
- Propagate is applicable to a constraint I' where $\text{top}(I') = x_j$
- C contains at least two divisibility constraints I_1, I_2 that have x_j as their top variable and Solve-Div-Left or Solve-Div-Right is applicable
- S contains a diophantine conflicting core (x_j, I_d) and Resolve-Weak-Cooper is applicable
- Decide is applicable to x_j because all conditions in Definition 3.5.6-(2) are fulfilled

Since S is eager top-level propagated and I is a conflict with top variable x_i, we know that state S contains no decided bound for x_i (Definition 3.6.7 and Lemma 3.6.8). W.l.o.g., we assume that C contains at most one divisibility constraint I_d with x_i as its top variable. Otherwise, Solve-Div-Left or Solve-Div-Right are applicable, which contradicts our initial assumption! Let $x_i \geq l_i$ be the strictest lower bound $l_i = \text{bound}(x_i, I_l, \geq, M)$ for an inequality $I_l \in C$ with top variable x_i or $l_i = -\infty$ if there is no inequality propagating a lower bound. Let $x_i \leq u_i$ be the strictest upper bound $u_i = \text{bound}(x_i, I_u, \leq, M)$ for an inequality $I_u \in C$ with top variable x_i or $u_i = \infty$ if there is no inequality propagating an upper bound. Since the strictly-two-layered strategy forbids the application of Forget to unguarded constraints, CutSat++ never removes an unguarded inequality. Furthermore, any bound $x_i \bowtie b_i$ ($\bowtie \in \{\leq, \geq\}$) propagated from a divisibility constraint requires another bound $x_i \bowtie b_i'$ propagated from an inequality. We deduce that $u_i \neq \infty$ if $\mathcal{U}(x_i, M) \neq \infty$ and $l_i \neq -\infty$ if $\mathcal{L}(x_i, M) \neq -\infty$. Next, we do a case distinction on whether the bounds u_i and l_i are finite:

1. Let $u_i = \infty$ and $l_i = -\infty$. Then $\mathcal{L}(a_i x_i + p_i) = -\infty$ holds for all inequalities $a_i x_i + p_i \leq 0$. Thus, the conflict I is no inequality. A divisibility constraint is a conflict only if $\mathcal{L}(x_i, M) \neq -\infty$ and $\mathcal{U}(x_i, M) \neq \infty$. This contradicts the assumption that I is a conflict.

2. Let $u_i = \infty$ and $l_i \in \mathbb{Z}$. Then $\mathcal{L}(a_i x_i + p_i) = -\infty$ holds for all inequalities $a_i x_i + p_i \leq 0$ with $a_i < 0$ and there exists no inequality $J = a_i x_i + p_i \leq 0$ in C with $a_i > 0$ and $\text{top}(J) = x_i$. Thus, the conflict I is not an inequality. A divisibility constraint is a conflict only if $\mathcal{L}(x_i, M) \neq -\infty$ and $\mathcal{U}(x_i, M) \neq \infty$. This contradicts the assumption that I is a conflict.

3. Let $l_i = -\infty$ and $u_i \in \mathbb{Z}$. Then $\mathcal{L}(a_i x_i + p_i) = -\infty$ holds for all inequalities $a_i x_i + p_i \leq 0$ with $a_i > 0$ and there exists no inequality $J = a_i x_i + p_i \leq 0$ in C with $a_i < 0$ and $\text{top}(J) = x_i$. Thus, the conflict I is not an inequality. A divisibility constraint is a conflict only if $\mathcal{L}(x_i, M) \neq -\infty$ and $\mathcal{U}(x_i, M) \neq \infty$. This contradicts the assumption that I is a conflict.

4. Let $l_i, u_i \in \mathbb{Z}$ and $u_i < l_i$. Then $(x_i, \{I_l, I_u\})$ is a conflicting core and Resolve-Weak-Cooper is applicable. This contradicts the assumption that no rule is applicable.

5. Let $l_i, u_i \in \mathbb{Z}$ and $l_i \leq u_i$. Then the conflict I must be the sole divisibility constraint I_d for x_i. If $(x_i, \{I_l, I_u, I_d\})$ is a conflicting core, then Resolve-Weak-Cooper is applicable contradicting our initial assumption. Therefore, there exists a solution $v_i \in \{l_i, \ldots, u_i\}$ for x_i satisfying I_d. Let D be the set of divisibility constraints used to propagate a bound for x_i in M. All constraints $D' \subseteq D$ not contained in C, i.e., $D' = D \setminus C = D \setminus \{I_d\}$, were eliminated with div-solve. Since div-solve preserves equivalence, it is easy to see that there exists a set of constraints $D^* = D^{**} \cup \{I_d\}$ contained in C that implies satisfiability of D:

$$D^* = D^{**} \cup \{I_d\} \to D,$$

and D^{**} contains only variables x_j smaller than x_i. However, the set of divisibility constraints D^* is fixed and satisfied under the partial assignment of M in state S. Otherwise, S would contain a conflict $I' \in D^* \subseteq C$ with $\text{top}(I') \prec x_i$, which contradicts our initial assumption! Thus, setting x_i to the solution $v_i \in \{l_i, \ldots, u_i\}$ satisfies I_d in M and also $D \cup \{I_d\}$. Furthermore, all propagated constraints are satisfied if x_i is set to v_i:

$$\mathcal{L}(x_i, M) \leq v_i \leq \mathcal{U}(x_i, M).$$

This contradicts the assumption that there exists a conflict I with $\text{top}(I) = x_i$. □

The remainder of the proof follows directly the proof outline from above:

Theorem 3.6.10 (CUTSAT++ Progress)**.** *Let $S = \langle M, C, J\rangle$ be a state reachable by* CutSat++*. Then S is not stuck.*

Proof. Assume for a contradiction that $S = \langle M, C, J\rangle$ is a stuck state. If CutSat++ is a conflict state, i.e., $J \neq \top$, then the proof for Theorem 2 in [87] shows why standard conflict resolution cannot get stuck on a state $S = \langle M, C, J\rangle$. Therefore, $S = \langle M, C, J\rangle$ must be a search state $S = \langle M, C\rangle$. Next, we assume that all guarded variables are fixed because CutSat++ can propagate at least two bounds for every guarded variable and afterwards use decided bounds to fix them. By Lemma 3.6.9, there is no conflict in state

S. Since there is no conflict, at least one variable is unfixed or rule Sat would be applicable. Therefore, there must exist a smallest unfixed and unguarded variable x_i. With the Slack-Intro rules CutSat++ introduces for all variables at least one lower or upper bound. Therefore, there exists a violation to the conditions in Definition 3.5.6-(2) or Decide would be applicable to x. Since x is the smallest unfixed variable, the condition in Definition 3.5.6-(2a) holds. Definition 3.5.6-(2c) is also easy to satisfy by applications of Solve-Div-Left and Solve-Div-Right. Therefore, Definition 3.5.6-(2b) is violated. Thus, there exists a constraint $I \in C$ that is a conflict in $S' = \langle [\![M, \gamma]\!], C \rangle$, where γ is a decided bound for $x_i = \text{top}(I)$. By Lemma 3.6.9, it is not possible that $I \in C$ is also a conflict in S or S would not be stuck. Finally, I is a conflict only in S' and not S if Propagate(-Div) is applicable to I. With Solve-Div-Left and Solve-Div-Right it is relatively easy to fulfil the conditions for Definition 3.5.6-(1) and, therefore, Propagate(-Div) is applicable. We conclude that CutSat++ has always one applicable rule, which is a contradiction to our assumption! □

3.6.3 Completeness

All CutSat++ rules are sound, i.e., if $\langle M_i, C_i, I_i \rangle \Longrightarrow_{\text{CS}} \langle M_j, C_j, I_j \rangle$, then any satisfiable assignment ν for C_j is also a satisfiable assignment for C_i. The rule Resolve-Weak-Cooper is sound because of the Lemmas 3.4.4 and 3.4.5. The soundness of Solve-Div-Left and Solve-Div-Right follows from the fact that div-solve is an equivalence preserving transformation. The soundness proofs for all other rules are either trivial or given by [87].

Summarizing, CutSat++ is terminating, sound, and never reaches a stuck state. In combination with the fact that Sat is applicable only if a satisfiable solution $\nu[M]$ is found and that Unsat and Unsat-Div detect trivially unsatisfiable constraints, these facts imply completeness:

Theorem 3.6.11 (CUTSAT++ Completeness)**.** *If* CutSat++ *starts from a start state* $\langle [\![]\!], C_0 \rangle$*, then it either terminates in the* unsat *state and* C_0 *is unsatisfiable, or it terminates with* $\langle \nu, \text{sat} \rangle$ *where* ν *is a satisfiable assignment for* C_0*.*

Proof. By Theorem 3.6.6, CutSat++ is terminating. By [87] and the Lemmas 3.2.1, 3.4.4, and 3.4.5, CutSat++ is sound. By Theorem 3.6.10, CutSat++ never reaches a stuck state. Since CutSat++ is terminating and never reaches a stuck state, every application of CutSat++ ends via the rules Sat, Unsat, or Unsat-Div in one of the end states. The rule Sat is only applicable in a state $\langle M, C \rangle$ where $\nu[M]$ satisfies C and because of soundness also C_0. The rules Unsat and Unsat-Div are only applicable to states

$\langle M, C, I\rangle$ where the constraint set C contains a trivially unsatisfiable constraint. When CutSat++ encounters a trivially unsatisfiable constraint, then the soundness of the CutSat++ rules guarantees that C_0 is unsatisfiable. □

3.7 Summary

The starting point of our work was an implementation of the CutSat [87] calculus as a theory solver for hierarchic superposition [64]. In that course, we observed divergence for some of our problems. The analysis of those divergences led to the development of the CutSat++ calculus presented in this chapter, which is, as far as we know, the first sound, complete, and terminating calculus for linear integer problems based on the model assumption and conflict learning approach motivated by CDCL style SAT solving.

CutSat++ efficiently handles problems over guarded variables, i.e., variables with a constant upper and lower bound. On problems with unguarded variables (e.g. unbounded problems) the CDCL style calculus alone is not guaranteed to terminate. Hence, we combine it with a lazy quantifier elimination procedure (called unguarded conflict resolution) that transforms a problem containing unguarded variables into one where feasibility depends only on guarded ones. The quantifier elimination procedure is called lazy because we only apply it to so-called conflicting cores, which are canonized forms of the unguarded conflicts encountered by the CDCL style algorithm. This allows us to avoid certain cases of worst-case exponential behavior that we would otherwise observe by using a quantifier elimination procedure alone.

Chapter 4

Fast Cube Tests (for Linear Arithmetic Constraint Solving)

Finding a mixed/integer solution for a polyhedron that is defined by a system of linear inequalities $Ax \leq b$ is a well-known NP-complete problem [119]. Systems of linear inequalities have many real-world applications so that this problem has been investigated in different research areas, e.g., in optimization via *(mixed) integer linear programming* (MILP) [89, 92, 109, 128] and in constraint solving via satisfiability modulo theories (SMT) [26, 32, 52, 76].

For commercial MILP implementations, it is standard to integrate preprocessing techniques, heuristics, and specialized tests [71, 72, 82, 89, 127, 128, 136]. Although these techniques are not complete, they are much more efficient on their designated target systems of linear inequalities than a complete algorithm alone. There actually exist specialized techniques for many classes of real-world problems representable as systems of linear inequalities. Therefore, commercial MILP solvers are efficient on many real-world inputs—even though the problem is NP-complete, in general.

The SMT community is still in the process of developing their own variety of specialized tests. It is even a big challenge to adopt the tests from the MILP community so that they still fit the requirements of SMT solving. One of those requirements is that SMT theory solvers have to solve a large number of incrementally connected, small systems of linear inequalities efficiently. Therefore, exploiting the incremental connection is key for making SMT theory solvers efficient [61]. In contrast, MILP solvers typically target one large system. The same holds for their specialized tests, which are not well suited to exploit incremental connections.

In this chapter, we present two tests tailored toward SMT solvers: the *largest cube test* and the *unit cube test*. The idea is to find hypercubes that are contained inside the input polyhedron and guarantee the existence of a mixed/integer solution. Due to computational complexity, we will restrict ourselves to only those hypercubes that are parallel to the coordinate axes. The largest cube test finds a hypercube with maximum edge length contained in the input polyhedron, determines its rational valued center, and rounds it to a potential mixed/integer solution. The unit cube test determines if a polyhedron contains a hypercube with edge length one, which is the minimal edge length that guarantees a mixed/integer solution.

Most linear arithmetic theory solvers for SMT are based on a branch-and-bound algorithm on top of a simplex algorithm (see Chapter 2.7). They search for a solution at the surface of a polyhedron. However, our tests search in the interior of the polyhedron. This gives them an advantage on polyhedra with a large number of integer solutions, e.g., absolutely unbounded inequality systems (see Chapter 2.8).

SMT theory solvers are designed to efficiently exchange bounds (see Chapter 2.7 and [57]). This efficient exchange is the main reason why SMT theory solvers exploit the incremental connection between the different inequality systems so well. Our unit cube test also requires only an exchange of bounds. After applying the test, we can easily recover the original system by reverting to the original bounds. In doing so, the unit cube test conserves the incremental connection to the different original systems. We make a similar observation about the largest cube test.

Our contributions are as follows: we define the linear cube transformation (Corollary 4.2.2) that allows us to efficiently compute whether a system of inequalities $Ax \leq b$ contains a hypercube of edge length e in Section 4.2. The most remarkable fact about this transformation is that it solely changes the bounds b of the inequalities. Based on this transformation, we develop in Section 4.3 two tests: the largest cube test and the unit cube test. For absolutely unbounded inequality systems, both tests always succeed (Lemma 4.4.1). Inside the SMT-LIB benchmarks [10], there are almost one thousand absolutely unbounded problem instances, and we show the advantage of our cube tests on these instances by comparing our implementation of the cube test with several state-of-the-art SMT solvers in Section 4.5. Our implementation is not only several orders of magnitudes faster, but it also solves all instances, which most SMT solvers do not (Figure 4.8).

4.1 Related Work and Preliminaries

This chapter is based on two publications with Christoph Weidenbach as co-author [34, 35] and the techniques in this chapter focus on the interaction of geometric objects. In the case of a system of inequalities, this means the polyhedron defined by $Ax \leq b$. This is the reason why we use in this chapter polyhedron as an alternative name for systems of inequalities $Ax \leq b$. The other geometric objects we are considering are cubes and flat cubes. Their definitions can be found in Chapter 2.9.

This chapter focuses on all three theories of linear arithmetic. Our tests themselves are formulated as problems in the theory of linear rational arithmetic. However, the goal of our tests is to find mixed/integer solutions for problems in the theory of linear mixed/integer arithmetic. We first present all of our techniques for linear integer arithmetic. In Section 4.3.3, we then extend them to linear mixed arithmetic. To avoid confusion between the various theories, we always specify the type of solution/satisfiability.

The constraints in this chapter are non-strict inequalities and they are either formatted according to the vector representation, i.e., $a_i^T x \leq b_i$ (see also Chapter 2.2.1), or the standard representation, $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ (see also Chapter 2.2.1). Other constraints have to be reduced to non-strict inequalities with the techniques presented in Chapter 2.3.

This chapter builds on the basics of linear algebra (Chapter 2.1) and linear arithmetic (Chapter 2.2), on the concept of implied constraints (Chapter 2.5), and on the definitions of (un)bounded and (un)guarded problems and variables (Chapter 2.8). Knowledge of standard arithmetic decision procedures for SMT solvers (Chapter 2.7) is not required to understand the tests that we propose here, but it is necessary to fully understand the practical usefulness of the tests with regard to SMT solving.

There also exist several publications by other authors that are highly relevant to the contributions presented in this chapter. The first one is by Hillier [82], who was aware of the unit cube test, but applied it only to cones (a special class of polyhedra) as a subroutine of the heuristic he presents in the same paper. His work never mentioned applications beyond cones, nor did he prove any structural properties connected to hypercubes. Hillier's heuristic tailored for MILP optimization lost popularity as soon as interior point methods became efficient in practice [93]. Nonetheless, our cube tests remain relevant for SMT theory solvers because there are no competitive incremental interior point methods known.

The second related work is by Bobot et al. [26]. They discuss in their paper [26] relations between hypercubes and polyhedra including absolute unboundedness and positive linear combinations between inequalities. Our largest cube test can also detect these relations because it is, with some minor changes, the dual of the linear optimization problem of Bobot et al.

In contrast to the linear optimization problem of Bobot et al., our tests are closer to the original polyhedron and, therefore, easier to construct. Our cube tests also produce sample points and find solutions for polyhedra that are bounded.

Another method that provides a sufficient condition for the existence of an integer solution is the dark shadow of the Omega Test [122]. The dark shadow is based on Fourier-Motzkin elimination and its worst-case runtime is double exponential. Although not practically advantageous, formulating the unit cube test through Fourier-Motzkin elimination allows us to put the sufficient conditions of the two methods in relation. Fourier-Motzkin elimination eliminates the variable x_j from a problem by combining each pair of inequalities $a_j x_j + p_j \leq 0$ and $-b_j x_j + q_j \leq 0$ (with $a_j, b_j > 0$) into a new inequality $a_j q_j + b_j p_j \leq 0$. The dark shadow creates a stronger version ($a_j q_j + b_j p_j \leq a_j + b_j - a_j b_j$) of the combined inequality to guarantee the existence of an integer solution for x_j. Formulating the unit cube test through Fourier-Motzkin elimination makes the combined inequality even stronger ($a_j q_j + b_j p_j \leq -a_j b_j$). This means that the sufficient condition of the dark shadow subsumes the condition of the unit cube test. Still, our unit cube test is definable as a linear program and it is, therefore, computable in polynomial time. So the better condition of the dark shadow comes at the cost of being much harder (doubly exponential) to compute.

4.2 Fitting Cubes into Polyhedra

We say that a cube $\mathcal{C}_e(z)$ (see Chapter 2.9.2 for the formal definition) *fits* into a polyhedron defined by $Ax \leq b$ if all points inside the cube $\mathcal{C}_e(z)$ are solutions of $Ax \leq b$, or formally: $\mathcal{C}_e(z) \subseteq \mathcal{Q}_\delta(Ax \leq b)$. In order to compute this, we transform the polyhedron $Ax \leq b$ into another polyhedron $Ax \leq b'$. For this new polyhedron, we merely have to test whether the cube's center point z is a solution ($z \in \mathcal{Q}_\delta(Ax \leq b')$) in order to also determine whether the cube $\mathcal{C}_e(z)$ fits into the original polyhedron. This is a simple test that requires only evaluation. We call this entire transformation the *linear cube transformation*.

We start explaining the linear cube transformation by looking at the case where the polyhedron is defined by a single inequality $a_i^T x \leq b_i$. A cube $\mathcal{C}_e(z)$ fits into the inequality $a_i^T x \leq b_i$ if all points inside the cube $\mathcal{C}_e(z)$ are solutions of $a_i^T x \leq b_i$, or formally: $\forall x \in \mathcal{C}_e(z).\ a_i^T x \leq b_i$.

We can think of $a_i^T x$ as an objective function that we want to maximize and see b_i as a guard for the maximum objective of any solution in the cube. Thus, we can express the universal quantifier in the above equation as an optimization problem (see Figure 4.1): $\max\{a_i^T x : x \in \mathcal{C}_e(z)\} \leq b_i$. This also means that all points in $x \in \mathcal{C}_e(z)$ satisfy the inequality $a_i^T x \leq b_i$ if a

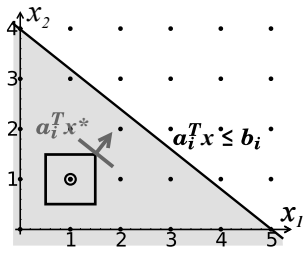


Figure 4.1: A square (two-dimensional cube) fitting into an inequality $a_i^T x \leq b_i$ and the cube's maximum $a_i^T x^*$ for the objective $a_i^T x$

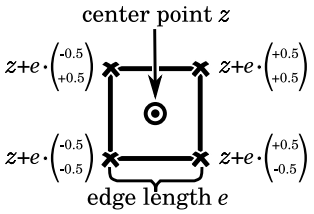


Figure 4.2: The vertices of an arbitrary axis-parallel square (two-dimensional cube with edge length e and center z)

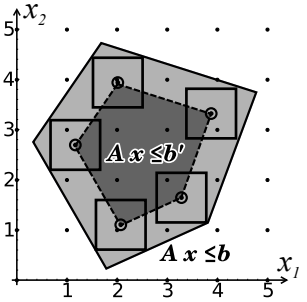


Figure 4.3: The transformed polyhedron $Ax \leq b'$ for edge length 1 together with the original polyhedron $Ax \leq b$

point $x^* \in \mathcal{C}_e(z)$ with maximum value $a_i^T x^* = \max\{a_i^T x : x \in \mathcal{C}_e(z)\}$ for the objective function $a_i^T x$ satisfies the inequality $a_i^T x^* \leq b_i$. We can formalize the above optimization problem as a linear program:

$$\begin{aligned} &\text{maximize} && a_i^T x \\ &\text{subject to} && z_j - \tfrac{e}{2} \leq x_j \leq z_j + \tfrac{e}{2} \quad \text{for } j = 1, \ldots, n\,. \end{aligned}$$

However, for the case of cubes, there is an even easier way to determine the maximum objective value. Since every cube is a bounded polyhedron, one of the points with maximum objective value is a vertex $v \in \mathcal{C}_e(z)$. A *vertex* v of the cube $\mathcal{C}_e(z)$ is one of the points with maximum distance to the center z (see Figure 4.2), or formally: $v = \left(z_1 \pm \frac{e}{2}, \ldots, z_n \pm \frac{e}{2}\right)^T$. If we insert the above equation into the objective function $a_i^T x$, we get:

$$a_i^T \left(z_1 \pm \tfrac{e}{2}, \ldots, z_n \pm \tfrac{e}{2}\right)^T = a_i^T z + \tfrac{e}{2} \textstyle\sum_{j=1}^n \pm a_{ij}\,,$$

which in turn is maximal if we choose v such that $\pm a_{ij}$ is always positive:

$$a_i^T v = a_i^T z + \tfrac{e}{2} \textstyle\sum_{j=1}^n |a_{ij}| = a_i^T z + \tfrac{e}{2} \left\|a_i\right\|_1\,.$$

Hence, we transform the inequality $a_i^T x \leq b_i$ into $a_i^T x \leq b_i - \frac{e}{2} \left\|a_i\right\|_1$, and $\mathcal{C}_e(z)$ fits into $a_i^T x \leq b_i$ if $a_i^T z \leq b_i - \frac{e}{2} \left\|a_i\right\|_1$.

Lemma 4.2.1 (Linear Cube Transformation)**.** *Let $\mathcal{C}_e(z)$ be a cube and $a_i^T x \leq b_i$ be an inequality. All $x \in \mathcal{C}_e(z)$ fulfill $a_i^T x \leq b_i$ if and only if $a_i^T z \leq b_i - \frac{e}{2} \left\|a_i\right\|_1$.*

Proof. $\Rightarrow$: Assume that all $x \in \mathcal{C}_e(z)$ fulfill

$$a_i^T x \leq b_i\,. \tag{4.1}$$

Since $\frac{e}{2} = \left|\frac{e}{2} \cdot \operatorname{sgn}(a_{ij})\right| \leq \frac{e}{2}$, it follows that the cube $\mathcal{C}_e(z)$ contains

$$v := \tfrac{e}{2} \left(\operatorname{sgn}(a_{i1}), \ldots, \operatorname{sgn}(a_{in})\right)^T + z\,. \tag{4.2}$$

Thus,

$$\begin{aligned} b_i &\geq^{(4.1)} a_i^T v =^{(4.2)} \tfrac{e}{2} a_i^T \left(\operatorname{sgn}(a_{i1}), \ldots, \operatorname{sgn}(a_{in})\right)^T + a_i^T z \\ &= \tfrac{e}{2}\left(\textstyle\sum_{j=1}^n a_{ij} \cdot \operatorname{sgn}(a_{ij})\right) + a_i^T z = \tfrac{e}{2}\left(\textstyle\sum_{j=1}^n |a_{ij}|\right) + a_i^T z\,. \end{aligned}$$

$\Leftarrow$: Assume that

$$\tfrac{e}{2}\textstyle\sum_{j=1}^{n} |a_{ij}| + a_i^T z \leq b_i \,. \tag{4.3}$$

By the definition of the cube $\mathcal{C}_e(z)$, it follows that for all points $x \in \mathcal{C}_e(z)$, there exists a vector $d \in \mathbb{R}^n$ such that $d_j \in [-1, 1]$ and

$$z := x - \tfrac{e}{2} d. \tag{4.4}$$

For $d_j \in [-1, 1]$, it follows that

$$|a_{ij}| - a_{ij} d_j \geq 0 \,. \tag{4.5}$$

Hence,

$$\begin{aligned} b_i &\geq^{(4.3)} \tfrac{e}{2}\textstyle\sum_{j=1}^{n} |a_{ij}| + a_i^T z =^{(4.4)} \tfrac{e}{2}\textstyle\sum_{j=1}^{n} |a_{ij}| + a_i^T x - \tfrac{e}{2} a_i^T d \\ &= \tfrac{e}{2}\textstyle\sum_{j=1}^{n} (|a_{ij}| - a_{ij} d_j) + a_i^T x \geq^{(4.5)} a_i^T x \,. \end{aligned}$$

□

Next, we look at the case where multiple inequalities $a_i^T x \leq b_i$ (for $i = 1, \ldots, m$) define the polyhedron $Ax \leq b$. Since $\mathcal{Q}_\delta(Ax \leq b)$ is the intersection of all $\mathcal{Q}_\delta(a_i^T x \leq b_i)$, the cube fits into $Ax \leq b$ if and only if it fits into all inequalities $a_i^T x \leq b_i$:

$$\forall i \in \{1, \ldots, m\}.\ \forall x \in \mathcal{C}_e(z).\ a_i^T x \leq b_i \,.$$

We can express this by m optimization problems:

$$\forall i \in \{1, \ldots, m\}.\ \max\{a_i^T x : x \in \mathcal{C}_e(z)\} \leq b_i$$

and, after applying Proposition 4.2.1, by the following m inequalities:

$$\forall i \in \{1, \ldots, m\}.\ a_i^T z \leq b_i - \tfrac{e}{2} \left\|a_i\right\|_1 \,.$$

Hence, the linear cube transformation transforms the polyhedron $Ax \leq b$ into the polyhedron $Ax \leq b'$, where $b'_i = b_i - \frac{e}{2} \left\|a_i\right\|_1$, and $\mathcal{C}_e(z)$ fits into $Ax \leq b$ if $Az \leq b'$.

Corollary 4.2.2 (Linear Cube Transformation)**.** *Let $\mathcal{C}_e(z)$ be a cube and $Ax \leq b$ be a polyhedron. $\mathcal{C}_e(z) \subseteq \mathcal{Q}_\delta(Ax \leq b)$ if and only if $Az \leq b'$, where $b'_i = b_i - \frac{e}{2} \left\|a_i\right\|_1$.*

Until now, we have discussed how to use the linear cube transformation to determine if one cube $\mathcal{C}_e(z)$ with fixed center point z fits into a polyhedron. A generalization of this problem determines whether a polyhedron $Ax \leq b$ contains a cube of edge length e at all. Actually, a closer look at the transformed polyhedron $Ax \leq b'$ reveals that the linear cube transformation ($b'_i = b_i - \frac{e}{2} \left\|a_i\right\|_1$) is dependent only on the edge length e of the cube. Therefore, the solutions $\mathcal{Q}_\delta(Ax \leq b')$ of the transformed polyhedron $Ax \leq b'$ are exactly all center points of cubes with edge length e that fit into the original polyhedron $Ax \leq b$ (see Figure 4.3). By determining the rational satisfiability of the transformed polyhedron $Ax \leq b'$, we can now also determine whether a polyhedron $Ax \leq b$ contains a cube of edge length e at all. If we choose a suitable algorithm, e.g., the simplex algorithm, then we even get the center point z of a cube $\mathcal{C}_e(z)$ that fits into $Ax \leq b$. This observation is the foundation for the cube tests that we present in Section 4.3.

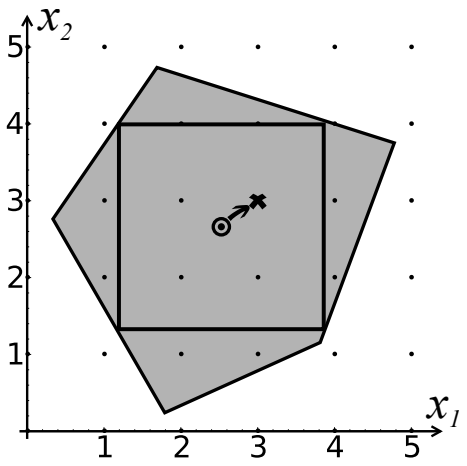


Figure 4.4: The largest cube inside a polyhedron, its center point, and a closest integer point to the center

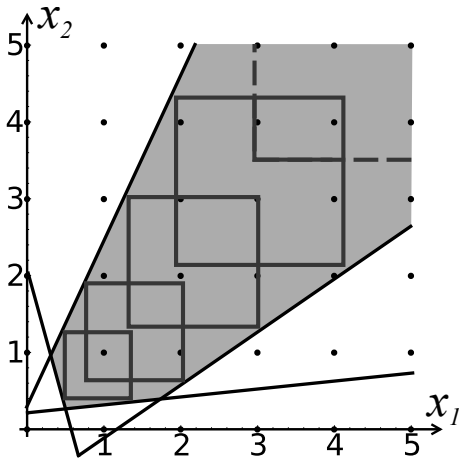


Figure 4.5: An absolutely unbounded polyhedron containing cubes for every edge length $e > 0$.

4.3 Fast Cube Tests

A polyhedron $Ax \leq b$ has an integer solution if and only if $\mathcal{Z}(Ax \leq b) = \mathcal{Q}_\delta(Ax \leq b) \cap \mathbb{Z}^n \neq \emptyset$, i.e., if the set of rational solutions contains an integer point. In this section, we show how to use the linear cube transformation to find such an integer solution. In contrast to arbitrary polyhedra, determining whether a cube $\mathcal{C}_e(z)$ contains an integer point is easy. Because of the cubes symmetry, it is enough to test whether it contains a closest integer point $\lceil z \rfloor$ to the center z (see also Corollary 2.9.2).

Note that every point $z \in \mathbb{Q}_\delta^n$ is also a cube $\mathcal{C}_0(z)$ of edge length 0. In order to be efficient, our tests look only at cubes with special properties. In the case of the largest cube test, we check for an integer solution in one of the largest cubes fitting into the polyhedron $Ax \leq b$. In the case of the unit cube test, we look for a cube of edge length one, which always guarantees an integer solution. Due to these restrictions, both tests are not complete but very fast to compute.

4.3.1 Largest Cube Test

A well-known test, implemented in most ILP solvers, is simple rounding (see also Chapter 2.1.2 & 2.7.4). For simple rounding, the ILP solver computes a rational solution x for a set of inequalities, *rounds* it to a closest integer $\lceil x \rfloor$, and determines whether this point is an integer solution. Not all types of rational solutions are good candidates for this test to be successful. Especially *surface points*, such as *vertices*, are not good candidates for roun-

ding. Vertices are, however, the usual output of the simplex algorithm. For many polyhedra, *center and interior points* z are a better choice because all integer points adjacent to z are solutions, including a closest integer point $\lceil z \rfloor$.

We now use the linear cube transformation (Section 4.2) to calculate a rational center point with the simplex algorithm. The center point we calculate is the center point of a largest cube that fits into the polyhedron $Ax \leq b$ (see Figure 4.4). We determine the center z of this largest cube and the associated edge length e with the following linear program (LP):

$$\begin{array}{ll} \text{maximize} & x_e \\ \text{subject to} & Ax + a' \frac{x_e}{2} \leq b, \text{ where } a'_i = \|a_i\|_1 \\ & x_e \geq 0\,. \end{array}$$

This linear program employs the linear cube transformation from Section 4.2. The only generalization is a variable x_e for the edge length instead of a constant value e. Additionally, this linear program maximizes the edge length as an optimization goal. If the resulting maximum edge length is unbounded, the original polyhedron contains cubes of arbitrary edge length (see Figure 4.5) and, thus, infinitely many integer solutions. Since the linear program contains all rational solutions of the original polyhedron (see $x_e = 0$), the original polyhedron is empty if and only if the above linear program is rationally unsatisfiable. If the maximum edge length is a finite value e, we use the resulting assignment z for the variables x as a center point and $\mathcal{C}_e(z)$ is a largest cube that fits into the polyhedron. From the center point, we round to a closest integer point $\lceil z \rfloor$ and determine if it fits into the original polyhedron. If this is the case, we are done because we have found an integer solution for $Ax \leq b$. Otherwise, the largest cube test does not know whether or not $Ax \leq b$ has an integer solution. An example for the latter case, are the following inequalities: $-2x_1 - 2x_2 \leq -3$, $4x_1 - x_2 \leq 1$, and $-3x_1 + 2x_2 \leq 3$. These inequalities have exactly one integer solution $(1, 3)^T$, but the largest cube contained by the inequalities has edge length $e = \frac{5}{12}$ and center point $(\frac{3}{8}, \frac{37}{24})^T$, which rounds to $(0, 2)^T$ (see Figure 4.6).

The largest cube test also upholds the incremental advantages of Dutertre and de Moura's version of the dual simplex algorithm (see also Chapter 2.7.1 and [57]). The only difference is the extra column $a' \frac{x_e}{2}$, which the theory solver can internally create while it is notified of all potential arithmetic literals. Adding this column from the start does not influence the correctness of the solution because $x_e \geq 0$ guarantees that the largest cube test is rationally satisfiable exactly when the original inequalities $Ax \leq b$ are rationally satisfiable. Even for explanations of rational unsatisfiability, it suffices to remove the bound $x_e \geq 0$ to obtain an explanation for the original inequalities $Ax \leq b$. The only disadvantage is the additional variable x_e, which only shrinks the search space when it is increased. Therefore, increasing x_e can never resolve any conflicts during the satisfiability search. The

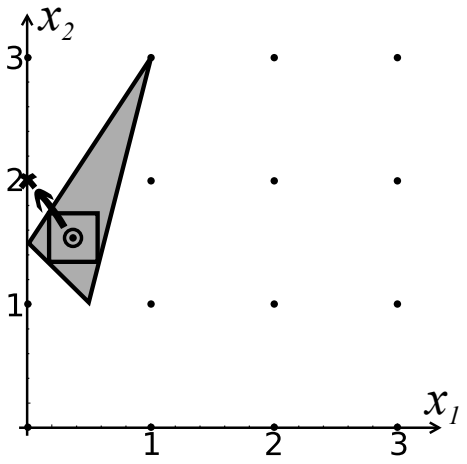


Figure 4.6: A polyhedron for which the cube tests fail

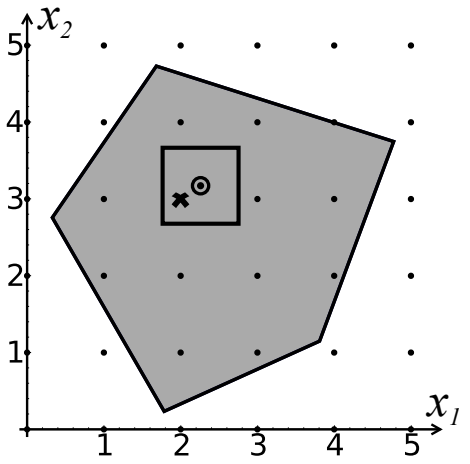


Figure 4.7: A unit cube inside a polyhedron, its center point, and a closest integer point to the center

simplex solver recognizes this with at least one additional pivot that sets x_e to 0. Hence, adding the extra column $a' \frac{x_e}{2}$ from the beginning has only constant influence on the theory solver's runtime, and is therefore negligible.

4.3.2 Unit Cube Test

Most SMT solvers [9, 40, 41, 49, 56] implement a simplex algorithm (see also Chapter 2.7.1 and [57]) that is specialized towards satisfiability and not towards optimization. Therefore, a test based on optimization, such as the largest cube test, does not fit well with existing implementations. As an alternative, we have developed a second test based on cubes that does not need optimization.

We avoid optimization by fixing the edge length e to the value 1 for all the cubes $\mathcal{C}_e(z)$ we consider (see Figure 4.7). We do so because cubes $\mathcal{C}_1(z)$ of edge length 1 are the smallest cubes to always guarantee an integer solution independent of the center point z. A cube with edge length 1 is also called a *unit cube*. To prove this guarantee, we first fix $e = 1$ in the definition of cubes, $\mathcal{C}_1(z) = \left\{x \in \mathbb{Q}_\delta^n : \forall j \in 1, \ldots, n.\ |x_j - z_j| \leq \frac{1}{2}\right\}$, and look at the following property for the rounding operator $\lceil . \rfloor$: $\forall z_j \in \mathbb{Q}_\delta . |\lceil z_j \rfloor - z_j| \leq \frac{1}{2}$. We see that any unit cube contains a closest integer $\lceil z \rfloor$ to its center point z. Furthermore, 1 is the smallest edge length that guarantees an integer solution for a cube with center point $z = (\ldots, \frac{1}{2}, \ldots)^T$. Thus, 1 is the smallest value that we can fix as an edge length to guarantee an integer solution for all cubes $\mathcal{C}_1(z)$.

Our second test tries to find a unit cube that fits into the polyhedron $Ax \leq b$ and, thereby, a guarantee for an integer solution for $Ax \leq b$. Again, we employ the linear cube transformation from Section 4.2 and obtain the linear program:

$$Az \quad \leq b', \text{ where } b'_i = b_i - \tfrac{1}{2} \left\| a_i \right\|_1 .$$

In addition to being a linear program without an optimization objective, we only have to change the row bounds b'_i of the original inequalities. In Dutertre and de Moura's version of the dual simplex algorithm (see also Chapter 2.7.1 and [57]), which is implemented in many SMT solvers [9, 40, 41, 49, 56], such a change of bounds is already part of the framework so that integrating the unit cube test into theory solvers is possible with only minor adjustments to the existing implementation. Since our unit cube test requires only an exchange of bounds, we can easily return to the original polyhedron by reverting the bounds. In doing so, the unit cube test upholds the incremental connection between the different original polyhedra.

4.3.3 Cube Tests for Linear Mixed Arithmetic

We can also extend our cube tests to the theory of linear mixed arithmetic. As mentioned in Chapter 2.2, our variables $x = (x_1, \ldots, x_n)^T$ are actually partitioned into two vectors: the rational variables $(x_1, \ldots, x_{n_1})^T$ and the integer variables $(x_{n_1+1}, \ldots, x_{n_1+n_2})^T$ with $n = n_1 + n_2$. Based on this partitioning, we also split the coefficient matrix A into two matrices $A = (R, S)$, where $R = (r_1, \ldots, r_m)^T \in \mathbb{Q}^{m \times n_1}$ defines the coefficients for the rational variables and $S = (s_1, \ldots, s_m)^T \in \mathbb{Q}^{m \times n_2}$ defines the coefficients for the integer variables.

Because only integer variables need to be assigned to integer values, tests like simple rounding should be restricted to integer variables. For instance, if z is a rational solution for the overall polyhedron, then simple rounding applies $\lceil . \rfloor$ only to the components of z that correspond to integer variables.[1] The same holds for our fast cube tests. Instead of looking for hypercubes of the same dimension n as the number of total variables, we are looking for hypercubes of dimension n_2 that expand in the directions that correspond to integer variables, but are flat in the directions that correspond to rational variables. Such a hypercube of dimension n_2 with center point z and edge length e is a *flat cube* $\mathcal{F}_e(z)$ (see Chapter 2.9.3 for the formal definition).

We can also modify the linear cube transformation so that we can compute whether a polyhedron $Ax \leq b$ contains a flat cube $\mathcal{F}_e(z)$ that is less than full dimensional:

[1] We call this operation *mixed simple rounding* and denote it by the function $\lceil z \rfloor_{n_2}$ as mentioned in Chapter 2.1.2.

Lemma 4.3.1 (Flat Cube Transformation)**.** *Let $\mathcal{F}_e(z)$ be a flat cube and $Ax \leq b$ be a polyhedron with $A = (R, S)$, $R = (r_1, \ldots, r_m)^T \in \mathbb{Q}^{m \times n_1}$, and $S = (s_1, \ldots, s_m)^T \in \mathbb{Q}^{m \times n_2}$. $\mathcal{F}_e(z) \subseteq \mathcal{Q}_\delta(Ax \leq b)$ if and only if $Az \leq b'$, where $b'_i = b_i - \frac{e}{2} \|s_i\|_1$.*

Proof. Works the same as the proofs in Section 4.2. □

Since the hypercube $\mathcal{F}_e(z)$ only expands in the directions that correspond to integer variables, the inequality bounds b' of the modified linear cube transformation are only influenced by the coefficients of the integer variables. Using Lemma 4.3.1, we can now modify our fast cube tests so that they work for linear mixed arithmetic. For the largest cube test, we compute the center point of a largest flat cube $\mathcal{F}_e(z)$ that fits into the polyhedron $Ax \leq b$. We determine the center z of this largest flat cube and the associated edge length e with the following LP:

$$\begin{array}{ll} \text{maximize} & x_e \\ \text{subject to} & Ax + s' \frac{x_e}{2} \leq b, \text{ where } s'_i = \|s_i\|_1 \\ & x_e \geq 0\,. \end{array}$$

From the resulting center point z we receive a candidate mixed solution by applying the rounding operator $\lceil . \rfloor$ to the components of z that correspond to integer variables. For the unit cube test, we search for a cube $\mathcal{F}_1(z)$ that is flat in the directions that correspond to rational variables, has edge length 1, and fits into the polyhedron $Ax \leq b$. A linear program that accomplishes this task is: $Ax \leq b'$, where $b'_i = b_i - \frac{1}{2} \|s_i\|_1$.

Again, 1 is the smallest value that we can fix as an edge length to guarantee a mixed solution for all cubes $\mathcal{F}_1(z)$.

4.4 Absolutely Unbounded Polyhedra

While our tests are useful for many types of polyhedra, the motivation for our tests stems from a special type of polyhedron, a so-called absolutely unbounded polyhedron or also called infinite lattice width polyhedron (see Chapter 2.8 and [90]). A polyhedron $Ax \leq b$ is absolutely unbounded if for every objective $h \in \mathbb{Q}^n \setminus \{0^n\}$, either its maximum or minimum objective value is unbounded:

$$\begin{array}{lll} \forall h \in \mathbb{Q}^n \setminus \{0^n\}. & \sup\left\{h^T x : x \in \mathcal{Q}_\delta(Ax \leq b)\right\} & = \infty \text{ or} \\ & \inf\left\{h^T x : x \in \mathcal{Q}_\delta(Ax \leq b)\right\} & = -\infty\,. \end{array}$$

Absolutely unbounded polyhedra seem trivial at first glance because their interior expands arbitrarily far in all directions (see Figure 4.5). Therefore, an absolutely unbounded polyhedron contains an infinite number of integer solutions [90]. Nonetheless, many SMT solvers are inefficient on those polyhedra because they use a branch-and-bound approach with an underlying simplex solver (see Chapter 2.7 or [57]). Although such an approach terminates inside finite *a priori* bounds see Chapter 2.7 and [119]), it does not explore the infinite interior, but rather directs the search along

the rational solutions suggested by the simplex solver: the vertices of the polyhedron. Thus, the SMT solvers concentrate their search on a bounded part of the polyhedron. This bounded part contains only a finite number of integer solutions, whereas the complete interior contains infinitely many integer solutions. The advantage of our cube tests is that they actually exploit the infinite interior because absolutely unbounded polyhedra contain cubes for every edge length (see Figure 4.5). Our tests are, therefore, always successful on absolutely unbounded polyhedra and usually need only a small number of pivoting steps before finding a solution.

Lemma 4.4.1 (Unbounded Cubes Lemma)**.** *Let $Ax \leq b$ be a polyhedron. Let $a' \in \mathbb{Z}^m$ be a vector such that its components are $a'_i = \|a_i\|_1$. Then the following two statements are equivalent:*
(1) $Ax \leq b$ contains a cube $\mathcal{C}_e(z)$ for every $e \geq 0$, and
(2) $Ax \leq b$ is absolutely unbounded.
Or formally:
(1) $\forall e \in \mathbb{Q}^n_\delta.(e \geq 0) \rightarrow (\exists z \in \mathbb{Q}^n_\delta.\, \mathcal{C}_e(z) \subseteq \mathcal{Q}_\delta(Ax \leq b))\,,$
(2) $\forall h \in \mathbb{Q}^n \setminus \{0^n\}.\sup\left\{h^T x : x \in \mathcal{Q}_\delta(Ax \leq b)\right\} = \infty$ *or*
$\inf\left\{h^T x : x \in \mathcal{Q}_\delta(Ax \leq b)\right\} = -\infty\,.$

Proof. (1) $\Rightarrow$ (2): We first assume that $Ax \leq b$ contains a cube $\mathcal{C}_e(z)$ for every $e \geq 0$. Note that the center point z depends on the edge length e. Furthermore, we define the function:

$$\text{width}(h, S) = \left(\sup\left\{h^T x : x \in S)\right\} + \sup\left\{-h^T x : x \in S)\right\}\right) \tag{4.6}$$

for every vector $h \in \mathbb{Q}^n \setminus \{0^n\}$ and for every set of points $S \subseteq \mathbb{Q}^n_\delta$. Then we prove that:

$$\lim_{e\to\infty} \text{width}(h, \mathcal{C}_e(.)) \rightarrow \infty\,.$$

In Section 4.2, we have shown that:

$$\sup\left\{h^T x : x \in \mathcal{C}_e(z)\right\} = h^T z + \tfrac{e}{2} \cdot \|h\|_1\,, \text{ and} \tag{4.7}$$

$$\sup\left\{-h^T x : x \in \mathcal{C}_e(z)\right\} = -h^T z + \tfrac{e}{2} \cdot \|h\|_1\,. \tag{4.8}$$

Therefore, $\text{width}(h, \mathcal{C}_e(z)) = e \cdot \|h\|_1$, which is independent of z. After inserting (4.7) and (4.8) into (4.6), we get:

$$\lim_{e\to\infty} \text{width}(h, \mathcal{C}_e(.)) = \lim_{e\to\infty} e \cdot \|h\|_1 \rightarrow \infty\,.$$

Since $Ax \leq b$ contains cubes $\mathcal{C}_e(z)$ for all $e \in \mathbb{R}$, it holds for all $e \in \mathbb{Q}_\delta$ that

$$\text{width}(h, \mathcal{Q}_\delta(Ax \leq b)) \geq \text{width}(h, \mathcal{C}_e(.))\,,$$

and, thus, $\text{width}(h, \mathcal{Q}_\delta(Ax \leq b)) = \infty$. Since $\mathcal{Q}_\delta(Ax \leq b)$ is also convex, it must hold that:

$$\sup\left\{h^T x : x \in \mathcal{Q}_\delta(Ax \leq b)\right\} = \infty \text{ or } \inf\left\{h^T x : x \in \mathcal{Q}_\delta(Ax \leq b)\right\} = -\infty\,.$$

(2) $\Rightarrow$ (1): By contradiction. Assume that $Ax \leq b$ is absolutely unbounded but that there exists an $e \in \mathbb{Q}_\delta$ (with $e \geq 0$) such that $Ax \leq b$ contains no cube $\mathcal{C}_e(z)$ of edge length e. By Corollary 4.2.2, $Ax \leq b$ contains no cube $\mathcal{C}_e(z)$ of edge length e implies that $Ax \leq b - \frac{e}{2} \cdot a'$ is rationally unsatisfiable. By Farkas' Lemma, $Ax \leq b - \frac{e}{2} \cdot a'$ is rationally unsatisfiable implies that there exists a $y \in \mathbb{Q}^m$ such that: (a) $y_i \geq 0$ for all $i \in \{1, \ldots, m\}$, (b) $y_k > 0$ for at least one $k \in \{1, \ldots, m\}$, (c) $y^T A = (0^n)^T$, and (d) $0 > y^T b - \frac{e}{2} \cdot y^T a'$.

Because of (b), we can transform the equality (c) into the following form:

$$a_k^T = -\sum_{i=1,i\neq k}^{m} \left(\frac{y_i}{y_k} a_i^T\right) . \tag{4.9}$$

By multiplying (4.9) with an $x \in \mathcal{Q}_\delta(Ax \leq b)$, we get:

$$a_k^T x = -\sum_{i=1,i\neq k}^{m} \left(\frac{y_i}{y_k} a_i^T x\right) .$$

Since $a_i^T x \leq b_i$ and $y_i \geq 0$, we get a finite lower bound for $a_k^T x$:

$$a_k^T x = -\sum_{i=1,i\neq k}^{m} \left(\frac{y_i}{y_k} a_i^T x\right) \geq -\sum_{i=1,i\neq k}^{m} \left(\frac{y_i}{y_k} b_i\right) .$$

Thus, the upper bound $\sup\left\{a_k^T x : x \in \mathcal{Q}_\delta(Ax \leq b)\right\} \leq b_k < \infty$ and the lower bound $\inf\left\{a_k^T x : x \in \mathcal{Q}_\delta(Ax \leq b)\right\} \geq -\sum_{i=1,i\neq k}^{m} \left(\frac{y_i}{y_k} b_i\right) > -\infty$ are finite, which contradicts the assumption that $Ax \leq b$ is absolutely unbounded. □

4.5 Experiments

We have found instances of absolutely unbounded polyhedra in some classes of the SMT-LIB benchmarks [10]. These instances are 229 of the 233 `dillig` benchmarks designed by Dillig et al. [52], 503 of the 591 `CAV-2009` benchmarks also by Dillig et al. [52], 229 of the 233 `slacks` benchmarks which are the `dillig` benchmarks extended with slack variables [87], and 19 of the 37 `prime-cone` benchmarks, that is, "a group of crafted benchmarks encoding a tight n-dimensional cone around the point whose coordinates are the first n prime numbers" [87]. The remaining problems (4 from `dillig`, 88 from `CAV-2009`, 4 from `slacks`, and 18 from `prime-cone`) are not absolutely unbounded because they are either tightly bounded or integer unsatisfiable. For our experiments, we look only at the instances of those benchmark classes that are actually absolutely unbounded.

Using these benchmark instances, we have confirmed our theoretical assumptions (Lemma 4.4.1) in practice. We integrated the unit cube test into our own branch-and-bound solver *SPASS-IQ* (v0.3) [2] and ran it on the absolutely unbounded instances; once with the unit cube test turned on (SPASS-IQ+uc) and once with the test turned off (SPASS-IQ). For every problem, SPASS-IQ+uc applies the unit cube test exactly once. This application happens before we start the branch-and-bound approach. To evaluate the efficiency of the unit cube test, we compared SPASS-IQ to several other solvers for systems of linear inequalities.

[2] `http://www.spass-prover.org/spass-iq`

Benchmark Name	CAV-2009		DILLIG		PRIME-CONE		SLACKS		ROTATE	
#Instances	503		229		19		229		229	
Solvers:	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**
SPASS-IQ+uc	**503**	**7.8**	**229**	**3.6**	**19**	**0.0**	**229**	**6.1**	**229**	**4.0**
SPASS-IQ	502	156	**229**	39	**19**	0.1	220	64	**229**	8.4
Ctrl-Ergo	**503**	9.7	**229**	4.2	**19**	0.1	**229**	20.2	228	185884
CVC4	426	19367	195	10646	**19**	1.4	138	1268	191	6062
MathSAT5+uc	**503**	39	**229**	17	**19**	0.2	**229**	38	**229**	19
MathSAT5	502	8607	228	3530	**19**	3.3	192	18055	**229**	1464
SMTInterpol	493	11348	225	3666	**19**	14	206	15975	178	4120
Yices	477	44236	211	11586	**19**	0	154	24525	198	52679
Z3+uc	**503**	281	**229**	97	**19**	0.1	**229**	100	**229**	97
Z3	472	3049	214	1639	**19**	0.1	160	368	214	1634

Figure 4.8: Experimental Results: SMT solvers

Comparison with State-Of-The-Art SMT Solvers

First, we compared SPASS-IQ with state-of-the-art SMT solvers for linear integer arithmetic: *CVC4* (v1.6) [9], *MathSAT5* (v5.5.2) [41], *SMTInterpol* (v2.5-19) [40], *Yices* (v2.6.0) [56], and *Z3* (v4.8.1) [49]. All these solvers employ a branch-and-bound approach with an underlying dual simplex solver [57]. The only exception are MathSAT5 and Z3, which, subsequent to our first publication on the unit cube test [34], now also perform the unit cube test in advance. That is why we also test MathSAT5 and Z3 once with the unit cube test turned on (MathSAT5+uc and Z3+uc) and once with the test turned off (MathSAT5 and Z3).

The solvers had to solve each problem in under 40 minutes. For the experiments, we used a Debian Linux cluster and allotted to each problem and solver combination 2 cores of an Intel Xeon E5620 (2.4 GHz) processor, 4 GB RAM, and 40 minutes. Figure 4.8 lists the results of the different solvers (column one) on the different benchmark classes (row one). Row two lists the number of benchmark instances we considered for our experiments. For each combination of benchmark class and solver, we have listed the number of instances the solver could solve in the given time as well as the total time (in seconds) of the instances solved (columns labelled with "solved" and "time", respectively).

Our solver that employs the unit cube test solves all instances with the application of the unit cube test and is 25 times faster than our solver without the test. The SMT theory solvers in their standard setting were not able to solve all instances within the allotted time. Moreover, our unit cube test was over 100 times faster than any state-of-the-art SMT solver without the unit cube test. The results for MathSAT5 and Z3 further support the superiority of the test.

Comparison with Ctrl-Ergo

Second, we compared our unit cube test with the *Ctrl-Ergo* solver, which includes a subroutine that is essentially the dual to our largest cube test [26]. As expected, both approaches are comparable for absolutely unbounded polyhedra. In order to also compare the two approaches on benchmarks that are not absolutely unbounded, we created the `rotate` benchmarks by adding the same four inequalities to all absolutely unbounded instances of the `dillig` benchmarks. These four inequalities essentially describe a square bounding the variables x_0 and x_1 in an interval $[-u, u]$. For a large enough choice of u (e.g., $u = 2^{10}$), the square is so large that the benchmarks are still integer satisfiable and not absolutely trivial for branch-and-bound solvers. To add a challenge, we rotated the square by a small factor $1/r$, which resulted in the following four inequalities:

$$-b \cdot r \cdot r + r \leq b \cdot r \cdot x_0 - x_1 \leq b \cdot r \cdot r - r\,, \text{ and}$$
$$-b \cdot r \cdot r + r \leq x_0 + b \cdot r \cdot x_1 \leq b \cdot r \cdot r - r\,.$$

These changes have nearly no influence on SPASS-IQ, and two SMT solvers even benefit from the proposed changes. For Ctrl-Ergo the *rotate* benchmarks are very hard because its subroutine detects only absolutely unbounded polyhedra. If the polyhedron is not absolutely unbounded, then Ctrl-Ergo starts its search from the boundaries of the polyhedron instead of looking at the polyhedron's interior. We can even control the number of iterations (r^2) Ctrl-Ergo spends on the parts of the boundary without any integer solutions if we choose r accordingly (e.g., $r = 2^{10}$). In contrast, we use our cube tests to also extract interior points for rounding. This difference makes our tests much more stable under small changes to the polyhedron.

Most problems in the linear integer arithmetic SMT-LIB benchmarks with finite lattice width (i.e., that are not absolutely unbounded) can be solved without using any actual integer arithmetic techniques. A standard simplex solver for the rationals typically finds a rational solution for such a problem that is also an integer solution. Applying the unit cube test on these trivial problem classes is a waste of time. In the worst case, it doubles the eventual solution time. For these examples it is beneficial to first compute a general rational solution and to check it for integer satisfiability before applying the unit cube test. This has the additional benefit that rational unsatisfiable problems are filtered out before applying the unit cube test. The unit cube test is also guaranteed to fail on problems containing boolean variables, i.e., variables that are either 0 or 1, unless they are absolutely trivial and describe a unit cube themselves. Whenever the problem contains a boolean variable, it is beneficial to skip the unit cube test. This is also the reason why we provide no experimental results for the theory of linear mixed arithmetic, i.e., the few mixed benchmarks available in the SMT-LIB all contain boolean variables.

Benchmark Name	CAV-2009		DILLIG		PRIME-CONE		SLACKS		ROTATE	
#Instances	503		229		19		229		229	
Solvers:	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**
SPASS-IQ+uc	**503**	7.8	**229**	3.6	**19**	**0.0**	**229**	6.1	**229**	4.0
SPASS-IQ	502	156	**229**	39	**19**	0.1	220	64	**229**	8.4
GLPK	**503**	24	**229**	13	**19**	**0.0**	121	4.3	**229**	9.8
Gurobi	**503**	**3.7**	**229**	**1.7**	**19**	0.1	**229**	**1.6**	**229**	**0.4**
SCIP	**503**	42	**229**	19	**19**	0.1	224	34	**229**	16

Figure 4.9: Experimental Results: MILP solvers

Comparison with MILP Solvers

Third, we compared our unit cube test with several solvers for mixed-integer programming (MILP) (see Figure 4.9): the two non-commercial solvers *GLPK* (v4.65) [106] and *SCIP* (v6.0.0) [70] as well as the commercial solver *Gurobi* (v7.52) [77]. For these experiments, we used the same benchmarks—although converted into the MPS (Mathematical Programming System) format—and the same experiment parameters as for our experiments with the SMT solvers. In General, mixed-integer programming solvers have an advantage over standard SMT theory solvers because (i) they are not required to be exact and sound, which allows them to use floating-point arithmetic, and (ii) they are not required to be incrementally efficient, which means they can use much more elaborate techniques. Despite these advantages, SPASS-IQ is faster and solves more problems from the absolutely unbounded benchmarks than GLPK and SCIP. The reason is that GLPK and SCIP rely—like the state-of-the-art SMT theory solvers—on a branch-and-bound approach with an underlying simplex solver, which means they also focus their search on the vertices of the polyhedron instead of the polyhedron's interior. Gurobi, on the other hand, is faster than SPASS-IQ on the absolutely unbounded benchmarks because (i) it uses an interior point method [93] to compute the first rational solution for its branch-and-bound approach and only then switches to its simplex solver and (ii) it uses floating-point arithmetic, which is more efficient than exact arithmetic.

The experiments with Gurobi give the impression that interior point methods are an efficient alternative to our unit cube test. At least for now, this impression is only correct for mixed-integer programming and not for SMT theory solvers. Interior point methods perform worse in the context of SMT theory solvers because the currently competitive interior point methods are not incrementally efficient, which is one of the most important properties for an efficient SMT theory solver [61].

On Other SMT-LIB Benchmarks

Most problems in the linear integer arithmetic SMT-LIB benchmarks with finite lattice width (i.e., that are not absolutely unbounded) can be solved without using any actual integer arithmetic techniques. A standard simplex solver for the rationals typically finds a rational solution for such a problem that is also an integer solution. Applying the unit cube test on these trivial problem classes is a waste of time. In the worst case, it doubles the eventual solution time. For these examples it is beneficial to first compute a general rational solution and to check it for integer satisfiability before applying the unit cube test. This has the additional benefit that rational unsatisfiable problems are filtered out before applying the unit cube test. The unit cube test is also guaranteed to fail on problems containing boolean variables, i.e., variables that are either 0 or 1, unless they are absolutely trivial and describe a unit cube themselves. Whenever the problem contains a boolean variable, it is beneficial to skip the unit cube test. This is also the reason why we provide no experimental results for the theory of linear mixed arithmetic, i.e., the few mixed benchmarks available in the SMT-LIB all contain boolean variables.

4.6 Summary

We have presented the linear cube transformation (Corollary 4.2.2), which allows us to efficiently determine whether a polyhedron contains a cube of a given edge length. Based on this transformation we have created two tests for linear integer arithmetic: the largest cube test and the unit cube test. Our tests can be integrated into SMT theory solvers without sacrificing the advantages that SMT solvers gain from the incremental structure of subsequent subproblems. Furthermore, our experiments have shown that these tests increase efficiency on certain polyhedra such that previously hard sets of constraints become trivial.

Chapter 5

Computing a Complete Basis for Equalities (Implied by a System of LRA Constraints)

Equalities are a special instance of linear arithmetic constraints. They are useful in simplifying systems of arithmetic constraints [76], and they are essential for the *Nelson-Oppen style combinations of theories* [36, 111, 118]. However, they are also an obstacle for our *fast cube tests* (see Chapter 4). If an inequality system implies an equality, then it has only a surface and no interior; so our cube tests cannot explore an interior and will certainly fail. In order to expand the applicability of our cube tests, we have to develop methods that find, isolate, and eliminate *implied equalities* from systems of linear arithmetic constraints.

We can detect the existence of an implied equality by searching for a hypercube in our polyhedron. If the maximal edge length of such a hypercube is zero, then there exists an implied equality. This test can be further simplified. By turning all inequalities into strict ones, the interior of the original polyhedron remains while the surface disappears. If the strict system is unsatisfiable, then the original system has no interior and implies an equality. Moreover, the method generates an implied equality as a proof based on an explanation of unsatisfiability for the strict system.

We are also able to extend the above method into an algorithm that computes an *equality basis*, i.e., a finite representation of all equalities implied by a satisfiable system of inequalities. For this purpose, the algorithm repeatedly applies the above method to find, collect, and eliminate equalities from our system of constraints. When the system contains no more equalities, then the collected equalities represent an equality basis, i.e., any implied equality can be obtained by a linear combination of the equalities in the basis. The equality basis has many applications. If transformed into a substitution, it eliminates all equalities implied by our system of constraints,

which results in a system of constraints with an interior and, therefore, improves the applicability of our cube tests. The equality basis also allows us to test whether a system of linear arithmetic constraints implies a given equality. We even extend this test into an efficient method that computes all pairs of equivalent variables inside a system of constraints. These pairs are necessary for the Nelson-Oppen style combination of theories.

This chapter is organized as follows: In Section 5.2, we show how to investigate equalities with the linear cube transformation. We do so by first introducing an efficient method for testing whether a system of linear arithmetic constraints implies a given equality (Section 5.2.1); then, we extend the method so that it computes an equality basis for our system of constraints (Section 5.2.2). In Section 5.3, we describe an implementation of our methods as an extension of Dutertre and de Moura's version of the simplex algorithm, which is integrated in many SMT solvers (see Chapter 2.7 and [57]). The implementation generates justifications and preserves incrementality. The efficient computation of an equality basis can then be used in identifying equivalent variables for the Nelson-Oppen combination of theories (Section 5.4). This is also the first application of the equality basis that we discuss in depth. The second application we present uses the equality basis for the computation of a bounded basis (Definition 2.8.5) and for the detection of bounded and unbounded directions (Section 5.5). The final application we present uses the equality basis for quantifier elimination (Section 5.6). Section 5.7 concludes the chapter with a summary of the presented results.

5.1 Related Work and Preliminaries

This chapter is based on two publications with Christoph Weidenbach as co-author [33, 35]. Only Section 5.5 has never appeared in any publication before this thesis.

This chapter focuses like the previous chapter on the geometric interpretation of systems of inequalities. This is the reason why we also use in this chapter polyhedron as an alternative name for systems of inequalities $Ax \leq b$. The other geometric objects we are considering are cubes. Their definition can be found in Chapter 2.9.

This chapter also focuses on the theory of linear rational arithmetic and not on the more general theory of linear mixed arithmetic. Therefore, we abbreviate in this chapter rational satisfiability/equivalence/entailment with satisfiability/equivalence/entailment. Nonetheless, some of the applications we present verge into the theory of linear mixed arithmetic. To avoid confusion, we always specify the type of solution/satisfiability if it is not rational.

The constraints in this chapter are non-strict inequalities and they are either formatted according to the vector representation, i.e., $a_i^T x \leq b_i$ (see also Chapter 2.2.1), or the standard representation, $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ (see also Chapter 2.2.1). This chapter also deals with equalities $a_i^T x = b_i$ besides non-strict inequalities. However, an equality is just our way of highlighting the set of inequalities $\{a_i^T x \leq b_i, -a_i^T x \leq -b_i\}$ as explained in Chapter 2.2.1. Other types of constraints have to be reduced to non-strict inequalities with the techniques presented in Chapter 2.3.

This chapter builds on the basics of linear algebra (Chapter 2.1) and linear arithmetic (Chapter 2.2), on the concept of implied constraints (Chapter 2.5), and on the definitions of (un)bounded and (un)guarded problems and variables (Chapter 2.8). Our example implementation (Section 5.3) also builds on the notions and definitions of standard arithmetic decision procedures for SMT solvers as presented in Chapter 2.7.

There also already exist several methods that find, isolate, and eliminate implied equalities [21, 123, 132, 81]. Hentenryck and Graf [81] define unique normal forms for systems of linear constraints with non-negative variables. To compute a normal form, they first eliminate all implied equalities from the system. To this end, they determine the lower bound for each inequality by solving one linear optimization problem. Similarly, Refalo [123] describes several incremental methods that use optimization to turn a satisfiable system of linear constraints in "revised solved form" into a system without any implied equalities. Rueß and Shankar also use this optimization scheme to determine a basis of implied equalities [125]. Additionally, they present a necessary but not sufficient condition for an inequality to be part of an equality explanation. During preprocessing, all inequalities not fulfilling this condition are eliminated, thus, reducing the number of optimization problems their method has to compute. However, this preprocessing step might be in itself expensive because it relies on a non-trivial fixed-point scheme. The method presented by Telgen [132] does not require optimization. He presents criteria to detect implied equalities based on the tableau used in the simplex algorithm, but he was not able to formulate an algorithm that efficiently computes these criteria. In the worst case, he has to pivot the simplex tableau until he has computed all possible tableaux for the given system of constraints. Another method that detects implied equalities was presented by Bjørner [21]. He uses Fourier-Motzkin variable elimination to compute linear combinations that result in implied equalities.

Our methods that detect implied equalities do not require optimization, which is advantageous because SMT solvers are usually not fine-tuned for optimization. Moreover, we defined our methods for a rather general formulation of linear constraints, which allows us to convert our results into other representations, e.g., the tableau-and-bound representation used in Dutertre and de Moura's version of the simplex algorithm (see Section 5.3), while

preserving efficiency. Finally, our method efficiently searches for implied equalities. We neither have to check each inequality independently nor do we have to blindly pivot the simplex tableau. This also makes potentially expensive preprocessing techniques obsolete.

5.2 From Cubes to Equalities

If a polyhedron implies an equality, then it has only surface points and neither an interior nor a center. There is no way such a polyhedron contains a unit cube and a largest cube has edge length zero and is just a point in the original polyhedron. Equalities are, therefore, a challenge for the applicability of our cube tests (see Chapter 4).

There even exist systems of inequalities that imply infinitely many equalities. For instance, the system consisting of the inequalities $-2x_1+x_2 \leq -2$, $x_1 + 3x_2 \leq 8$, and $x_1 - 2x_2 \leq -2$ has only one rational solution: the point $(x_1, x_2) = (2, 2)$. Therefore, it implies the equalities $-2x_1 + x_2 = -2$ and $x_1 + 3x_2 = 8$, and all linear combinations of those two equalities, i.e., $\lambda_1 \cdot (-2x_1+x_2)+\lambda_2 \cdot (x_1+3x_2) = \lambda_1 \cdot (-2)+\lambda_2 \cdot 8$ for all $\lambda_1, \lambda_2 \in \mathbb{Q}$. The above example also points us to another fact about equalities: there exists a finite representation of all equalities implied by a system of inequalities—even if the system implies infinitely many equalities.

One such finite representation is the *equality basis* for a satisfiable system of inequalities $Ax \leq b$. An equality basis is a system of equalities $D'x = c'$ such that all (explicit and implicit equalities) implied by $Ax \leq b$ are linear combinations of equalities from $D'x = c'$. We prefer to represent each equality basis $D'x = c'$ as an equivalent system of equalities $y - Dz = c$ such that $y = (y_1, \ldots, y_{n_y})^T$ and $z = (z_1, \ldots, z_{n_z})^T$ are a partition of the variables in x, $D \in \mathbb{Q}^{n_y \times n_z}$, and $c \in \mathbb{Q}^{n_y}$. The existence of such an equivalent system of equalities is guaranteed by Gaussian elimination. Moreover, each variable y_i appears exactly once in the system $y - Dz = c$, that is to say, y_i appears only in the row $y_i - d_i^T z_i = c_i$. We choose to represent our equality bases in this manner because this form also correlates to a distinct substitution $\sigma_{y,z}^{D,c}$ that replaces variable y_i with $c_i + d_i^T z$ (see also Chapter 2.2.4):

$$\sigma_{y,z}^{D,c} := \{y_i \mapsto c_i + d_i^T z : i \in \{1, \ldots, n_y\}\}.$$

The substitution $\sigma_{y,z}^{D,c}$ is important because it allows us to eliminate all equalities from $Ax \leq b$. We simply apply the substitution $\sigma_{y,z}^{D,c}$ to $Ax \leq b$ and receive a new system $A'z \leq b'$ that neither contains the variables y nor implies any equalities.[1] And the substitution $\sigma_{y,z}^{D,c}$ for the equality basis $y - Dz = c$ has even further applications. For instance, we can directly

[1] If we combine the equality basis with a *diophantine equation handler* [76], then we even receive a substitution σ' that eliminates the equalities in such a way that we can reconstruct an integer solution from them. The result is a new system of inequalities that implies no equalities and has an integer solution if and only if $Ax \leq b$ has one.

check whether an equality $h^T x = g$ is a linear combination of $y - Dz = c$ and, therefore, implied by both $Ax \leq b$ and $y - Dz = c$. We simply apply $\sigma_{y,z}^{D,c}$ to $h^T x = g$ and see if it simplifies to $0 = 0$. Other applications of $\sigma_{y,z}^{D,c}$, e.g., for the *Nelson-Oppen* style combination of theories, will be discussed in the Sections 5.4–5.6.

5.2.1 Finding Equalities

The first step in computing an equality basis for a polyhedron $Ax \leq b$ is to detect whether the system contains any equalities. At the beginning of this section, we have already stated a criterion that detects this:

Lemma 5.2.1 (Cube-Equality)**.** *Let $Ax \leq b$ be a polyhedron. Then exactly one of the following statements is true:*
(1) $Ax \leq b$ implies an equality $h^T x = g$ with $h \neq 0^n$, or
(2) $Ax \leq b$ contains a cube with edge length $e > 0$.

Proof. This proof is a case distinction over the sign of x_e for the following slightly simplified version of the largest cube test (see Chapter 4.3.1 for the original definition):

$$\begin{array}{ll} \text{maximize} & x_e \\ \text{subject to} & Ax + a' x_e \leq b, \text{ where } a'_i = \frac{1}{2} \left\| a_i \right\|_1 . \end{array} \tag{5.1}$$

If the maximum objective value is positive, $Ax \leq b$ contains a cube with edge length $e > 0$. Therefore, we have to prove that $Ax \leq b$ contains no equality $h^T x = g$ with $h \neq 0^n$, which we will do by contradiction. Assume $Ax \leq b$ contains an equality $h^T x = g$ with $h \neq 0^n$. Then, by transitivity of the subset relation, the polyhedron consisting of the inequalities $h^T x \leq h$ and $-h^T x \leq -g$ must also contain a cube of edge length e. However, applying the transformation from Corollary 4.2.2 to this new polyhedron results in two contradicting inequalities: $h^T x \leq g - \left\| h \right\|_1 \cdot \frac{e}{2}$ and $-h^T x \leq -g - \left\| h \right\|_1 \cdot \frac{e}{2}$. Thus, (1) and (2) cannot hold at the same time.

If the maximum objective value is zero, then $Ax \leq b$ is satisfiable but contains no cube with edge length $e > 0$. Therefore, we have to prove that $Ax \leq b$ contains an equality $h^T x = g$ with $h \neq 0$. Consider the dual linear program [128] of (5.1):

$$\begin{array}{ll} \text{minimize} & y^T b \\ \text{subject to} & y^T A = (0^n)^T , \\ & y^T a' = 1 , \text{ where } a'_i = \frac{1}{2} \left\| a_i \right\|_1 , \\ & y \geq 0 . \end{array} \tag{5.2}$$

Due to strong duality, the objectives of the dual and primal linear programs are equal [128]. Therefore, there exists a $y \in \mathbb{Q}^m$ that has objective $y^T b = 0$ and that satisfies the dual (5.2). Since $y^T a' = 1$ and $a'_i \geq 0$ and $y_i \geq 0$ holds, there exists a $k \in \{1, \ldots, m\}$ such that $y_k > 0$. By multiplying $y^T A = (0^n)^T$ with an $x \in \mathcal{Q}_\delta(Ax \leq b)$ and isolating $a_k^T x$, we get:

$$a_k^T x = -\textstyle\sum_{i=1, i \neq k}^{m} \left(\frac{y_i}{y_k} a_i^T x \right) .$$

Using $y_i \geq 0$, and our original inequalities $a_i^T x \leq b_i$, we get a finite lower bound for $a_k^T x$:

$$a_k^T x = -\sum_{i=1, i\neq k}^{m} \left(\frac{y_i}{y_k} a_i^T x\right) \geq -\sum_{i=1, i\neq k}^{m} \left(\frac{y_i}{y_k} b_i\right) .$$

Now, we reformulate $y^T b = 0$ analogously and get: $b_k = -\sum_{i=1, i\neq k}^{m} \left(\frac{y_i}{y_k} b_i\right)$. Thus, $a_k^T x = b_k$ is an equality contained in the original inequalities $Ax \leq b$.

If the maximum objective value is negative, $Ax \leq b$ is unsatisfiable and contains no cube with edge length $e > 0$. Since $\mathcal{Q}_\delta(Ax \leq b)$ is now empty, $Ax \leq b$ contains all equalities. □

A cube with positive edge length is enough to prove that there exists no implied equality. The actual edge length e of this cube is not relevant. Therefore, we can assume that the edge length e is arbitrarily small. We can even assume that our edge length is so small that we can ignore the different multiples $\|a_i\|_1$ and any infinitesimals introduced by strict inequalities. We just have to turn all of our inequalities into strict inequalities.

Lemma 5.2.2 (Strict-Cube)**.** *Let $Ax \leq b$ be a polyhedron, where $a_i \neq 0^n$, $b_i = (p_i, q_i)$, $q_i \leq 0$, and $b_i^\delta = (p_i, -1)$ be the strict versions of the bounds b_i for all $i \in \{1, \ldots, m\}$. Then the following statements are equivalent:*
(1) $Ax \leq b$ contains a cube with edge length $e > 0$, and
(2) $Ax \leq b^\delta$ is satisfiable.

Proof. (1) ⇒ (2): If $Ax \leq b$ contains a cube of edge length $e > 0$, then $Ax \leq b - a'$ is satisfiable, where $a'_i = \frac{e}{2} \|a_i\|_1$. By Lemma 2.3.1, we know that there exists a $\delta \in \mathbb{Q}$ such that $Ax \leq p + q\delta - a'$. Now, let

$$\delta' = \min\{a'_i - q_i\delta : i = 1, \ldots, m\} .$$

Since $a'_i - q_i\delta \geq \delta'$, it holds that $Ax \leq p - \delta' 1^m$. Since $a'_i = \|a_i\|_1 > 0$ and $q_i \leq 0$, it also holds that $\delta' > 0$. By Lemma 2.3.1, we deduce that $Ax < p$ and, therefore, $Ax \leq b^\delta$ holds.

(2) ⇒ (1): If $Ax \leq b^\delta$ is satisfiable, then we know by Lemma 2.3.1 that there must exist a $\delta > 0$ such that $Ax \leq p - \delta 1^m$ holds. Let

$$a_{\max} = \max\{\|a_i\|_1 : i = 1, \ldots, m\} ,$$

$\delta' = \frac{\delta}{2}$, and $e = \frac{\delta}{a_{\max}}$. Then $p_i - \delta = p_i - \delta' - \frac{e}{2} a_{\max} \leq b_i - \frac{e}{2} \|a_i\|_1$. Thus, $Ax \leq b$ contains a cube with edge length $e > 0$. □

In case $Ax \leq b^\delta$ is unsatisfiable, $Ax \leq b$ contains no cube with positive edge length and, therefore by Lemma 5.2.1, an equality. In case $Ax \leq b^\delta$ is unsatisfiable, the algorithm returns an explanation, i.e., a minimal set C of unsatisfiable constraints $a_i^T x \leq b_i^\delta$ from $Ax \leq b^\delta$ (see also Chapter 2.5.1). If $Ax \leq b$ itself is satisfiable, we can extract equalities from this explanation:

Lemma 5.2.3 (Equality Explanation)**.** *Let $Ax \leq b$ be a satisfiable polyhedron, where $a_i \neq 0^n$, $b_i = (p_i, q_i)$, $q_i \leq 0$, and $b_i^\delta = (p_i, -1)$ for all $i \in \{1, \ldots, m\}$. Let $Ax \leq b^\delta$ be unsatisfiable. Let C be a minimal set of unsatisfiable constraints $a_i^T x \leq b_i^\delta$ from $Ax \leq b^\delta$. Then it holds for every $a_i^T x \leq b_i^\delta \in C$ that $a_i^T x = b_i$ is an equality implied by $Ax \leq b$.*

Proof. Because of transitivity of the subset and implies relationships, we can assume that $Ax \leq b$ and $Ax \leq b^\delta$ contain only the inequalities associated with the explanation C. Therefore, $C = \{a_1^T x \leq b_1^\delta, \ldots, a_m^T x \leq b_m^\delta\}$. By Lemma 2.5.4 and $Ax \leq b^\delta$ being unsatisfiable, we know that there exists a $y \in \mathbb{Q}^m$ with $y \geq 0$, $y^T A = (0^n)^T$, and $y^T b^\delta < 0$. By Lemma 2.5.4 and $Ax \leq b$ being satisfiable, we know that $y^T b \geq 0$ is also true. By Lemma 2.5.6, we know that $y_k > 0$ for every $k \in \{1, \ldots, m\}$.

Now, we use $y^T b^\delta < 0$, $y^T b \geq 0$, and the definitions of $<$ and $\leq$ for $\mathbb{Q}_\delta$ to prove that $y^T b = 0$ and $b = p$. Since $y^T b^\delta < 0$, we get that $y^T p \leq 0$. Since $y^T b \geq 0$, we get that $y^T p \geq 0$. If we combine $y^T p \leq 0$ and $y^T p \geq 0$, we get that $y^T p = 0$. From $y^T p = 0$ and $y^T b \geq 0$, we get $y^T q \geq 0$. Since $y > 0$ and $q_i \leq 0$, we get that $y^T q = 0$ and $q_i = 0$. Since $q_i = 0$, $b = p$.

Next, we multiply $y^T A = (0^n)^T$ with an $x \in \mathcal{Q}_\delta(Ax \leq b)$ to get $y^T Ax = 0$. Since $y_k > 0$ for every $k \in \{1, \ldots, m\}$, we can solve $y^T Ax = 0$ for every $a_k^T x$ and get:

$$a_k^T x = -\sum_{i=1, i\neq k}^{m} \left(\frac{y_i}{y_k} a_i^T x\right).$$

Likewise, we solve $y^T b = 0$ for every b_k to get: $b_k = -\sum_{i=1, i\neq k}^{m} \left(\frac{y_i}{y_k} b_i\right)$.
Since $x \in \mathcal{Q}_\delta(Ax \leq b)$ satisfies all $a_i^T x \leq b_i$, we can deduce b_k as the lower bound of $a_k^T x$:

$$a_k^T x = -\sum_{i=1, i\neq k}^{m} \left(\frac{y_i}{y_k} a_i^T x\right) \geq -\sum_{i=1, i\neq k}^{m} \left(\frac{y_i}{y_k} b_i\right) = b_k\,,$$

which proves that $Ax \leq b$ implies $a_k^T x = b_k$. □

Lemma 5.2.3 justifies simplifications on $Ax \leq b^\delta$. We can eliminate all inequalities in $Ax \leq b^\delta$ that cannot appear in the explanation of unsatisfiability, i.e., all inequalities $a_i^T x \leq b_i^\delta$ that cannot form an equality $a_i^T x = b_i$ that is implied by $Ax \leq b$. For example, if we have an assignment $v \in \mathbb{Q}_\delta^n$ such that $Av \leq b$ is true, then we can eliminate every inequality $a_i^T x \leq b_i^\delta$ for which $a_i^T v = b_i$ is false. According to this argument, we can also eliminate all inequalities $a_i^T x \leq b_i^\delta$ that were already strict inequalities in $Ax \leq b$.

5.2.2 Computing an Equality Basis

We now present the algorithm `EqBasis`$(A'x \leq b')$ (Figure 5.1) that computes an equality basis for a polyhedron $A'x \leq b'$. In a nutshell, `EqBasis` iteratively detects and removes equalities from our system of inequalities and collects them in a system of equalities until it has a complete equality

Algorithm 11: `EqBasis`$(Ax \leq b)$

Input : A satisfiable system of inequalities $Ax \leq b$, where $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m_\delta$

Output: An equality basis $y - Dz = c$ for $Ax \leq b$

$l := 0$;
$n_z := n$;
$(z_1, \ldots, z_{n_z}) := (x_1, \ldots, x_n)$;
$y := ()^T$;
$(y - Dz = c) := \emptyset$;
Remove all rows $a_i^T z \leq b_i$ from $Az \leq b$ with $a_i = 0^n$ and $b_i = 0$;
while $Az \leq b^\delta$ *is unsatisfiable* **do**
    Let C be an explanation for $Az \leq b^\delta$ being unsatisfiable;
    Select $(a_i^T z \leq b_i^\delta) \in C$;
    `/* by Lemma 5.2.3,` $a_i^T z = b_i$ `is implied by` $Az \leq b$ `*/`
    Select a variable z_k such that $a_{ik} \neq 0$;
    $\sigma' := \{z_k \mapsto \frac{b_i}{a_{ik}} - \sum_{j=1, j \neq k}^{n} \frac{a_{ij}}{a_{ik}} z_j\}$;
    $z' := (z_1, \ldots, z_{k-1}, z_{k+1}, \ldots, z_n)^T$;
    $y' := (y_1, \ldots, y_l, z_k)^T$;
    $l := l + 1$;
    $(A'z' \leq b') := (Az \leq b)\sigma'$;
    $(y' - D'z' = c') := (y - Dz = c)\sigma' \cup \{z_k + \sum_{j=1, j \neq k}^{n} \frac{a_{ij}}{a_{ik}} z_j = \frac{b_i}{a_{ik}}\}$;
    $z := z'$;
    $y := y'$;
    $(Az \leq b) := (A'z' \leq b')$;
    $(y - Dz = c) := (y' - D'z' = c')$;
    Remove all rows $a_i^T z \leq b_i$ from $Az \leq b$ with $a_i = 0^n$ and $b_i = 0$;
end
return $y - Dz = c$;

Figure 5.1: `EqBasis` computes an equality basis

basis. To this end, `EqBasis` computes in each iteration one system of inequalities $Az \leq b$ and one system of equalities $y - Dz = c$ such that $A'x \leq b'$ is equivalent to $(y - Dz = c) \cup (Az \leq b)$. While the variables z are completely defined by the inequalities $Az \leq b$, the equalities $y - Dz = c$ extend any assignment from the variables z to the variables y. Initially, z is just x, $y - Dz = c$ is empty, and $Az \leq b$ is just $A'x \leq b'$.

In every iteration l of the while loop, `EqBasis` eliminates one equality $a_i^T z = b_i$ from $Az \leq b$ and adds it to $y - Dz = c$. `EqBasis` finds this equality based on the techniques we presented in the Lemmas 5.2.2 & 5.2.3 (line 7). If the current system of inequalities $Az \leq b$ implies no equality, then `EqBasis` is done and returns the current system of equalities $y - Dz = c$. Otherwise, `EqBasis` turns the found equality $a_i^T z = b_i$ into a substitution $\sigma' := \{z_k \mapsto \frac{b_i}{a_{ik}} - \sum_{j=1, j\neq k}^{n} \frac{a_{ij}}{a_{ik}} z_j\}$ (line 11) and applies it to $Az \leq b$ (line 15). This has the following effects: (i) the new system of inequalities $A'z' \leq b'$ implies no longer the equality $a_i^T z = b_i$; and (ii) it no longer contains the variable z_k. Next, we apply σ' to our system of equalities (line 16) and concatenate the equality $z_k + \sum_{j=1, j\neq k}^{n} \frac{a_{ij}}{a_{ik}} z_j = \frac{b_i}{a_{ik}}$ to the end of it. This has the following effects: (i) the new system of equalities $y' - D'z' = c'$ implies $a_i^T z = b_i$; and (ii) the variable z_k appears exactly once in $y' - D'z' = c'$. This means that we can now re-partition our variables so that $z := (z_1, \ldots, z_{k-1}, z_{k+1}, \ldots, z_n)^T$ and $y_{l+1} := z_k$ to get two new systems $Az \leq b$ and $y - Dz = c$ that are equivalent to our original polyhedron (line 20). Finally, we remove all rows $0 \leq 0$ from $Az \leq b$ because those rows are trivially satisfied but would obstruct the detection of equalities with Lemma 5.2.2.

To prove the correctness of the algorithm, we first need to prove that moving the equality from our system of inequalities to our system of equalities preserves equivalence, i.e, the systems $(Az \leq b) \cup (y - Dz = c)$ and $(A'z' \leq b') \cup (y' - D'z' = c')$ are equivalent in line 16.

Lemma 5.2.4 (Equality Elimination)**.** *Let:*
(i) $Az \leq b$ *be a system of inequalities;*
(ii) $y - Dz = c$ *be a system of equalities;*
(iii) $h^T z = g$ *be an equality implied by* $Az \leq b$ *with* $h_k \neq 0$*;*
(iv) $\sigma' := \{z_k \mapsto \frac{g}{h_k} - \sum_{j=1, j\neq k}^{n} \frac{h_j}{h_k} z_j\}$ *be a substitution based on this equality;*
(v) $y' := (y_1, \ldots, y_l, z_k)^T$ *and* $z' := (z_1, \ldots, z_{k-1}, z_{k+1}, \ldots, z_n)^T$*;*
(vi) $(A'z' \leq b') := (Az \leq b)\sigma'$*;*
(vii) $(y' - D'z' = c') := (y - Dz = c)\sigma' \cup \{z_k + \sum_{j=1, j\neq k}^{n} \frac{h_j}{h_k} z_j = \frac{g}{h_k}\}$*;*
(viii) $u \in \mathbb{Q}_\delta^{n_y}$*,* $v \in \mathbb{Q}_\delta^{n_z}$*;*
(ix) $u' = (u_1, \ldots, u_{n_y}, v_k)^T$ *and* $v' = (v_1, \ldots, v_{k-1}, v_{k+1}, \ldots, v_{n_z})^T$*.*
Then $(Av \leq b) \cup (u - Dv = c)$ *is true if and only if* $(A'v' \leq b') \cup (u' - D'v' = c')$ *is true.*

Proof. First, we create a new substitution $\sigma_v := \{z_k \mapsto \frac{g}{h_k} - \sum_{j=1,j\neq k}^{n} \frac{h_j}{h_k} v_j\}$ that is equivalent to σ' except that it directly assigns the variables z_i to their values v_i. Let us now assume that either $(Av \leq b) \cup (u - Dv = c)$ or $(A'v' \leq b') \cup (u' - D'v' = c')$ is true. This means that $h^T v = g$ is also true, either by definition of $(Av \leq b)$ or $(u' - D'v' = c')$. But $h^T v = g$ is true also implies that $v_k = \frac{g}{h_k} - \sum_{j=1,j\neq k}^{n} \frac{h_j}{h_k} v_j$ is true. Therefore, σ_v simplifies to the assignment $z_k \mapsto v_k$. So $(Av \leq b) \cup (u - Dv = c)$ and $(A'v' \leq b') \cup (u' - D'v' = c')$ simplify to the same expressions and if one combined system is true, so is the other. □

The algorithm `EqBasis`$(A'x \leq b')$ decomposes the original system of inequalities $A'x \leq b'$ into a reduced system $Az \leq b$ that implies no equalities, and an equality basis $y - Dz = c$. The algorithm is guaranteed to terminate because the variable vector z decreases by one variable in each iteration. Note that `EqBasis`$(A'x \leq b')$ constructs $y - Dz = c$ in such a way that the substitution $\sigma_{y,z}^{D,c}$ is the concatenation of all substitutions σ' from every previous iteration. Therefore, we also know that $\sigma_{y,z}^{D,c}$ applied to $A'x \leq b'$ results in the system of inequalities $Az \leq b$ that implies no equalities. We exploit this fact to prove the correctness of `EqBasis`$(A'x \leq b')$, but first we need two more auxiliary lemmas.

Lemma 5.2.5 (Equivalent Substitution)**.** *Let $y - Dz = c$ be a satisfiable system of equalities. Let $Ax \leq b$ and $A^*x \leq b^*$ be two systems of inequalities, both implying the equalities in $y - Dz = c$. Let $A'z \leq b' := (Ax \leq b)\sigma_{y,z}^{D,c}$ and $A^{**}z \leq b^{**} := (A^*x \leq b^*)\sigma_{y,z}^{D,c}$. Then $A'z \leq b'$ is equivalent to $A^{**}z \leq b^{**}$ if $Ax \leq b$ is equivalent to $A^*x \leq b^*$.*

Proof. Let $Ax \leq b$ be equivalent to $A^*x \leq b^*$. Suppose to the contrary that $A'z \leq b'$ is not equivalent to $A^{**}z \leq b^{**}$. This means that there exists a $v \in \mathbb{Q}_\delta^{n_z}$ such that either $A'v \leq b'$ is true and $A^{**}v \leq b^{**}$ is false, or $A'v \leq b'$ is false and $A^{**}v \leq b^{**}$ is true. Without loss of generality we select the first case that $A'v \leq b'$ is true and $A^{**}v \leq b^{**}$ is false. We now extend this solution by $u \in \mathbb{Q}_\delta^{n_y}$, where $u_i := c_i + d_i^T v$, so $(A'v \leq b') \cup (u - Dv = c)$ is true. Based on the definition of the substitution $\sigma_{y,z}^{D,c}$ and n_y recursive applications of Lemma 5.2.4, the four constraint systems $Ax \leq b$, $A^*x \leq b^*$, $(A'z \leq b') \cup (y - Dz = c)$, and $(A^{**}z \leq b^{**}) \cup (y - Dz = c)$ are equivalent. Therefore, $(A^{**}v \leq b^{**}) \cup (u - Dv = c)$ is true, which means that $A^{**}v \leq b^{**}$ is also true. The latter contradicts our initial assumptions. □

Now we can also prove what we have already explained at the beginning of this section. The equality $h^T x = g$ is implied by $Ax \leq b$ if and only if $y - Dz = c$ is an equality basis and $(h^T x = g)\sigma_{y,z}^{D,c}$ simplifies to $0 = 0$. An equality basis is already defined as a set of equalities $y - Dz = c$ that implies exactly those equalities implied by $Ax \leq b$. So we only need to prove that $h^T x = g$ is implied by $y - Dz = c$ if $(h^T x = g)\sigma_{y,z}^{D,c}$ simplifies to $0 = 0$.

Lemma 5.2.6 (Equality Substitution)**.** *Let $y - Dz = c$ be a satisfiable system of equalities. Let $h^T x = g$ be an equality. Then $y - Dz = c$ implies $h^T x = g$ iff $(h^T x = g)\sigma_{y,z}^{D,c}$ simplifies to $0 = 0$.*

Proof. First, let us look at the case where $h^T x = g$ is an explicit equality $y_i - d_i^T z = c_i$ in $y - Dz = c$. Then $(y_i - d_i^T z = c_i)\sigma_{y,z}^{D,c}$ simplifies to $0 = 0$ because $\sigma_{y,z}^{D,c}$ maps y_i to $d_i^T z + c_i$ and the variables z_j are not affected by $\sigma_{y,z}^{D,c}$.

Next, let us look at the case where $h^T x = g$ is an implicit equality in $y - Dz = c$. Since $y - Dz = c$ implies $h^T z = g$, $y - Dz = c$ and $(y - Dz = c) \cup (h^T z = g)$ must be equivalent. By Lemma 5.2.5, it follows that $(y - Dz = c)\sigma_{y,z}^{D,c}$ and $((y - Dz = c) \cup (h^T z = g))\sigma_{y,z}^{D,c}$ are also equivalent. As we stated at the beginning of this proof, $(y_i - d_i^T z = c_i)\sigma_{y,z}^{D,c}$ simplifies to $0 = 0$. An equality $h'^T z = g'$ that simplifies to $0 = 0$ is true for all $v \in \mathbb{Q}_\delta^{n_z}$. Moreover, only equalities that simplify to $0 = 0$ are true for all $v \in \mathbb{Q}_\delta^{n_z}$. This means $(y - Dz = c)\sigma_{y,z}^{D,c}$ is satisfiable for all assignments and, therefore, $(h^T z = g)\sigma_{y,z}^{D,c}$ must simplify to $0 = 0$.

Finally, let us look at the case where $h^T x = g$ is not an equality implied by $y - Dz = c$. Suppose to the contrary that $(h^T z = g)\sigma_{y,z}^{D,c}$ simplifies to $0 = 0$. From the first part of this prove, we know that $(y - Dz = c)\sigma_{y,z}^{D,c}$ also simplifies to $0 = 0$. Therefore, $((y - Dz = c) \cup (h^T z = g))\sigma_{y,z}^{D,c}$ simplifies to $0 = 0$ and is satisfiable for all assignments. Based on Lemma 5.2.4 and transitivity of equivalence, it follows that $(y - Dz = c) \cup (h^T z = g)$ and $(y - Dz = c)$ are equivalent. Therefore, $h^T z = g$ is implied by $y - Dz = c$, which contradicts our initial assumption. □

With Lemma 5.2.6, we have now all auxiliary lemmas needed to prove that the algorithm `EqBasis` is correct:

Lemma 5.2.7 (Equality Basis)**.** *Let $A'x \leq b'$ be a satisfiable system of inequalities. Let $y - Dz = c$ be the output of* `EqBasis`*$(A'x \leq b')$. Then $y - Dz = c$ is an equality basis of $A'x \leq b'$.*

Proof. Let $Az \leq b$ be the result of applying $\sigma_{y,z}^{D,c}$ to $A'x \leq b'$. Then the condition in line 7 of `EqBasis` and $y - Dz = c$ being the output of `EqBasis`$(A'x \leq b')$ guarantee us that $Az \leq b$ implies no equalities. Let us now suppose to the contrary of our initial assumptions that $A'x \leq b'$ implies an equality $h'^T x = g'$ that $y - Dz = c$ does not imply. Since $h'^T x = g'$ is not implied by $y - Dz = c$, the output of $(h'^T x = g')\sigma_{y,z}^{D,c}$ is an equality $h^T z = g$, where $h \neq 0^{n_z}$. This also implies that $(Az \leq b) \cup (h^T z = g)$ is the output of $((A'x \leq b') \cup (h'^T x = g'))\sigma_{y,z}^{D,c}$. By Lemma 5.2.5, $Az \leq b$ and $(Az \leq b) \cup (h^T z = g)$ are equivalent. Therefore, $Az \leq b$ implies the equality $h^T z = g$, which contradicts the condition in line 7 of `EqBasis` and, therefore, our initial assumptions. □

5.3 Implementation

It is not straight forward how to efficiently integrate our method that finds an equality basis into an SMT solver. Therefore, we now explain how to implement our method as an extension of Dutertre and de Moura's version (see Chapter 2.7.1 and [57]) of the dual simplex algorithm [15, 103]. Our goal for the extension is that it keeps all benefits of the simplex algorithm, i.e., it stays incremental and produces justifications. Both are properties necessary for an efficient theory solver.

We already outlined the simplex algorithm in Chapter 2.7.1. We also mention in Chapter 2.7.1 that this algorithm uses a different input format than the one we prefer for our theory definitions. We defined the theory for the equality basis by representing our input constraints through inequalities $Ax \leq b$ because inequalities represent the set of solutions more intuitively. In the simplex algorithm, the input constraints are represented instead in the tableau representation (Chapter 2.4.2). Therefore, the input consists of two sets of variables $z_1, \ldots, z_n \in \mathcal{N}$ and $y_1, \ldots, y_m \in \mathcal{B}$, a tableau $Az = y$, and two sets of bound functions $\mathcal{L}$ and $\mathcal{U}$, which define a set of bounds $\mathcal{L}(x_j) \leq x_j \leq \mathcal{U}(x_j)$ for the variables $x_j \in \mathcal{N} \cup \mathcal{B}$. (We explain the conversion between the two formats also in Chapter 2.4.2.) Due to the change in format, it might seem difficult to efficiently integrate our method in the simplex algorithm. The truth, however, is that the tableau representation grants us several advantages for the implementation of our equality basis method. For example, we do not have to explicitly eliminate variables via substitution, but we do so automatically via pivoting.

5.3.1 Integration in the Simplex Algorithm

The tableau representation of an equality basis is a tableau $Az = y$ and a set of *tightly bounded* variables, i.e., a set of variables x_j such that $\beta(x_j) := \mathcal{L}(x_j)$ or $\beta(x_j) := \mathcal{U}(x_j)$ for all satisfiable assignments β. Therefore, one way of determining an equality basis is to find all tightly bounded variables.

To find all tightly bounded variables, we present a new extension of the simplex algorithm called `FindTBnds`() (Figure 5.2). This extension uses our Lemmas 5.2.2 & 5.2.3 to iteratively find all bounds $\mathcal{L}(x_j) \leq x_j$ ($x_j \leq \mathcal{U}(x_j)$) that hold tightly for all satisfiable assignments β, and then turns them into explicit equalities by setting $\mathcal{U}(x_j) := \mathcal{L}(x_j)$ ($\mathcal{L}(x_j) := \mathcal{U}(x_j)$). But first of all, `FindTBnds`() determines if our constraint system is actually satisfiable with a call of `Check`(). If the system is unsatisfiable, then it has no solutions and implies all equalities. In this case, `FindTBnds`() returns *false*.

Otherwise, we get a satisfiable assignment β from `Check`() and we use this assignment in `Initialize`() (Figure 5.3) to eliminate all bounds that do not hold tightly under β (i.e., $\beta(x_i) > \mathcal{L}(x_i)$ or $\beta(x_i) < \mathcal{U}(x_i)$). We know that we can eliminate these bounds without losing any tightly bounded va-

Algorithm 12: `FindTBnds()`

Effect : Finds as many tightly bounded variables as possible
Output : *false* iff the system of linear arithmetic constraints is unsatisfiable

```
1 if Check() returns (false,y_i) then return false;
2 Initialize()
3 while Check() returns (false,y_i) do
4 |  (L',U') := FixEqs(y_i)
5 end
  /* For all variables x_k with L(x_k) < U(x_k) recover their
     original bounds L'(x_k), U'(x_k)                          */
6 for x_k ∈ B ∪ N do
7 |  if L(x_k) ≠ U(x_k) then L(x_k) := L'(x_k);U(x_k) := U'(x_k) ;
8 end
9 return true
```

Figure 5.2: The main function for computing an equality basis in tableau representation

riables because we only need the bounds that can be part of an equality explanation, i.e., only bounds that hold tightly for all satisfiable assignments (see Lemma 5.2.3). For the same reason, `Initialize()` eliminates all originally strict bounds, i.e., bounds with a non-zero delta part.

Next, `Initialize()` tries to turn as many variables x_i with $\mathcal{L}(x_i) = \mathcal{U}(x_i)$ into non-basic variables. We do so because x_i is guaranteed to stay a non-basic variable if $\mathcal{L}(x_i) = \mathcal{U}(x_i)$ (see lines 6 & 12 of `Check`). Pivoting like this essentially eliminates the tightly bounded non-basic variable x_i and replaces it with the constant value $\mathcal{L}(x_i)$. There only exists one case when `Initialize()` cannot turn the variable y_i with $\mathcal{L}(y_i) = \mathcal{U}(y_i)$ into a non-basic variable. This case occurs whenever all non-basic variables z_j with non-zero coefficient a_{ij} also have tight bounds $\mathcal{L}(z_j) = \mathcal{U}(z_j)$. In this case, the complete row $y_i = \sum_{z_j \in \mathcal{N}} a_{ij} z_j$ simplifies to $x_i = \mathcal{L}(y_i)$, so it never produces a conflict and we can also ignore this row.

As its last action, `Initialize()` turns the bounds of all variables x_j with $\mathcal{L}(x_j) \neq \mathcal{U}(x_j)$ into strict bounds. Since `Initialize()` transformed these bounds into strict bounds, the condition of the while loop in line 3 of `FindTBnds()` checks whether the system contains another tightly bounded variable (see also Lemma 5.2.2). If `Check` returns (*false*, y_i), then the row y_i represents an equality explanation and all variables z_j with a non-zero coefficient in the row hold tightly (see Lemma 5.2.3). `FindTBnds()` uses `FixEqs`(y_i) (Figure 5.4) to turn these tightly bounded variables x_k into explicit equalities by setting $\mathcal{L}(x_k) = \mathcal{U}(x_k)$. After `FixEqs`$(y_i)$ is done, we go back to the beginning of the loop in `FindTBnds()` and do another call to `Check`.

Algorithm 13: `Initialize()`

Effect : Removes all bounds l_k and u_k that cannot produce equalities; turns as many basic variables x_i with $l_i = u_i$ into non-basic variables as is possible; the bounds for all variables x_k are turned into strict bounds if $l_k < u_k$

Output : A backup of the original bound functions

```
1  L' := L ;          // Backup of the original lower bound function
2  U' := U ;          // Backup of the original upper bound function
3  for x_k ∈ B ∪ N do
4  |  if β(x_k) > L(x_k) then L(x_k) := −∞;
5  |  if β(x_k) < U(x_k) then U(x_k) := +∞;
6  |  if β(x_k) = p_k + q_k δ such that q_k ≠ 0 then L(x_k) := −∞; U(x_k) := ∞;
7  end
8  for y_i ∈ B do
9  |  if L(y_i) = U(y_i) then
10 |  |  select the smallest non-basic variable z_j such that a_ij is non-zero
   |  |    and L(z_j) < U(z_j)
11 |  |  if there is such a z_j then pivot(y_i, z_j);
12 |  end
13 end
14 for x_k ∈ B ∪ N do
15 |  if L(x_k) < U(x_k) then
16 |  |  if L(x_k) ≠ −∞ then L(x_k) := L(x_k) + δ;
17 |  |  if U(x_k) ≠ +∞ then U(x_k) := U(x_k) − δ;
18 |  |  if x_k ∈ N and L(x_k) > β(x_k) then update(x_k, L(x_k));
19 |  |  if x_k ∈ N and U(x_k) < β(x_k) then update(x_k, U(x_k));
20 |  end
21 end
22 return (L', U')
```

Figure 5.3: Initialization function for computing an equality basis in tableau representation

If `Check` returns *true*, then the original system of inequalities implies no further tightly bounded variables (Lemma 5.2.2). We exit the loop and revert the bounds of the remaining variables x_k with $\mathcal{L}(x_k) \neq \mathcal{U}(x_k)$ to their original values. As a result, we have also reverted to a linear system equivalent to our original constraint system. The only difference is that now all tightly bounded variables x_k are explicit equalities because $\mathcal{L}(x_k) = \mathcal{U}(x_k)$. Moreover, the tableau $Az = y$ and the non-basic variables that are tightly bounded represent an equality basis for our original constraint system. The simplex algorithm even represents the current tableau and the tightly bounded non-basic variables in such a way that they also describe

Algorithm 14: `FixEqs`(y_i)

Input : A basic variable y_i that explains the conflict
Effect : Turns the bounds of all variables responsible for the conflict into equalities

```
1  for z_j ∈ 𝒩 do
2      if 𝓛(z_j) ≠ 𝓤(z_j) then
3          if β(y_i) < 𝓛(y_i) then
4              if a_ij > 0 then
5                  𝓤(z_j) := 𝓤'(z_j);   𝓛(z_j) := 𝓤'(z_j);   update(z_j, 𝓤(z_j));
6              end
7              if a_ij < 0 then
8                  𝓛(z_j) := 𝓛'(z_j);   𝓤(z_j) := 𝓛'(z_j);   update(z_j, 𝓛(z_j));
9              end
10         end
11         if β(y_i) > 𝓤(y_i) then
12             if a_ij > 0 then
13                 𝓛(z_j) := 𝓛'(z_j);   𝓤(z_j) := 𝓛'(z_j);   update(z_j, 𝓛(z_j));
14             end
15             if a_ij < 0 then
16                 𝓤(z_j) := 𝓤'(z_j);   𝓛(z_j) := 𝓤'(z_j);   update(z_j, 𝓤(z_j));
17             end
18         end
19     end
20 end
21 if β(y_i) > 𝓤(y_i) then 𝓤(y_i) := 𝓤'(y_i);   𝓛(y_i) := 𝓤'(y_i);
22 if β(y_i) < 𝓛(y_i) then 𝓛(y_i) := 𝓛'(y_i);   𝓤(y_i) := 𝓛'(y_i);
```

Figure 5.4: Supporting function for computing an equality basis in tableau representation

a substitution σ for the elimination of equalities: the rows of the tableau map each basic variable y_i to their row definition $\sum_{z_j \in \mathcal{N}} a_{ij} z_j$ and the tightly bounded non-basic variables z_j, i.e., all variables $z_j \in \mathcal{N}$ with $\mathcal{L}(z_j) = \mathcal{U}(z_j)$, are mapped to their tight bound $\mathcal{L}(z_j)$.

5.3.2 Incrementality and Explanations

Note that asserting additional bounds to our system can increase the number of tightly bounded variables. In this case, we have to apply `FindTBnds`() again to find all tightly bounded variables and to complete the new equality basis. We already mentioned that `Check`() never pivots a non-basic variable z_j into a basic one if $\mathcal{L}(z_j) = \mathcal{U}(z_j)$ because of the conditions in the lines 6 & 12 of `Check`(). So the tightly bounded non-basic variables we have computed in the last call to `FindTBnds`() stay non-basic even if the SMT solver asserts additional bounds for the variables and applies `Check`()

again. Hence, our next application of `FindTBnds`() does not perform any computations for the tightly bounded variables that were detected by earlier applications of `FindTBnds`(). This means that our algorithm to compute the equality basis is highly incremental.

Another important feature of an efficient SMT theory solver is that it produces good—maybe even minimal—conflict explanations. In a typical SMT solver, a SAT solver based on CDCL (conflict-driven clause-learning) selects and asserts a set of theory literals that satisfy the boolean model. Then the theory solvers verify that the asserted literals that belong to their theory are consistently satisfiable. If the theory solver finds a conflict between the asserted literals, then it returns a conflict explanation. The SAT solver uses the conflict explanation to start a conflict analysis that determines a good point for back jumping so it can select a new set of theory literals. Naturally, a good conflict explanation greatly enhances the conflict analysis and, therefore, the remaining search.

The literals asserted in our simplex based theory solver are bounds for our variables.[2] Our algorithm `FindTBnds`() asserts bounds independently of the SAT solver. This leads to problems in the conflict analysis because the conflict explanation is no longer comprehensible for the SAT solver. Hence, we have to extend `FindTBnds`() so it produces *justifications* (for the bounds it asserts in `FixEqs`(y_i)) in the form of clauses that the SAT solver can comprehend and learn.

We only need to justify bounds asserted by `FixEqs`(y_i) because all other bounds asserted by `FindTBnds`() are reverted in line 7 to their original bounds $x_k \geq \mathcal{L}'(x_k)$ and $x_k \leq \mathcal{U}'(x_k)$. And even in `FixEqs`($y_i$), we only have to justify the bounds $x_k \leq \mathcal{L}'(x_k)$ ($x_k \geq \mathcal{U}'(x_k)$) that make the tight bounds $x_k \geq \mathcal{L}'(x_k)$ ($x_k \leq \mathcal{U}'(x_k)$) explicit. We also see that the bounds asserted by `FixEqs`(y_i) are just linear combinations of existing bounds if we look again at the proof of Lemma 5.2.3. The proof also shows that we can derive this linear combination from the conflict explanation C of the strict system. For instance, if the call to `Check`() from line 3 of `FindTBnds`() exits in line 7 with (*false*, y_i), then the conflict explanation is

$$C = \{y_i > \mathcal{L}'(y_i)\} \cup \{z_j < \mathcal{U}'(z_j) : z_j \in \mathcal{N} \text{ and } a_{ij} > 0\} \cup \{z_j > \mathcal{L}'(z_j) : z_j \in \mathcal{N} \text{ and } a_{ij} < 0\}. \text{ [57]}$$

If the call to `Check`() exits instead in line 13 with (*false*, x_i), then the conflict explanation is

$$C = \{y_i < \mathcal{U}'(y_i)\} \cup \{z_j > \mathcal{L}'(z_j) : z_j \in \mathcal{N} \text{ and } a_{ij} > 0\} \cup \{z_j < \mathcal{U}'(z_j) : z_j \in \mathcal{N} \text{ and } a_{ij} < 0\}. \text{ [57]}$$

[2] Actually, the literals we assert are full inequalities $a_i^T x \leq b_i$. Due to slacking, the left side of those constraints is abstracted to a slack variable s such that $s = a_i^T x$. The definition of the slack variable $s = a_i^T x$ is directly stored in the simplex tableau and only a bound $s \leq b_i$ remains as the literal for the SMT solver (see Chapter 2.4.2 for more details).

We receive the set of tightly propagating bounds that we found with the last call to `Check`() if we turn all bounds in C into non-strict bounds:

$$C' = \{x_k \leq \mathcal{U}'(x_k) : (x_k < \mathcal{U}'(x_k)) \in C\} \cup \{x_k \geq \mathcal{L}'(x_k) : (x_k > \mathcal{L}'(x_k)) \in C\}.$$

`FixEqs`(x_i) asserts now for every bound $(x_k \geq \mathcal{L}'(x_k)) \in C'$ the bound $x_k \leq \mathcal{L}'(x_k)$. From the proof of Lemma 5.2.3, we see that the bound $x_k \leq \mathcal{L}'(x_k)$ is a linear combination of the bounds $C' \setminus \{x_k \geq \mathcal{L}'(x_k)\}$. Hence, $x_k \leq \mathcal{L}'(x_k)$ is implied by the bounds $C' \setminus \{x_k \geq \mathcal{L}'(x_k)\}$ and, therefore, the clause

$$\left(\bigvee_{(x_i \geq \mathcal{L}'(x_i)) \in C', x_i \neq x_k} x_i < \mathcal{L}'(x_i)\right) \vee \left(\bigvee_{(x_i \leq \mathcal{U}'(x_i)) \in C'} x_i > \mathcal{U}'(x_i)\right) \vee (x_k \leq \mathcal{L}'(x_k))$$

justifies the asserted bound $x_k \leq \mathcal{L}'(x_k)$. Together with the slack variable definitions stored in the simplex tableau, this clause is a tautology and the SAT solver can learn it without restrictions. Moreover, all literals in this clause except for $x_k \leq \mathcal{L}'(x_k)$ are asserted as unsatisfiable in the current model of our SAT solver. Therefore, the SAT solver can assert the literal $x_k \leq \mathcal{L}'(x_k)$ on its own through unit propagation.

Symmetrically, `FixEqs`(y_i) asserts for every bound $(x_k \leq \mathcal{U}'(x_k)) \in C'$ the bound $x_k \geq \mathcal{U}'(x_k)$ and the justification for this bound is the clause:

$$\left(\bigvee_{(x_i \geq \mathcal{L}'(x_i)) \in C'} x_i < \mathcal{L}'(x_i)\right) \vee \left(\bigvee_{(x_i \leq \mathcal{U}'(x_i)) \in C', x_i \neq x_k} x_i > \mathcal{U}'(x_i)\right) \vee (x_k \geq \mathcal{U}'(x_k)).$$

Note also that all justifications we defined are in some sense minimal: each of the above clauses is a tautology and, if we remove one literal from the clause, then it is no longer a tautology. This fact is another property that enhances any potential conflict analysis.

5.4 Application: The Nelson-Oppen Method

In this Section, we explain how the integration of our methods in the simplex algorithm can be used for the combination of theories with the Nelson-Oppen method. For the Nelson-Oppen style combination of theories inside an SMT solver [36], each theory solver has to return all valid equations between variables in its theory. Linear arithmetic theory solvers sometimes guess these equations based on one satisfying assignment. Then the equations are transferred according to the Nelson-Oppen method without verification. This leads to a backtrack of the combination procedure in case the guess was wrong and eventually led to a conflict. With the availability of an equality basis, the guesses can be verified directly and efficiently. Therefore, the method helps the theory solver in avoiding any conflicts due to wrong guesses together with the overhead of backtracking. This comes at the price of computing the equality basis, which should be negligible because the integration we propose is incremental and includes justified simplifications.

5.4.1 Finding Valid Equations Between Variables

We can efficiently find all valid equations between variables as needed for the Nelson-Oppen style combination of theories if we first apply `FindTBnds`(). Then, we use the substitution σ that we get from the tableau and the tightly bounded variables to get a normalized term that represents each variable x_i. If the variable z_j is non-basic and tightly bounded (i.e., $\mathcal{L}(z_j) = \mathcal{U}(z_j)$), then the normalized term is the constant value $\mathcal{L}(z_j)$. If the variable z_j is non-basic and not tightly bounded (i.e., $\mathcal{L}(z_j) \neq \mathcal{U}(z_j)$), then the normalized term is the variable z_j itself. If the variable y_i is basic, then the normalized term is

$$\left(\sum_{z_j \in \mathcal{N}, \mathcal{L}(z_j) \neq \mathcal{U}(z_j)} a_{ij} z_j\right) + \left(\sum_{z_j \in \mathcal{N}, \mathcal{L}(z_j) = \mathcal{U}(z_j)} a_{ij} \mathcal{L}(z_j)\right),$$

where all basic mathematical operations between constant values are replaced by the results of those operations.

We know from Lemma 5.2.6 that $x_i\sigma = x_k\sigma$ simplifies to $0 = 0$ if σ is the substitution we get from an equality basis and $x_i = x_k$ is implied by our constraints. Therefore, both $x_i\sigma$ and $x_k\sigma$ must be represented by the same normalized term if x_i and x_k are equivalent. So the equality basis together with a normalization procedure has turned semantic equivalence into syntactic equivalence. It is very easy to find variables x_i represented by the same normalized term if we store these terms in a DAG, which most SMT solvers already provide for assigning slack variables during the transformation into tableau representation (see Chapter 2.4.2).

5.4.2 Nelson-Oppen Justifications

If we use the equality basis computed by the function `FindTBnds`() for a Nelson-Oppen style combination of theories, then we also assert equalities $x_i = x_k$ for all pairs of equivalent variables x_i, x_k. Since these equalities have been asserted independently of the SAT solver, we have to justify these assertions to the SAT solver.

We get these justifications by looking at the normalized representations of the variables x_i and x_k that are equivalent. The current set of non-basic variables defines a basis and, therefore, already on its own a normalized representation for all variables. Since this normalized representation only depends on the current tableau $Az = y$, it is also independent of any of the asserted bounds. The normalized representation we use for the Nelson-Oppen style combination is only an extension of this representation by the tight bounds $z_j = c_j$ of all tightly bounded non-basic variables. Therefore, the equality $x_i = x_k$ is implied by those tight bounds $z_j = c_j$ that were actively used to compute this representation.

For instance, if z_i and z_k are both non-basic and equivalent, then both variables must be tightly bounded so that $z_i = z_k = v$. Otherwise, they cannot have the same normalized representation. Therefore, $z_i = v$ and $z_k = v$ imply $z_i = z_k$, or as a clause:

$$(z_i < v \vee z_i > v) \vee (z_k < v \vee z_k > v) \vee (z_i = z_k)\,.$$

Next, we look at the case where two basic variables y_i and y_k are equivalent. But before we give the complete formal justification, let us look at an example. Let the variables z_1, z_2, z_3, z_4, z_5 be non-basic and the variables y_1 and y_2 be basic. In this example, the basic variables are defined by the non-basic variables as follows: $y_1 = 2z_1 - z_2 + 3z_4$ and $y_2 = 2z_1 - z_2 + z_5$. Moreover, let the variables z_2, z_3, z_4, and z_5 be tightly bounded such that $z_2 = 1$, $z_3 = 0$, $z_4 = 1$, and $z_5 = 3$. If we now replace the tightly bounded non-basic variables, in the definitions of y_1 and y_2 we get that both of their normalized representations are $2z_1$ and we have actively used the tight bounds $z_2 = 1, z_4 = 1, z_5 = 3$ to compute this normalization. Hence, $y_1 = y_2$ is implied by the tight bounds $z_2 = 1$, $z_4 = 1$, and $z_5 = 3$. The variables y_1 and y_2 are also equivalent if we had not asserted that $z_2 = 1$ because the normalized representation of both variables without $z_2 = 1$ is $2z_1 - z_2$. Hence, $z_2 = 1$ is redundant for the justification and $y_1 = y_2$ is also implied by just the tight bounds $z_4 = 1$ and $z_5 = 3$.

To find which tightly bounded variables are redundant, we can just look at the coefficients. If a_{ij} and a_{kj} are the same, then any tight bound $z_j = c_j$ is redundant in the justification. This gives us the following clause as a general justification:

$$\Big(\bigvee_{z_j \in \mathcal{N}, \mathcal{L}(z_j) = \mathcal{U}(z_j), a_{ij} \neq a_{kj}} z_j < \mathcal{L}(z_j) \vee z_j > \mathcal{U}(z_j)\Big) \vee y_i = y_k \tag{5.3}$$

From this clause, we also get the justification for the case where y_i is basic and z_k non-basic. We simply treat z_k as if it were defined as a basic variable by itself ($z_k = 1 \cdot z_k$), so $a_{kk} = 1$ and all other $a_{kj} = 0$. If we simplify these restrictions in the clause justification (5.3) for the case with two basic variables, then we receive the following general justification for the mixed case:

$$\Big(\bigvee_{z_j \in \mathcal{N} \setminus \{z_k\}, \mathcal{L}(z_j) = \mathcal{U}(z_j), a_{ij} \neq 0} z_j < \mathcal{L}(z_j) \vee z_j > \mathcal{U}(z_j)\Big) \vee \Big(\bigvee_{z_j \in \{z_k\}, \mathcal{L}(z_j) = \mathcal{U}(z_j), a_{ij} \neq 1} z_j < \mathcal{L}(z_j) \vee z_j > \mathcal{U}(z_j)\Big) \vee y_i = z_k$$

All literals in the above clauses except for the variable equivalences $x_i = x_k$ are asserted as unsatisfiable in the current model of our SAT solver. This holds because these literals contain only tightly bounded variables. Hence, the SAT solver can assert the literal $x_i = x_k$ on its own through unit propagation. Note also that all justifications we defined are in some sense minimal: each of the above clauses is a tautology and, if we remove one literal from the clause, then it is no longer a tautology. This fact is another property that enhances any potential conflict analysis.

5.5 Application: Exploration of (Un)Bounded Directions

Another application of the equality basis lies in the exploration of *bounded and unbounded directions* [3]. In this section, we present techniques based on the equality basis that (i) find/detect (un)bounded directions and (ii) compute a *bounded basis* (Definition 2.8.5), i.e., a finite representation of all bounded directions in a system of inequalities. In Chapter 6, we use these methods to extend branch-and-bound into a complete decision procedure for linear mixed arithmetic.

For the exploration of (un)bounded directions, we exploit connections between the two inequality systems $Ax \leq b$ and $Ax \leq 0^m$. Both systems imply only inequalities that are related to one another because both systems have the same coefficients (see Lemma 2.5.5). For instance, if $Ax \leq b$ implies $h^T x \leq g$, then $Ax \leq 0^m$ also implies an inequality $h^T x \leq g'$, and vice versa. This means that both polyhedra also share the same bounded directions h. However, $Ax \leq 0^m$ has an advantage over $Ax \leq b$. If we look again at Lemma 2.5.5, then we see that $Ax \leq 0^m$ also implies $h^T x \leq 0$ whenever it implies an inequality $h^T x \leq g$. This leads us to the conclusion that all bounded directions h in $Ax \leq 0^m$—and, therefore, also in $Ax \leq b$—correspond to implied equalities $h^T x = 0$ in $Ax \leq 0^m$.

Lemma 5.5.1 (Bounds and Equalities)**.** *Let $\mathcal{Q}_\delta(Ax \leq b) \neq \emptyset$. Then h is bounded in $Ax \leq b$ iff $Ax \leq 0^m$ implies that $h^T x = 0$.*

Proof. By Definition 2.8.1, h is bounded in $Ax \leq b$ means that there exists $l, u \in \mathbb{Q}_\delta$ such that $Ax \leq b$ implies $h^T x \leq u$ and $-h^T x \leq -l$. By Lemma 2.5.5, this is equivalent to: there exist $l, u \in \mathbb{Q}_\delta$, $y, z \in \mathbb{Q}^m$ with $y, z \geq 0^m$, and $y^T A = h^T = -z^T A$ so that $y^T b \leq u$ and $z^T b \leq -l$. Symmetrically, $Ax \leq 0$ implies that $h^T x = 0$ is equivalent to: there exist a $y, z \in \mathbb{Q}^m$ with $y, z \geq 0^m$ and $y^T A = h^T = -z^T A$ so that $y^T 0^m \leq 0$ and $z^T 0^m \leq 0$. Since u and l only have to exist, we can trivially choose them as $u := y^T b$ and $l := -z^T b$. This means that $y^T b \leq u$, $z^T b \leq -l$, $y^T 0^m \leq 0$, and $z^T 0^m \leq 0$ are all trivially satisfied by any pair of linear combinations $y, z \in \mathbb{Q}^m$ with $y, z \geq 0^m$ such that $y^T A = h^T = -z^T A$. Hence, the two definitions are equivalent and our lemma holds. □

In Section 5.2, we presented our method that finds an equality basis $Dx = c$ for any inequality system. Together with the above lemma, we can also use it to compute a bounded basis for any inequality system $Ax \leq b$: we simply compute an equality basis $Dx = 0$ for $Ax \leq 0^m$ and D is our bounded basis.

[3] See Chapter 2.8 for an introduction into the terminology and formal basis of (un)bounded directions.

We can then use the bounded basis to determine whether a direction h is bounded in $Ax \leq b$. A direction h is bounded in $Ax \leq b$ if and only if it is a *linear combination of* the row *vectors* in D, i.e., there exist a $y \in \mathbb{Q}^n$ such that $y^T D = h^T$.

5.6 Application: Quantifier Elimination

The methods and theories presented in this chapter have a surprisingly wide range of application. Besides our original goal of investigating equalities, we have also already presented applications for theory combinations with the Nelson-Oppen method and for the exploration of (un)bounded directions. In this section, we show that our methods around equalities are also useful for quantifier elimination.

In general, a *quantifier elimination* (QE) procedure takes a formula $\exists y.\phi(y)$, where $\phi(y)$ itself is quantifier-free but may contain extra variables x called *parameters*, and returns an equivalent formula ϕ' that is quantifier-free. Quantifier elimination procedures typically eliminate the quantifier $\exists y$ by introducing a case distinctions over the values the variable y can assume.

Linear virtual substitution is a complete QE procedure for the theory of linear rational arithmetic [105]. It eliminates the variable y by creating a case distinction [4] exploiting the following fact: a linear rational arithmetic formula $\phi(y)$ is rationally satisfiable if and only if $\phi(l)$ is satisfiable, where l is the strictest lower bound (or upper bound) of y, i.e., the smallest value for y in any solution to the problem. This value is either represented by $-\infty$ ($+\infty$) or one of the inequalities in $\phi(y)$ containing y. There are only finitely many inequalities in $\phi(y)$, so by a case distinction over all inequalities containing y satisfiability can be preserved:

$$\exists y \in \mathbb{Q}.\phi(y) \equiv \phi(-\infty) \vee \bigvee_{a_{iy}y + a_i^T x \leq b_i \text{ in } \phi(y) \text{ with } a_{iy} < 0} \phi\left(-\frac{b_i}{a_{iy}} + \frac{a_i^T x}{a_{iy}}\right).$$

This case distinction is the source of the worst case doubly exponential complexity of the procedure in case of quantifier alternations. At the same time, there are also instances that we can resolve without case distinctions. For instance, if the formula $\phi(y)$ implies an equality $h_y \cdot y + h^T x = g$ where $h_y \neq 0$, then we already know one guaranteed definition for the strictest lower bound of y:

$$\frac{g}{h_y} - \frac{h^T x}{h_y}.$$

A quantifier-free formula that is equivalent to the original one is simply:

$$\exists y \in \mathbb{Q}.\phi(y) \equiv \phi\left(\frac{g}{h_y} - \frac{h^T x}{h_y}\right).$$

This technique is well-known and integrated in many QE implementati-

[4] There actually exist various versions of linear virtual substitution, each based on a different case distinction over the values the variable y can assume. For the presented application of the equality basis, the actual version of linear virtual substitution is irrelevant.

ons [54, 105, 131]. Even so, we are unaware of any implementation that makes use of non-explicit equalities for this purpose. This is where our methods that find implicit equalities come into play. Our methods are applicable because QE procedures typically keep ϕ in a disjunctive form and the respective disjuncts contain often only conjuncts of inequalities. This allows us to efficiently search for an equality.

A similar trick also exists for the theory of linear integer/mixed arithmetic, i.e., for eliminating an integer variable. We only have to additionally verify that the lower bound of the integer variable is an actual integer value. which we can do through one additional divisibility constraint. Therefore,

$$\phi\left(\frac{g}{h_y} - \frac{h^T x}{h_y}\right) \cup \{(h_y | g + h^T x)\}$$

is a quantifier-free formula that is equivalent to $\exists y \in \mathbb{Z}.\phi(y)$ if $\phi(y)$ implies the equality $h_y \cdot y + h^T x = g$ where $h_y \neq 0$.

Note that our methods detect only equalities implied by rational solutions of $Ax \leq b$, i.e., only those equalities $h^T x = g$ that are satisfied by all solutions $x \in \mathcal{Q}_\delta(Ax \leq b)$. There might be integer/mixed entailed equalities that are not generally implied, which means that they cannot be computed with our equality basis method.

5.7 Summary

We have presented in Chapter 4 the linear cube transformation (Corollary 4.2.2), which allows us to efficiently determine whether a polyhedron contains a cube of a given edge length. One obstacle for our cube tests are equalities. Resolving these obstacles led to an additional application of the linear cube transformation: investigating equalities. Through Lemmas 5.2.2 & 5.2.3, we have presented a method that efficiently checks whether a system of linear arithmetic constraints implies an equality at all. We use this method in the algorithm `EqBasis`$(Ax \leq b)$ to compute an equality basis $y - Dz = c$, which is a finite representation of all equalities implied by the inequalities $Ax \leq b$.

We also presented various applications for the equality basis $y - Dz = c$. (i) We can use the equality basis to eliminate all equalities from $Ax \leq b$. It is, therefore, useful as a preprocessing step for our cube tests. (ii) We can use the equality basis to directly check whether an equality $h^T x = g$ is implied by $Ax \leq b$. (iii) We can use the equality basis to efficiently compute all pairs of equivalent variables in $Ax \leq b$ (Section 5.4). These pairs are necessary for a backjump-free *Nelson-Oppen* style combination of theories. (iv) We can use the equality basis to efficiently compute a bounded basis for $Ax \leq b$, i.e., a finite representation of all bounded directions in $Ax \leq b$ (Section 5.5). (v) We can use the equality basis for quantifier elimination (Section 5.6).

Chapter 6

A Reduction (from Unbounded Linear Mixed Arithmetic Problems) into Bounded Problems

We have already presented two complete decision procedures for linear integer arithmetic: branch-and-bound extended by *a priori* bounds (Chapter 2.8) and CutSat++ (Chapter 3).

A priori bounds complete branch-and-bound (see Chapter 2.7.3) by extending the input problem with approximated variable bounds $-g \leq x_i \leq g$ for every variable x_i. Adding the variable bounds creates an equisatisfiable problem because the bounds $-g \leq x_i \leq g$ are approximated large enough, i.e., the original problem has an integer/mixed solution within the bounds whenever it has one outside the bounds. Since the *a priori* bounds bound all variables, they also reduce the search space for a branch-and-bound solver (and many other integer/mixed arithmetic decision procedures) to a finite search space. So branch-and-bound is guaranteed to terminate. However, these bounds are so large that the resulting search space cannot be explored in reasonable time (e.g. the life time of earth) even for small problems. One reason why the *a priori* bounds are so impractically large is that they only take parameter sizes into account and not actually the structure of each problem. Therefore, the approximated bound values are by far larger than is necessary.

CutSat++ is a CDCL style algorithm combined with a lazy quantifier elimination procedure. This combination allows us to use a much more structure-oriented approach for completeness. The price for this result is an algorithm that is by far more complicated than the branch-and-bound approach. In particular, (i) it has to consider divisibility constraints in addition to inequalities; (ii) CutSat++ is missing several features that make

CDCL style SAT solving so efficient in practice, e.g., exhaustive propagation cannot be done in CutSat++ or we encounter propagation divergence; (iii) it is very inflexible, i.e., there is not much room to improve its search strategy because of its strict termination strategy; and (iv) an extension to mixed arithmetic would make it even more complex and make it even more inflexible.

Due to their disadvantages, neither *a priori* bounds or CutSat++ have been integrated in any state-of-the-art SMT solvers [9, 40, 41, 49, 56]. As far as we know, none of the state-of-the-art SMT solvers use any method that guarantees termination for linear integer or mixed arithmetic. Therefore, they are all incomplete.

In this chapter, we present a different approach to reach a complete decision procedure. Like *a priori* bounds, our approach consists of mixed equisatisfiable transformations that turn every problem into a bounded problem. This means that we can also use them to make branch-and-bound (and many other integer/mixed arithmetic decision procedures) complete. In contrast to *a priori* bounds, our transformations do not use approximations, which also means that we never have to deal with over-approximation and the resulting loss of efficiency. Instead, our transformations orient themselves on the structure of the input problem by eliminating the unbounded directions in the problem. This sounds similar to CutSat++, but our transformations manage to avoid the shortcomings of the CutSat++ calculus. This means that (i) they do not introduce new types of constraints; (ii) the combination of branch-and-bound and our transformations stays flexible; and (iii) the extension to mixed arithmetic is intuitive and requires no major changes. As a result, our approach is efficient in practice and we can even support this claim with benchmark experiments.

We already explained in Chapter 2.8 that the *(un)boundedness* of a problem indicates whether branch-and-bound terminates. For this analysis, we partitioned our input problems into four categories: *guarded* problems, *unguarded but bounded* problems, *absolutely unbounded* problems, and *partially unbounded* problems. Guarded problems and unguarded but bounded problems compose together all bounded problems, i.e., all problems for which branch-and-bound has a finite search space and, therefore, always terminates. Absolutely unbounded problems are unbounded, which causes some problems for branch-and-bound alone. The truth is, however, that they are quite trivial because they always have an integer solution. In Chapter 4, we described two *cube tests* that detect and solve absolutely unbounded problems in polynomial time. Therefore, branch-and-bound becomes also complete for absolutely unbounded problems if we extended it with our cube tests. The actual difficult case is when some directions are bounded and others unbounded, i.e., when the problem is partially unbounded. Here, branch-and-bound and most other algorithms diverge or become inefficient in practice. The transformations, which we present, are designed to ef-

ficiently handle this category of problems. Our approach is also efficient because we only actually apply our transformations on partially unbounded problems. We avoid the application to the other types of problems because we use the method from Chapter 5.5 to efficiently determine the type of (un)boundedness of our input problems.

The contributions of this chapter are as follows: We present satisfiability preserving transformations that reduce unbounded problems into bounded problems. On these bounded problems, most linear mixed decision procedures become terminating, which we show on the example of branch-and-bound. The transformations are, therefore, extensions that complete other decision procedures. Our transformations work by eliminating unbounded variables. First, we use the Double-Bounded reduction (Section 6.3) to eliminate all unbounded inequalities from our constraint system. Then we use the Mixed-Echelon-Hermite transformation (Section 6.2) to shift the variables of our system to ones that are either bounded or do not appear in the new inequalities and are, therefore, eliminated. With Corollary 6.2.9 & Lemma 6.3.5 we explain how to efficiently convert certificates of (un)satisfiability between the transformed and the original system. Our method is efficient because it is fully guided by the structure of the problem. This is confirmed by experiments (Section 6.5). Finally, we explain in Section 6.4 how to implement the presented procedures in an incrementally efficient way. This is relevant for an efficient SMT implementation.

6.1 Related Work and Preliminaries

This chapter is based on the publication [30] and focuses on the theory of linear mixed arithmetic. Nonetheless, we always specify the type of solution/satisfiability. Only implications are always assumed to be rational/general entailments (see Chapter 2.5).

The constraints in this chapter are non-strict inequalities and they are either formatted according to the vector representation, i.e., $a_i^T x \leq b_i$ (see also Chapter 2.2.1), or the standard representation, $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ (see also Chapter 2.2.1). Other types of constraints have to be reduced to non-strict inequalities with the techniques presented in Chapter 2.3.

This chapter builds on the basics of linear algebra (Chapter 2.1) and linear arithmetic (Chapter 2.2), on the concept of implied constraints (Chapter 2.5), and on the definitions of (un)bounded and (un)guarded problems and variables (Chapter 2.8). Our incremental implementation (Section 6.4) also builds on the notions and definitions of standard arithmetic decision procedures for SMT solvers as presented in Chapter 2.7.

There also exist several publications by other authors that are highly relevant to the contributions presented in this chapter. The first related work is by Bobot et al. [26]. They present in their paper [26] a complete decision procedure for linear integer arithmetic, which is also the most similar approach to our transformations that we found in the literature. The decision procedure by Bobot et al. dynamically eliminates one linear independent bounded direction at a time via transformation. For comparison, our own transformations statically eliminate all unbounded directions at once. The disadvantage of the dynamic approach is that it is very restrictive and does not leave enough freedom to change strategies or to add complementing techniques. Moreover, Bobot et al.'s implementation of the decision procedure in the SMT solver Ctrl-Ergo uses this transformation approach for all problems and not only the partially unbounded ones, which sometimes leads to a massive overhead on bounded problems.

The second related work is by Christ and Hoenicke [39]. They present in their paper [39] an extension to branch-and-bound that uses the Mixed-Echelon-Hermite transformation to find more versatile branches and cuts. Since one of our proposed transformations is also the Mixed-Echelon-Hermite transformation, there exist some interesting similarities and relationships between their approach and our approach. For instance, our Double-Bounded reduction alone would be sufficient to make Christ and Hoenicke's extension complete.

The third related work is by Philipp Rümmer [126]. He presents in his paper [126] a constraint sequent calculus that is complete for quantified linear integer arithmetic. It terminates because it simulates in the worst case the complete and terminating Omega Test [122]. Moreover, it can be extended so it produces interpolants for ground formulas [29]. This means Rümmer's calculus performs tasks that are more general than our intended task: determining satisfiability of systems of inequalities. This broader focus also leads to one big disadvantage: Rümmer's calculus performs our intended task less efficiently than more specialized methods. This is especially true for determining satisfiability of unbounded problems (see Section 6.5).

6.2 Mixed-Echelon-Hermite Transformation

Our goal is to present an equisatisfiable transformation that turns any constraint system into a system that is bounded, i.e., a system on which branch-and-bound and many other arithmetic decision procedures terminate. In this section, we only present such a transformation for a subset of constraint systems, which we call *double-bounded constraint systems*. We then show in

the next section that each constraint system can be reduced to an equisatisfiable double-bounded system. We also show how to efficiently transform a mixed solution from the double-bounded reduction to a mixed solution for the original system.

Definition 6.2.1 (Double-Bounded Constraint System)**.** A constraint system $Dx \leq u$ is *double-bounded* if $Dx \leq u$ implies $Dx \geq l$ for $l \in \mathbb{Q}^m$. For such a double-bounded system, we call the bounds u the *upper bounds* of Dx and the bounds l the lower bounds of Dx. Moreover, we typically write $l \leq Dx \leq u$ instead of $Dx \leq u$ although the *lower bounds* l are only implicit.

Note that only the inequalities in a double-bounded constraint system are guaranteed to be bounded. Variables might still be unbounded. For instance, in the constraint system $1 \leq 3x_1 - 3x_2 \leq 2$ both inequalities are bounded but the variables x_1 and x_2 are not. Moreover, the above constraint system is also an example where branch-and-bound diverges (see Example 2.8.3 in Chapter 2.8). This means that even bounding all inequalities does not yet guarantee termination. So for our purposes, a double-bounded constraint system is still too complex.

This changes, however, if we also require that the coefficient matrix D of our constraint system is a *lower triangular matrix with gaps*:

Definition 6.2.2 (Lower Triangular with Gaps)**.** A matrix $A \in \mathbb{Q}^{m \times n}$ is *lower triangular with gaps* if it holds for each column j that $\text{piv}(A, j) > m$ or that $\text{piv}(A, j) < \text{piv}(A, k)$ for all columns k with $j < k \leq n$, i.e., column j either has only zero entries or all pivoting entries right of j have a higher row index.

A matrix is *lower triangular* if and only if the row indices of its pivots are strictly increasing, i.e., $\text{piv}(A, 1) < \ldots < \text{piv}(A, n)$. If we also allow it to have gaps, only the row indices of pivots with non-zero columns have to be strictly increasing. Now we get termination for free because of our restrictions:

Lemma 6.2.3 (Lower Triangular Double-Bounded Systems)**.** *Let the matrix $D \in \mathbb{Q}^{m \times n}$ be lower triangular with gaps and $l \leq Dx \leq u$ be a double-bounded constraint system. Then each variable x_j is either bounded, i.e., $l \leq Dx \leq u$ implies that $l'_j \leq x_j \leq u'_j$, or its column in D has only zero entries.*

Proof. Proof by induction. Assume that the above property already holds for all variables x_k with $k < j$. Let $p = \text{piv}(D, j)$. If $p > m$, then the column j of D is zero and we are done. If $p \leq m$, then the pivoting entry d_{pj} of column j is non-zero. Because of Definition 6.2.2 and our induction hypothesis, this also means that each column k with $k < j$ has either a zero entry in row p or the variable x_k is bounded by our induction hypothesis,

i.e., $l \leq Dx \leq u$ implies $l'_k \leq x_k \leq u'_k$. Since Definition 6.2.2 also implies that row p has only zero entries to the right of d_{pj}, the row p has only one unbounded variable with a non-zero entry, viz., x_j. This means we can transform the row $l_p \leq d_p^T x \leq u_p$ into the following two inequalities: $l_p - \sum_{k=1}^{j-1} d_{pk}x_k \leq d_{pj}x_j$ and $u_p - \sum_{k=1}^{j-1} d_{pk}x_k \geq d_{pj}x_j$, where the variables x_k on the left sides are either bounded or $d_{pk} = 0$. Hence, we can derive an upper and lower bound for x_j via bound propagation/refinement (see Chapter 2.7.2). □

Corollary 6.2.4 (BnB-LTDB-Termination)**.** *Branch-and-bound terminates on every double-bounded system $l \leq Dx \leq u$ where D is lower triangular with gaps.*

Our next goal is to efficiently transform every double-bounded system $l \leq Dx \leq u$ into an equisatisfiable system that also has a lower triangular coefficient matrix with gaps. We start by defining a class of transformations that do not only preserve mixed equisatisfiability, but are also very expressive.

Definition 6.2.5 (Mixed Column Transformation Matrix [39])**.** Given a mixed constraint system. A matrix $V \in \mathbb{Q}^{n \times n}$ is a *mixed column transformation matrix* if it is invertible and consists of an invertible matrix $V_{(\mathbb{Q})} \in \mathbb{Q}^{n_1 \times n_1}$, a unimodular matrix $V_{(\mathbb{Z})} \in \mathbb{Z}^{n_2 \times n_2}$, and a matrix $V_{(M)} \in \mathbb{Q}^{n_1 \times n_2}$ such that

$$V = \begin{pmatrix} V_{(\mathbb{Q})} & V_{(M)} \\ 0^{n_2 \times n_1} & V_{(\mathbb{Z})} \end{pmatrix}.$$

The formal definition of mixed column transformation matrices may seem anything but intuitive. However, they actually describe a straightforward class of transformations, viz., any combination of *mixed equisatisfiable column transformations* that can be performed on a matrix A. Mixed equisatisfiable column transformations are either (i) multiplying a column by -1; (ii) the swapping of two columns i, j of the same type, i.e., $i, j \leq n_1$ or $i, j > n_1$; (iii) multiplying a rational column j (i.e., $j \leq n_1$) with a non-zero rational factor; (iv) adding a rational multiple of a column j with $j \leq n_1$ to any other column i; and (v) adding an integer multiple of a column $j > n_1$ to a different column i with $i > n_1$. If we perform the same mixed equisatisfiable column transformations that resulted in H from A to an $n \times n$ identity matrix, then the transformed identity matrix V is a mixed column transformation matrix and $H = AV$. We just compacted the column transformations into a matrix. We cannot just use V to redo the column transformations, but we can also use its inverse to undo them:

Lemma 6.2.6 (Mixed Column Transformation Inversion [39])**.** *Given a mixed constraint system. Let $V \in \mathbb{Q}^{n \times n}$ be a mixed column transformation matrix. Then V^{-1} is also a mixed column transformation matrix.*

This means that each mixed column transformation matrix defines a bijection from $(\mathbb{Q}^{n_1} \times \mathbb{Z}^{n_2})$ to $(\mathbb{Q}^{n_1} \times \mathbb{Z}^{n_2})$. Hence, they guarantee mixed equisatisfiability:

Lemma 6.2.7 (Mixed Column Transformation Equisatisfiability [39])**.** *Let $Ax \leq b$ be a mixed constraint system. Let $V \in \mathbb{Q}^{n \times n}$ be a mixed column transformation matrix. Then $(AV)y \leq b$ and $Ax \leq b$ are mixed equisatisfiable and every solution $y \in \mathcal{M}((AV)y \leq b))$ can be converted into a solution $Vy = x \in \mathcal{M}(Ax \leq b)$ and vice versa.*

Moreover, the mixed column transformation matrix V also establishes a direct relationship between the linear combinations of the original constraint system and the transformed one:

Lemma 6.2.8 (Mixed Column Transformation Implications)**.** *Let $Ax \leq b$ be a constraint system. Let $V \in \mathbb{Q}^{n \times n}$ be a mixed column transformation matrix. Let $Ax \leq b$ imply $h^T x \leq g$. Then $AVz \leq b$ implies $h^T Vz \leq g$.*

Proof. By Lemma 2.5.5, $Ax \leq b$ implies $h^T x \leq g$ iff there exists a non-negative linear combination $y \in \mathbb{Q}^n$ such that $y \geq 0$, $y^T A = h^T$ and $y^T b \leq g$. Multiplying $y^T A = h^T$ with V results in $y^T AV = h^T V$ and thus y is also the non-negative linear combination of inequalities $AVz \leq b$ that results in $h^T Vz \leq g$. □

Corollary 6.2.9 (Mixed Column Transformation Certificates)**.** *Let $Ax \leq b$ be a constraint system. Let $V \in \mathbb{Q}^{n \times n}$ be a mixed column transformation matrix. Then y is a certificate of unsatisfiability*[1] *for $Ax \leq b$ iff it is one for $AVz \leq b$.*

Now we only need a mixed column transformation matrix V for every coefficient matrix A such that $H = AV$ is lower triangular with gaps. One such matrix V is the one that transforms A into *Mixed-Echelon-Hermite normal form*:

Definition 6.2.10 (Mixed-Echelon-Hermite Normal Form [39])**.** A matrix $H \in \mathbb{Q}^{m \times n}$ is in *Mixed-Echelon-Hermite normal form* if

$$H = \begin{pmatrix} E & 0^{r \times (n_1 - r)} & 0^{r \times n_2} \\ E' & 0^{(m-r) \times (n_1 - r)} & H' \end{pmatrix},$$

where E is an $r \times r$ identity matrix (with $r \leq n_1$), $E' \in \mathbb{Q}^{(m-r) \times r}$, and $H' \in \mathbb{Q}^{(m-r) \times n_2}$ is a matrix in hermite normal form, i.e., a lower triangular matrix without gaps, where each entry h'_{pk} in the row $p = \text{piv}(H', j)$ is non-negative and smaller than h'_{pj}).

The following proof for the existence of the Mixed-Echelon-Hermite normal form is constructive and presents the Mixed-Echelon-Hermite transformation.

[1] See Lemma 2.5.4 for the definition of a *certificate of unsatisfiability*

Lemma 6.2.11 (Mixed-Echelon-Hermite Transformation)**.** *Let $A \in \mathbb{Q}^{m\times n}$ be a matrix, where the upper left $r \times n_1$ submatrix has the same rank r as the complete left $m \times n_1$ submatrix. Then there exists a mixed transformation matrix $V \in \mathbb{Q}^{n\times n}$ such that $H = AV$ is in* Mixed-Echelon-Hermite normal form*.*

Proof. Proof from [39] with slight modifications so it also works for singular matrices. We subdivide A into

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

such that $A_{11} \in \mathbb{Q}^{r\times n_1}$, $A_{12} \in \mathbb{Q}^{r\times n_2}$, $A_{21} \in \mathbb{Q}^{m-r\times n_1}$, and $A_{21} \in \mathbb{Q}^{m-r\times n_2}$. Then we bring A_{11} with an invertible matrix $V_{11} \in \mathbb{Q}^{n_1\times n_1}$ into reduced echelon column form $H_{11} = (E\ 0^{r\times(n_1-r)}) = A_{11}V_{11}$, where E is an $r \times r$ identity matrix. We get V_{11} and H_{11} by using Bareiss algorithm instead of the better known Gaussian elimination as it is polynomial in time [8].[2] Note that the last $n_1 - r$ columns of $H_{21} = (H'_{21}\ 0^{(m-r)\times(n_1-r)}) = A_{21}V_{11}$ are also zero because all rows in A_{21} are linear dependent of A_{11} (due to the rank). Next we notice that

$$A_{12} - A_{11}V_{11}\begin{pmatrix} A_{12} \\ 0^{(n_1-r)\times n_2} \end{pmatrix} = A_{12} - (E\ 0^{r\times(n_1-r)})\begin{pmatrix} A_{12} \\ 0^{(n_1-r)\times n_2} \end{pmatrix} = 0^{r\times n_2}$$

so we can reduce the upper right submatrix A_{12} to zero by adding multiples of the n_1 columns with rational variables to the n_2 columns with integer variables. However, this also transforms the lower right submatrix A_{22} into

$$H'_{22} = A_{22} - A_{21}V_{11}\begin{pmatrix} A_{12} \\ 0^{(n_1-r)\times n_2} \end{pmatrix}.$$

Finally, we transform this new submatrix H'_{22} into hermite normal form H_{22} via the algorithm of Kannan and Bachem [91] (or a similar polynomial time algorithm). This algorithm also returns a unimodular matrix $V_{22} \in \mathbb{Z}^{n_2\times n_2}$ such that $H_{22} = H'_{22}V_{22}$. To summarize: our total mixed transformation matrix is

$$V = \begin{pmatrix} V_{11} & -V_{11} \cdot \begin{pmatrix} A_{12} \\ 0^{(n_1-r)\times n_2} \end{pmatrix} \cdot V_{22} \\ 0^{n_2\times n_1} & V_{22} \end{pmatrix} \text{ and } H = \begin{pmatrix} H_{11} & 0^{r\times n_2} \\ H_{21} & H_{22} \end{pmatrix}. \qquad \square$$

It is not possible to transform every matrix $A \in \mathbb{Q}^{m\times n}$ into Mixed-Echelon-Hermite normal form. We have to restrict ourselves to matrices, where the upper left $r \times n_1$ submatrix has the same rank r as the complete left $m \times n_1$ submatrix. However, this is very easy to accomplish for a system of linear mixed arithmetic constraints $l \leq Ax \leq u$. The reason is that the order of inequalities does not change the set of satisfiable solutions. Hence,

[2] In our implementation, we do actually use less efficient, Gaussian-elimination-based transformations instead of Bareiss algorithm and the algorithm of Kannan and Bachem. The reason is that these transformations are incrementally efficient (see Section 6.4). Our experiments show that the transformation cost still remains negligible in practice.

we can swap the inequalities and, thereby, the rows of A until its upper left $r \times n_1$ submatrix has the desired form. This also means that there are usually multiple possible inequality orderings that each have their own Mixed-Echelon-Hermite normal form H.

To conclude this section: whenever we have a double-bounded constraint system $l \leq Dx \leq u$, we can transform it (after some row swapping) into an equisatisfiable system $l \leq Hy \leq u$ where $H = DV$ is in Mixed-Echelon-Hermite normal form and $Vy = x$. Since H is also a lower triangular matrix with gaps, branch-and-bound terminates on $l \leq Hy \leq u$ with a mixed solution t or it will return unsatisfiable (Corollary 6.2.4). Moreover, we can convert any mixed solution t for $l \leq Hy \leq u$ into a mixed solution s for $l \leq Dx \leq u$ by setting $s := Vt$. Hence, we have a complete algorithm for double-bounded constraint systems.

6.3 Double-Bounded Reduction

In the previous Section, we have shown how to solve a double-bounded constraint system. Now we show how to reduce any system of inequalities $A'x \leq b'$ to a mixed equisatisfiable double-bounded system $l \leq Dx \leq u$. Moreover, we explain how to take any solution of $l \leq Dx \leq u$ and turn it into a solution for $A'x \leq b'$.

As the first step of our reduction, we reformulate the constraint system into a so-called *split system*:

Definition 6.3.1 (Split System)**.** $(Ax \leq b) \cup (l \leq Dx \leq u)$ is a *split system* if: (i) all directions are unbounded in $Ax \leq b$; and (ii) all row vectors a_i from A are also unbounded in $(Ax \leq b) \cup (l \leq Dx \leq u)$. Moreover, we call $Ax \leq b$ the *(absolutely) unbounded part* and $l \leq Dx \leq u$ the *(double-)bounded part* of the split system.

A split system consists of an (absolutely) unbounded part $Ax \leq b$ that is guaranteed to have (infinitely many) integer solutions (see Lemma 2.8.8) and a double-bounded part $l \leq Dx \leq u$ (see Figures 6.1—6.3 for an example). Any constraint system can be brought into the above form. We just have to move all unbounded inequalities into the unbounded part and all bounded inequalities into the bounded part.

Lemma 6.3.2 (Split Equivalence)**.** *Let $A'x \leq b'$ be a system of inequalities. Then there exists an equivalent split system $(Ax \leq b) \cup (l \leq Dx \leq u)$ where: (i)$A' \in \mathbb{Q}^{m \times n}$, $A \in \mathbb{Q}^{m_1 \times n}$, and $D \in \mathbb{Q}^{m_2 \times n}$ so that $m_1 + m_2 = m$; (ii) all rows d_i^T of D and a_k^T of A appear as rows in A'; and (iii) $Dx \leq u$ implies $l \leq Dx$.*

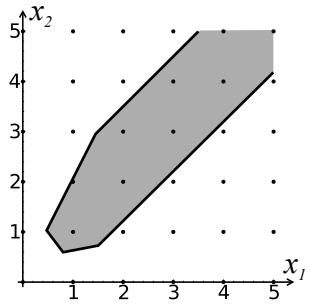


Figure 6.1: A partially bounded system

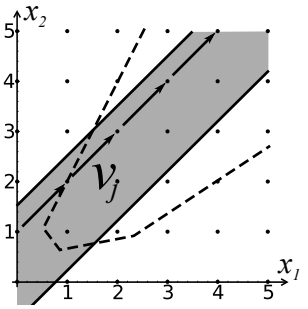


Figure 6.2: The (double-)bounded part of Figure 6.1; $v_j := (1,1)^T$ is an orthogonal direction to the bounding row vectors

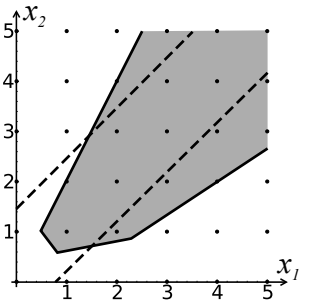


Figure 6.3: The unbounded part of Figure 6.1

Proof. For (i), (ii), and the equivalence, it is enough to move all bounded inequalities $a_i'^T x \leq b_i'$ of $A'x \leq b'$ into $Dx \leq u$ and all unbounded inequalities into $Ax \leq b$. For (iii), we assume for a contradiction that $Dx \leq u$ does not imply $l_i \leq d_i^T x$ but $(Dx \leq u) \cup (Ax \leq b)$ does. By Lemma 2.5.5, this means that there exists a $y \in \mathbb{Q}^{m_2}$ with $y \geq 0^{m_2}$ and a $z \in \mathbb{Q}^{m_1}$ with $z \geq 0^{m_1}$ so that $y^T D + z^T A = -d_i^T$ and $y^T u + z^T b \leq -l_i$. We also know that there exists a $z_k > 0$ because $Dx \leq u$ alone does not imply $l_i \leq d_i^T x$. We use this fact to reformulate $y^T D + z^T A = -d_i^T$ into $-a_k^T = \frac{1}{z_k}\left[y^T D + d_i^T + \sum_{j=1, j\neq k}^{m_1} z_j a_j^T\right]$, and use the bounds of the inequalities in $Dx \leq u$ and $Ax \leq b$ to derive a lower bound for $a_k^T x$: $-a_k^T x \leq \frac{1}{z_k}\left[y^T u + u_i + \sum_{j=1, j\neq k}^{m_1} z_j b_j\right]$. Hence, a_k^T is bounded in $A'x \leq b'$ and we have our contradiction. □

The above Lemma also shows that the bounded part of a constraint system is self-contained, i.e., a constraint system implies that a direction is bounded if and only if its bounded part does. The actual difficulty of reformulating a system into a split system is not the transformation per se, but finding out which inequalities are bounded or not. There are many ways to detect whether an inequality is bounded by a constraint system. Most work even in polynomial time. For instance, solving the linear rational optimization problem "minimize $a_i^T x$ such that $Ax \leq b$" returns $-\infty$ if a_i is unbounded, ∞ if $Ax \leq b$ has no rational solution, and the optimal lower bound l_i for $a_i^T x$ otherwise. However, even with this technique we still need to solve m linear optimization problems to determine for all inequalities $a_i^T x \leq b_i$ whether they are bounded.

A, in our opinion, more efficient alternative is based on our algorithm for finding equality bases (see Chapter 5). In Lemma 5.5.1 of Chapter 5.5, we have shown that $\mathcal{Q}_\delta(Ax \leq b) \neq \emptyset$ implies the following equivalence: the direction h is bounded in $Ax \leq b$ iff $Ax \leq 0^m$ implies $h^T x = 0$. It is easy and efficient to compute an equality basis for $Ax \leq 0^m$ and to determine with it the inequalities in $Ax \leq b$ that are bounded (see also Chapter 5.5). The only disadvantage towards the optimization approach is that we do not derive an optimal lower bound l for the inequalities. This is no problem because only the existence of lower bounds is relevant for our transformations and not the actual bound values.

Although we now know how to transform every system of inequalities into a split system $(Ax \leq b) \cup (l \leq Dx \leq u)$, we still need to reduce the split system to a mixed equisatisfiable double-bounded system. However, this task is trivial because the unbounded part $(Ax \leq b)$ is actually inconsequential to the rational/mixed satisfiability of the system. It may reduce the number of rational/mixed solutions, but it never removes them all. Hence, $(Ax \leq b) \cup (l \leq Dx \leq u)$ is equisatisfiable to just $l \leq Dx \leq u$. We first show this equisatisfiability for the rational case:

Lemma 6.3.3 (Rational Extension)**.** *Let $(Ax \leq b) \cup (l \leq Dx \leq u)$ be a split system. Let $s \in \mathbb{Q}_\delta^n$ be a rational solution to the bounded part $l \leq Dx \leq u$ such that $Ds = g$, where $g \in \mathbb{Q}_\delta^{m_2}$. Then $(Ax \leq b) \cup (Dx = g)$ has a rational solution $s' \in \mathbb{Q}_\delta^n$.*

Proof. Assume for a contradiction that $(Ax \leq b) \cup (Dx = g)$ has no solution. By Lemma 2.5.4, this means that there exist a $y \in \mathbb{Q}^{m_1}$ with $y \geq 0^{m_1}$ and $z, z' \in \mathbb{Q}^{m_2}$ with $z, z' \geq 0^{m_2}$ such that $y^T A + z^T D - z'^T D = (0^n)^T$ and $y^T b + z^T g - z'^T g < 0$. Since $Dx = g$ is satisfiable by itself, there must exist a $y_i > 0$. Now we use this fact to reformulate the equation $y^T A + z^T D - z'^T D = (0^n)^T$ into

$$-a_i^T = \tfrac{1}{y_i} \left[\left(\textstyle\sum_{j=1 j \neq i}^{m_1} y_j a_j^T \right) + z^T D - z'^T D \right],$$

from which we deduce a lower bound for $a_i^T x$ in $(Ax \leq b) \cup (l \leq Dx \leq u)$:

$$-a_i^T x \leq \tfrac{1}{y_i} \left[\left(\textstyle\sum_{j=1 j \neq i}^{m_1} y_j b_j \right) + z^T u - z'^T l \right].$$

Therefore, a_i is bounded in $(Ax \leq b) \cup (l \leq Dx \leq u)$, which is a contradiction. □

Note that the bounded part $l \leq Dx \leq u$ of a split system can still have unbounded directions (not inequalities). Some of these unbounded directions in $l \leq Dx \leq u$ are the orthogonal directions to the row vectors d_i, i.e., vectors $v_j \in \mathbb{Z}^n$ such that $d_i^T v_j = 0$ for all $i \in \{1, \ldots, m_2\}$. This also means that the existence of one mixed solution $s \in (\mathbb{Q}^{n_1} \times \mathbb{Z}^{n_2})$ and one unbounded direction proves the existence of infinitely many mixed solutions. We just need to follow the orthogonal directions, i.e., $s' = \lambda \cdot v_j + s$ is also a mixed solution for all $\lambda \in \mathbb{Z}$ because $d_i^T s' = \lambda \cdot d_i^T v_j + d_i^T s = d_i^T s$.

(See Figures 6.1—6.3 for an example.) In the next two steps, we prove that $Ax \leq b$ cannot cut off all of these orthogonal solutions because it is completely unbounded. The first step proves that $Ax \leq b$ remains absolutely unbounded even if we settle on one set of orthogonal solutions, i.e., enforce $Dx = Ds$ for some solution s.

Lemma 6.3.4 (Persistence of Unboundedness)**.** *Let $(Ax \leq b) \cup (l \leq Dx \leq u)$ be a split system. Let $s \in \mathbb{Q}_\delta^n$ be a rational solution for $l \leq Dx \leq u$ such that $Ds = g$ (with $g \in \mathbb{Q}_\delta^{m_2}$). Then all row vectors a_i from A are still unbounded in $(Ax \leq b) \cup (Dx = g)$.*

Proof. By Lemma 6.3.3, $(Ax \leq b) \cup (Dx = g)$ has at least a rational solution s^*. Moreover, $(Ax \leq 0) \cup (Dx = 0)$ does not imply $a_i^T x = 0$ because of Lemma 5.5.1 and the assumption that the row vectors a_i from A are unbounded in $(Ax \leq b) \cup (l \leq Dx \leq u)$. In reverse, $(Ax \leq b) \cup (Dx = g)$ having a rational solution, $(Ax \leq 0) \cup (Dx = 0)$ does not imply $a_i^T x = 0$, and Lemma 5.5.1 prove together that the row vectors a_i from A are also unbounded in $(Ax \leq b) \cup (Dx = g)$. □

The next step proves how to extend the mixed solution from the bounded part to the complete system with the help of the Mixed-Echelon-Hermite normal form and the absolute unboundedness of $Ax \leq b$.

Lemma 6.3.5 (Mixed Extension)**.** *Let $(Ax \leq b) \cup (l \leq Dx \leq u)$ be a split system. Let $s \in (\mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_2})$ be a mixed solution for $l \leq Dx \leq u$. Then $(Ax \leq b) \cup (l \leq Dx \leq u)$ has a mixed solution $s' \in (\mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_2})$.*

Proof. Let $g = Ds$. Without loss of generality we assume that the upper left $r \times n_1$ submatrix of D has the same rank r as the complete left $m_1 \times n_1$ submatrix of D. (Otherwise, we just reorder the rows accordingly.) Therefore, there exists a mixed column transformation matrix V such that $H = DV$ is in Mixed-Echelon-Hermite normal form (see Lemma 6.2.11). By Lemma 6.2.7, there exists a mixed vector $t \in (\mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_2})$ such that $s = Vt$ and t is a mixed-solution to $l \leq Hy \leq u$ as well as $Hy = g$. Let the set $\mathcal{B}$ contain all column indices j in H that correspond to bounded variables y_j. Let the set $\mathcal{U}$ contain all column indices j in H that correspond to columns with only zeroes as entries. Then $\mathcal{B} \cup \mathcal{U}$ contains all column indices in H (Lemma 6.2.3). Moreover, the equation system $(Hy = g)$ fixes each variable y_j with $j \in \mathcal{B}$ to the value t_j because H is lower triangular with gaps. Hence, $((AV)y \leq b) \cup (Hy = g)$ is equivalent to

$$A \left[\sum_{j \in \mathcal{U}} \begin{pmatrix} v_{1j} \\ \vdots \\ v_{nj} \end{pmatrix} \cdot y_j \right] \leq b - A \left[\sum_{j \in \mathcal{B}} \begin{pmatrix} v_{1j} \\ \vdots \\ v_{nj} \end{pmatrix} \cdot t_j \right]. \tag{6.1}$$

Due to Lemma 6.3.4 and 6.2.8, all directions are unbounded in (6.1). This means (6.1) has an integer solution (Lemma 2.8.8) assigning each variable y_j with $j \in \mathcal{U}$ to a $t'_j \in \mathbb{Z}$. (Can be computed via the unit cube

test form Chapter 4). We extend this solution to all variables y by setting $t'_j := t_j$ for $j \in \mathcal{B}$ and we have a mixed solution $t' \in (\mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_1})$ for $((AV)y \leq b) \cup (l \leq Hy \leq u)$. Hence, we have via Lemma 6.2.7 a mixed solution $s' \in (\mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_2})$ for $(Ax \leq b) \cup (l \leq Dx \leq u)$ with $s' = Vt'$. □

The above proof is constructive and, thereby, also explains how to convert a satisfiable solution (i.e., a certificate of satisfiability) from the bounded part to the complete constraint system. The remainder of the proof of equisatisfiability is trivial:

Corollary 6.3.6 (Double-Bounded Reduction)**.** *A split system* $(Ax \leq b) \cup (l \leq Dx \leq u)$ *is mixed equisatisfiable to its bounded part* $(l \leq Dx \leq u)$.

6.4 Incremental Implementation

Suppose an SMT theory solver has to solve $(Ax \leq b) \cup (Dx \leq c)$. Moreover, the last problem it has solved was $(Ax \leq b)$. Then we call the runtime advantage it gains from having already solved a subset of the problem its *incremental efficiency*. Since all problems sent to an SMT theory solver are incrementally connected, its incremental efficiency is a major factor in determining its total efficiency.

In this section, we explain how to implement our transformation based approach more incrementally efficient and what limits there are with regard to incremental efficiency. We start our discussion with incrementally efficient implementations of the subcomponents of our procedure: finding bounded inequalities and computing the Mixed-Echelon-Hermite normal form.

6.4.1 Finding Bounded Inequalities Incrementally

In Section 6.3, we explained via Lemma 5.5.1 that all bounded directions in $Ax \leq b$ are equalities in $Ax \leq 0^m$ and vice versa. Based on this fact, we recommend to use the method outlined in Chapter 5 to compute an equality basis for $Ax \leq 0^m$ and to determine with it the inequalities in $Ax \leq b$ that are bounded.

We also recommend the equality basis method because it is incrementally efficient (see Chapter 5.3). This incremental efficiency directly translates to determining bounded inequalities. Since determining the bounded inequalities is the bottleneck of splitting (Section 6.3), the incremental efficiency also translates to splitting a constraint system.

Algorithm 15: `ExtendMEH`$(Hy \leq u, V, a_{m+1}^T x \leq b_{m+1})$

Input : A system of inequalities $Hy \leq u$, where $H \in \mathbb{Q}^{m \times n}$ is in Mixed-Echelon-Hermite normal form and $u \in \mathbb{Q}_\delta^m$; a mixed column transformation matrix $V \in \mathbb{Q}^{n \times n}$; and an inequality $a_{m+1}^T x \leq b_{m+1}$, where $a_{m+1} \in \mathbb{Q}^n$ and $b_{m+1} \in \mathbb{Q}_\delta$

Effect : Extends $Hy \leq u$ by $a_{m+1}^T V y \leq b_{m+1}$ and transforms it into Mixed-Echelon-Hermite normal form $H'y \leq u'$ via mixed column transformation operations, which are stored in V'. H' has at most one more non-zero column than H and $(Hy \leq u) \subset (H'y \leq u')$.

Output: $(H'y \leq u', V')$

```
1 h_{m+1}^T := a_{m+1}^T V;
2 p := PivR(m, H);
3 j := PivR(1, h_{m+1}^T);
4 if j > p then return ExtendR(Hy ≤ u, V, h_{m+1}^T x ≤ b_{m+1}, j);
5 p := PivI(m, H);
6 j := PivI(1, h_{m+1}^T);
7 if j > p then return ExtendI(Hy ≤ u, V, h_{m+1}^T x ≤ b_{m+1}, j);
8 return ((Hy ≤ u) ∪ (h_{m+1}^T y ≤ b_{m+1}), V);
```

Figure 6.4: `ExtendMEH`() extends a Mixed-Echelon-Hermite normal form by one inequality. All algorithms used in the transformation are based on algorithms from [128].

6.4.2 An Incremental Version of the Mixed-Echelon-Hermite Transformation

The Mixed-Echelon-Hermite normal form (MEHNF) can also be computed incrementally efficient. However, the polynomial time algorithms [8, 91] for computing the reduced echelon column form and the hermite normal form are typically less incrementally efficient than the ones based on Gaussian elimination [59]. So to achieve incremental efficiency, we have to accept a worst-case exponential runtime. Fortunately, the Gaussian based transformations rarely seem to reach their exponential worst-case in practice. Our experiments support this assumption since the transformation cost is negligible (if not immeasurable) on all of the tested benchmarks (see Section 6.5).

Algorithm 16: `PivR`(k, H)

Input : A row dimension k and a matrix $H \in \mathbb{Q}^{k \times n}$

1 **return** If all columns $j \leq n_1$ in H are 0^k, then 0 is returned. Otherwise, this function returns the largest j such that column j in H is not 0^k and $0 < j \leq n_1$.

Algorithm 17: `PivI`(k, H)

Input : A row dimension k and a matrix $H \in \mathbb{Q}^{k \times n}$

1 **return** If all columns $j > n_1$ in H are 0^k, then n_1 is returned. Otherwise, this function returns the largest j such that column j in H is not 0^k and $n_1 < j \leq n$.

Figure 6.5: `PivR`(k, H) and `PivI`(k, H) determine the largest column index of a non-zero rational/integer column in H

The incrementally efficient version of the Mixed-Echelon-Hermite transformation (see `ExtendMEH`() in Figure 6.4)) works as follows: let $Hy \leq u$ be a system of inequalities in MEHNF and let V be an $n \times n$ mixed column transformation matrix such that $AVy \leq b$ is equivalent to $Hy \leq u$.[3] Moreover, let $A'x \leq b'$ be $Ax \leq b$ incrementally extended by one inequality $a_{m+1}^T x \leq b_{m+1}$, i.e., $A' = (a_1, \ldots, a_m, a_{m+1})^T$ and $b' = (b_1, \ldots, b_m, b_{m+1})^T$. Then $(H'y \leq u', V') :=$ `ExtendMEH`$(Hy \leq u, V, a_{m+1}^T x \leq b_{m+1})$ returns a system of inequalities in MEHNF $H'y \leq u'$ and a mixed column transformation matrix V' such that $A'V'y \leq b'$ is equivalent to $H'y \leq u'$.

As already mentioned in Section 6.2, it is not possible to transform every matrix $A \in \mathbb{Q}^{m \times n}$ into Mixed-Echelon-Hermite normal form. We have to restrict ourselves to matrices, where the upper left $r \times n_1$ submatrix has the same rank r as the complete left $m \times n_1$ submatrix. However, this condition is very easy to fulfill because we are looking at systems of inequalities $Ax \leq b$ and not just matrices. This means we can simply swap the inequalities in $Ax \leq b$ to get the equivalent system $Cx \leq u$, where C's upper left $r \times n_1$ submatrix has the desired form. This swapping is also the reason why the input MEHNF $Hy \leq u$ and output MEHNF $H'y \leq u'$ of `ExtendMEH`() are only equivalent to $AVy \leq b$ and $A'V'y \leq b'$ instead of the stronger conditions $H = AV$ and $H' = A'V'$ we previously used for MEHNF's of matrices.

Otherwise, `ExtendMEH`() works the same as one step of the Gaussian based Mixed-Echelon-Hermite transformation for matrices: `ExtendMEH`() first applies the previous column transformations V to $a_{m+1}^T x \leq b_{m+1}$ to get the inequality $h_{m+1}^T y \leq b_{m+1}$ (line 1). Next, `ExtendMEH`() checks whether h_{m+1}^T has any non-zero entries h_{m+1j} in one of the *zero columns* j of H, i.e., in one

[3] Initially, our system of inequalities $Ax \leq b$ is the empty set, its MEHNF $Hy \leq u$ is also the empty set, and our transformation matrix V is just the $n \times n$ identity matrix.

Algorithm 18: `ExtendR`$(Hy \leq u, V, h_{m+1}^T y \leq b_{m+1}, j)$

Input : A system of inequalities $Hy \leq u$, where $H \in \mathbb{Q}^{m \times n}$ is in Mixed-Echelon-Hermite normal form and $u \in \mathbb{Q}_\delta^m$; a mixed column transformation matrix $V \in \mathbb{Q}^{n \times n}$; an inequality $h_{m+1}^T y \leq b_{m+1}$, where $h_{m+1} \in \mathbb{Q}^n$ and $b_{m+1} \in \mathbb{Q}_\delta$; and a column index $j \leq n_1$ such that $h_{m+1j} \neq 0$ and $h_{ij} = 0$ for all $i \leq m$

Effect : Extends $Hy \leq u$ by $h_{m+1}^T y \leq b_{m+1}$ and transforms it into Mixed-Echelon-Hermite normal form $H'y \leq u'$ via mixed column transformation operations, which are stored in V'. H' has one more non-zero rational column than H and $(Hy \leq u) \subset (H'y \leq u')$.

Output: $(H'y \leq u', V')$

1 $V' := V$;
2 $p :=$ `PivR`$(m, H) + 1$;
3 $H'y \leq u' :=$ Insert $h_{m+1}^T y \leq b_{m+1}$ between rows $p-1$ and p of $Hy \leq u$;
4 $V' :=$ Swap Column j and p in V';
5 $H' :=$ Swap Column j and p in H';
6 $V' :=$ Divide Column p of V' by h'_{pp};
7 $H' :=$ Divide Column p of H' by h'_{pp};
8 **for** $j \in \{1, \ldots, p-1, p+1, \ldots, n\}$ **do**
9 | $V' :=$ Subtract h'_{pj} times column p from column j of V';
10 | $H' :=$ Subtract h'_{pj} times column p from column j of H';
11 **end**
12 **return** $(H'y \leq u', V')$;

Figure 6.6: `ExtendR`$(Hy \leq u, V, h_{m+1}^T y \leq b_{m+1}, j)$

of the columns j of H that consists only of zero entries. If h_{m+1}^T does not have any such entries, then no column transformations are necessary and $H'y \leq u' := (Hy \leq u) \cup (h_{m+1}^T y \leq b_{m+1})$ with $H' = (h_1, \ldots, h_m, h_{m+1})^T$ and $u' = (u_1, \ldots, u_m, u_{m+1})^T$ is in MEHNF (line 8). If h_{m+1}^T fills, however, one of the gaps of H, i.e, has a non-zero coefficient in a zero column of H, then $(Hy \leq u) \cup (h_{m+1}^T y \leq b_{m+1})$ is not in MEHNF. In order to resolve this, we have to distinguish between two cases:

Case 1: h_{m+1}^T fills a *rational gap* of H, i.e., there exists a zero column $0 < j \leq n_1$ in H such that $h_{k+1j} \neq 0$. In this case, we have to extend H from $p-1$ non-zero rational columns to p non-zero rational columns. We do so with the function `ExtendR`() (Figure 6.6). `ExtendR`() first inserts the inequality $h_{m+1}^T y \leq b_{m+1}$ at an appropriate position p (line 3), to solve the rank requirements we discussed before. So in the new constraint system

Algorithm 19: `ExtendI`$(Hy \leq u, V, h_{m+1}^T y \leq b_{m+1}, j)$

Input : A system of inequalities $Hy \leq u$, where $H \in \mathbb{Q}^{m \times n}$ is in Mixed-Echelon-Hermite normal form and $u \in \mathbb{Q}_\delta^m$; a mixed column transformation matrix $V \in \mathbb{Q}^{n \times n}$; an inequality $h_{m+1}^T y \leq b_{m+1}$, where $h_{m+1} \in \mathbb{Q}^n$ and $b_{m+1} \in \mathbb{Q}_\delta$; and a column index $j > n_1$ such that $h_{m+1j} \neq 0$ and $h_{ij} = 0$ for all $i \leq m$

Effect : Extends $Hy \leq u$ by $h_{m+1}^T y \leq b_{m+1}$ and transforms it into Mixed-Echelon-Hermite normal form $H'y \leq u'$ via mixed column transformation operations, which are stored in V'. H' has one more non-zero integer column than H and $(Hy \leq u) \subset (H'y \leq u')$.

Output: $(H'y \leq u', V')$

1 $p :=$ `PivI`$(m, H) + 1$;
2 $H'y \leq u' :=$ Insert $h_{m+1}^T y \leq b_{m+1}$ between rows $p-1$ and p of $Hy \leq u$;
3 $(H', V') :=$ `ReduceLeftI`(H', V, p);
4 $(H', V') :=$ `ReduceRightI`(H', V', p);
5 **return** $(H'y \leq u', V')$;

Figure 6.7: `ExtendI`$(Hy \leq u, V, h_{m+1}^T y \leq b_{m+1}, j)$

$(H'y \leq u')$ the inequality $h_{m+1}^T y \leq b_{m+1}$ is located in row p. Then `ExtendR`() swaps column j with column p and uses column operations to eliminate all other coefficients in h_{m+1}^T that fill gaps in H. The result $H'y \leq u'$ is then again in MEHNF (and V' is the mixed column transformation matrix as specified above). Since all column operations are performed on columns with gaps in H, all inequalities in $Hy \leq u$ also appear in $H'y \leq u'$, i.e., $(Hy \leq u) \subset (H'y \leq u')$.

Case 2: h_{m+1}^T fills no rational gap, but an *integer gap* of H, i.e., there exists a zero column $n_1 < j \leq n$ in H such that $h_{m+1j} \neq 0$. In this case, we have to extend H from $p-1$ non-zero integer columns to p non-zero integer columns. We do so with the function `ExtendI`() (Figure 6.7). `ExtendI`() first inserts the inequality $h_{m+1}^T y \leq b_{m+1}$ at an appropriate position p (line 2), to solve the rank requirements we discussed before. So in the new constraint system $(H'y \leq u')$ the inequality $h_{m+1}^T y \leq b_{m+1}$ is located in row p. Then `ExtendI`() swaps column j with column p and uses column operations to eliminate all other coefficients in h_{m+1}^T that fill gaps in H. The resulting system of inequalities $H'y \leq u'$ is then again in MEHNF (and V' is the transformation matrix as specified above). Since all column operations are performed on columns with gaps in H, all inequalities in $Hy \leq u$ also appear in $H'y \leq u'$, i.e., $(Hy \leq u) \subset (H'y \leq u')$.

Algorithm 20: `ReduceLeftI`(H', V, p)

Input : A matrix $H' \in \mathbb{Q}^{m+1 \times n}$, a mixed column transformation matrix $V \in \mathbb{Q}^{n \times n}$, and a row and column index p.

Effect : Applies mixed column transformations to H' until all entries h'_{pi} with $i > p$ are zero. The transformations are combined with the previous transformations into V'. The overall algorithm is based on the Euclidean algorithm for GCD computation.

Output: (H',V')

```
1  V' := V;
   /* Since this algorithm is based on GCD computation, we
      need to abstract the coefficients h'_pi to integers.
      (Stored in S.)                                          */
2  (H', V', S) := AbstractToInt(H', V', p);
   /* Next we perform the Euclidean algorithm via column
      operations on the coefficients stored in S.             */
3  while |S| ≠ 1 do
4  |  (i, s_pi) := an (j, s_pj) ∈ S with the smallest s_pj;
5  |  for (j, s_pj) ∈ S do
6  |  |  if j = i then continue ;
7  |  |  S := S \ {(j, s_pj)};
8  |  |  d_pj := ⌊s_pj ÷ s_pi⌋;
9  |  |  s_pj := s_pj − d_pj · s_pi;
10 |  |  V' := Subtract d_pj times column i from column j of V';
11 |  |  H' := Subtract d_pj times column i from column j of H';
12 |  |  if s_pj ≠ 0 then S := S ∪ {(j, s_pj)} ;
13 |  end
14 end
   /* We have found the gcd s_pi as soon as S contains only
      one element (i, s_pi). We swap it to column p.          */
15 (i, s_pi) := the only (i, s_pi) ∈ S;
16 V' := Swap Column i and p in V';
17 H' := Swap Column i and p in H';
18 return (H',V');
```

Figure 6.8: `ReduceLeftI`(H', V, p)

Algorithm 21: `AbstractToInt`(H', V', p)

Input : A matrix $H' \in \mathbb{Q}^{m+1 \times n}$, a mixed column transformation matrix $V' \in \mathbb{Q}^{n \times n}$, and a row and column index p.

Effect : Negate all columns $i \geq p$ with $h'_{pi} < 0$. Extract the integer part s_{pi} of each coefficient h'_{pi}. Store all non-zero integer parts s_{pi} and their column index i in a set S.

Output: (H', V', S)

```
1  S := ∅;
2  c := lcm{d_pj | j ∈ {n_1 + 1, ..., n} and d_pj := denominator of h'_pj};
3  for j ∈ {p, ..., n} do
4  |  if h'_pj < 0 then
5  |  |  V' := Negate column i of V';
6  |  |  H' := Negate column i of H';
7  |  end
8  |  if h'_pj > 0 then
9  |  |  S := S ∪ {(j, h'_pj · c)};
10 |  end
11 end
12 return (H', V', S);
```

Figure 6.9: `AbstractToInt`(H', V', p)

The case distinction over the algorithms `ExtendR`() and `ExtendI`() is necessary because of the restrictions we have on our column transformations[4], e.g., we can add multiples of rational columns to integer columns but not vice versa.

Since `ExtendR`() and `ExtendI`() change only the new inequality, it holds that $(Hy \leq u)$ is equivalent to $AV'y \leq b$. This means that an extended transformation matrix still transforms the previous constraint system into an equisatisfiable MEHNF. We can use this fact to be also decrementally efficient, i.e., to efficiently remove inequalities in the order they were added. In order to remove $a_{m+1}^T x \leq b_{m+1}$ from $H'y \leq u'$, we simply remove the last inequality that was added to the constraint system (can be efficiently marked with a flag) to get again $(Hy \leq u)$. Since $(Hy \leq u)$ is equivalent to $AV'y \leq b$, it is not necessary to change the transformation matrix[5]. Thus, we have found an incrementally and decrementally efficient way to compute the MEHNF of a constraint system.

[4]Without these restrictions, our transformations would not be mixed equisatisfiable!

[5]When the size of coefficients in V gets too large, it can make sense to recompute H and V to get a smaller transformation matrix.

Algorithm 22: `ReduceRightI`(H', V', p)

Input : A matrix $H' \in \mathbb{Q}^{m+1 \times n}$, a mixed column transformation matrix $V' \in \mathbb{Q}^{n \times n}$, and a row and column index p.

Effect : Applies mixed column transformations to H' until all entries h'_{pi} with $n_1 < i < p$ are non-negative and less than h'_{pp}. The transformations are also added to V'.

Output: ($H'y \leq b'$,V')

```
1 c := lcm{d_pj | j ∈ {n_1 + 1, ..., n} and d_pj := denominator of h'_pj};
2 s_pp := h'_pp · c;
3 for j ∈ {n_1 + 1, ..., p − 1} do
4 |   s_pj := h'_pj · c;
5 |   d_pj := ⌊s_pj ÷ s_pp⌋;
6 |   V' := Subtract d_pj times column p from column j of V';
7 |   H' := Subtract d_pj times column p from column j of H';
8 end
9 return (H',V');
```

Figure 6.10: `ReduceRightI`(H', V', p)

6.4.3 The Complete Incremental Procedure

Now that we have incrementally efficient subprocedures, we can describe a version of our complete procedure that is incrementally efficient. As a reminder, the non-incremental version of our total procedure works as follows: Our input is a constraint system $Ax \leq b$ and we want to find a mixed solution for it. To this end, we first compute the equality basis of $Ax \leq 0^m$ to find the inequalities and directions in $Ax \leq b$ that are bounded. Next we do a case distinction depending on whether $Ax \leq b$ is bounded, absolutely unbounded or partially unbounded. If $Ax \leq b$ is bounded, we find the mixed solution via branch-and-bound [6]. If $Ax \leq b$ is absolutely unbounded, we find the mixed solution via the unit cube test (see Chapter 4). The only slightly complicated case is if $Ax \leq b$ is partially unbounded. In this case, we first split $Ax \leq b$ into a split system and transform the double-bounded part into its MEHNF. The double-bounded system in MEHNF is then solved with branch-and-bound.

Now assume that we have done all of the above for $Ax \leq b$, but need to incrementally extend it to $(Ax \leq b) \cup (A'x \leq b')$. This means we want to find a mixed solution for $(Ax \leq b) \cup (A'x \leq b')$. If $(Ax \leq b)$ was already bounded, then we know that $(Ax \leq b) \cup (A'x \leq b')$ will also be

[6] As mentioned in Chapter 2.7, we recommend to use the version of the dual simplex algorithm presented by Dutertre and de Moura (see Chapter 2.7.1 and [57]) as the basis for the underlying branch-and-bound solver (see Chapter 2.7.3). We do so because this version is highly incrementally efficient.

bounded and we simply apply branch-and-bound to it. Otherwise, we find the inequalities and directions in $(Ax \leq b) \cup (A'x \leq b')$ that are bounded by extending the equality basis of $Ax \leq 0^m$ to the equality basis of $(Ax \leq 0^m) \cup (A'x \leq 0^{m'})$. In Section 6.4.1, we have shown how to do this incrementally efficient. Next we do a case distinction depending on whether $(Ax \leq b) \cup (A'x \leq b')$ is bounded, absolutely unbounded or partially unbounded. If $(Ax \leq b) \cup (A'x \leq b')$ is now bounded, we find the mixed solution via branch-and-bound. If $(Ax \leq b) \cup (A'x \leq b')$ is still absolutely unbounded, we find the mixed solution via the unit cube test (also an incrementally efficient procedure; see Chapter 4). If $(Ax \leq b) \cup (A'x \leq b')$ is still partially unbounded, we continue as follows: We still have the split system for $(Ax \leq b)$ and can now use our extended equality basis for $(Ax \leq 0^m) \cup (A'x \leq 0^{m'})$ to efficiently extend it to a split system for $(Ax \leq b) \cup (A'x \leq b')$. Since adding new inequalities can only add bounded directions, the double-bounded part of the extended split system still contains all bounded inequalities from the previous double-bounded part. This means we can incrementally extend the MEHNF $l \leq Hy \leq u$ by the new inequalities in the double-bounded part of $(Ax \leq b) \cup (A'x \leq b')$. In Section 6.4.2, we have shown how to do this incrementally efficient. Finally, we solve the extended double-bounded constraint system $(l \leq Hy \leq u) \cup (l' \leq H'y \leq u')$ with branch-and-bound. Since we only add inequalities to the running constraint system $(l \leq Hy \leq u)$, we can continue our branch-and-bound search incrementally efficient.

This shows that most parts of our procedure can be implemented incrementally efficient. However, there are two limits to the incremental efficiency. First of all, we have to store multiple constraint systems in our memory to stay incrementally efficient: we need one system to store the current equality basis, so we can later extend it; we need one system to store the current MEHNF, so we can later extend it; we need the current transformation matrix of the MEHNF transformation, so we can later extend it; and we need one copy of the MEHNF to perform branch-and-bound on. Secondly, we do not know how to make the assignment/solution conversion incrementally efficient, i.e., how to convert the mixed solution of the transformed system to a mixed solution of the original system in an incrementally efficient way (see Lemma 6.3.5 for the non-incremental subprocedure). However, this second limitation is in reality not a problem because there are ways to avoid the conversion until we know that the complete problem is satisfiable. So the conversion is used at most once for each SMT input problem. In the next subsection, we will elaborate why this is the case.

6.4.4 Avoiding the Solution Conversion

In order to explain why we can avoid the solution conversion, we first have to distinguish the origin of the incrementally connected problems, i.e., the origin of the problems sent from the SMT solver to the SMT theory solver. There are typically two reasons a theory solver might receive incrementally connected problems from the SMT solver:

(1) The SMT solver tries to prune some partial models (i.e., conjunctions of literals) that are theory unsatisfiable. This case is actually not necessary for a complete SMT solver[7]. It is just a trick to speed-up the boolean search of the SMT solver. In fact, it is already too expensive for the theory solver to check all partial models. Instead, theory solvers typically just check partial models when the SAT solver is about to do a decision. And even then the check is often just a sound approximation of the complete theory solver because the complete check is too expensive for some theories. One of those theories is in fact linear mixed/integer arithmetic. For this theory, most SMT solvers check only the rational relaxation of the partial models for theory satisfiability. So this source of incrementally connected problems is not relevant to our complete approach.

(2) The SMT solver combines multiple theory solvers via the Nelson-Oppen method. As part of the Nelson-Oppen method, (2.1) each theory solver has to first determine the satisfiability of their own conjunctions of literals. (2.2) Then the theory solvers incrementally send to each other (negated) equalities over constant function symbols and test these extended problems for satisfiability. (2.3) This continues until they find a complete and satisfiable equivalence class over the constant function symbols. All of the above can be done with our transformation scheme without converting the intermediate solutions to the original system. Unfortunately, most SMT solvers rely on the intermediate solutions to the original system to guess the (negated) equalities they send in step (2.2).

At a first glance, case (2) seems like it actually needs the solution conversion via Lemma 6.3.5. However, there is an easy and reasonable way to avoid it. Instead of using the intermediate solution to the complete original system, we just use the intermediate solution to the double-bounded part of the original system. This solution can be efficiently computed with the transformation matrix V, i.e., $x := Vy$ is the solution to the double-bounded part of the original system if y is the solution to the transformed system. This is a reasonable approximation for the guesses in (2.2) because we know that the unbounded part is irrelevant to the satisfiability of the original system (Corollary 6.3.6).

[7] Only incomplete models do not have to be checked. Complete models still need to be checked for complete theory satisfiability!

We conclude that our total incremental procedure never has to convert a complete solution more than once. So the procedure should be incrementally efficient in practice. However, we are unable to test this claim with experiments since we do not have a SMT solver that supports other theories besides linear arithmetic.

6.5 Experiments

We integrated the Mixed-Echelon-Hermite transformation and the Double-Bounded reduction into our own theory solver *SPASS-IQ v0.3* and ran it on four families of newly constructed benchmarks.[8] Once with the transformations turned on (*SPASS-IQ*) and once with the transformations turned off (*SPASS-IQ-Off*). If SPASS-IQ encounters a system $Ax \leq b$ that is not explicitly bounded, i.e., where not all variables have an explicit upper and lower bound, then it computes an equality basis for $Ax \leq 0^m$ (see Chapter 5). This basis is used to determine whether the system is implicitly bounded, absolutely unbounded or partially bounded, as well as which of the inequalities are bounded. Our solver only applies our two transformations if the problem is partially unbounded. The resulting equisatisfiable but bounded problem is then solved via branch-and-bound. The other two cases, absolutely unbounded and implicitly bounded, are solved respectively via the unit cube test (see Chapter 4) and branch-and-bound on the original system. Our solver also converts any mixed solutions from the transformed system into mixed solutions for the original system following the proof of Lemma 6.3.5. Rational conflicts are converted between the two systems by using Corollary 6.2.9.

To evaluate the efficiency of our transformations, we compared SPASS-IQ with several other solvers for systems of linear inequalities. The focus of our experiments are partially unbounded problems since SPASS-IQ applies our transformations only on this class of problems. However, before our first publication on the Mixed-Echelon-Hermite transformation and the Double-Bounded reduction, the SMT-LIB (Satisfiability Modulo Theories Library) benchmarks for QF_LIA (quantifier-free linear integer arithmetic) [10] contained only a few partially unbounded problems: two in `arctic-matrix`, one in `CAV-2009`, five in `cut_lemmas`, three in `slacks`, and four in `tropical-matrix`. Since there were only so few partially unbounded problems in the SMT-LIB, we created in addition four new benchmark families:[9]

[8] SPASS-IQ and all benchmark files, benchmark scripts, and benchmark results are available on `http://www.spass-prover.org/spass-iq`

[9] Our benchmarks for quantifier-free linear integer arithmetic (`SlackedQFLIA` and `RandomUnbd`) were recently added to the SMT-LIB as part of the `20180326-Bromberger` benchmark family.

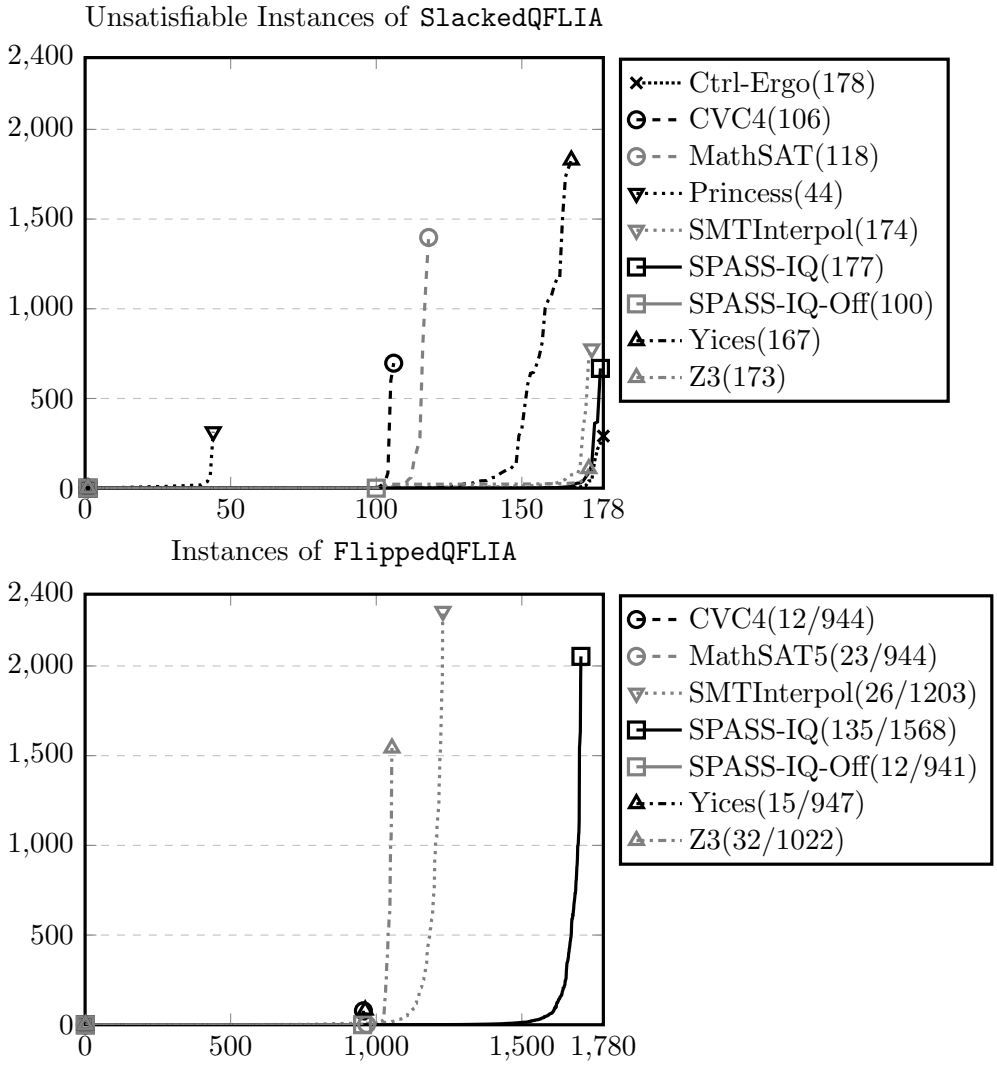


Figure 6.11: SPASS-IQ compared to SMT solvers on the `SlackedQFLIA` and `FlippedQFLIA` benchmark families

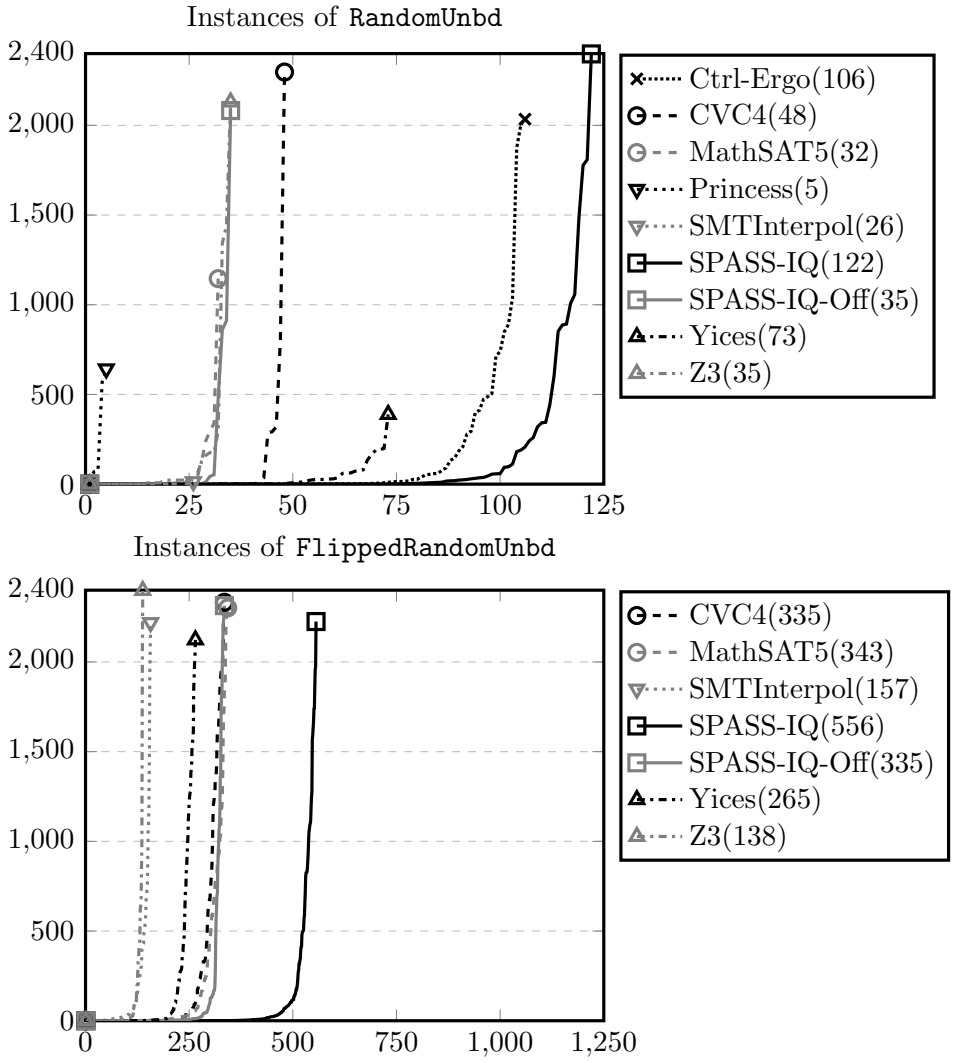


Figure 6.12: SPASS-IQ compared to various SMT solvers on the `RandomUnbd` and `FlippedRandomUnbd` benchmark families

`SlackedQFLIA`: are linear integer benchmarks based on the SMT-LIB classes `CAV-2009` [52], `cut_lemmas` [76], and `dillig` [52]. We simply took all of the unsatisfiable benchmarks and replaced in them all variables x with $x_+ - x_-$ where x_+ and x_- are two new variables such that $x_+, x_- \geq 0$. This transformation, called slacking, is equisatisfiable and the slacked version of the `dillig` benchmarks, called `slacked` [87], is already in the SMT-LIB. Slacking turns any unsatisfiable problem into a partially unbounded one. Hence, all problems in `SlackedQFLIA` are partially unbounded. Slacking is commonly used to integrate absolute values into linear systems or for solvers that require non-negative variables [128].

`RandomUnbd`: are linear integer benchmarks that are all partially unbounded and satisfiable with 10, 25, 50, 75, and 100 variables. All problems in this benchmark family are randomly created via a sagemath script.

`FlippedQFLIA` and `FlippedRandomUnbd`: are linear mixed benchmarks that are all partially unbounded. They are based on `SlackedQFLIA` and `RandomUnbd`. We constructed them by first copying ten versions of the integer benchmarks and then randomly flipping the type of some of the variables to rational (probability of 20%). Some of the flipped instances of `SlackedQFLIA` became satisfiable.

For the experiments, we used a Debian Linux cluster and allotted to each problem and solver combination 1 core of an Intel Xeon E5620 (2.4 GHz) processor, 4 GB RAM, and 40 minutes. The plots in Figures 6.11, 6.12, and 6.13 depict the results of the different solvers. In the legends of the plots, the numbers behind the solver names are the number of solved instances. For `FlippedQFLIA`, there are two numbers to indicate the number of satisfiable/unsatisfiable instances solved. This is only necessary for `FlippedQFLIA` because it is the only tested benchmark family with satisfiable and unsatisfiable instances. (We verified that the results match if two solvers solved the same problem.)

6.5.1 Comparison with SMT Solvers

We compared our solver, SPASS-IQ, with some of the state-of-the-art SMT solvers currently available for linear arithmetic: *CVC4* (v1.6) [9], *MathSAT5* (v5.5.2) [41], *Princess* (version from 2018-10-26) [126], *SMTInterpol* (v2.5-19) [40], *Yices* (v2.6.0) [56], and *Z3-4.6.0* (v4.8.1) [49]. (See Figures 6.11 and 6.12 for the results.)

Most of these solvers employ a branch-and-bound approach with an underlying dual simplex solver [57], which is also the basis for our own solver. (As far as we are aware, our own solver, SPASS-IQ, is the only branch-and-bound based solver that employs a technique that guarantees termination.) We are, however, particularly interested in the three solvers not based on this approach: Ctrl-Ergo, Princess, and SMTInterpol.

Ctrl-Ergo uses an approach that is complete over linear integer arithmetic but cannot handle linear mixed arithmetic [26]. The approach works by dynamically eliminating one linear independent bounded direction at a time via transformation. The disadvantages of the dynamic approach are that it is very restrictive and does not leave enough freedom to change strategies or to add complementing techniques. Moreover, Ctrl-Ergo uses this transformation approach for all problems and not only the partially unbounded ones, which sometimes leads to a massive overhead on bounded problems.

Princess uses a constraint sequent calculus that is not only complete for linear integer arithmetic but also for quantified linear integer arithmetic. It terminates because it simulates in the worst case the complete and terminating Omega Test [122]. The broader focus of the constraint sequent calculus also has one big disadvantage: it is less efficient than more specialized methods when it only has to determine satisfiability of a problem. This is especially true for determining satisfiability of (partially) unbounded problems, which is confirmed by our experiments. Moreover, the constraint sequent calculus cannot handle linear mixed arithmetic.

SMTInterpol extends branch-and-bound via the cuts from proofs approach, which uses the Mixed-Echelon-Hermite transformation to find more versatile branches and cuts [39]. Although the procedure is not complete, the similarities to our own approach make an interesting comparison. Actually, the Double-Bounded reduction alone would be sufficient to make SMTInterpol terminating since it already builds branches via a Mixed-Echelon-Hermite transformation.

Although our solver could not solve all problems (due to time and memory limits) it was still able to solve more problems than the other solvers. It was also faster on most instances than the other solvers. On some of the unsatisfiable, partially unbounded benchmarks, Ctrl-Ergo is better than SPASS-IQ. This is due to its conflict focused, dynamic approach. For the same reason, Ctrl-Ergo is slower on the satisfiable, partially unbounded benchmarks. Only SPASS-IQ, Ctrl-Ergo, and Yices solved all of the fifteen original SMT-LIB benchmarks that are partially unbounded, though the complete methods were still a lot faster (SPASS-IQ took 23s, Ctrl-Ergo took 42s, and Yices took 1273s). On one of these benchmarks, 20-14.slacks.smt2 from `slacks`, all other solvers seem to diverge. Another interesting result of our experiments is that relaxing some integer variables to rational variables seems to make the problems harder for SMT solvers instead of easier. We expected this for our transformations because the resulting systems become more complex and less sparse, but it is also true for the other solvers. The reason might be that bound refinement, a technique used in most branch-and-bound implementations (see also Chapter 2.7.4), is less effective on mixed problems.

6.5.2 Comparison with MILP Solvers

We compared our solver with several solvers for mixed-integer programming (MILP) (see Figure 6.13): the two non-commercial solvers *GLPK* (v4.65) [106] and *SCIP* (v6.0.0) [70] as well as the commercial solver *Gurobi* (v7.52) [77]. For these experiments, we used the same benchmarks—although converted into the MPS (Mathematical Programming System) format—and the same experiment parameters as for our experiments with the SMT solvers. In General, mixed-integer programming solvers have an advantage over standard SMT theory solvers because (i) they are not required to be exact and sound, which allows them to use floating-point arithmetic, and (ii) they are not required to be incrementally efficient, which means they can use much more elaborate techniques. (As far as we are aware, none of the MILP solvers uses a technique that guarantees termination.)

Despite these advantages, SPASS-IQ is faster and solves more problems from the `RandomUnbd` benchmarks than GLPK, Gurobi, and SCIP. On the `SlackedQFLIA` benchmarks, SPASS-IQ is again faster and solves more problems than GLPK and SCIP, but is less efficient than Gurobi. Gurobi is more efficient on the `SlackedQFLIA` benchmarks because it uses a preprocessing technique that undoes the slacking transformation. This means Gurobi solves the original bounded versions of the `SlackedQFLIA` benchmarks, which are by far easier. If we turn this preprocessing technique off (see Gurobi-Off), then SPASS-IQ is also faster and solves more problems than Gurobi.

We also tested the MILP solvers on the linear mixed arithmetic benchmark families `FlippedQFLIA` and `FlippedRandomUnbd`. At a first glance, GLPK, Gurobi, and SCIP seemed to improve compared to their results on the integer versions of the benchmarks. However, this improvement was only possible due to the unsoundness of the three solvers. We determined that GLPK returned the wrong result on at least 72 problems, Gurobi returned the wrong result on at least 320 problems, and SCIP returned the wrong result on at least 249 problems. Since GLPK, Gurobi, and SCIP were so unreliable on the linear mixed arithmetic problems, no fair comparison with SPASS-IQ was possible.

6.5.3 Further Remarks on SPASS-IQ

The time SPASS-IQ needs to detect the bounded inequalities and to apply our transformations is negligible. This is even true for the implicitly bounded problems we tested. As mentioned before, we do not have to apply our transformations to terminate on bounded problems. This is also the only advantage we gain from detecting that a problem is implicitly bounded. Since there is no noticeable difference in the runtime, we do not further elaborate the results on bounded problems, e.g. with graphs.

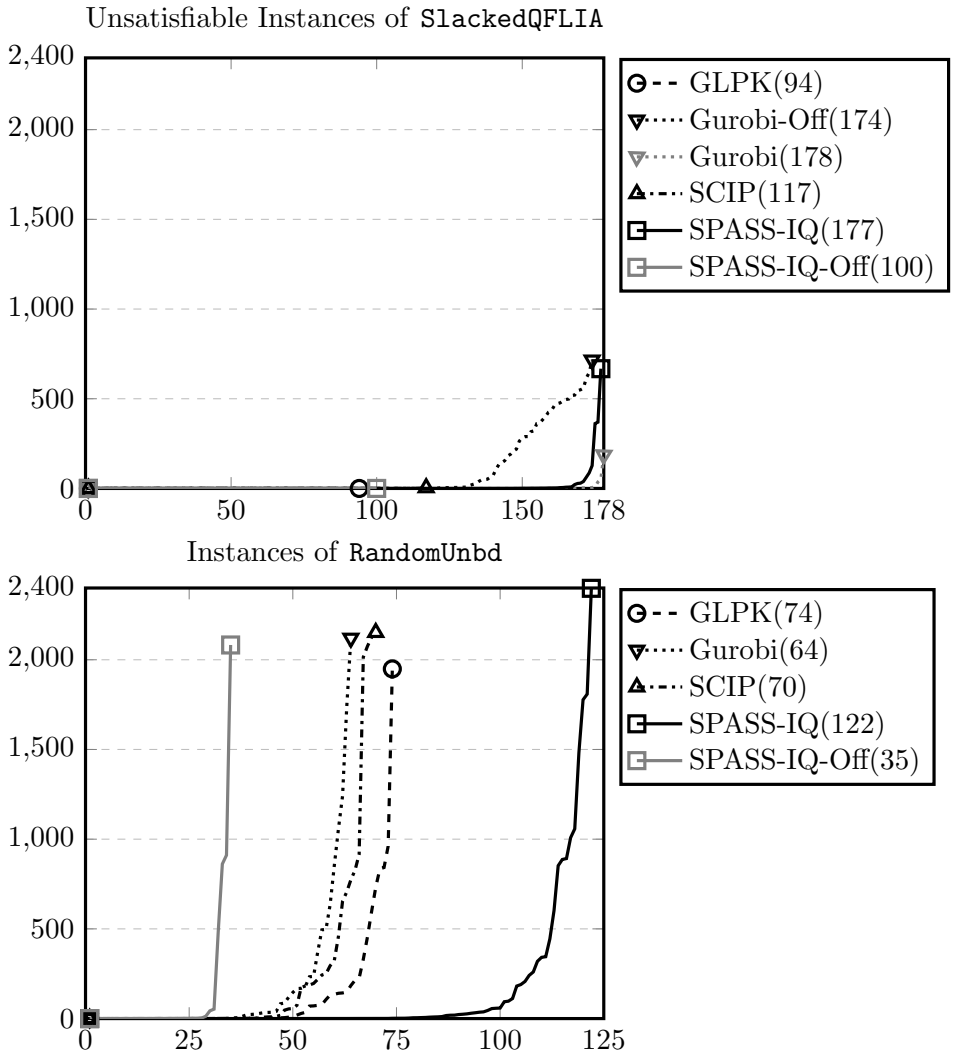


Figure 6.13: SPASS-IQ compared to various MILP solvers on the `SlackedQFLIA` and `RandomUnbd` benchmark families

An actual disadvantage of our approach is that the Mixed-Echelon-Hermite transformation almost always increases the density of the coefficient matrix as well as the absolute size of the coefficients. Both are important factors for the efficiency of the underlying simplex solver.

6.6 Summary

We have presented the Double-Bounded reduction (Lemma 6.3.2 & Corollary 6.3.6) and the Mixed-Echelon-Hermite transformation (Lemma 6.2.11). We have shown that both transformations together turn any constraint system into a mixed equisatisfiable system that is also bounded (Lemma 6.2.3). This is sufficient to make branch-and-bound, and many other linear mixed decision procedures, complete and terminating. We have also shown how to convert certificates of rational (un)satisfiability efficiently between the transformed and original systems (Corollary 6.2.9 & Lemma 6.3.5). Moreover, experimental results on partially unbounded benchmarks [10] show that our approach is also efficient in practice. Our approach can be nicely combined with the extensive branch-and-bound framework and its many extensions, where other complete techniques cannot be used in a modular way [26, 32].

Chapter 7

Implementation of SPASS-IQ

We have presented in the previous chapters several benchmark evaluations for our decision procedures. These evaluations are mostly based on our linear arithmetic theory solver SPASS-IQ. Moreover, they all focus on one single technique relevant to the respective chapter, but never on their combination in one system. In this chapter, we remedy this negligence and describe and evaluate the overall implementation of SPASS-IQ.

SPASS-IQ is a theory solver for linear arithmetic. Its input problems are finite sets of inequalities containing arithmetic variables, which is equivalent to our definition of standard input problems from Chapter 2.4. Given such an input problem, SPASS-IQ's goal is to either (i) find an assignment for the variables that satisfies all of the input inequalities and the variable types (integer/rational), or (ii) to determine that no such assignment exists.

Depending on the types of the input variables, we split the theory of linear arithmetic into three highly related, but ultimately different subtheories: the *theory of linear rational arithmetic*, which contains all input problems with only rational variables; the *theory of linear integer arithmetic*, which contains all input problems with only integer variables; and *the theory of linear mixed arithmetic*, which contains all input problems with both types of variables. SPASS-IQ is divided analogously into two main components: an implementation of the simplex algorithm for handling linear rational arithmetic (see Section 7.1) and an implementation of the branch-and-bound algorithm for handling linear mixed/integer arithmetic (see Section 7.2).

The division between the two components is in all truth not that strict. The main focus of the simplex algorithm may be handling linear rational arithmetic, but we also use it for the overall representation and storage of variables, bounds, and constraints. It is, therefore, the foundation of SPASS-IQ.

The branch-and-bound implementation is more of a supervisor for the simplex implementation. We say so because the branch-and-bound implementation may coordinate the search for a mixed or integer solution, but the majority of the actual search/calculation is still done by the simplex implementation. To be more precise, the branch-and-bound implementation mostly evaluates the assignments and conflicts returned by the simplex implementation and based on this evaluation provides the simplex implementation with new subproblems to solve. This means that the efficiency of our branch-and-bound implementation highly depends of the efficiency of our simplex implementation.

By far not all techniques implemented in SPASS-IQ are also unique features of SPASS-IQ. The techniques that appeared first in SPASS-IQ are the unit cube test (Chapter 4) and the bounding transformations (Chapter 6). Further important techniques implemented in SPASS-IQ have already been available in other SMT (theory) solvers such as CVC4 [9], MathSAT [41], Yices [56], and Z3 [49], but not all in one tool: the implementation of branch-and-bound as a separate theory solver and a number of improvements to the simplex implementation such as a priority queue for pivot selection, integer coefficients instead of rational coefficients, dynamically switching between native and arbitrary precision integers, and backing-up versus recalculating simplex states. Although these techniques are contained in existing SMT theory solvers, not all have been described in the respective literature.

Experimental Setup

For this chapter, we have performed several experiments in order to estimate the impact of SPASS-IQ's various features. However, it is hard to estimate the impact of a theory solver (e.g., incremental efficiency and conflict generation) that is intended to be a module of a more complex solver (e.g., a CDCL(LA) solver) if we test it outside of that complex solver. Therefore, we do not test SPASS-IQ directly, but SPASS-SATT, the CDCL(LA) extension of SPASS-IQ. (For more details on SPASS-SATT see Chapter 8.)

The benchmarks for our experiments are the 1649 SMT-LIB benchmarks for quantifier free linear rational arithmetic (QF_LRA) and the 6947 SMT-LIB benchmarks for quantifier free linear integer arithmetic (QF_LIA) from the SMT-LIB (satisfiability modulo theories library) [10]. All experiments compare different configurations of SPASS-SATT with each other. The experiments are preformed on a Debian Linux cluster from which we allotted one core of an Intel Xeon E5620 (2.4 GHz) processor, 8 GB RAM, and 40 minutes to each combination of problem and SPASS-SATT configuration.

7.1 Simplex Implementation

The basis of SPASS-IQ is an implementation of the simplex algorithm, which we explained in detail in Chapter 2.7. The general efficiency of the simplex algorithm is heavily influenced by the data structures that we use in our implementation. An implementation with optimal data structures can be several orders of magnitude faster than a naive implementation. This section, therefore, includes descriptions on the data structures we use to represent variables, bounds, and the simplex tableau (Subsection 7.1.4). In addition, we present a data structure that efficiently selects violated basic variables for pivoting (Subsection 7.1.2).

There also exist other design choices, besides mere data structures, that have a heavy influence on the efficiency of our implementation. For instance, the pivoting strategy. In Subsection 7.1.1, we not only present the pivoting strategy used by SPASS-IQ, but we also compare it with several other pivoting strategies from various SMT solvers. Moreover, we describe an alternative backtracking technique (Subsection 7.1.3) to the backup technique we have presented in Chapter 2.7. SPASS-SATT, the CDCL(LA) solver based on SPASS-IQ, solves more instances from the SMT-LIB benchmarks with this alternative technique backtracking than with the original backtracking technique.

7.1.1 Pivoting

The simplex algorithm pivots variables in every iteration. Therefore, *pivoting* is one of the (if not the) most important operation(s) for the simplex algorithm. Pivoting itself is a substitution that turns a basic variable y_i into a non-basic variable and, in exchange, a non-basic variable z_j into a basic variable. For this reason, we also call the two variables y_i and z_j the *pivoting variables*.

The pivots done by the simplex algorithm are not absolutely arbitrary, i.e., the pivoting variables actually have to fulfill certain conditions to be selected for pivoting (see also `Check`() in Figure 2.2). These conditions are as follows:

The simplex algorithm first selects the basic variable y_i for pivoting. Any basic variable y_i is a potential candidate for pivoting as long as it is *violated*, i.e., the current assignment $\beta(y_i)$ for the variable y_i violates either the upper bound (i.e., $\beta(y_i) > \mathcal{U}(y_i)$) or the lower bound (i.e., $\beta(y_i) < \mathcal{L}(y_i)$). After the basic variable y_i is selected for pivoting, the non-basic variable z_j is selected. Any non-basic variable z_j is a potential candidate for pivoting as long as it fulfills the following two conditions: (i) the variable z_j has a non-

zero coefficient a_{ij} in the row of the selected basic variable y_i and (ii) there is no tight bound in the variables *changing direction*, i.e., $\beta(z_j) < \mathcal{U}(z_j)$ if the changing direction is upper and $\beta(z_j) > \mathcal{L}(z_j)$ if the changing direction is lower.

The last point is very vague so let us explain it in more detail. If a bound of a basic variable y_i is violated, then we need to change the variable's assigned value $\beta(y_i)$ so the bound is no longer violated. However, we cannot change the assignment of a basic variable directly because the assigned value $\beta(y_i)$ is defined through the tableau and the values assigned to the non-basic variables. This means we have to change the values assigned to the non-basic variables in order to fix the violated basic variable.

If the lower bound of the selected basic variable y_i is violated, then we need to increase the value $\beta(y_i)$ assigned to y_i. This is possible by either increasing the value $\beta(z_j)$ assigned to a non-basic variable z_j with a positive coefficient in y_i's row ($a_{ij} > 0$) or by decreasing the value $\beta(z_j)$ assigned to a non-basic variable z_j with a negative coefficient in y_i's row ($a_{ij} < 0$). Symmetrically, we need to somehow decrease the value $\beta(y_i)$ assigned to y_i if the upper bound of the selected basic variable y_i is violated. Again, this is possible by either decreasing the value $\beta(z_j)$ assigned to a non-basic variable z_j with a positive coefficient in y_i's row ($a_{ij} > 0$) or by increasing the value $\beta(z_j)$ assigned to a non-basic variable z_j with a negative coefficient in y_i's row ($a_{ij} < 0$).

Based on this change in assignment, we define the changing direction of a non-basic variable. The *changing direction* of a non-basic variable z_j is "upper" if the value $\beta(z_j)$ needs to be increased to fix the assignment for y_i. And the changing direction of a non-basic variable z_j is "lower" if the value $\beta(z_j)$ needs to be decreased to fix the assignment for y_i. This means that the second non-basic pivoting condition—there is no tight bound in the variables *changing direction*—prevents us from selecting a non-basic variable that cannot be changed without violating one of its own bounds.

Pivoting Strategies

The pivoting conditions of the simplex algorithm are necessary to prevent the simplex algorithm from repeatedly pivoting the same two variables, which would cause divergence. Nonetheless, the conditions are still too weak to always guarantee exactly one candidate pair of pivoting variables and to prevent all types of divergence. Therefore, we need a *pivoting strategy*, i.e., a strategy for selecting the pivoting variables, that resolves the non-determinism and prevents the divergence.

The simplex algorithm can only diverge if it runs into a *pivoting cycle*, i.e., a series of pivots that leads back to the same set of non-basic variables and the same assignment. The pivoting conditions alone are enough to prevent all pivoting cycles consisting of upto two pivots. We need, however, a *terminating pivoting strategy*, e.g., Bland's rule [25], to avoid all other cycles.

The terminating pivoting strategies have the disadvantage that they typically need more pivots to reach a solution if there are no cycles than *greedy* but potentially diverging *strategies*. We, therefore, have to find a compromise between termination and efficiency in practice. Our solution is to first do a finite number of pivots according to a greedy but potentially diverging strategy and afterwards continue with a terminating pivoting strategy.[1] This combination of greedy and terminating pivoting strategy is also commonly used by other SMT theory solvers, e.g., in CVC4 [9], SMTInterpol [40], veriT [27], Yices [56], and Z3 [49].

We have actually implemented several pivoting strategies inside SPASS-IQ (command line option −`LAPR` <`id`>) and compared. The most promising strategies are listed below. Most of them rely on a strict total variable order $\prec$.

Bland's rule (−`LAPR 1`):
Basic variable strategy: select the smallest candidate variable according to $\prec$.
Non-basic variable strategy: select the smallest candidate variable according to $\prec$.

CVC4's rule [9] (−`LAPR 2`):[2]
basic variable strategy: primarily prefer candidate variables with *minimum error*, i.e., variables y_i with minimum absolute difference $|b_i - \beta(y_i)|$ between the violated bound value b_i and the assigned value $\beta(y_i)$; then prefer the smallest candidate variable according to $\prec$.
non-basic variable strategy: primarily prefer candidate variables without any bounds; then prefer candidate variables with the least number of non-zero coefficients; finally prefer the smallest candidate variable according to $\prec$.
additional comments: switches after a finite number of pivoting steps to Bland's rule.

SMTInterpol's rule [40] (−`LAPR 3`):
Basic variable strategy: primarily prefer candidate variables with the least number of non-zero coefficients in their tableau row; then prefer the smallest candidate variable according to $\prec$.
non-basic variable strategy: primarily prefer candidate variables without any

[1] In our implementation, this finite number is equal to the number of non-basic variables.
[2] This pivoting rule is only used in CVC4 for the Dutertre and de Moura version of the simplex algorithm. CVC4 typically uses a different version of the simplex algorithm called sum of infeasibility simplex [97] and we cannot reproduce its pivoting strategy in the Dutertre and de Moura version of the simplex algorithm.

bounds; then prefer candidate variables without a bound in their changing direction; then prefer candidate variables with the least number of non-zero coefficients; finally prefer the smallest candidate variable according to $\prec$.
additional comments: switches after a finite number of pivoting steps to Bland's rule.

Maximum error rule [9] (−LAPR 4):[3]
basic variable strategy: primarily prefer candidate variables with *maximum error*, i.e., variables y_i with maximum absolute difference $|b_i - \beta(y_i)|$ between the violated bound value b_i and the assigned value $\beta(y_i)$; then prefer the smallest candidate variable according to $\prec$.
non-basic variable strategy: primarily prefer candidate variables without any bounds; then prefer candidate variables with the least number of non-zero coefficients; finally prefer the smallest candidate variable according to $\prec$.
additional comments: switches after a finite number of pivoting steps to Bland's rule. The maximum error rule is not used by any SMT solver as far as we are aware.

Prefer unbounded rule (−LAPR 5):
basic variable strategy: primarily prefer candidate variables that have non-basic variables in their row with an unbounded changing direction; then prefer the smallest candidate variable according to $\prec$.
non-basic variable strategy: primarily prefer candidate variables without any bounds; then prefer candidate variables without a bound in their changing direction; then prefer candidate variables with the least number of non-zero coefficients; finally prefer the smallest candidate variable according to $\prec$.
additional comments: switches after a finite number of pivoting steps to Bland's rule. The prefer unbounded rule is not used by any SMT solver as far as we are aware.

SPASS-SATT's rule (−LAPR 0):
basic variable strategy: select the smallest candidate variable according to $\prec$.
non-basic variable strategy: primarily prefer candidate variables without any bounds; then prefer candidate variables with the least number of non-zero coefficients; finally prefer the smallest candidate variable according to $\prec$.
additional comments: switches after a finite number of pivoting steps to Bland's rule.

We implemented these strategies into SPASS-IQ and compared their overall efficiency through benchmark experiments with SPASS-SATT, the CDCL(LA) extension of SPASS-IQ. We present the benchmark results in the cactus plots in Figures 7.1—7.4. In the legends of the plots, the two numbers in brackets behind each pivoting strategy name are the numbers of satisfiable and unsatisfiable instances that we can solve with the respective strategy.

[3] This pivoting rule is also implemented in CVC4 [9] but not activated by default.

From the cactus plots, we can see that Bland's rule alone performs much worse than most of the other pivoting strategies. Moreover, the pivoting strategies that do not switch to Bland's rule at some point perform worse than their counterparts that do switch to Bland's rule. This supports our initial assumption that we need to combine a greedy pivoting strategy with a terminating pivoting strategy in order to receive a generally efficient pivoting strategy. The best pivoting strategies for QF_LRA and QF_LIA are SMTInterpol's rule and SPASS-SATT's rule, respectively. This is also the reason why we selected SMTInterpol's rule and SPASS-SATT's rule as SPASS-IQ's pivoting strategies for QF_LRA and QF_LIA, respectively. However, we do want to remark that the difference between the best strategies and their runner-ups is so small that they could be attributed to performance fluctuations of our cluster. Therefore, the runner-up pivoting strategies are most likely equally good choices.

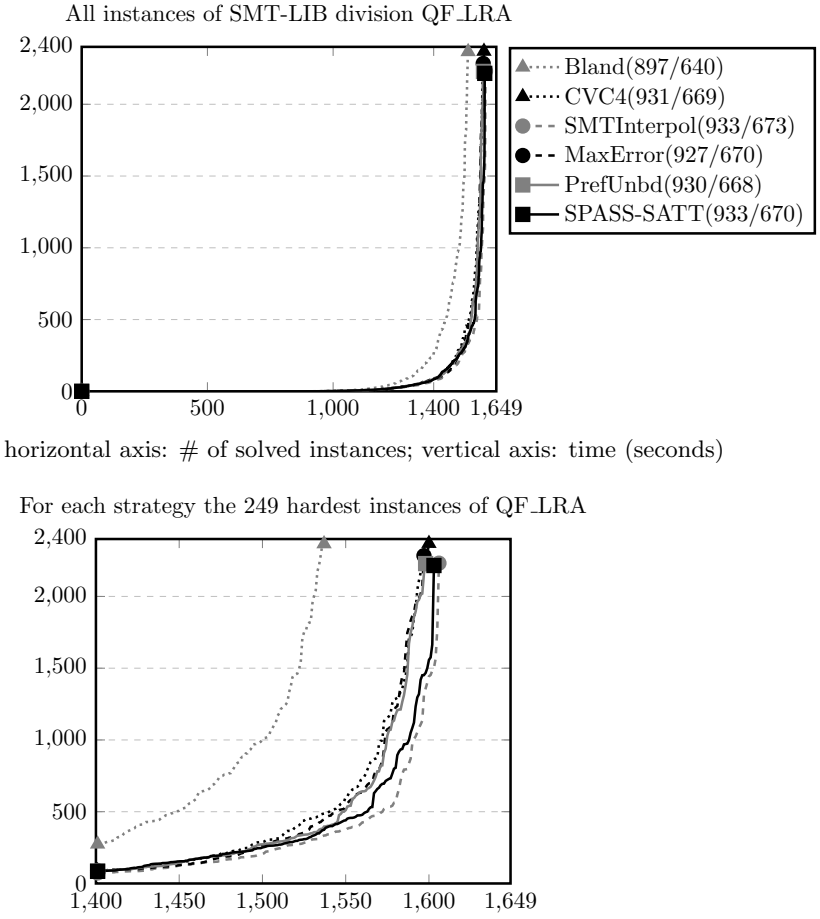


Figure 7.1: Results for SPASS-IQ's pivoting strategies on the SMT-LIB benchmark division QF_LRA

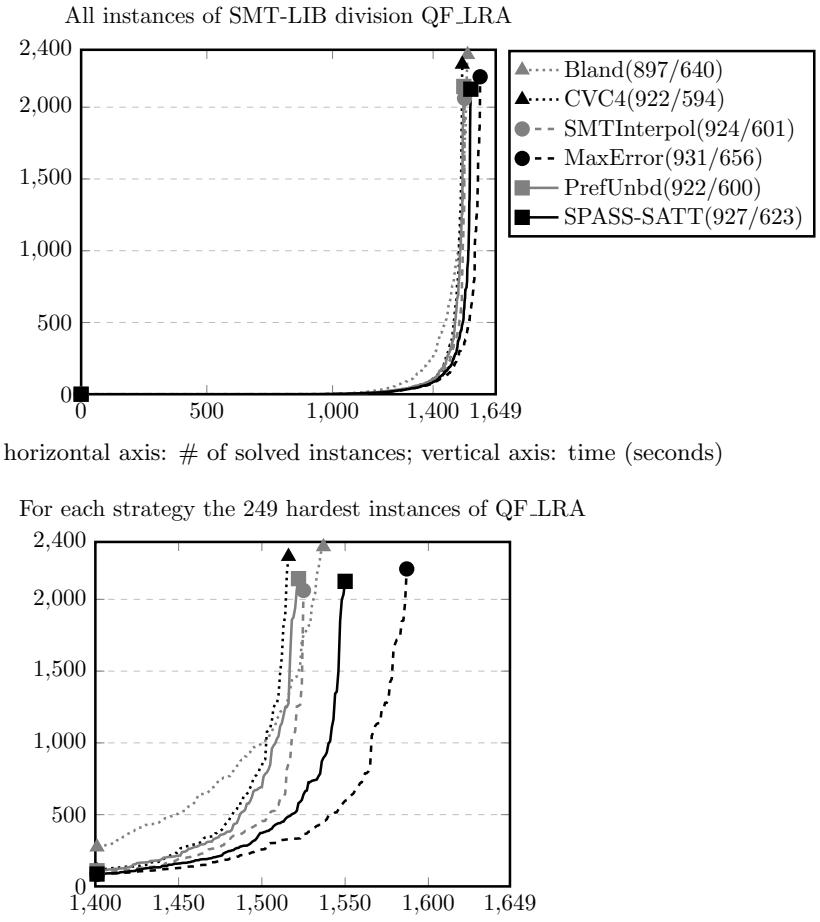


Figure 7.2: Results for SPASS-IQ's pivoting strategies on the SMT-LIB benchmark division QF_LRA without switching to Bland's rule after finitely many pivots

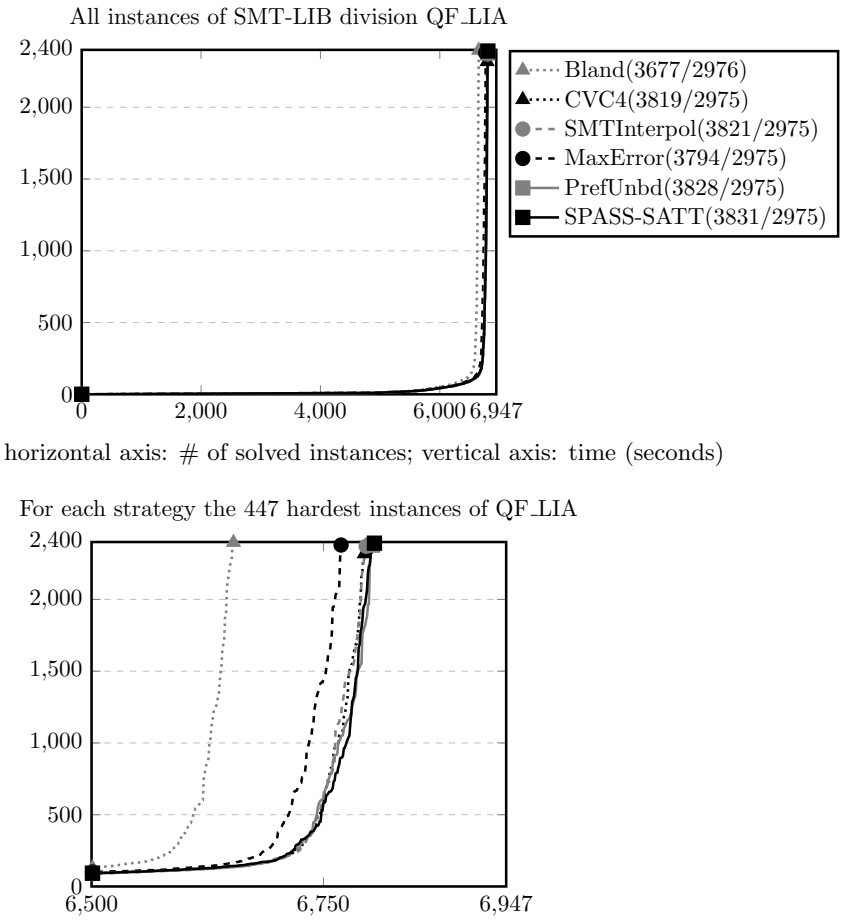


Figure 7.3: Results for SPASS-IQ's pivoting strategies on the SMT-LIB benchmark division QF_LIA

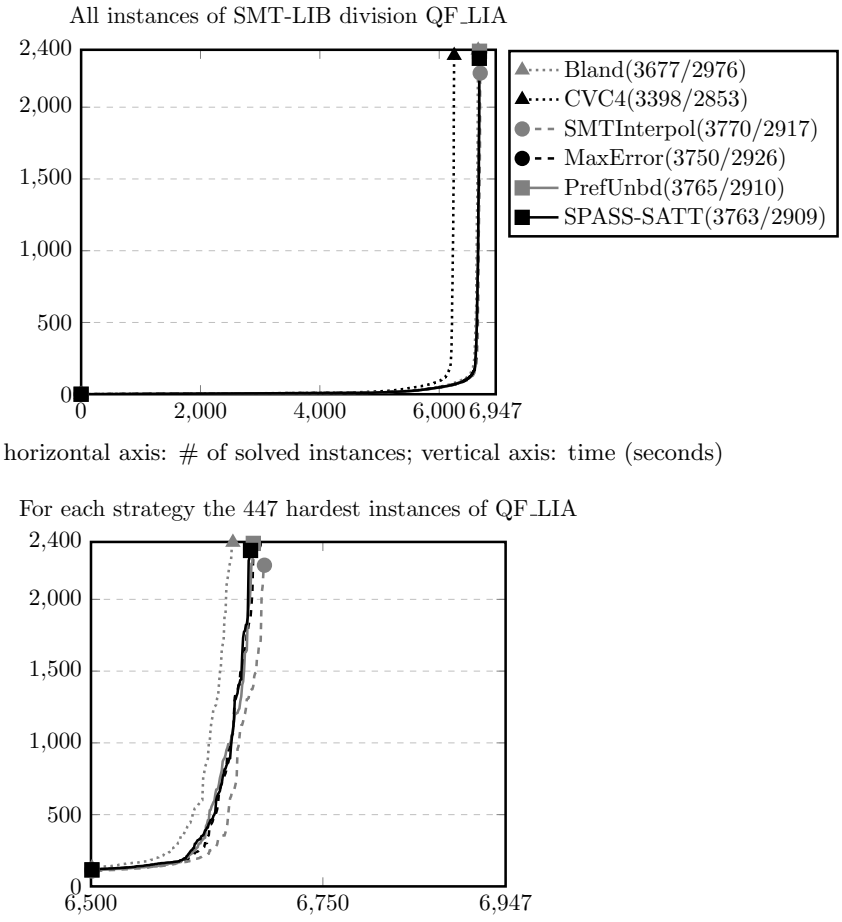


Figure 7.4: Results for SPASS-IQ's pivoting strategies on the SMT-LIB benchmark division QF_LIA without switching to Bland's rule after finitely many pivots

7.1.2 Violated Variable Heap

The `Check`() function selects in every iteration of its while-loop a violated variable, i.e., a basic variable with a violated bound (see Figure 2.2 line 2). Typically, we have a large number of basic variables, but only a small subset of them is actually violated in a given iteration of the simplex algorithm. Our implementation goal is, therefore, to reduce the number of basic variables visited in each iteration and, if possible, reduce it even to the subset of actually violated variables.

We have decided to implement our variable selection through a violated variable heap (VV heap). (Similar heaps can be found in all of the state-of-the-art theory solvers for SMT [9, 27, 40, 41, 49, 56].)

The VV heap contains exactly all currently violated variables during variable selection. Inside the heap, the variables are ordered according to their selection preference. The heap is, therefore, mostly a priority queue and its top element is the violated variable we want to select according to our selection preference.

However, our heap has one big difference compared to other priority queues: it alternates between two phases. The first phase is called the *assertion or buffering phase*. Our heap is in the assertion phase while we are making a series of assertions. During this phase, our heap is empty and we do not actually communicate with it directly. Instead, we fill a buffer with all basic variables that change their bounds or their assignment during the assertions.

The second phase is called the check or pivoting phase. Our heap is in the check phase whenever we are inside a `Check`() call. At the beginning of the check phase, we fill the heap with the basic variables in our buffer that are actually violated. This means that the top element on the heap is also the violated variable we want to select according to our selection preference. We extract this variable from the heap, `pivotAndUpdate`() with it, and inform the heap of all basic variables that have changed their assignment during the pivot. Based on this information, we are able to add all newly violated variables to the heap and remove all variables from the heap that are no longer violated. This means our heap contains again exactly all currently violated variables.

We continue selecting violated variables in this way until we have found a conflict or there are no more violated variables. This variable selection implementation is more efficient than the naive implementation because it only visits a basic variable when it actually changes in some way.

Greedy Conflict Detection and Returning Multiple Conflicts

There exists one extension that should improve almost all selection preferences: prefer a violated variable if it also describes a conflict. We call this extension *greedy conflict detection* and it should be beneficial because `Check`() stops as soon as it detects a conflict.

However, we do not extend all of our selection preferences explicitly with greedy conflict detection. Instead, we added greedy conflict detection directly into our VV heap implementation. This makes sense because we already inform the VV heap of all basic variable changes and, therefore, of all changes that could turn a basic variable into a conflict.

Adding greedy conflict detection to the VV heap also has another advantage. If we implement it in this way, then we detect not only one conflict variable but all conflict variables. This means that we can return *multiple conflicts* at once without any overhead.[4] (Similar greedy and multiple conflicts schemes can be found in other theory solvers for SMT, e.g., in CVC4 [9].)

In spite of these arguments, we were unable to confirm a guaranteed benefit from either greedy conflict detection or learning multiple conflicts in our benchmark experiments. We present the benchmark results in the four scatter plots in Figures 7.5—7.8. All figures compare two versions of SPASS-SATT, the CDCL(LA) extension of SPASS-IQ. Figure 7.5 examines the effect of greedy conflict detection on the QF_LRA benchmarks. To this end, we compare the default version of SPASS-SATT with greedy conflict detection turned on (horizontal axis; command-line option `-LAGC 1`) with the version of SPASS-SATT that does not use greedy conflict detection (vertical axis; command-line option `-LAGC 0`). Symmetrically, Figure 7.6 examines the effect of greedy conflict detection on the QF_LIA benchmarks with the same solver configurations. Figure 7.7 examines the effect of learning (if possible) multiple conflicts after each unsatisfiable `Check`() call on the QF_LRA benchmarks. To this end, we compare the default version of SPASS-SATT that can produce multiple conflicts at once (horizontal axis; command-line option `-LAMC 1`) with the version of SPASS-SATT that does not (vertical axis; command-line option `-LAMC 0`). Symmetrically, Figure 7.8 examines the effect of learning (if possible) multiple conflicts after each unsatisfiable `Check`() call on the QF_LIA benchmarks with the same solver configurations.

In all four of our experiments, the default version of SPASS-SATT, which employs greedy conflict detection and learns (if possible) multiple conflicts, solves slightly more problem instances. However, the increase in solved instances happens so close to the timeout that it could be attributed to performance fluctuations of our cluster. Apart from that, the figures show that

[4] Note that this is a property that impacts the efficiency of SPASS-IQ as a theory solver inside a CDCL(LA) extension (e.g., SPASS-SATT) but not its efficiency as a standalone solver.

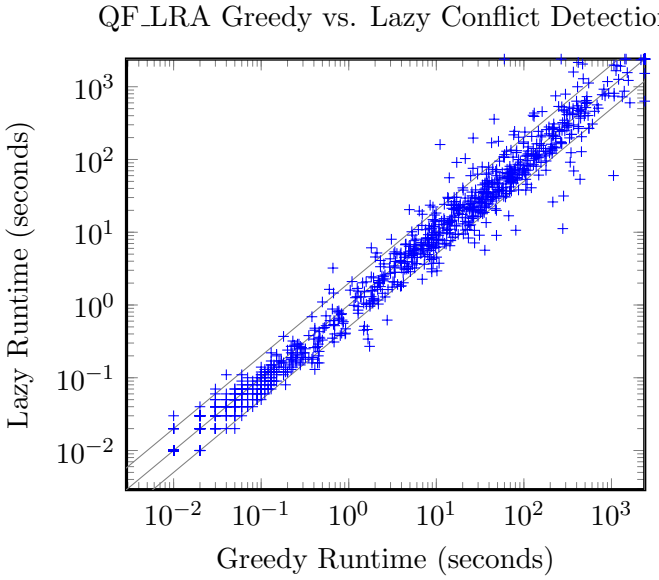
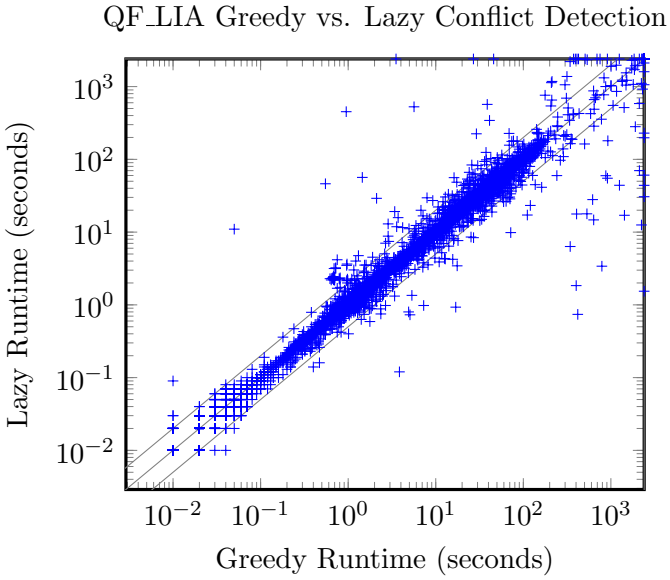


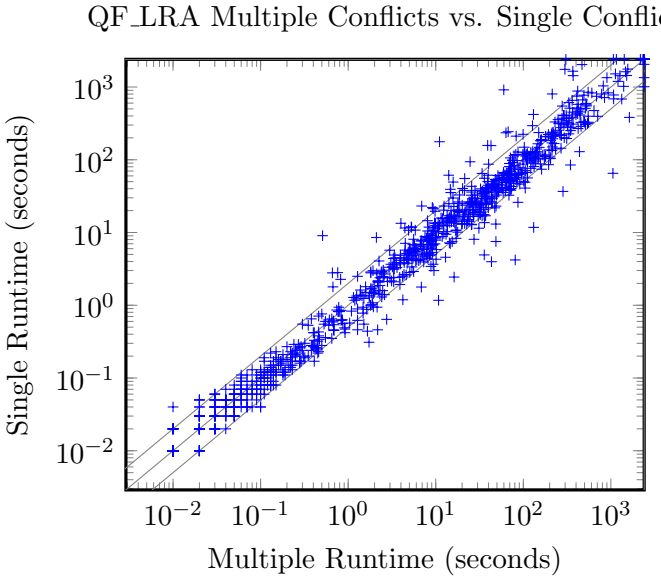
Figure 7.5: Greedy vs. lazy conflict detection on the QF_LRA SMT-LIB benchmarks

there are roughly as many problem instances where greedy conflict detection helps as there are problem instances where it hinders SPASS-SATT's search. We assume that the desired benefit of greedy conflict detection, i.e., ending `Check`() earlier, is so minor that it is easily outweighed by the unpredictable changes it causes in the remainder of the search. The same can be said about learning multiple conflicts.



SPASS-SATT with greedy conflict detection solves 6806 instances
SPASS-SATT with lazy conflict detection solves 6798 instances

Figure 7.6: Greedy vs. lazy conflict detection on the QF_LIA SMT-LIB benchmarks



SPASS-SATT with multiple conflicts per check solves 1607 instances
SPASS-SATT with a single conflict per check solves 1605 instances

Figure 7.7: Multiple conflicts vs. single conflict on the QF_LRA SMT-LIB benchmarks

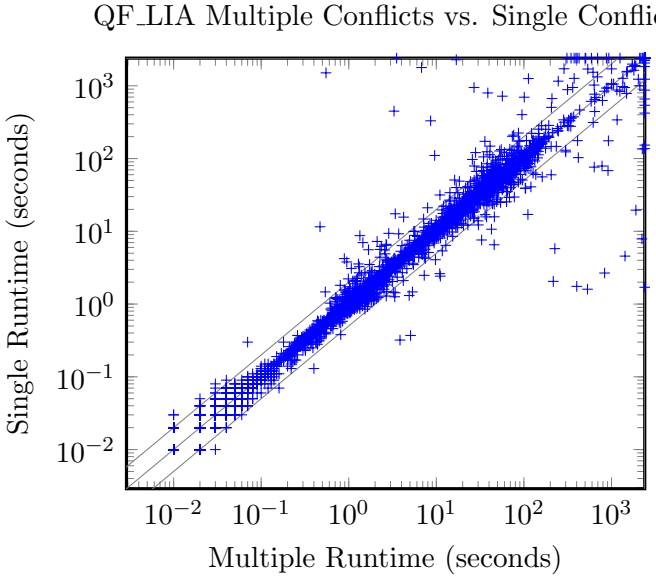


Figure 7.8: Multiple conflicts vs. single conflict on the QF_LIA SMT-LIB benchmarks

Algorithm 23: `HardBacktrack`(d')

Input : The decision level to backtrack to.
Effect : Reverts the bound value functions so they map to the bounds upto decision level d'. Recomputes a satisfiable assignment for the decision level d'.

```
1 if d ≤ d' then return;
2 while O = [[O', (γ, d*)]] with d* > d' do
3 |   O := O'
4 |   if γ = (x_i ≥ l_i) then L(x_i) := l_i;
5 |   if γ = (x_i ≤ u_i) then U(x_i) := u_i;
6 end
7 d := d';
8 Check();
```

Figure 7.9: Another simplex backtrack function

7.1.3 Backtracking through Recalculation

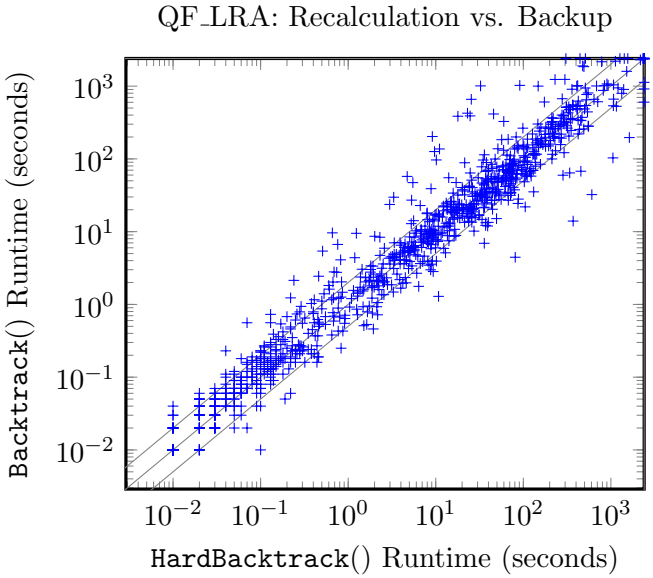
We have presented in Chapter 2.7.1 the original backtracking technique for Dutertre and de Moura's version of the simplex algorithm (see `Backtrack`() in Figure 2.5). We argued that this technique is efficient because it only performs two inexpensive steps: it reverts the bound values and it reverts the assignment to a backup solution assignment. However, this argument only guarantees that the backtracking function itself is fast and cheap. It does not guarantee that the backtracked state is a smooth continuation point for the simplex algorithm.

This is also the reason why most SMT theory solvers (e.g. CVC4 [9] and veriT [27]) use a different backtracking technique (see `HardBacktrack`() in Figure 7.9). The `HardBacktrack`() function reverts the bound values the same way the original `Backtrack`() function does, but it does not revert the assignment to the backup solution assignment. Instead, it computes a potentially new solution with the `Check`() function.

`HardBacktrack`() itself is obviously slower than `Backtrack`(). However, the backtracked state returned by `HardBacktrack`() is sometimes a better continuation point for the simplex algorithm. The reason is that a good continuation point for the simplex algorithm depends not only on the current assignment, but also on the current tableau. Recomputing the solution with `Check`() keeps the assignment and the tableau in sync, which results in a good continuation point. Reverting to the backup solution does the opposite and sometimes disconnects the assignment and the tableau, which results in a bad continuation point.

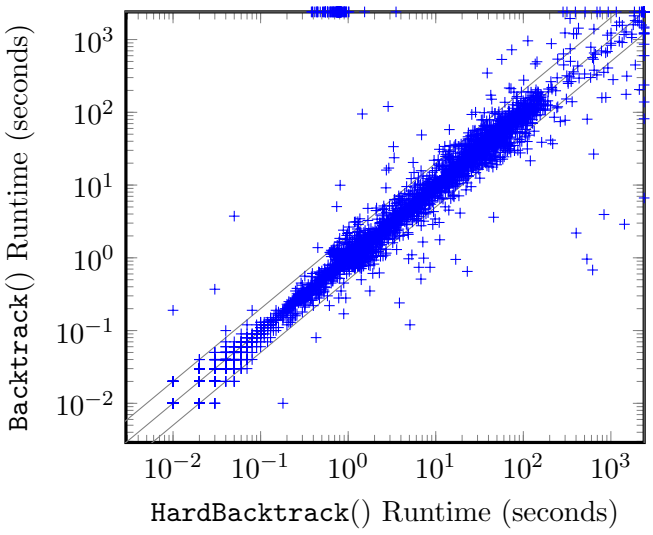
The smoother continuation point has the result that SPASS-IQ with the `HardBacktrack`() function is on average more efficient than SPASS-IQ with the standard `Backtrack`() function. Again, we support this argument by performing benchmark experiments with SPASS-SATT. We present the benchmark results through two scatter plots (Figures 7.10 & 7.11). Figure 7.10 compares the two backtrack functions on the QF_LRA benchmarks. To this end, we compare the default version of SPASS-SATT, i.e., SPASS-SATT with `HardBacktrack`() as its backtrack function (horizontal axis; command-line option `-b 0`), with the version of SPASS-SATT that uses `Backtrack`() as its backtrack function (vertical axis; command-line option `-b 1`). Symmetrically, Figure 7.11 compares the two backtrack functions on the QF_LIA benchmarks with the same solver configurations.

As mentioned before, SPASS-SATT with `HardBacktrack`() performs on average better than SPASS-SATT with `Backtrack`(). In the QF_LIA benchmarks, there even exists one benchmark family, viz., the `convert` benchmarks, where SPASS-SATT with `HardBacktrack`() solves much more problems than with `Backtrack`() (see Figure 7.12). In all other benchmark families, there are roughly as many problem instances where `HardBacktrack`() helps as there are problem instances where it hinders SPASS-SATT's search. The latter is most likely caused by benchmark instances where the backup solution is already a good continuation point and the recalculation is therefore just more expensive. Moreover, the increase in solved instances from the QF_LRA benchmarks could be attributed to performance fluctuations of our cluster because those instance are solved so close to the timeout.



SPASS-SATT with `HardBacktrack`() solves 1607 instances
SPASS-SATT with `Backtrack`() solves 1602 instances

Figure 7.10: Backtracking via recalculation (`HardBacktrack`()) vs. backtracking via backup (`Backtrack`()) on the QF_LRA SMT-LIB benchmarks



SPASS-SATT with `HardBacktrack`() solves 6806 instances
SPASS-SATT with `Backtrack`() solves 6683 instances

Figure 7.11: Backtracking via recalculation (`HardBacktrack`()) vs. backtracking via backup (`Backtrack`()) on the QF_LIA SMT-LIB benchmarks

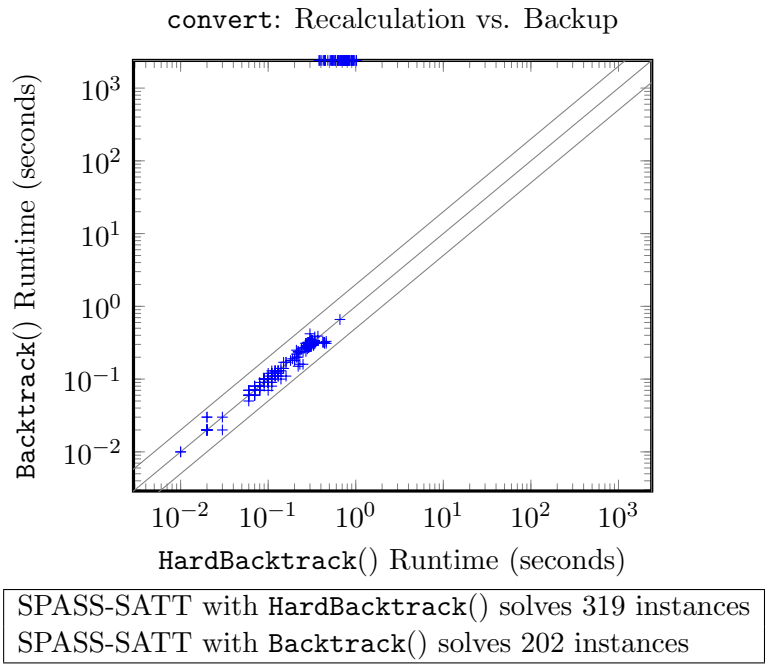


Figure 7.12: Backtracking via recalculation (`HardBacktrack`()) vs. backtracking via backup (`Backtrack`()) on the QF_LIA `convert` benchmark family

7.1.4 Data Structures

The most basic data structures in our linear arithmetic solver are based on the simplex implementation. They represent numbers, variables, bounds, and simplex tableaux. However, they are not just used in the simplex implementation but in fact in the whole implementation because they embody the most basic constructs of linear arithmetic, i.e., the variables that we need to assign and the constraints that define our satisfiable assignments, but which we transformed into bounds and tableau rows.

Multiprecision Arithmetic

SPASS-IQ is an arithmetic theory solver. This means that we have to represent numbers internally, which might sound easier than it actually is. First of all, SPASS-IQ has to handle three types of numbers: integers $\mathbb{Z}$, rationals $\mathbb{Q}$, and δ-rationals $\mathbb{Q}_\delta$.

The first two types are generally very popular and there already exist several libraries that contain efficient representations for them [75, 79]. The most obvious one are the integer and float types that are native to the *C* language. This representation is very efficient because their operations are optimized on the hardware level. This is also why we call them *hardware integers and floats*. They are, however, not arbitrarily precise, e.g., hardware integers have a finite range and hardware floats have to round numbers to reach a higher range than integers. This is problematic because SMT solvers—and, therefore, theory solvers—were originally designed for various verification tasks, e.g., hardware verification, software verification, and theorem proving. These task require that our implementation is sound, which means that we have to avoid rounding (errors), and, if possible, complete, which means that our numbers cannot be restricted to a finite range. Therefore, hardware integers and floats are not a good choice for our implementation.

Instead of hardware arithmetic, we are using *multiple/arbitrary precision arithmetic libraries* to represent our integers and rationals [75, 79].[5] Multiple/arbitrary precision arithmetic libraries typically represent integers and rational numbers through variable-length arrays of hardware integers. Therefore, their range is only limited by the available memory of the executing machine. Their operations stay efficient both for big and small numbers because they are using state-of-the-art algorithms written in highly optimized assembly code.

[5] Most other SMT theory solvers also represent integers and rationals internally with the help of arbitrary precision arithmetic libraries [9, 40, 49, 56].

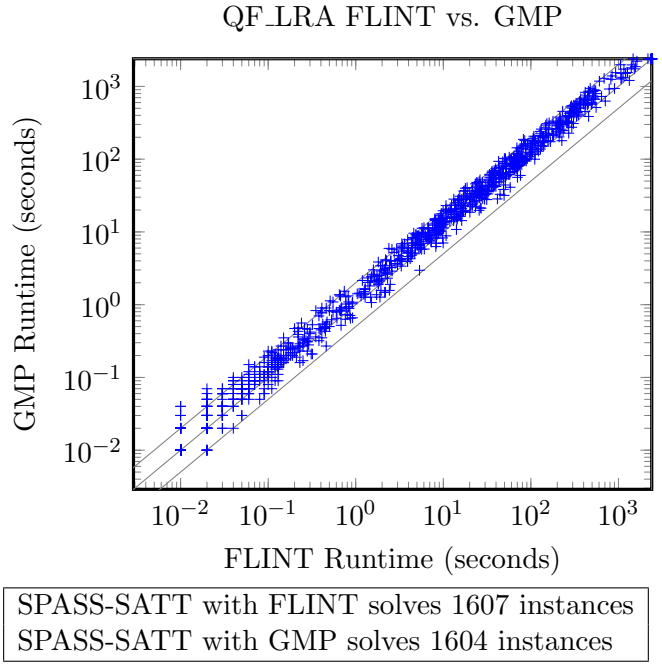


Figure 7.13: FLINT vs. GMP on the QF_LRA SMT-LIB benchmarks

SPASS-IQ can be compiled with one of two arbitrary precision arithmetic libraries: the *GNU Multiple Precision Library (GMP)* [75] and the *Fast Library for Number Theory* (FLINT) [79, 80]. Both libraries have their own advantages and disadvantages. For instance, GMP is mainly focused on representing big numbers and, therefore, on efficient algorithms for big number arithmetic. GMP is, however, much slower on small numbers than the native hardware types. This is problematic because verification problems might break the limits of hardware types, but rarely do so.

In contrast, FLINT's performance on small numbers comes much closer to the performance of hardware arithmetic. This is possible because FLINT is able to dynamically switch between hardware types and arbitrary precision types. Otherwise, FLINT is mostly an extension of GMP. This means that FLINT performs similarly to GMP on big numbers, with the exception of the overhead caused by the checks for the dynamic switch to hardware types. Therefore, FLINT is faster on problems with mostly small numbers and GMP is faster on problems with mostly big numbers. We personally prefer FLINT because we typically have to deal with more small numbers than big numbers.

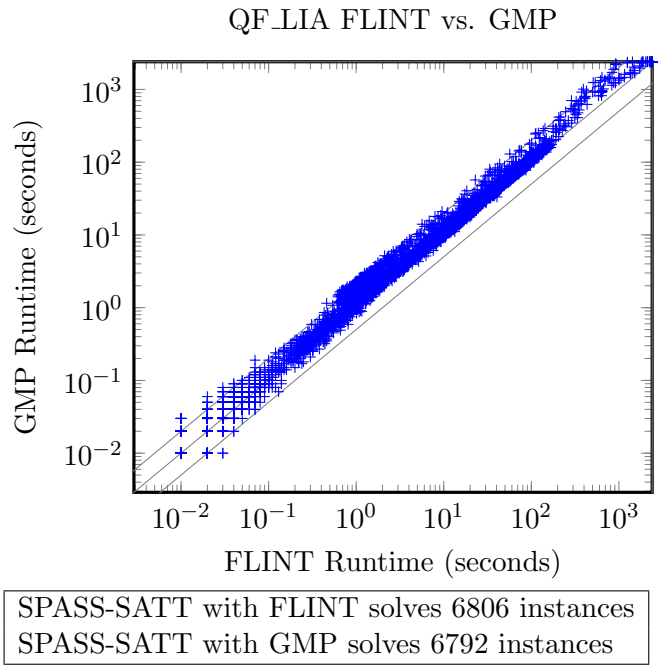


Figure 7.14: FLINT vs. GMP on the QF_LIA SMT-LIB benchmarks

For a more detailed comparison of the performance of GMP and FLINT see Figures 7.28 & 7.14. In these figures we compare two versions of SPASS-SATT over the SMT-LIB benchmarks [10]. The first version uses GMP (vertical axis) the second version FLINT (horizontal axis). Clearly, SPASS-SATT with FLINT is slightly superior on most problems and performs on the few problems with big numbers well enough to also win overall.

In contrast to rationals and integers, we could not find a satisfactory library for the extension of rationals to δ-rationals. Therefore, we had to implement the δ-rationals and the operations over them ourselves. Our implementation for the δ-rationals and the necessary operations follow the description from Chapter 2.3.2. This means we implemented δ-rationals as a pair of rationals and simply extended the operations over the rationals to δ-rationals.

For the remainder of this thesis, we refer to the hardware integers as `int`, to the arbitrary precision integers as `Integer`, to the arbitrary precision rationals as `Rational`, and to the arbitrary precision δ-rationals as `DRational`.

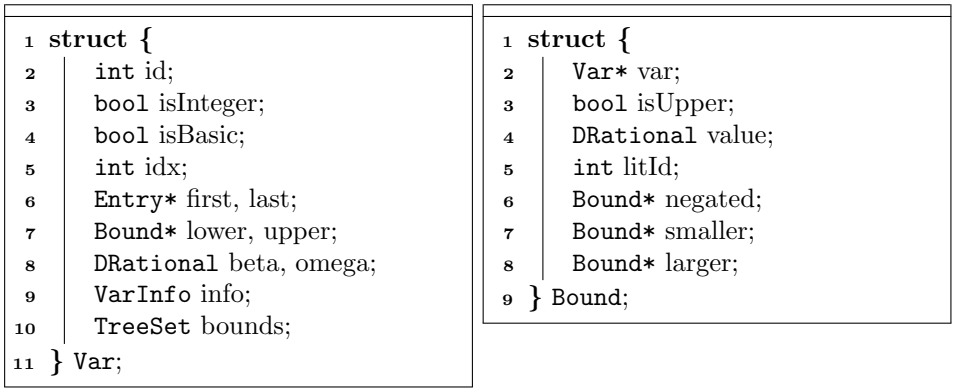


Figure 7.15: Data structures for variables and bounds

Variable Implementation

The variable names we use in linear arithmetic are typically just identifiers. There is, however, other information directly associated with our variables that we want to integrate into their representation. We have, therefore, chosen a more complicated data structure `Var` for our variables (see Figure 7.15). Naturally, our variables still need an identifier, which is represented by the field "`int` id". The next field "`bool` isInteger" marks the type of the variable, i.e., the field is set to true if the variable is an integer variable and to false if it is a rational variable. `Var` also includes several fields with simplex specific information: The field "`bool` isBasic" marks whether the variable is basic or non-basic. The field "`int` idx" represents the index of the variable in the simplex tableau. The index is a row index if the variable is basic and a column index otherwise. The fields "`Entry*` first, last" represent the first and last non-zero entry in the row/column associated with the variable (the definition of `Entry` can be found later in this section). Both fields are set to `NULL` if the row/column does not contain a non-zero entry. The fields "`Bound*` lower, upper" represent the current lower and upper bound of the variable (the definition of `Bound` can be found later in this section). This means we have implemented the bound value functions $\mathcal{L}$ and $\mathcal{U}$ as local fields instead of actual functions. Note that we express the lower (upper) bound value $-\infty$ (∞) by setting the field lower (upper) to `NULL`. Similarly, we have also implemented the current assignment β and the backup assignment ω as local fields "`DRational` beta, omega". The field "`VarInfo` info" stores redundant information (the definition of `VarInfo` can be found later in this section); This information is redundant because it can also be computed on the fly. However, the computation cost is high enough that it is more efficient to store and incrementally update this information. The final field "`TreeSet` bounds" is not relevant for SPASS-IQ but for its ex-

tension SPASS-SATT. It is an ordered tree set that stores all upper bounds $x_i \leq u_j$ constructed for the variable and orders them by their bound values. This ordering establishes a relationship that is especially useful for unate propagation (see Chapter 8).

Bound Implementation

We implement variable bounds $x_i \leq b_i$ or $x_i \geq b_i$ with the data structure `Bound` (see Figure 7.15). The first three fields are the most important fields for this chapter: The field "`Var*` var" defines the variable that is bounded; The field "`bool` isUpper" defines the bound direction, i.e., whether the bound is an upper or a lower bound; and the field "`DRational` value" defines the bound value of the bound.

The remaining fields of `Bound` are not relevant for SPASS-IQ but for its extension SPASS-SATT. The field "`int` litId" defines the link between the arithmetic representation of a literal as a bound and the boolean representation of a literal as a propositional variable. The field "`Bound*` negated" is also based on the literal definition and points to the negated version of the bound. This means an upper bound over a rational variable $x_i \leq u_i$ points to the lower bound $x_i \geq \bar{u_i}$ and vice versa. Similarly, an upper bound over an integer variable $x_i \leq u_i$ points to the lower bound $x_i \geq u_i + 1$ and vice versa. The final two fields "`Bound*` smaller" and "`Bound*` larger" are needed for unate propagation (see Chapter 8). The field "`Bound*` smaller" points to the largest smaller bound for the same variable and bound direction and the field "`Bound*` larger" points to the smallest larger bound for the same variable and bound direction. This means the bounds for the same variable and bound direction are linked together in a double-linked list ordered by the bound values. We can efficiently extend this list because of the ordered tree set "`TreeSet` bounds" stored in the associated variable. This tree set allows us to compute in logarithmic time the next largest/smallest bound in the set and from there we only need a constant number of operations for adding the new bound to the double-linked list.

Tableau Implementation

The simplex tableau $Az = y$ consists of three parts: the coefficient matrix A, the non-basic variables z, and the basic variables y. We implemented the tableau with the data structure `Tableau` (see Figure 7.16). The `Tableau` contains arrays "`Var[ ]` tableau.rows" and "`Var[ ]` tableau.columns" for our basic and non-basic variables, respectively. The position of a variable var in its respective array corresponds to the variable index var.idx, i.e., tableau.rows[var.idx] = var if var.isBasic = *true* and tableau.columns[var.idx] = var otherwise.

```
1 struct {
2 |   Var[ ] columns;
3 |   Var[ ] rows;
4 |   Integer[ ] rowCoeffs;
5 } Tableau;
```

```
1 struct {
2 |   int rowIdx, colIdx;
3 |   Entry* rowPrev, rowNext;
4 |   Entry* colPrev, colNext;
5 |   Integer coeff;
6 } Entry;
```

Figure 7.16: Data structures for simplex tableaux and their entries

The coefficient matrix A of the tableau $Az = y$ is implemented through several double-linked lists. To be more precise, each row and each column of the matrix A correspond to one double-linked list consisting of entries build from the data structure `Entry` (see Figure 7.16). Each such entry corresponds to one non-zero coefficient a_{ij} of the coefficient matrix A.[6] Each entry is also part of two double-linked lists: one row list and one column list. To this end, the data structure `Entry` contains the two fields "`int` rowIdx, colIdx" that indicate the entries position in the coefficient matrix A and, therefore, also the row and column it belongs to.

Based on these indices, a row list contains all entries with the same row index rowIdx. The list itself is formed by the fields "`Entry*` rowPrev, rowNext" that point to the previous and next entry in the row. If rowPrev of a given entry points to `NULL`, then the entry is the first entry in the row. If rowNext of a given entry points to `NULL`, then the entry is the last entry in the row. Moreover, we can access a given row with index i from the tableau through its corresponding basic variable, i.e., rows[i].first is the first entry of the row list and rows[i].last the last entry.

The column lists are defined symmetrically, i.e., a column list contains all entries with the same column index colIdx. The list itself is formed by the fields "`Entry*` colPrev, colNext" that point to the previous and next entry in the column. If colPrev of a given entry points to `NULL`, then the entry is the first entry in the column. If colNext of a given entry points to `NULL`, then the entry is the last entry in the column. Moreover, we can access a given column with index j from the tableau through its corresponding non-basic variable, i.e., columns[j].first is the first entry of the column list and columns[j].last the last entry.

We chose double-linked and not just single-linked lists because they make row summations more efficient. Or to be more precise, double-linked lists are more efficient at adding and removing single entries, especially, if the entries are linked in two lists. There are, however, three other steps that make row summations more efficient. As our first step, we reduce the number of mathematical operations needed for row summations by turning our rational

[6] Only representing the non-zero coefficient is advantageous because our coefficient matrices are typically sparse, i.e., many coefficients are zero.

Algorithm 24: `AddMultipleOfRowToRow`(var_i, c_i, var_k)

Input : A basic `Var` var_i, an `Integer` c_i, and another basic `Var` var_k
Effect : Adds the row of var_i multiplied by c_i to the row of var_k.

```
1  if c_i = 0 then return;
2  Integer d_i, d_k;
3  d_i := rowCoeffs[var_i.idx] ;
4  d_k := rowCoeffs[var_k.idx] ;
5  rowCoeffs[var_k.idx] := lcm(d_i, d_k);
6  d_i := c_i · lcm(d_i, d_k) ÷ d_i;
7  d_k := lcm(d_i, d_k) ÷ d_k;
8  Entry entry_i, entry_k;
9  entry_i := var_i.first;
10 entry_k := var_k.first;
11 while entry_i ≠ NULL and entry_k ≠ NULL do
12 |   int colvar_i, colvar_k;
13 |   colvar_i := columns[entry_i.colIdx].id;
14 |   colvar_k := columns[entry_k.colIdx].id ;
15 |   if colvar_i = colvar_k then
16 |   |   entry_k.coeff := d_k · entry_k.coeff + d_i · entry_i.coeff;
17 |   |   if entry_k.coeff = 0 then
18 |   |   |   Entry entry_h;
19 |   |   |   entry_h := entry_k.rowNext;
20 |   |   |   RemoveEntry(entry_k);
21 |   |   |   entry_k := entry_h;
22 |   |   else
23 |   |   |   entry_k := entry_k.rowNext;
24 |   |   end
25 |   |   entry_i := entry_i.rowNext;
26 |   else if colvar_i < colvar_k then
27 |   |   CreateEntryBeforeEntry(entry_i.colIdx, d_i · entry_i.coeff,
   |   |     entry_k);
28 |   |   entry_i := entry_i.rowNext;
29 |   else
30 |   |   entry_k.coeff := d_k · entry_k.coeff;
31 |   |   entry_k := entry_k.rowNext;
32 |   end
33 end
34 while entry_i ≠ NULL do
35 |   CreateEntryAtEndOfRow(entry_i.colIdx, d_i · entry_i.coeff, var_k);
36 |   entry_i := entry_i.rowNext;
37 end
38 while entry_k ≠ NULL do
39 |   entry_k.coeff := d_k · entry_k.coeff;
40 |   entry_k := entry_k.rowNext;
41 end
```

Figure 7.17: An efficient row summation function

matrix coefficients a_{ij} into integer coefficients.[7] To do so, we compute a common row denominator d_i for all matrix coefficients a_{ij} in the row i. Then we multiply all coefficients in the row with the common row denominator and get our integer entry coefficients $a'_{ij} := a_{ij} \cdot d_i$, which are saved in the fields "`Integer` coeff" of the entries. We also save the common row denominators in an array "`Integer[ ]` rowCoeffs" of the `Tableau` structure so that we are able to reconstruct our original coefficients. With the common row denominators, we now have $1 + L$ integers values per row, where L is the number of row entries. With the rational coefficients, we would have $2 \cdot L$ integers values per row instead (i.e., two per rational coefficient). This means we almost reduced the number of integer operands by half.

As our second step, we sort our row lists and keep them sorted as an invariant. We do so because the row ordering allows us to efficiently identify which columns are shared by two rows, i.e., for which columns both rows have non-zero coefficients. The actual ordering is not really relevant for optimizing row summations it is just important that all rows are sorted by the same well-founded total order. We personally choose for our order that columns[entry.colIdx].id is strictly increasing because this order optimizes our pivoting strategy, which we discuss later in more detail.

As our third step, we forbid any order invariants on the column lists, i.e., we expect that no part of our implementation ever requires that our column lists stay sorted in a specific way. We do so because it allows us to efficiently add a new entry to a given column, i.e., we can add it in constant time to the end of the column.

Based on these three steps, we are now able to define an efficient algorithm `AddMultipleOfRowToRow`() for row summation (Figure 7.17).[8] It is linear in the number of non-zero entries of both rows and keeps both rows sorted. The algorithm itself is based on the classic merge function of the merge sort algorithm. There are two reasons why the algorithm is linear. First of all, the double-linked nature of our rows and columns allows us to implement the functions `RemoveEntry`(), `CreateEntryBeforeEntry`(), and `CreateEntryAtEndOfRow`() in such a way that they have a constant runtime. (We omit the pseudocode for these functions because their names are descriptive enough.) Secondly, the sorted row lists allow us to efficiently identify which columns are (not) shared by the two rows.

```
1  struct {
2  |   int boolInfo;
3  |   int numEntries;
4  |   int numLow;
5  |   int numUpp;
6  |   int numTightLow;
7  |   int numTightUpp;
8  |   DRational difference;
9  |   int direction;
10 |   int pos;
11 } VarInfo;
```

```
1 enum {
2 |   hasLower = 1;
3 |   hasUpper = 2;
4 |   isTightAtLower = 4;
5 |   isTightAtUpper = 8;
6 |   isViolating = 16;
7 } VarInfoIndices;
```

Figure 7.18: Data structures for incrementally updated variable information

Storing and Incrementally Updating Information

The simplex algorithm regularly checks whether our tableau, bounds, and variables fulfill certain properties. For instance, whether and which bound of a basic variable is violated and whether this violation is already a conflict. All of these properties can be computed on a need to know basis. However, we typically check them more often than they actually change. This is bad because it can be quite expensive if we compute some of these properties regularly (e.g., every time linear in the number of non-zero row entries).

For this reason, we decided on a runtime efficient alternative. Instead of recomputing too expensive to compute information, we store a redundant copy of the information and update it when the actual changes happen. (Similar information storage and update schemes can be found in other SMT theory solvers, e.g., in SMTInterpol [40].) The drawback of this method is that it requires more memory. However, memory typically is a more readily available resource.

All of the information that we store and update is associated with one variable. We, therefore, store it as part of the variables structure `Var`. However, we try to avoid any confusion between the actually necessary fields and the redundantly stored information by moving the redundantly stored information into its own data structure `VarInfo` (see Figure 7.18).

Let us now discuss the fields of `VarInfo` and, therefore, the redundantly stored information. The first field "`int` boolInfo" of `VarInfo` is just a single integer. However, each of its bits corresponds to one boolean property. The stored boolean properties and their bit positions are stored in the enumeration `VarInfoIndices`: hasLower/hasUpper indicates whether the variable

[7] This scheme is also used in other SMT theory solvers, e.g., in veriT [27].

[8] Our actual implementation also always minimized the common row denominators. We skip this step in the pseudocode of `AddMultipleOfRowToRow`() to keep the pseudocode concise and readable.

has a finite lower/upper bound; isTightAtLower/isTightAtUpper indicates whether the variable has a tight lower/upper bound, i.e., whether the value assigned to the variable matches the lower/upper bound; and isViolating is true iff the variable assignment violates the upper or lower bound of the variable. The second field "`int` numEntries" counts simply the number of entries between var.first and var.last. The next four fields are only used when the variable is a basic variable. They count the number of non-basic variables that share an entry with this basic variable and fulfill certain properties: "`int` numLow" considers only non-basic variables with a finite lower bound when the associated entry coefficient is positive and with a finite upper bound when the associated entry coefficient is negative; "`int` numUpp" considers only non-basic variables with a finite upper bound when the associated entry coefficient is positive and with a finite lower bound when the associated entry coefficient is negative; "`int` numTightLow" considers only non-basic variables with a tight lower bound when the associated entry coefficient is positive and with a tight upper bound when the associated entry coefficient is negative; and "`int` numTightUpp" considers only non-basic variables with a tight upper bound when the associated entry coefficient is positive and with a tight lower bound when the associated entry coefficient is negative. The next two fields are also only used when the variable is a basic variable. During a call of the `Check`() function (see Figure 2.2), "`DRational` difference" stores the absolute difference between the variable assignment and its violated bound or zero if no bound is violated. Outside of a call to the `Check`() function, i.e., during a sequence of assertions (see Figure 2.4), "`DRational` difference" stores the difference between the variable assignment before the sequence and the variable assignment at the moment. During a call of the `Check`() function (see Figure 2.2), "`int` direction" indicates what bound is violated, i.e., 0 means no bound is violated, > 0 means the upper bound is violated, and < 0 means the lower bound is violated. Outside of a call to the `Check`() function, i.e., during a sequence of assertions (see Figure 2.4), "`int` direction" stores the directions of the newly asserted bounds, i.e., 0 means no new bounds for the variable, 1 and 3 mean a new lower bound has been asserted for the variable, and 2 and 3 mean a new upper bound has been asserted for the variable. The final field "`int` pos" indicates the position of the variable in the violating variable heap, which we discuss in more detail in Section 7.1.2.

Based on `VarInfo`, we are now able to compute some properties in constant runtime that were previously only computable in linear runtime. For instance, we can compute in constant time whether a basic variable and its row describe a conflict (see Figure 7.19). We previously defined a conflict by the statements in lines 7 and 13 of the `Check`() function (see Figure 2.2). These lines correspond to the checks in function `IsConflict`() (see Figure 7.19).

Algorithm 25: `IsConflict`(var_i)

Input : A basic `Var` var_i
Output : Returns *true* iff the basic variable and its defining row describe a conflict.

```
1 if var_i.info.direction > 0 then
2 |   return var_i.info.numTightLow = var_i.info.numEntries
3 else if var_i.info.direction < 0 then
4 |   return var_i.info.numTightUpp = var_i.info.numEntries
5 return false;
```

Figure 7.19: A function that efficiently determines whether a basic variable and its defining row describe a conflict

7.2 Branch-and-Bound Implementation

SPASS-IQ's second set of decision procedures revolves around an implementation of the branch-and-bound algorithm, which we already explained in Chapter 2.7. We need this implementation of branch-and-bound because our simplex implementation alone can only handle linear rational arithmetic. If we, however, extend the simplex implementation with a branch-and-bound implementation, then we also get a decision procedure for the theories of linear integer and linear mixed arithmetic.

Most SMT solvers (e.g., [9, 40, 49, 56]) implement branch-and-bound through a technique called *splitting-on-demand* [11], which delegates some of the branch-and-bound reasoning to a SAT solver. In order to keep more control over the branch-and-bound reasoning, we decided against splitting-on-demand and implemented branch-and-bound without the help of a SAT solver (see Subsection 7.2.1).

This also made it easier to complement branch-and-bound with other decision procedures: a simple rounding test (Chapter 2.7.4), a unit cube test (Chapter 4), a bound propagation procedure (Chapters 2.7.2 and 8.1.2), and our two transformations that reduce any problem into a bounded problem (Chapter 6). We have already discussed these extensions as standalone procedures in previous chapters and in these chapters we have also presented efficient ways to implement them. But we have not yet explained how we efficiently combine them. We remedy this in Subsection 7.2.4.

Moreover, we discuss in Subsection 7.2.2 the selection strategies that we use in our branch-and-bound implementation. To be more precise, we explain the selection strategies that SPASS-IQ uses for selecting branching variables and active nodes.

```
1 struct {
2 |   BNode* parent;
3 |   int depth;
4 |   bool branched;
5 |   BNodeType type;
6 |   Var* bvar;
7 |   DRational bval;
8 } BNode;
```

```
1 enum {
2 |   rootNode = 1;
3 |   lowerBranch = 2;
4 |   upperBranch = 3;
5 } BNodeType;
6 struct {
7 |   BNode* root;
8 |   BNode[ ] activeNodes;
9 } BTree;
```

Figure 7.20: Data structures for the branching tree

7.2.1 Branching Tree Implementation

Most SMT solvers implement branch-and-bound through a technique called *splitting-on-demand* [11]. This means they typically implement their branching tree through branching clauses

$$x_i \leq \lfloor c_i \rfloor \vee x_i \geq \lceil c_i \rceil$$

and let a SAT solver guide the selection of nodes through the selection of literals.

Using a SAT solver for branch-and-bound enables better conflict learning in the CDCL(LA) framework (see Chapter 2.6). To be more precise, a SAT solver can globally learn local conflict explanations for pruned nodes. This results in two advantages: (i) the SAT solver dynamically combines the pruning conflicts into one conflict for the whole branching tree and (ii) the learned pruning conflicts enable the SAT solver to automatically repeat similar prunings in later branch-and-bound searches. The second advantage sounds useful in theory but similar prunings are actually rare in practice. One reason is that CDCL(LA) solvers rarely need more than a small number of subsequent branch-and-bound searches in practice.

Splitting-on-demand also results in one major disadvantage: the SMT solver has to add at runtime new literals to the SAT solver because the bounds in a branching clause are not necessarily existing literals. These new literals confuse the regular SAT solver search because (i) they are disconnected from the original clauses and (ii) their actual relevance is only short term (during the current branch-and-bound search) but they are added long term (for the remaining run). Moreover, they also confuse later branch-and-bound searches because the new literals force us to repeat old branches without considering the changed model. In our opinion, this disadvantage outweighs the advantages of splitting-on-demand and, therefore, our own branch-and-bound implementation is separate from the SAT solver.

The main structure of the branch-and-bound algorithm is a so-called branching tree. We implement branching trees in SPASS-IQ with the help of the data structure `BNode` (Figure 7.20). Each `BNode` represents a node in a branching tree. The field "`BNode*` parent" points to the parent node of the given node and it points only to `NULL` if the given node is a root node. The field "`int` depth" stores the depth of the given node in its branching tree. The field "`bool` branched" is true when the node has been branched, i.e., our current node is the parent node for some other nodes. The field "`BNodeType` type" is the type of the given node and states whether the node is the root node ("rootNode") or whether the node was generated because of a lower or upper branch ("lowerBranch" or "upperBranch"). All nodes that point to the same parent node have different types, i.e., a node has at most one lower and one upper branch as its children. If the branch-and-bound method creates branches for the given node, then the field "`Var*` bvar" points to the selected branching variable and the field "`DRational` bval" stores the branching value, i.e., the assignment of the branching variable at the moment of branching.

Based on the `BNode` structure, we are now able to represent the whole branching tree `BTree` (Figure 7.20 through one root node (`BNode*` root) and one dynamically growing array that stores all active nodes (`BNode*` activeNodes).

Combining Pruning Conflicts

We mentioned before that a theory solver should return a conflict, i.e., an unsatisfiable subset of asserted constraints, if the problem is unsatisfiable. We construct such a conflict in our branch-and-bound implementation by combining the rational conflicts the simplex solver returns for each pruned node. This allows us to recursively generate one conflict for the whole branching tree. The problems associated with our nodes do, however, contain branching bounds and propagated bounds in addition to the original constraints from our input problem. We do not want these bounds in our combined conflict because we would otherwise need to add new literals on the fly, which is also the reason why we avoided splitting-on-demand.

In order to avoid all but the original constraints, we eliminate propagated bounds in all explanations by replacing them with their propagation explanation, i.e., a set of constraints that imply the propagation of the bound (see Chapter 2.7.2). This also means that we replace previously propagated bounds in the propagation explanation of a newly propagated bound. Thereby, all conflict/pruning and propagation explanations contain only branching bounds and original constraints.

Next, we simultaneously combine conflict/pruning explanations and remove the branching bounds from our explanations. For this purpose, let us look at the abstract recursive case of the procedure. As our input we get (i) a node C (ii) that has two child nodes $C_l := C \cup \{x_i \geq \lceil c_i \rceil\}$ and $C_u := C \cup \{x_i \leq \lfloor c_i \rfloor\}$ (iii) such that C'_l and C'_u are the (combined) conflict explanations for C_l and C_u, respectively. As the invariant of our procedure, we assume that $C'_l \subseteq C_l$, i.e., C'_l contains only constraints that also appear in C_l, and $C'_u \subseteq C_u$, i.e., C'_u contains only constraints that also appear in C_u. Based on this invariant, we have to differentiate three cases: Case 1: C'_l does not contain the branching bound $x_i \geq \lceil c_i \rceil$, which means that C'_l is also a conflict for C because $C'_l \subset C$. Therefore, $C' := C'_l$ is the combined conflict for C and we can stop. Case 2: C'_u does not contain the branching bound $x_i \leq \lfloor c_i \rfloor$, which means that C'_u is also a conflict for C because $C'_u \subset C$. Therefore, $C' := C'_u$ is the combined conflict for C and we can stop. Case 3: C'_l contains the branching bound $x_i \geq \lceil c_i \rceil$ and C'_u contains the branching bound $x_i \leq \lfloor c_i \rfloor$. Since $x_i \geq \lceil c_i \rceil$ is mixed equivalent to $\neg(x_i \leq \lfloor c_i \rfloor)$, we can apply boolean resolution to combine C'_l and C'_u. The result, $C' := (C'_l \cup C'_u) \setminus \{x_i \geq \lceil c_i \rceil, x_i \leq \lfloor c_i \rfloor\}$, is then the combined conflict for C and still mixed unsatisfiable. In all three cases, the invariant is maintained, i.e., the resulting combine conflict C' contains only constraints that also appear in C. This also means that the combined explanation C'_0 for the root node C_0 contains only original constraints.

7.2.2 Branch-And-Bound Selection Strategies

The branch-and-bound algorithm as defined in Chapter 2.7.3 is non-deterministic, e.g., with regard to the active node selection and the selection of a branching variable. Here we present the strategies that SPASS-IQ uses to resolve the non-deterministic selection. Moreover, we explain whether the choice of active node and branching variable has a measurable impact on the runtime performance of branch-and-bound.

Active Node Selection

At the beginning of every iteration of the branch-and-bound algorithm, an active node (corresponding to a set of asserted bounds C) has to be selected. Naturally, there are many different selection strategies [89, 128], but the most obvious one is depth-first.

We distinguish three cases for the *depth-first selection strategy*:

- if the root node C_0 is active, then we select it next;
- if branch-and-bound has created two child nodes C_l and C_u for the last selected node C, then we select either of them next;
- if the last selected node is pruned, then we select one of the active nodes with the largest number of ancestor nodes next.

The depth-first selection strategy still has some non-determinism because it does not say which child is selected in the second case [9]. In our own implementation of branch-and-bound, we resolve this non-determinism with the following case distinction on the sets of asserted bounds corresponding to the nodes $C_l := C \cup \{x_j \geq \lceil \beta(x_j) \rceil\}$ and $C_u := C \cup \{x_j \leq \lfloor \beta(x_j) \rfloor\}$:

- if x_j is *fixed* in C_l, i.e., there exists $v_j \in \mathbb{Q}_\delta$ such that $\{x_j = v_j\} \subseteq C_l$, but not fixed in C_u, i.e., $\{x_j = v_j\} \not\subseteq C_u$ for all $v_j \in \mathbb{Q}_\delta$, then we select C_l next;
- if x_j is fixed in C_u but not fixed in C_l, then we select C_u next;
- if both or none of the branches fix x_j, then we choose C_l if the input constraint set C_0 contains more constraints of the form $a_{ij}x_j + p_{ij} \leq 0$ with $a_{ij} > 0$ than $a_{ij} < 0$, and otherwise C_u.[10]

The number of constraints of the form $a_{ij}x_j + p_{ij} \leq 0$ with $a_{ij} > 0$ is a good indicator for the potential number of bound propagations between the nodes C and C_u. The number of constraints of the form $a_{ij}x_j + p_{ij} \leq 0$ with $a_{ij} < 0$ is a good indicator for the potential number of bound propagations between the nodes C and C_l. This means we try to select the branch with the smallest number of bound propagations. This, in turn, is a good indicator for the rational satisfiability of the selected node.

As mentioned before, there exist many other active node selection strategies and we compared many of them in practice. We were, however, unable to find a strategy that made a significant difference in performance. The only exception is the node we select directly after a branching. For this specific case, our proposed rule seems to give us a relevant advantage. We, therefore, conclude that it is more relevant to find a good successor after a successful branching than after a pruning.

Branching Variable Selection

Branch-and-bound also has to select a branching variable x_j for every selected node with a rational but not a mixed solution. Again there are many different selection strategies [89, 128].

In our own implementation of branch-and-bound, we select the most fractional integer variable. A variable x_j is more fractional than variable x_k under assignment $\beta(x) := s$, if:

$$|s_j - \lceil s_j \rfloor| > |s_k - \lceil s_j \rfloor| .$$

Any remaining non-determinism is resolved with an order over the variables.

[9] The third case might also look like it contains non-determinism, but it does not because we explore the nodes depth-first and every node has at most two children.

[10] We compute and store this preference before we start branch-and-bound as the variables preferred branching direction.

We selected this strategy after comparing several other branching variable selection strategies in practice. This includes selecting the minimal fractional integer variable, several fixed variable orders, and other more complex strategies. We were, however, unable to find a strategy that performed better. (Although the fixed variable orders performed worse.)

7.2.3 Extensions to Branch-And-Bound

We have already presented the most popular branch-and-bound extensions used by the SMT community in Chapter 2.7.4. Moreover, we have presented several of our own branch-and-bound extensions in the previous chapters. In this subsection, we will give again a short summary on each of the branch-and-bound extensions implemented in SPASS-IQ. The extensions used by SPASS-IQ are: a simple rounding test (Chapter 2.7.4), a unit cube test (Chapter 4), a bound propagation procedure (Chapters 2.7.2 and 8.1.2), and our two transformations that reduce any problem into a bounded problem (Chapter 6). The exact way in which they are integrated in SPASS-IQ and in which they are combined with branch-and-bound will be explained in the next subsection.

Simple Rounding

Simple Rounding (Chapter 2.7.4) is an easy way to turn any rational solution $\beta(x) = s$ computed during branch-and-bound, into a mixed assignment $s' \in \mathbb{Q}_\delta^{n_1} \times \mathbb{Z}^{n_2}$ that fulfills the types of all variables. We simply round all s_j with $j > n_1$ to an integer value $s'_j = \lceil s_j \rfloor$ and keep all other values as before. Beware that this new assignment might very well violate the node or input constraints. We, therefore, have to evaluate whether s' is a mixed solution for our original problem C_0. If the heuristic solution s' satisfies the original problem, then we stop our branch-and-bound search and return the solution instead. Otherwise, branch-and-bound ignores s' and continues the branch-and-bound search with s. This technique can also be found in other SMT theory solvers, e.g., in Yices [56].

Unit Cube Test

The unit cube test (Chapter 4) determines in polynomial time whether a polyhedron, i.e., the geometric representation of a system of inequalities, contains a hypercube parallel to the coordinate axes with edge length one. This information is useful because such a hypercube guarantees a mixed/integer solution for the system of inequalities.

The unit cube test is only a sound and not a complete decision procedure. However, there is at least one class of inequality systems, viz., absolutely unbounded inequality systems (see also Chapter 2.8), that are trivial for the unit cube test and much harder for many complete decision procedures. These problems are also the reason why we extend our own branch-and-bound implementation with the unit cube test.

The unit cube test itself replaces only the bounds in our original system of inequalities. The resulting problem over the theory of linear rational arithmetic is then solved internally with our simplex implementation. Note also that our implementation of the simplex algorithm is designed to efficiently exchange bounds (see Chapter 2.7). After applying the test, we can use this design to easily recover the original system by reverting to the original bounds. In doing so, the unit cube test conserves the incremental connection to the different original systems. It is, therefore, an incrementally efficient technique.

Bound Propagation

Bound propagation (Chapter 2.7.2) is a technique for linear arithmetic but in itself not a decision procedure.[11] It is mainly used to extend other decision procedure so they perform better in practice. Bound refinement takes as input an inequality $a_{ij}x_j + p_{ij} \leq 0$ and several variable bounds from our current set of constraints C. Then it tries to propagate an entailed variable bound for (at least) one of the variables x_j with non-zero coefficient a_{ij}. In the case that bound refinement succeeds, it produces a variable bound $x_j \leq u_j$ ($x_j \geq l_j$) that is not subsumed by an existing bound in C.[12] The produced bound $x_j \leq u_j$ ($x_j \geq l_j$) can then be explicitly added to C. The computation of the entailed bounds is described in detail in Chapters 2.7.2.

For linear mixed/integer arithmetic, we combine bound propagation with *constraint tightening*. This means we round any bound $x_j \leq u_j$ ($x_j \geq l_j$) for an integer variable x_j to its closest integer value $x_j \leq \lfloor u_j \rfloor$ ($x_j \geq \lceil l_j \rceil$). In many cases, bound refinement combined with constraint tightening is enough to cutaway some of the rational solutions while keeping all of the mixed/integer solutions. These cuts are not very strong but they have the advantage that they always subsume another inequality/bound. Thus, bound propagation does not increase the number of constraints and, therefore, not the time for computing a rational solution. In SPASS-IQ, we always perform a bounded number of bound propagations before calculating a rational solution for a new branch.

[11] Bound Refinement/Propagation is used in most SMT theory solvers [9, 40, 41, 49, 56].

[12] A bound $x_j \leq u_j$ ($x_j \geq l_j$) is subsumed if $(x_j \leq u'_j) \in C$ with $u'_j \leq u_j$ ($(x_j \geq l'_j) \in C$ with $l'_j \geq l_j$)

Reduction into Bounded Problems

Branch-and-bound alone is an incomplete decision procedure and we already explained in Chapter 2.8 that the unboundedness of a problem indicates whether branch-and-bound terminates. However, we also presented two transformations in Chapter 6 (the Mixed-Echelon-Hermite transformation and the Double-Bounded reduction) that reduce any unbounded problem into a mixed equisatisfiable problem that is bounded. The transformed problem can then be solved with our branch-and-bound implementation because it is complete for bounded problems.

Although the transformed problem is only mixed equisatisfiable, there still exist ways (Corollary 6.2.9 & Lemma 6.3.5) to efficiently convert certificates of (un)satisfiability between the transformed and the original system. Our method is efficient, compared to other extensions that complete branch-and-bound, because it is fully guided by the structure of the problem. Moreover, it can be implemented in an incrementally efficient way (see Section 6.4).

Finding Unbounded Directions Efficiently

The Mixed-Echelon-Hermite transformation and the Double-Bounded reduction are useful to prevent branch-and-bound from diverging. However, they might reduce the efficiency of branch-and-bound when applied to already bounded problems. We, therefore, use our bounded basis techniques from Chapter 5.5 to efficiently determine whether an input problem is (un)bounded and which directions are (un)bounded.

7.2.4 An Efficient Combination of Techniques

We combine in SPASS-IQ several decision procedures in order to handle problems over the theory of linear mixed/integer arithmetic. This combination of decision procedures is divided into two parts: the *preprocessing procedures*, i.e., the procedures we apply once before we actually apply branch-and-bound, and the *inprocessing procedures*, i.e., the procedures we apply during every iteration of branch-and-bound.

The *preprocessing procedures* (Figure 7.21) have two purposes: they either prevent us from unnecessarily constructing a branching tree because there exists an easy to find mixed solution; or they transform the input problem because it is otherwise too hard for branch-and-bound. To this end, we first search for a rational solution with the simplex algorithm. If we already fail to find a rational solution, then there cannot exist a mixed solution. Therefore, we can stop and return unsatisfiable with the conflict from the

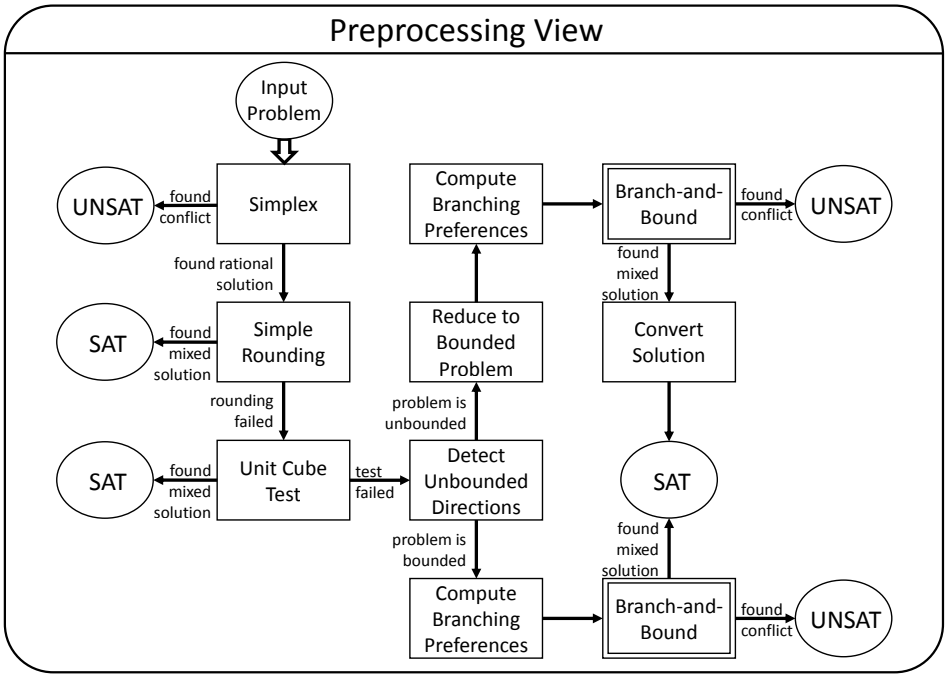


Figure 7.21: A control flow graph of the combination of techniques focused on the preprocessing procedures. The calls to the main loop (Figure 7.22) are labeled as branch-and-bound.

simplex algorithm as our proof. Next we try to round our rational solution to a mixed assignment. If this assignment is a solution, then we are also already done. We stop and return satisfiable with the rounded assignment as our proof.

Otherwise, we try to apply the unit cube test as another shortcut to a potentially mixed solution. This test is important because it alone automatically solves all absolutely unbounded problems and some other problems too. We do, however, skip the unit cube test if one of the original integer variables has bounds $\mathcal{U}(x_i) - \mathcal{L}(x_i) = 1$ or if one of the slacked variables has bounds $\mathcal{U}(x_i) - \mathcal{L}(x_i) = 0$. In these cases, the unit cube test is guaranteed to fail and would just cost time. If some of the original integer variables are fixed, i.e., $\mathcal{U}(x_i) - \mathcal{L}(x_i) = 0$, then we treat them as if they were replaced by their constant values. This avoids another case where the unit cube test would otherwise automatically fail.

If the unit cube test fails, then we determine whether the problem is bounded and which directions are unbounded. To this end, we compute a bounded basis with the techniques proposed in Chapter 5.5.[13] If the bounded basis is not full-dimensional, i.e., if there are unbounded directions,

[13]We skip this step if all original integer variables are explicitly bounded.

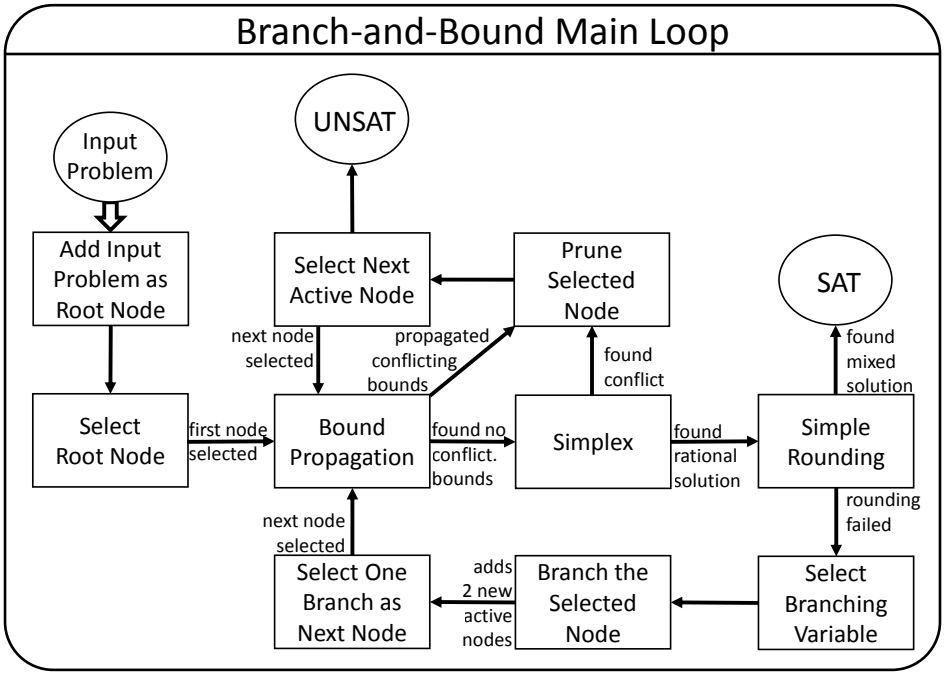


Figure 7.22: A control flow graph that describes the main loop of our combination of techniques, i.e., the procedures we apply in SPASS-IQ during every iteration of branch-and-bound.

then we use the transformations from Chapter 6 to reduce our problem to an equisatisfiable but bounded problem. The bounded problem (whether it is the original or the transformed problem) is then analyzed and we compute the preferred branching directions for all integer variables as explained in Subsection 7.2.2.

As our next step, we apply branch-and-bound extended by *inprocessing procedures* to the bounded problem (Figure 7.22). This is also the main loop of our combined procedure. We start the loop by adding our input problem as the root node of the branching tree and select it as our first selected node. The remaining branch-and-bound algorithm is structured as an iterative process for every selected node. In every iteration, we take our selected node and perform bound propagations on the subproblem C represented by the selected node (Chapter 2.7.2). We do, however, limit the number of propagated bounds so that we get at most a hundred propagated lower/upper bounds for each variable at the root node and at most ten propagated lower/upper bounds for each variable at all other nodes. Depending on the propagated bounds C', we now have to distinguish between two cases.

In case one, the propagation results in two asserted bounds for a variable x_i that are contradicting (i.e., $\mathcal{U}(x_i) < \mathcal{L}(x_i)$). This means that the subproblem C has no mixed solution and we can prune the selected node. Next we try to select one of the still active nodes based on the selection strategy presented in Subsection 7.2.2. If there are no more active nodes, then we have proven that the input problem has no mixed solution and we can return unsatisfiable together with a conflict constructed as described in Subsection 7.2.1. Otherwise we select and remove one of the active nodes from the set of active nodes, and continue with a new iteration of branch-and-bound.

In case two, the propagation did not result in two conflicting bounds and we continue our execution by applying the simplex algorithm to the subproblem C extended by the propagated bounds, i.e., $C \cup C'$. Depending on the result of the simplex algorithm, we have to distinguish again between two cases. We prune the selected node in the case that the simplex algorithm returns unsatisfiable. As in the previous pruning case, we now try to select one of the still active nodes based on the selection strategy presented in Subsection 7.2.2. If there are no more active nodes, then we have proven that the input problem has no mixed solution and we can return unsatisfiable together with a conflict constructed as described in Subsection 7.2.1. Otherwise we select and remove one of the active nodes from the set of active nodes, and continue with a new iteration of branch-and-bound.

In the case that the simplex algorithm returns satisfiable, we get at least an assignment β that is a rational solution for $C \cup C'$ and, therefore, also for C. We use this rational solution β for a simple rounding test (Chapter 2.7.4) and determine whether the rational solution rounded to a mixed assignment β' is also a mixed solution.[14] If it is a mixed solution, then we can stop branch-and-bound and return satisfiable because the mixed solution for the subproblem is also a mixed solution for the original problem.

If the mixed assignment is not a solution, then we select a branching variable x_i based on the rational solution β (see Subsection 7.2.2). Next we branch the node C into two new nodes that represent the subproblems $C_l := C \cup C' \cup x_i \geq \lceil \beta(x_i) \rceil$ and $C_u := C \cup C' \cup x_i \leq \lfloor \beta(x_i) \rfloor$, respectively. Our algorithm then selects one of the nodes C_l and C_u as the next selected node and adds the other one to the list of active nodes (see Subsection 7.2.2 for the selection strategy). We continue with the newly selected node in a new iteration of branch-and-bound.

[14] Note that simple rounding does not change the assignment if the rational solution is already a mixed solution.

Multiple Tableau Instances

The basis of our implementation is the simplex algorithm and the basic representation of our constraints in the simplex implementation was through the `Tableau` structure. The simplex algorithm does, however, change the coefficient matrix of the `Tableau` and, thereby, the explicit constraints it represents. These changes, also called pivots, are equivalence preserving so our original constraints are still implicitly represented. We need, however, the original versions of the constraints for some of our decision procedures, i.e., the simple rounding test (Chapter 2.7.4), the unit cube test (Chapter 4), and the Mixed-Echelon-Hermite transformation (Chapter 6). To be more precise, we need the original versions in order to (i) test the rounded assignments from our simple rounding test efficiently; (ii) compute the changed bound values for the unit cube test based on the original coefficient matrix; (iii) transform the original coefficient matrix into its Mixed-Echelon-Hermite normal form.

For this purpose, we have at all times two instances of the `Tableau` structure: the static instance and the dynamic instance. The static instance explicitly represents our original constraints and is never pivoted nor do we add branching or propagating bounds to it. The dynamic instance implicitly represents our original constraints and we use it for our actual simplex computations.[15]

We need three more instances of the `Tableau` structure when we encounter a partially unbounded problem.[16] The first and second instances are again static and dynamic instances, but this time they represent our original constraints after the Mixed-Echelon-Hermite transformation. The static instance explicitly represents our original transformed constraints and is never pivoted nor do we add branching or propagating bounds to it. It is used for the efficient simple rounding test on the transformed constraints and it is needed for the iterative extension of the Mixed-Echelon-Hermite transformation.[17] The dynamic instance implicitly represents our transformed constraints and we use it for our actual simplex computations on the transformed system. The third instance represents the column transformation matrix for our Mixed-Echelon-Hermite transformation. It is needed

[15] This also includes the simplex computations for the unit cube test. We adjusted the bound values of the dynamic instance based on the static instance and let the simplex algorithm run on the dynamic instance.

[16] Note that we cannot simply overwrite the static and the dynamic instance of the original constraints. We need the original static instance as a copy of all original constraints for the iterative extension of the Mixed-Echelon-Hermite transformation. We need the original dynamic instance to store and to incrementally extend the current bounded basis, which is also needed for the iterative extension of the Mixed-Echelon-Hermite transformation.

[17] Note that the iterative extension of the Mixed-Echelon-Hermite transformation occurs only if we combine multiple theories, which we do not do in the current version of SPASS-SATT.

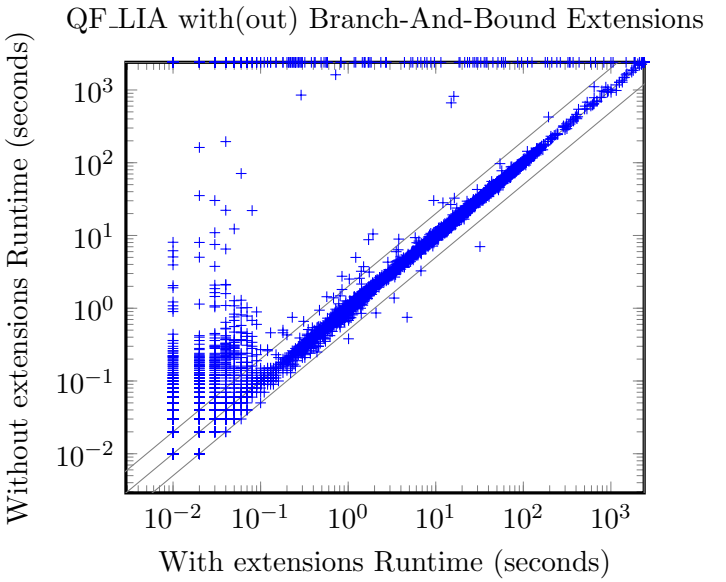


Figure 7.23: With(out) branch-and-bound extensions on the QF_LRA SMT-LIB benchmarks

for the efficient solution conversion between the original and the transformed system and for the iterative extension of the Mixed-Echelon-Hermite transformation. (See Chapter 6.4.3 for more details on why we need these instances for the iterative extension of the Mixed-Echelon-Hermite transformation.)

7.2.5 Branch-And-Bound Experiments

In order to measure the impact of our various branch-and-bound extensions, we performed a series of benchmark experiments with SPASS-SATT, the CDCL(LA) extension of SPASS-IQ. Our first experiment (see Figure 7.23) in this series of experiments examines the impact that the combination of our extensions have on SPASS-SATT's performance on the QF_LIA benchmarks. To this end, we compare the default version of SPASS-SATT with all four extensions (horizontal axis) and the version of SPASS-SATT without any of the extensions (vertical axis). Clearly, our extensions improve the number of solved instances by a significant amount. This amount becomes even more significant if we look at Figure 7.24 that illustrates how rarely we actually need branch-and-bound at all (641 out of 6947 instances

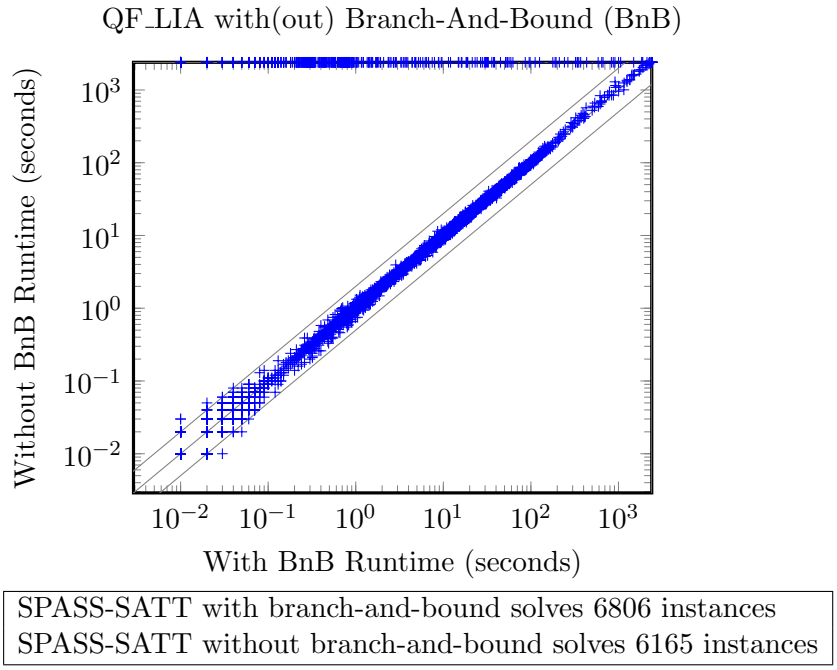


Figure 7.24: With(out) branch-and-bound (BnB) on the QF_LRA SMT-LIB benchmarks

or 1912 out of 6947 instances if we also turn off the unit cube test).[18] In fact, SPASS-SATT needs branch-and-bound only for the benchmark families `20180326-Bromberger` (for 200 out of 806 instances or 618 out of 806 instances if we also turn off the unit cube test), `arctic-matrix` (for 3 out of 100 instances), `calypto` (for 21 out of 37 instances), `CAV-2009` (for 9 out of 591 instances or 465 out of 591 if we also turn off the unit cube test), `CIRC` (for 11 out of 51 instances), (can only solve 36 of them) `convert` (for 282 out of 319 instances), `cut_lemmas` (for 69 out of 93 instances), `dillig` (for 4 out of 233 instances or 216 out of 233 if we also turn off the unit cube test), `miplib2003` (for 2 out of 16 instances), `prime-cone` (for 18 out of 27 instances or 26 out of 27 if we also turn off the unit cube test), `slacks` (for 4 out of 233 instances or 204 out of 233 if we also turn off the unit cube test), `tightrhombus` (for 12 out of 12 instances), and `tropical-matrix` (for 6 out of 107 instances).

The remaining figures in this section illustrate the impact of each of our four branch-and-bound extensions separately.

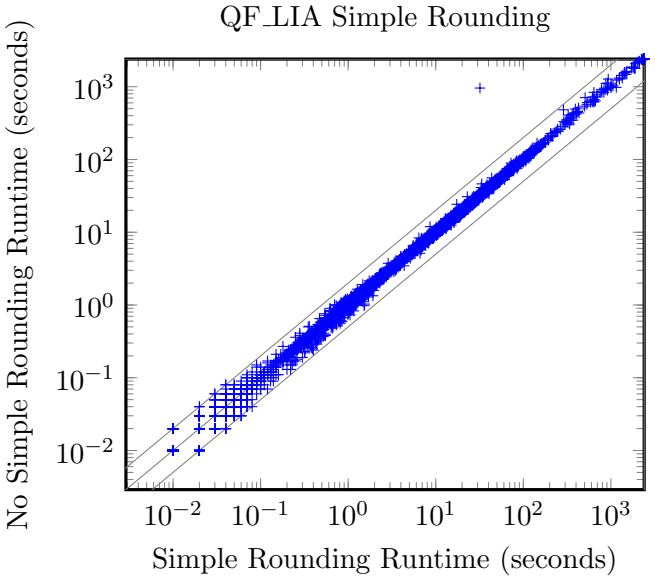


Figure 7.25: With(out) simple rounding on the QF_LIA SMT-LIB benchmarks

Simple Rounding

Figure 7.25 examines the impact that simple rounding has on SPASS-SATT's performance on the QF_LIA benchmarks. In it, we compare the default version of SPASS-SATT with simple rounding (horizontal axis; command-line option `-LASR 1`) and the version of SPASS-SATT without simple rounding (vertical axis; command-line option `-LASR 0`). The plot shows that simple rounding has no impact on the QF_LIA benchmarks with the exception of one single problem, which belongs to the `miplib2010` benchmark family.

[18] These statistics do not include instances that may require branch-and-bound but could not be solved in the given time limit.

Bound Propagation

We mentioned before that we put a threshold on the number of bound propagations. To be more precise, we limit at every branching node the number of bounds we propagate per variable. In Figure 7.26, we examine the impact that different bound propagation thresholds have on SPASS-SATT's performance on the QF_LIA benchmarks. In the legends of the plots, we list the different bound propagation settings, i.e., the thresholds on the maximum number of bounds we propagate per branching node and variable (command-line option −`LABP <threshold>`). Moreover, the two numbers in brackets behind each threshold are the numbers of satisfiable and unsatisfiable instances that we can solve with the respective threshold. Naturally, a threshold of 0 means that SPASS-SATT performs no bound propagations and a threshold of ∞ means that SPASS-SATT performs arbitrarily many bound propagations.

The figures show that activating bound propagation (i.e., selecting a non-zero threshold value) is beneficial, although it has no major impact because SPASS-SATT solves only a few more problems. Moreover, the results for the different positive threshold values are too close and can be attributed to performance fluctuations of our cluster. Therefore, we assume that the actual threshold on the number of bound propagations is irrelevant as long as it is non-zero. This observation changes if we also turn of our bounding transformations. If we turn them off, then SPASS-SATT might waste time on unnecessary propagations if the threshold is set too large and might even diverge on some instances if the threshold is set to ∞.

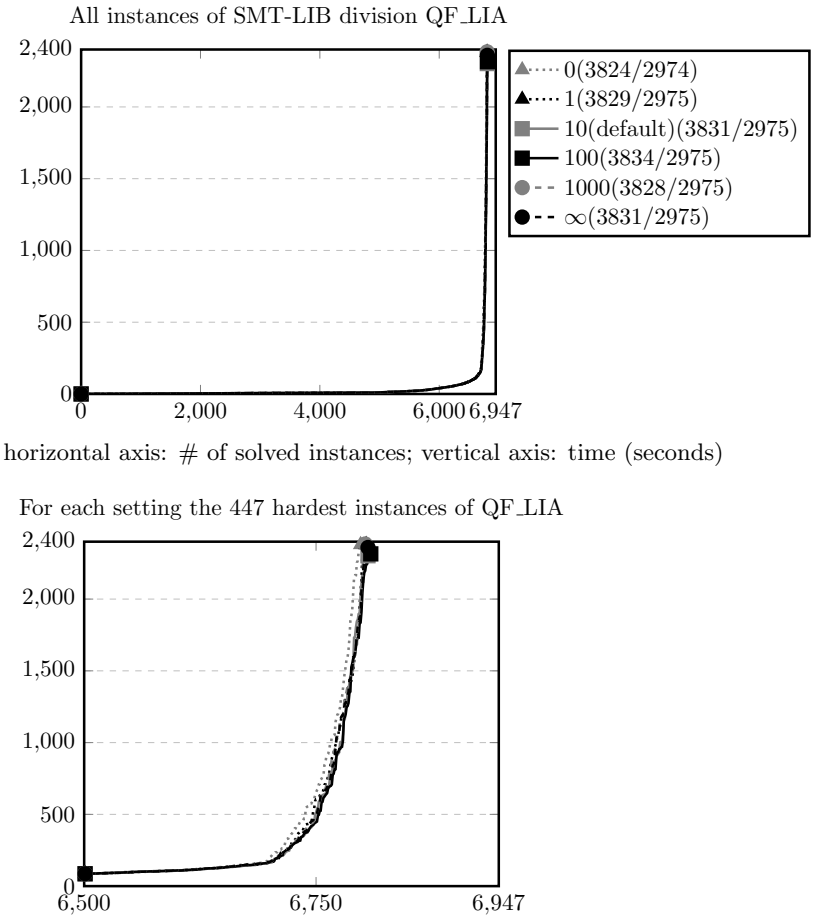
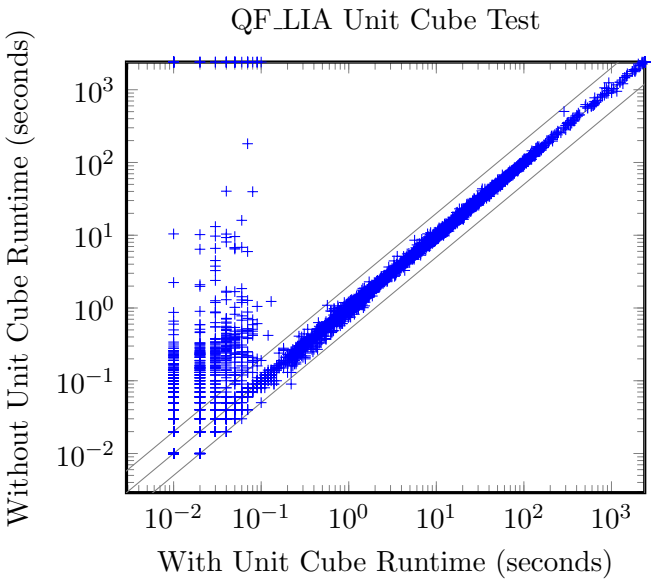


Figure 7.26: Results for different bound propagation settings on the SMT-LIB benchmark division QF_LIA

Unit Cube Test

We discuss unit cube tests in more detail in Chapter 4. In that Chapter, we present several benchmark experiments over the unit cube test that focus on absolutely unbounded problems. In Figure 7.27, we examine the impact that the unit cube test has on SPASS-SATT's performance on all QF_LIA benchmarks. To do so, we compare the default version of SPASS-SATT employing the unit cube test (horizontal axis; command-line option `-C 1`) and the version of SPASS-SATT without the unit cube test (vertical axis; command-line



SPASS-SATT with unit cube test solves 6806 instances
SPASS-SATT without unit cube test solves 6750 instances

Figure 7.27: With(out) unit cube test on the QF_LIA SMT-LIB benchmarks

option `-C 0`). The plot shows that SPASS-SATT employing the unit cube test is faster on many problem instances in the QF_LIA benchmarks. Moreover, the unit cube test causes only a minor, almost immeasurable overhead on problem instances where it is not successfully applicable.

What the plot does not show is that there are actually over 1483 instances of absolutely unbounded problems in the QF_LIA benchmarks[19] and that SPASS-SATT without unit cube tests is still efficient enough to solve the majority of them on its own. However, this is only possible because the majority of absolutely unbounded problems in the QF_LIA benchmarks consist of a reasonably small number of variables and constraints. Therefore, general improvements to SPASS-SATT's simplex and branch-and-bound implementation (e.g. bound propagation and simple rounding) are enough to handle these problems. However, these general improvements do not scale

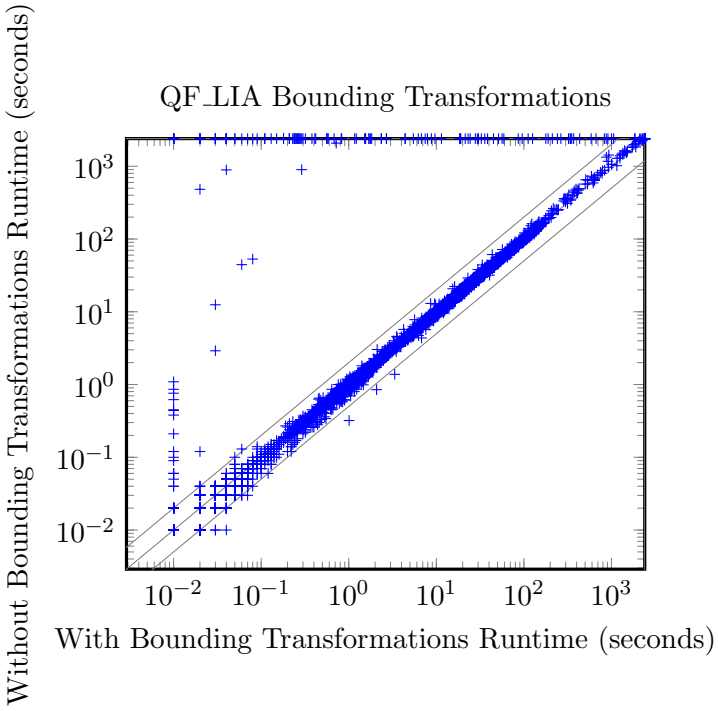
[19]These instances are 229 of the 233 `dillig` benchmarks designed by Dillig et al. [52], 503 of the 591 `CAV-2009` benchmarks also by Dillig et al. [52], 229 of the 233 `slacks` benchmarks which are the `dillig` benchmarks extended with slack variables [87], 503 of the 806 `20180326-Bromberger` benchmarks which include among other benchmarks the `CAV-2009` benchmarks extended with slack variables [30], and 19 of the 37 `prime-cone` benchmarks, that is, "a group of crafted benchmarks encoding a tight n-dimensional cone around the point whose coordinates are the first n prime numbers" [87].

on larger absolutely unbounded problems, which can also be observed in the plot. In comparison, the unit cube test scales on absolutely unbounded problems and stays efficient for much larger absolutely unbounded problems than those found in the QF_LIA benchmarks.

Reduction into Bounded Problems

In Chapter 4, we have presented two transformations (the Mixed-Echelon-Hermite transformation and the Double-Bounded reduction) that reduce any unbounded problem into a mixed equisatisfiable problem that is bounded. For this reason, we also call the combination of these two transformations a *bounding transformation*. We need a bounding transformation because branch-and-bound alone is only guaranteed to terminate on bounded problems. In Figure 7.27, we examine the impact that the bounding transformation has on SPASS-SATT's performance on all QF_LIA benchmarks. To do so, we compare the default version of SPASS-SATT employing the bounding transformation (horizontal axis; command-line option `−B 1`) and the version of SPASS-SATT without the bounding transformation (vertical axis; command-line option `−B 0`). The plot shows that SPASS-SATT can solve 169 additional benchmark instances from the QF_LIA benchmarks if it employs the bounding transformation.[20] Moreover, the bounding transformation causes only a minor, almost immeasurable overhead on problem instances where it is not successfully applicable.

[20] These instances are 162 of the 806 `20180326-Bromberger` benchmarks which include 303 partially unbounded problems [30], 1 of the 100 `arctic-matrix` benchmarks which include at least 2 partially unbounded problems [42], 3 of the 93 `cut_lemmas` benchmarks which include 5 partially unbounded problems [76], 1 of the 233 `slacks` benchmarks which include 3 partially unbounded problems [87], and 2 of the 107 `tropical-matrix` benchmarks which include at least 4 partially unbounded problems [42] .



SPASS-SATT with bounding transformations solves 6806 instances
SPASS-SATT without bounding transformations solves 6637 instances

Figure 7.28: With(out) bounding transformations on the QF_LIA SMT-LIB benchmarks

Chapter 8

Implementation of SPASS-SATT

SPASS-SATT uses at its core a CDCL(LA) implementation that combines the CDCL (conflict-driven-clause-learning)-based SAT solver SPASS-SAT from the SPASS Workbench [3] with our LA theory solver SPASS-IQ. In this chapter, we explain the construction of SPASS-SATT in more detail. We start in Section 8.1 by outlining the interaction between the theory solver and the SAT solver in the CDCL(LA) implementation. This also includes extensions to the theory reasoning that enhance and guide the search of the SAT solver.

In Section 8.2, we explain the preprocessing techniques incorporated into SPASS-SATT. We start by giving a short introduction to the input language used by SPASS-SATT (SMT-LIB standard v2.0) [12] and specifically the term expressions we need to handle because of this language. The two most troublesome types of expressions introduced by the SMT-LIB language are *let expressions* and *if-then-else expressions*. They are so troublesome because CDCL(LA) cannot handle them directly and because their removal can cause an exponential blow-up in the formula size. The purpose of our first two sets of preprocessing techniques is, therefore, the intelligent removal of let expressions and if-then-else expressions.

The first preprocessing technique, which we present in Subsection 8.2.2, is just our internal term representation. This term representation is *shared*, i.e., we only need one copy for equivalent subterms; even if we have exponentially many occurrences of the same subterm. This allows us to remove let expressions and ignore the resulting exponential blow-up of the formula size because the number of different subterms does not increase.

Based on the shared term representation, there also exist some efficient simplifications that can reduce the overall formula size upto an exponential factor. The most important simplifications that reduce our formula size are presented in Subsection 8.2.3. Here we explain the intelligent removal

and simplification of if-then-else expressions. This includes simplifications for constant if-then-else expressions, compression of if-then-else expressions, lifting shared terms in nested if-then-else expressions, bounding constant if-then-else expressions, as well as the reconstruction of naively eliminated if-then-else expressions.

With the shared term representation, we can also perform some other preprocessing techniques more efficiently, e.g., the elimination of certain variables through substitution (see Subsection 8.2.4) and the transformation of certain inequalities into clauses (see Subsection 8.2.4). Moreover, our shared term representation brings all arithmetic inequalities into a canonical form.

The last preprocessing technique that we mention is the small-clause-normal-form transformation (see Subsection 8.2.4). It is also the only unique feature of SPASS-SATT that we present in this chapter.[1] All other techniques presented in this chapter have already been available in other SMT solvers such as CVC4 [9], MathSAT [41], Yices [56], and Z3 [49], but not all in one tool. Although these techniques are contained in existing SMT solvers, not all have been described in the respective literature.

Experimental Setup

For this chapter, we have performed several experiments in order to estimate the impact of SPASS-SATT's various features. The benchmarks for our experiments are the 1649 SMT-LIB benchmarks for quantifier free linear rational arithmetic (QF_LRA) and the 6947 SMT-LIB benchmarks for quantifier free linear integer arithmetic (QF_LIA) from the SMT-LIB (satisfiability modulo theories library) [10]. All experiments compare different configurations of SPASS-SATT with each other. The experiments are preformed on a Debian Linux cluster from which we allotted one core of an Intel Xeon E5620 (2.4 GHz) processor, 8 GB RAM, and 40 minutes to each combination of problem and SPASS-SATT configuration.

[1] Other unique features of SPASS-SATT, viz., unit cube test and bounding transformations, are presented in Chapter 7, where we describe SPASS-IQ, the theory solver used inside SPASS-SATT.

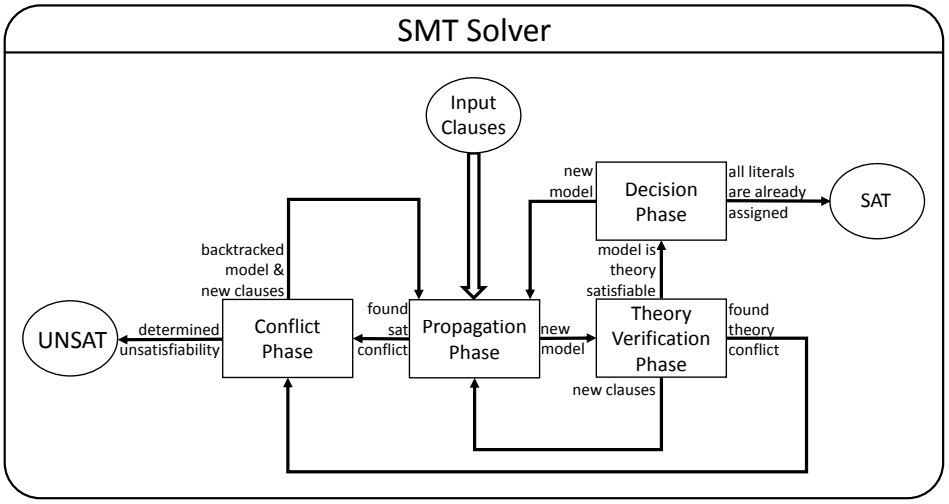


Figure 8.1: A control flow graph that describes the four phases of CDCL(LA) in SPASS-SATT.

8.1 CDCL(LA) Implementation

Our implementation of CDCL(LA) combines SPASS-IQ, our theory solver for ground and conjunctive linear arithmetic formulae, with a CDCL-based (conflict-driven-clause-learning-based) SAT solver from the SPASS Workbench [3].[2] The result is a decision procedure for ground linear arithmetic formulae in clause normal form. In this section, we will not describe our theory solver and SAT solver as separate entities.[3] Instead, we explain how our theory solver and SAT solver interact in our CDCL(LA) implementation.

8.1.1 Interaction Between SAT and Theory Solver

The CDCL(LA) implementation in SPASS-SATT is actually divided into four phases (Figure 8.1). Three of those phases—propagation, decision, and conflict— are handled mostly by the CDCL-based SAT solver with some minor interactions with the theory solver. The fourth phase—theory verification—is handled completely by the theory solver.

CDCL(LA) loops between these four phases until it determines whether the problem is satisfiable or unsatisfiable. Each iteration of the loop starts with the (unit) *propagation phase* (Figure 8.2). At the start of CDCL(LA), unit propagation gets the original clauses and an empty model as its input. In later iterations, it receives as its input an equisatisfiable set of clauses and

[2]For a general description of CDCL(T) see Chapter 2.6.

[3]We gave a stand-alone and in-depth description of the theory solver SPASS-IQ in Chapter 7 and an in-depth description of the SAT solver used by SPASS-SATT can be found in [135].

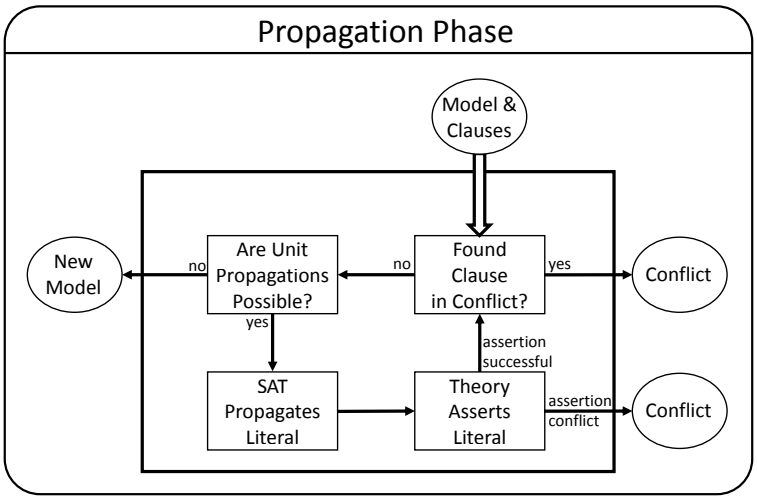


Figure 8.2: A control flow graph that describes SPASS-SATT's solver interactions during the propagation phase.

a partial model that is satisfiable on the propositional level. The propagation phase itself is also an iterative process. In every iteration of the propagation phase, the SAT solver first checks whether there is a conflict clause, i.e., whether one of the clauses is unsatisfiable under the current model.[4] If there is a conflict clause, then CDCL(LA) exits the propagation phase and handles the conflict clause in the conflict phase. Otherwise, the SAT solver checks whether a unit propagation is possible, i.e., whether there exists a clause $L \vee C$ such that L is still undecided (i.e., not assigned to a truth value) and all literals in C are assigned to *false*.[5] If there is no such clause, then CDCL(LA) exits the propagation phase and sends the current model to the theory solver for theory verification. Otherwise, the SAT solver propagates L, i.e., the SAT solver extends the model by setting L to *true*. Moreover, CDCL(LA) notifies the theory solver, so it can assert the corresponding bound to the literal L. The theory assertion is either successful and we continue at the start of the propagation phase or the theory assertion returns a conflict clause. In the latter case, CDCL(LA) exits again the propagation phase and handles the conflict clause in the conflict phase.

The *theory verification phase* takes the current (partial or full) model as its input. At the start of this phase, the theory solver checks whether its conflict backlog contains any previously found but not yet analyzed conflict explanations (sets of literals). If the backlog contains explanations, then the theory solver searches for an explanation that is still a conflict, i.e., an expla-

[4] Can be done automatically and efficiently with the two-watched literals scheme [110]

[5] Can also be done automatically and efficiently with the two-watched literals scheme [110]

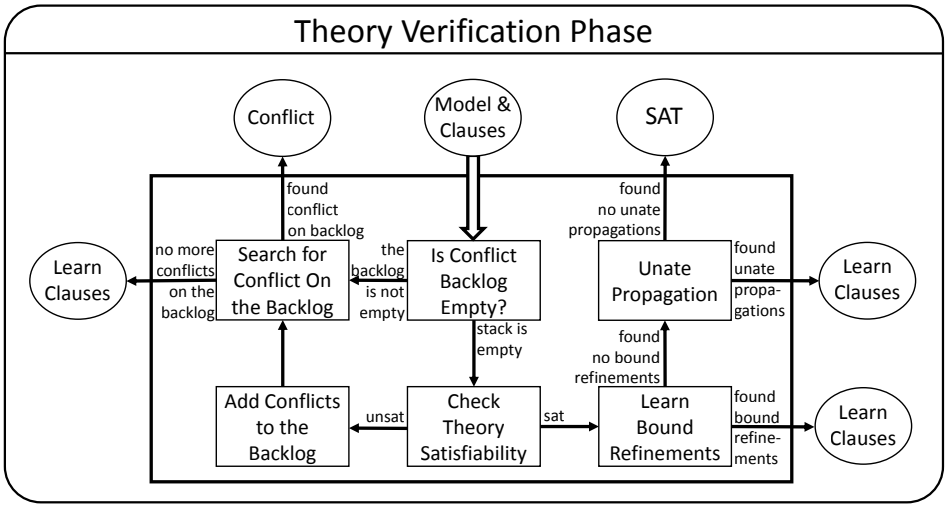


Figure 8.3: A control flow graph that describes SPASS-SATT's solver interactions during the theory verification phase.

nation with all of its literals still asserted. The explanation $\{L_1, \ldots, L_n\}$ is then removed from the backlog and turned into a conflict clause $\bar{L_1} \vee \ldots \vee \bar{L_n}$. Moreover, CDCL(LA) then exits the theory verification phase and handles the conflict clause in the conflict phase. If the backlog contains explanations $\{L_1, \ldots, L_n\}$ and all of them are no longer conflicts, then the theory solver learns all of them, i.e., turns them into clauses $\bar{L_1} \vee \ldots \vee \bar{L_n}$ and adds them to the current clause set. Moreover, CDCL(LA) then exits the theory verification phase and tries to propagate the new clauses in the propagation phase.

Otherwise, the backlog contains no explanations and the theory solver verifies the theory satisfiability of the current model with SPASS-IQ. Note, however, that SPASS-SATT verifies only for full models whether they are linear mixed/integer satisfiable. Partial models are checked only for rational satisfiability. If SPASS-IQ returns unsatisfiable, then SPASS-IQ also found at least one conflict explanation. The theory solver then adds the conflict explanations to the conflict backlog and processes the conflict backlog as described before. If SPASS-IQ returns satisfiable instead, then the theory solver tries to refine the current model if it is partial.

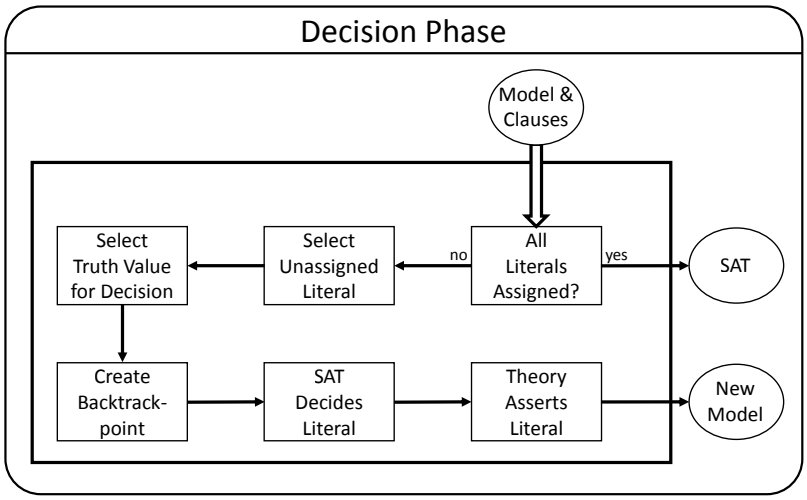


Figure 8.4: A control flow graph that describes SPASS-SATT's solver interactions during the decision phase.

For this purpose, it first tries to compute a series of bound refinements (see Chapter 2.7.2).[6] If it succeeds, then bound refinement returns a propagation explanation $\{L_1, \ldots, L_n\}$ for each refined bound, which the theory solver then learns. Moreover, CDCL(LA) then exits the theory verification phase and tries to propagate the new clauses in the propagation phase.

If bound refinement fails, then the theory solver tries to greedily perform a series of *unate propagations* [57]. A unate propagation is possible whenever the current model (i) contains a literal L corresponding to a bound $x_i \leq u$ ($x_i \geq l$), (ii) the set of clauses contains another literal L' corresponding to a bound $x_i \leq u'$ ($x_i \geq l'$) with $u < u'$ ($l > l'$), and (iii) neither L' nor $\bar{L}'$ is part of the current model. Clearly, the bound $x_i \leq u$ ($x_i \geq l$) is only satisfied if the bound $x_i \leq u'$ ($x_i \geq l'$) with $u < u'$ ($l > l'$) is also satisfied. The clause $\bar{L} \vee L'$ is, therefore, a tautology and we call it a unate propogation clause. If CDCL(LA) actually finds any unate propagations, then it learns the unate propogation clauses $\bar{L} \vee L'$, exits the theory verification phase, enters the propagation phase, and lets the SAT solver propagate the literals L'.[7] If CDCL(LA) finds no unate propagations, then it fails to refine the current model, exits the theory verification phase and enters the propagation phase.

[6] This series of bound refinements only propagates bounds that correspond to already existing literals in the current set of clauses. Moreover, it propagates at most 100 bounds for each variable.

[7] In SPASS-SATT we do not add the unate propagation clauses explicitly to the set of clauses. This is not necessary because the theory solver can efficiently recompute them on the fly when we need them. Therefore, storing them explicitly would only wast space [69].

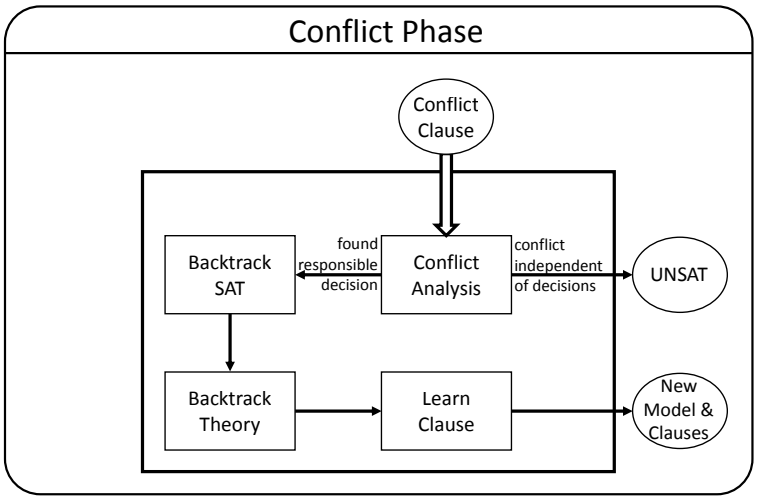


Figure 8.5: A control flow graph that describes SPASS-SATT's solver interactions during the conflict phase.

The *decision phase* takes the current clause set and the current (partial or full) model as its input. At the start of this phase, the SAT solver checks whether the current model satisfies all clauses.[8] If the current model satisfies all clauses, then CDCL(LA) has determined that the original clause set was satisfiable and it, therefore, returns satisfiable. Otherwise, at least one propositional variable appearing in the current clause set is not yet assigned a truth-value and the SAT solver selects one of them based on a heuristic. If the selected propositional variable corresponds to one of the original propositional variables, then the SAT solver uses a phase saving scheme to *decide*, whether the propositional variable should be assigned to true or false. If the selected propositional variable corresponds to a bound, then the SAT solver asks the theory solver, whether the propositional variable should be assigned to true or false. The theory solver then recommends a truth value for the propositional variable that corresponds to (i) a bound that cannot cause a conflict if asserted by the theory solver (this is always possible) and (ii) that is (if possible) satisfied by the current assignment for the arithmetic variables. We also call this technique a *decision recommendation*. Next the SAT solver and theory solver create a backtrack point and then they actually assign/assert the selected propositional variable to the selected truth value. CDCL(LA) then exits the theory verification phase and continues in the propagation phase with the new model.

[8] Can be done efficiently (i.e., constant time) with the watched literals scheme [110].

The *conflict phase* takes a conflict clause C as input as well as the current set of clauses and the current model. The SAT solver starts the conflict phase with a conflict analysis based on UIP (unique implication point) resolution [130, 139]. This conflict analysis tries to find a previous decision L (i.e., decided literal) that was responsible for the detected conflict. If there exists no such decision, then the original set of clauses is unsatisfiable and CDCL(LA) returns unsatisfiable. Otherwise, CDCL(LA) backtracks to the backtrack point before the decision, i.e., the SAT solver reduces the current model to the prefix without the decided literal and the theory solver reverts the asserted bound values and recomputes a rational solution. The conflict analysis also transforms the conflict clause into a clause $\bar{L} \vee C'$ implied by the original clause set such that all literals in C' are false under the backtracked model. The SAT solver learns this clause, i.e., adds it to the current set of clauses, after which CDCL(LA) exits the conflict phase and starts again with the propagation phase. The learned clause $\bar{L} \vee C'$ prevents that L is decided again because it must be used to propagate $\bar{L}$.

Key Changes to the General CDCL(T) Framework

There are four key points that we have changed in SPASS-SATT compared to the more general frameworks for CDCL(T) (see Chapter 2.6).[9] First of all, we rely on "weakened early pruning" [129], i.e., we only use a weaker but faster check to determine theory satisfiability for partial (propositionally abstracted) models. We do so because checking for an integer solution is too expensive and not incrementally efficient enough to be done more than once per complete (propositionally abstracted) model. As a compromise, we at least check whether the partial model has a rational solution, before we add a(nother) decision literal to the model. (Weakened early pruning is used by most SMT solvers [9, 27, 40, 41, 49, 56].)

As our second key change, we let the theory solver select the phase of the next decision literal L, i.e., whether the SAT solver will add the positive or the negated version of L to the model. We call this technique a *decision recommendation*. (The SMT solver Yices also uses decision recommendations [56].)

As our third key change, we allow our theory solver to return more than one conflict per theory check. (As mentioned in Chapter 7.1.2, CVC4's theory solver can also return multiple conflicts [9].)

Finally, we use theory reasoning to find and learn new clauses implied by the input formula. The reasoning techniques we use for this purpose are *unate propagations* and *bound refinements* as proposed in [57]. (Unate propagations and bound refinements are used by most SMT solvers [9, 27, 40, 41, 49, 56].)

[9] All of these changes appeared first in other SMT solvers.

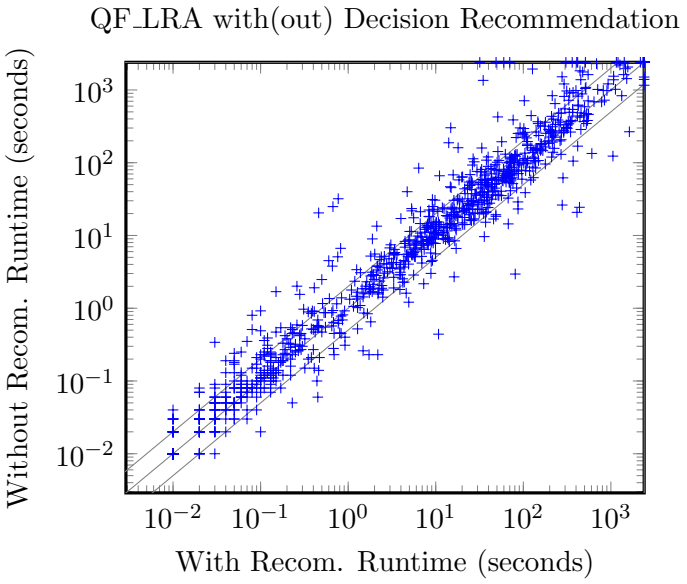


Figure 8.6: With(out) Decision Recommendation on the QF_LRA SMT-LIB benchmarks

8.1.2 CDCL(LA) Experiments

In this section, we evaluate the impact of decision recommendations, unate propagations, and bound refinements. In order to measure their impact, we performed a series of benchmark experiments with SPASS-SATT.

Decision Recommendation

In a previous part of this section, we explained *decision recommendations*, i.e., an interaction technique in our CDCL(LA) implementation that lets our theory solver decide whether the SAT solver assigns a decision literal to true or false. Figures 8.6 and 8.7 examine the impact that decision recommendations have on SPASS-SATT's performance on the QF_LRA and QF_LIA benchmarks. To this end, we compare the default version of SPASS-SATT with decision recommendations (horizontal axis; command-line option `-p 0`) and the version of SPASS-SATT without decision recommendations (vertical axis; command-line option `-p 1`). SPASS-SATT with decision recommendations is on average better in both benchmark divisions.

On the QF_LRA benchmarks, SPASS-SATT with decision recommendations can solve 11 more problems than SPASS-SATT without decision recommendations—1 more problem form the `2017-Heizmann-UltimateInvariantSynthesis` benchmark family, 10 more problems from the `Lasso-`

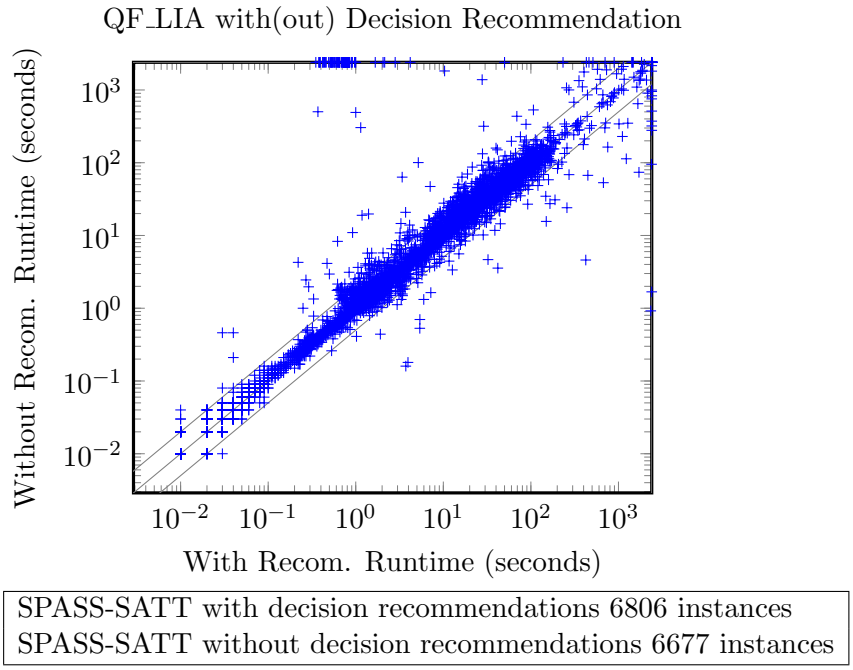


Figure 8.7: With(out) Decision Recommendation on the QF_LIA SMT-LIB benchmarks

`Ranker` benchmark family, and 2 more problems from the `miplib` benchmark family, although also 2 less problems from the `tropical-matrix` benchmark family. Moreover, SPASS-SATT with decision recommendations is more than twice as fast as SPASS-SATT without decision recommendations on 160 problems from the QF_LRA benchmarks. For comparison, SPASS-SATT without decision recommendations is more than twice as fast as SPASS-SATT with decision recommendations on only 50 problems from the QF_LRA benchmarks.

On the QF_LIA benchmarks, SPASS-SATT with decision recommendations can solve 129 more problems than SPASS-SATT without decision recommendations—1 more problem from the `bofill-scheduling` benchmark family, 116 more problems from the `convert` benchmark family, 1 more problem from the `miplib2003` benchmark family, 11 more problems form the `tropical-matrix` benchmark family, 2 more problems from the `nec_smt` benchmark family, but 1 less from both the `201803026-Bromberger` benchmark family and the `arctic-matrix` benchmark family. Moreover, SPASS-SATT with decision recommendations is more than twice as fast as SPASS-SATT without decision recommendations on 389 problems from the QF_LIA benchmarks. For comparison, SPASS-SATT without decision recommendations is more than twice as fast as SPASS-SATT with decision recommendations on only 58 problems from the QF_LIA benchmarks.

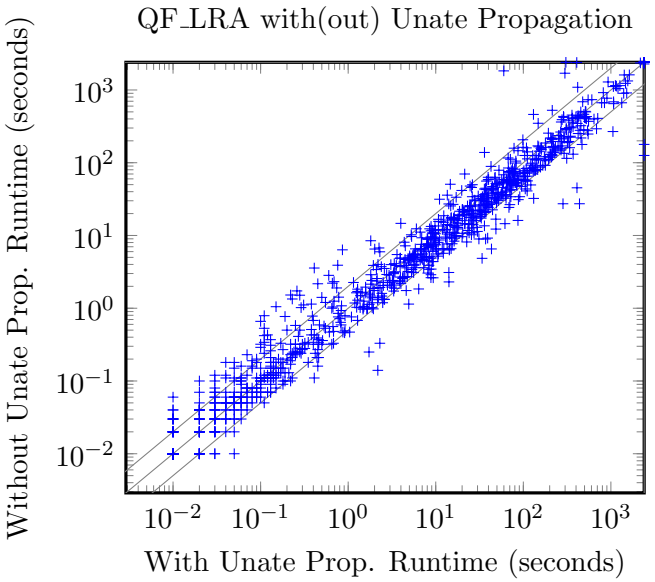
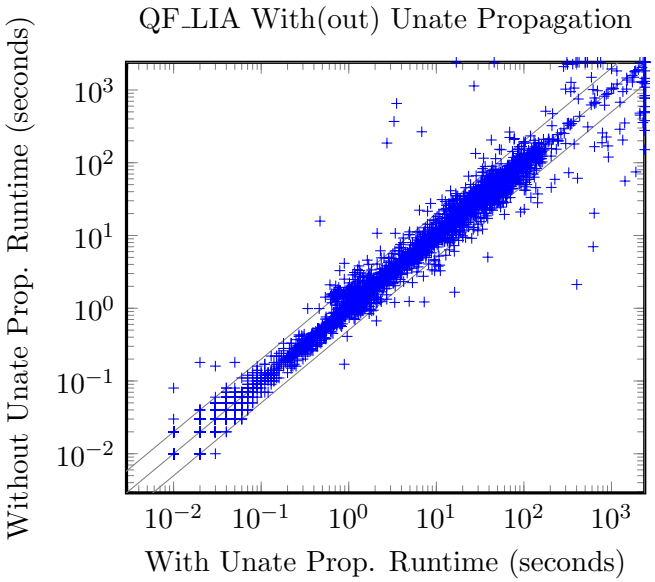


Figure 8.8: With(out) Unate Propagation on the QF_LRA SMT-LIB benchmarks

Therefore, we conclude that SPASS-SATT is a very beneficial interaction technique overall, although it is not the optimal strategy for every problem instance.

Unate Propagation

Unate propagation is another technique that we presented in a previous part of this section (for more details see also [57]). Figures 8.8 and 8.9 examine the impact that unate propagation has on SPASS-SATT's performance on the QF_LRA and QF_LIA benchmarks. To this end, we compare the default version of SPASS-SATT with unate propagations (horizontal axis; command-line option `-p 0`) and the version of SPASS-SATT without decision recommendations (vertical axis; command-line option `-p 1`). Although SPASS-SATT frequently and regularly performs unate propagations in both benchmark divisions, we are unable to observe any consistent benefit from this interaction technique. SPASS-SATT with unate propagations cannot solve more benchmark instances than SPASS-SATT without unate propagations (in fact it solves one instance less due to performance fluctuations of our cluster) and any speed-up gained on some of the benchmark instances is countered by a slow-down on other benchmark instances. We still keep unate propagations by default in SPASS-SATT because we assume that there exist some theory combinations where it is beneficial.



SPASS-SATT with unate propagation solves 6806 instances
SPASS-SATT without unate propagation solves 6806 instances

Figure 8.9: With(out) Unate Propagation on the QF_LIA SMT-LIB benchmarks

Bound Refinements

The final interaction technique that we have mentioned in this section, are bound refinements (see Chapter 2.7.2 for a formal definition). These bound refinements are preformed in addition to the bound propagations of our branch-and-bound solver (see Chapter 7.2.3). However, in contrast to the branch-and-bound bound propagations, the interaction bound refinements propagate only bounds that correspond to already existing literals in the current set of clauses. Moreover, we test more frequently for interaction bound refinements than for branch-and-bound bound propagations. In fact, we test during every theory verification phase for interaction bound refinements, but only during the branch-and-bound search for bound propagations.

The two methods have, however, in common that we put a threshold on the number of bound propagations per refinement/propagation phase. To be more precise, we limit at every branching node the number of bounds we refine/propagate per variable as part of the branch-and-bound bound propagation. Similarly, we limit during every theory verification phase the number of bounds we refine/propagate per variable as part of the interaction bound refinements. In Figure 7.26 of Chapter 7.2.5, we examined the impact

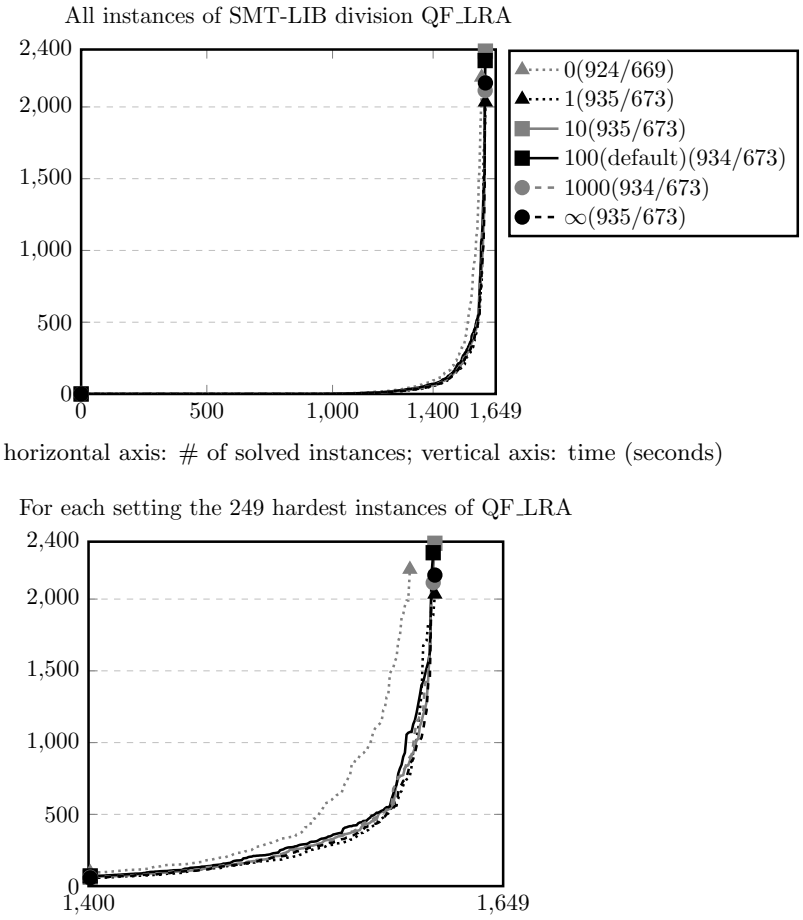


Figure 8.10: Results for different bound refinement settings on the SMT-LIB benchmark division QF_LRA

that different bound propagation thresholds have on SPASS-SATT's performance on the QF_LIA benchmarks. In Figures 8.8 and 8.9 of this chapter, we now examine the impact that bound refinements have on SPASS-SATT's performance on the QF_LRA and QF_LIA benchmarks.

In the legends of the plots, we list the different bound refinement settings, i.e., the thresholds on the maximum number of bounds we propagate per theory verification phase and variable (command-line option −`LABR <threshold>`). Moreover, the two numbers in brackets behind each threshold are the numbers of satisfiable and unsatisfiable instances that we can solve with the respective threshold. Naturally, a threshold of 0 means that SPASS-SATT performs no bound refinements and a threshold of ∞ means that SPASS-SATT performs arbitrarily many bound refinements.

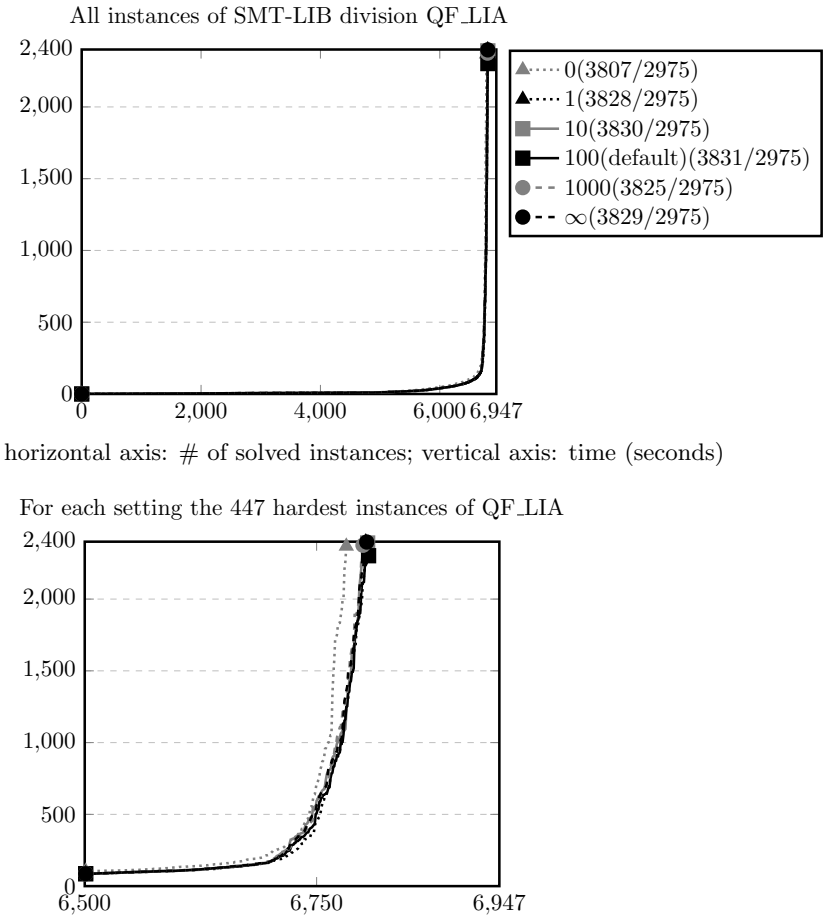


Figure 8.11: Results for different bound refinement settings on the SMT-LIB benchmark division QF_LIA

The figures show that activating bound refinements (i.e., selecting a non-zero threshold value) is beneficial and allows SPASS-SATT to solve several more problems. Moreover, the results for the different positive threshold values are too close and can be attributed to performance fluctuations of our cluster. Therefore, we assume that the actual threshold on the number of bound refinements is irrelevant as long as it is non-zero.

We also observe, that interaction bound refinements seem to have a much larger impact on the overall performance of SPASS-SATT than branch-and-bound bound propagations. For instance, we can solve 24 more problems with interaction bound refinements on the QF_LIA benchmarks, but only 8 more problems with branch-and-bound bound propagations.

8.2 Preprocessing

The satisfiability modulo theories library (SMT-LIB) [10] offers a large number of benchmarks for ground linear arithmetic formulae and is, therefore, an important resource for testing the efficiency of our decision procedures. All SMT-LIB benchmarks are written in the SMT-LIB language [12], which can construct all ground linear arithmetic formulae consisting of the standard boolean operators (and, or, not, implication, equivalence), the standard arithmetic operators (plus, minus, multiplication, division, less than, greater than, less than or equal, greater than or equal, equal), as well as some special operators (let, if-then-else). This is a problem because our CDCL(LA) implementation can only handle ground linear arithmetic formulae in clause normal form. We, therefore, have to preprocess the input problems, given to us in the SMT-LIB language, into an equisatisfiable formula in clause normal form.

The specific preprocessing steps that SPASS-SATT applies before it sends the problem to our CDCL(LA) implementation are outlined in the remainder of this section. But first, let us take a closer look at the SMT-LIB language.

8.2.1 SMT-LIB Language

A linear arithmetic *input problem* constructed with the SMT-LIB language consists of four sets: a *set of propositional variables* P, a *set of integer variables* Z, a *set of rational variables* Q, and a *set of formulae* F over those variables (which is semantically interpreted as a conjunction of formulae). The three sets of variables are disjoint and consist of identifiers $\langle identifier \rangle$. The set of formulae consists of formulae $\langle formula \rangle$ constructed according to the following subset of the SMT-LIB language:

$$
\begin{array}{rl}
\langle formula \rangle ::= & (\text{and } \langle formula \rangle^*) \mid (\text{or } \langle formula \rangle^*) \mid \\
& (\text{not } \langle formula \rangle) \mid (=> \ \langle formula \rangle \ \langle formula \rangle) \mid \\
& (= \ \langle formula \rangle \ \langle formula \rangle^+) \mid \\
& (\text{ite } \langle formula \rangle \ \langle formula \rangle \ \langle formula \rangle) \mid \\
& (\text{let } \langle let\_defs \rangle \ \langle formula \rangle) \mid \langle let\_var \rangle \mid \\
& \langle prop\_var \rangle \mid \text{true} \mid \text{false} \mid \\
& (<= \ \langle la\_term \rangle \ \langle la\_term \rangle) \mid (>= \ \langle la\_term \rangle \ \langle la\_term \rangle) \mid \\
& (< \ \langle la\_term \rangle \ \langle la\_term \rangle) \mid (> \ \langle la\_term \rangle \ \langle la\_term \rangle) \mid \\
& (= \ \langle la\_term \rangle \ \langle la\_term \rangle) \\
\langle la\_term \rangle ::= & (+ \ \langle la\_term \rangle \ \langle la\_term \rangle^+) \mid \\
& (- \ \langle la\_term \rangle \ \langle la\_term \rangle) \mid \\
& (- \ \langle la\_term \rangle) \mid \\
& (* \ \langle la\_term \rangle \ \langle la\_term \rangle^+) \mid \\
& (/ \ \langle la\_term \rangle \ \langle la\_term \rangle) \mid \\
& \langle number \rangle \mid \langle let\_var \rangle \mid \langle la\_var \rangle \mid
\end{array}
$$

$$
\begin{array}{rl}
 & (\text{ite } \langle formula \rangle \ \langle la\_term \rangle \ \langle la\_term \rangle) \mid \\
 & (\text{let } \langle let\_defs \rangle \ \langle la\_term \rangle) \\
\langle let\_defs \rangle ::= & (\langle let\_def \rangle^+) \\
\langle let\_def \rangle ::= & (\langle let\_var \rangle \ \langle formula \rangle) \mid (\langle let\_var \rangle \ \langle la\_term \rangle) \\
\langle prop\_var \rangle ::= & \langle identifier \rangle \\
\langle let\_var \rangle ::= & \langle identifier \rangle \\
\langle la\_var \rangle ::= & \langle identifier \rangle \\
\langle number \rangle ::= & \langle integer \rangle \mid \langle rational \rangle \\
\langle integer \rangle ::= & \langle digit \rangle \mid \langle nzdigit \rangle \langle digit \rangle^+ \mid (- \ \langle nzdigit \rangle \langle digit \rangle^+) \\
\langle rational \rangle ::= & \langle digit \rangle . \langle digit \rangle^+ \mid \\
 & \langle nzdigit \rangle \langle digit \rangle^+ . \langle digit \rangle^+ \mid \\
 & (- \ \langle digit \rangle . \langle digit \rangle^+) \mid \\
 & (- \ \langle nzdigit \rangle \langle digit \rangle^+ . \langle digit \rangle^+) \\
\langle digit \rangle ::= & 0 \mid \langle nzdigit \rangle \\
\langle nzdigit \rangle ::= & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle identifier \rangle ::= & \langle start\_char \rangle \langle char \rangle^* \\
\langle letter \rangle ::= & \text{A} \mid \text{B} \mid \text{C} \mid \text{D} \mid \text{E} \mid \text{F} \mid \text{G} \mid \text{H} \mid \text{I} \mid \text{J} \mid \text{K} \mid \text{L} \mid \text{M} \mid \\
 & \text{N} \mid \text{O} \mid \text{P} \mid \text{Q} \mid \text{R} \mid \text{S} \mid \text{T} \mid \text{U} \mid \text{V} \mid \text{W} \mid \text{X} \mid \text{Y} \mid \text{Z} \mid \\
 & \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \mid \text{e} \mid \text{f} \mid \text{g} \mid \text{h} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{l} \mid \text{m} \mid \\
 & \text{n} \mid \text{o} \mid \text{p} \mid \text{q} \mid \text{r} \mid \text{s} \mid \text{t} \mid \text{u} \mid \text{v} \mid \text{w} \mid \text{x} \mid \text{y} \mid \text{z} \mid \\
\langle start\_char \rangle ::= & \langle letter \rangle \mid \tilde{} \mid ! \mid @ \mid \$ \mid \% \mid \hat{} \mid \& \mid * \mid \_ \mid - \mid + \mid = \mid \\
 & < \mid > \mid . \mid ? \mid / \\
\langle char \rangle ::= & \langle start\_char \rangle \mid \langle digit \rangle
\end{array}
$$

The operators in the SMT-LIB language are all prefix operators. The *standard boolean operators* in the ground linear arithmetic subset are:

- a *conjunction operator* (and $t_1 \ldots t_n$) with arbitrarily many operands, which is semantically interpreted as $t_1 \wedge \ldots \wedge t_n$;
- a *disjunction operator* (or $t_1 \ldots t_n$) with arbitrarily many operands, which is semantically interpreted as $t_1 \vee \ldots \vee t_n$;
- a *boolean negation operator* (not t_1), which is semantically interpreted as $\neg t_1$;
- an *implication operator* ($=> \ t_1 \ t_2$), which is semantically interpreted as $t_1 \rightarrow t_2$;
- an *equivalence operator* ($= \ t_1 \ldots t_n$), which is semantically interpreted as $(t_1 \equiv t_2) \wedge \ldots \wedge (t_{n-1} \equiv t_n)$.

The *standard arithmetic operators* in the ground linear arithmetic subset are:

- a *summation operator*($+ \ t_1 \ldots t_n$) with arbitrarily many operands, which is semantically interpreted as $t_1 + \ldots + t_n$;
- a *multiplication operator* ($* \ t_1 \ldots t_n$) with arbitrarily many operands, which is semantically interpreted as $t_1 \cdot \ldots \cdot t_n$ and which has at most one subterm that is a variable;

- a *subtraction operator* $(-\ t_1\ t_2)$, which is semantically interpreted as $t_1 - t_2$;
- an *arithmetic negation operator* $(-\ t_1)$, which is semantically interpreted as $-t_1$;
- a *division operator* $(/\ t_1\ t_2)$, which is semantically interpreted as t_1/t_2 with t_2 containing no subterms that are variables.

The *arithmetic comparison operators* in the ground linear arithmetic subset are:

- an *equality operator* $(=\ t_1\ t_2)$, which is semantically interpreted as $t_1 = t_2$;
- a *greater than operator* $(>\ t_1\ t_2)$, which is semantically interpreted as $t_1 > t_2$;
- a *greater than or equal operator* $(>=\ t_1\ t_2)$, which is semantically interpreted as $t_1 \geq t_2$;
- a *less than operator* $(<\ t_1\ t_2)$, which is semantically interpreted as $t_1 < t_2$;
- a *less than or equal operator* $(<=\ t_1\ t_2)$, which is semantically interpreted as $t_1 \leq t_2$.

The ground linear arithmetic subset of the SMT-LIB language also contains two special operators. The first special operator is the *if-then-else operator* (ite $t_1\ t_2\ t_3$), which is semantically interpreted as the function that returns t_2 if the formula t_1 is true and returns t_3 if the formula t_1 is false. We also assign to the arguments of the if-then-else operator special names so we can better differentiate them. The first argument t_1 is the *condition*, the second argument t_2 is the *consequence*, and the third argument t_3 is the *alternative*. Moreover, we call (ite $t_1\ t_2\ t_3$) a *formula if-then-else* expression if its consequence t_2 and alternative t_3 are formulae $\langle$*formula*$\rangle$ and *arithmetic if-then-else* expression if its consequence t_2 and alternative t_3 are arithmetic terms $\langle$*la_term*$\rangle$.

The second special operator is the *let operator* (let $((x_1\ t_1)\ldots(x_n\ t_n))t$). It is actually the most complicated operator that the linear arithmetic subset of the SMT-LIB language contains. It is so complicated because it assigns terms to, potentially new, local variables $x_1, \ldots, x_n$ called *let variables* L. It is semantically interpreted as $t\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, i.e., the term t except that all occurrences of $x_1, \ldots, x_n$ in t are substituted by $t_1, \ldots, t_n$, respectively. The only exception happens, when a different let operator in t overwrites the replacement term for one of the variables. For instance, (let $((x_1\ t_1)(x_2\ t_2))$(let $((x_2\ t_3))t_4)$) is interpreted as $t_4\{x_1 \mapsto t_1, x_2 \mapsto t_3\{x_1 \mapsto t_1, x_2 \mapsto t_2\}\}$, i.e., all occurrences of x_1 and x_2 are replaced in t_4 by t_1 and t_3', respectively, where t_3' is equivalent to t_3 except that all occurrences of x_1 and x_2 are replaced by t_1 and t_2. This means that let operators substitute bottom-up, i.e., (let $((x_1\ t_1)\ldots(x_n\ t_n))t$) is interpreted as $t'\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where t' is equivalent to t except that all let operators in s are already substituted away.

To avoid confusion, let operator can only overwrite other let variables and not operators or propositional, integer, or rational variables. In general, propositional variables, integer variables, rational variables, and let variables cannot be named after operators and all four sets of variables are disjoint.

As mentioned before, SPASS-SATT is given an input problem defined by the above language and it needs to transform it into an equisatisfiable formula in clause normal form. But first of all, this means that SPASS-SATT has to represent these formulae and their subterms internally. In SPASS-SATT we differentiate between four types of terms: constant truth values, variables, numbers, and complex terms. A *constant truth value* term is either *true* or *false* and we can represent them internally as an enumeration structure with two values. Variables are in the input language string identifiers, but we can simplify the identifiers internally to distinct numeral ids.[10] When we are specifying an abstract variable, we simply write p_i, i.e., the letter p with and index i, for a *propositional variable* $\langle prop\_var \rangle$, x_i, i.e., the letter x with and index i, for an *arithmetic variable* $\langle la\_var \rangle$, l_i, i.e., the letter l with and index i, for a *let variable* $\langle let\_var \rangle$, and y_i, i.e., the letter y with and index i, for an arbitrary variable. *Numbers* are also represented by strings in the input language, but internally we use the data types `Integer` and `Rational` to represent $\langle integer \rangle$ and $\langle rational \rangle$ numbers, respectively. When we are specifying an abstract number, we simply write a_i, i.e., the letter a with and index i. *Complex terms* make up all other terms. They consist of two things: an *operator* (e.g., and, or, +, <=, represented through distinct numeral ids) and an array of *arguments*, which are pointers to other terms.[11] When we are specifying an abstract *complex term*, then we simply write it in prefix form $(o\ t_1 \ldots t_n)$, as we did in the input language. Here, o is the operator and t_i are the abstract term arguments.

[10] Note that we use two different ids if two variables are defined by two different let operations. This means our internal representation does not have to handle overwritten variables.

[11] We also implement the other types of terms internally through a pair consisting of an operator and an argument array. To do so, we create operators for the variables, the truth values, and one operator each for rational and integer numbers. The variables, the truth values have an empty argument array. The number operators store the numerical value as their first argument.

8.2.2 Term Sharing

When we first parse our problem from the input language into internal terms, the resulting set of terms is not yet shared, i.e., two syntactically equivalent subterms are still represented by two different internal terms. We do, however, start to share our internal (sub)terms on the formula level as part of our let operator elimination. The *shared term* representation has the additional benefit that it saves memory; in some cases even an exponential amount.[12]

We start our *let operator elimination* and sharing procedure by pushing negations top-down with the following set of equivalence preserving transformations:

(not (not t))	$\mapsto$	t
(not false)	$\mapsto$	true
(not true)	$\mapsto$	false
(not (and $t_1 \dots t_n$))	$\mapsto$	(or (not t_1) $\dots$ (not t_n))
(not (or $t_1 \dots t_n$))	$\mapsto$	(and (not t_1) $\dots$ (not t_n))
(=> t_1 t_2)	$\mapsto$	(or (not t_1) t_2)
(not (= t_1 t_2))	$\mapsto$	(= t_1 (not t_2))
(not (ite t_1 t_2 t_3))	$\mapsto$	(ite t_1 (not t_2) (not t_3))
(ite (not t_1) t_2 t_3)	$\mapsto$	(ite t_1 t_3 t_2)
(ite t_1 false true)	$\mapsto$	(not t_1)
(not (let ((l_1 t_1) $\dots$ (l_n t_n))t))	$\mapsto$	(let ((l_1 t_1) $\dots$ (l_n t_n))(not t))

As a result, the negation operator appears only on top of propositional variables, let variables, and arithmetic comparators (i.e., <=, >=, <, >, =). From now on, we call (not y_i) a *negative occurrence* of the variable y_i and any occurrence of y_i that is not below a negation a *positive occurrence*.

Next we traverse the sub-terms in all formulae f from our set of formulae F bottom-up. For each let operator $t' :=$ (let ((l_1 t_1) $\dots$ (l_n t_n))t) that we find in f, (i) we replace all t' by t in f, (ii) we create terms (not t_i) for all t_i if t_i is a $\langle$*formula*$\rangle$, (iii) we push the negations top-down in the new terms (not t_i) so we get equivalent terms t_i', (iv) we transform the unshared terms t_i and t_i' into shared terms s_i and s_i' (will be explained in more detail later), and (v) we link l_i with its positive shared replacement s_i and its negative shared replacement s_i' in a map M. Finally, we also transform the unshared formulae f from our set of formulae F into shared formulae.

Transforming an unshared term t into a shared term works as follows: First we need three maps. The first map M, was discussed previously, and contains the shared replacement terms s_i, s_i' for all let variables l_i occurring in t. M already contains these replacements because we first traversed and shared the let operations bottom up. The second map V maps all truth

[12] Most SMT solvers use similar shared term representations [9, 27, 40, 49, 56].

values, as well as all propositional and arithmetic variable ids to shared term versions. This map and the contained shared terms are constructed after we push-down the negations. The third map S maps all already shared terms s to their *superterm set*, i.e., a set that contains all shared terms s' that have s as an argument. This map will be filled and updated as part of the sharing transformation itself.

Now with these three maps in hand, we traverse the subterms t' of t bottom-up and replace them by shared versions. If $t' := l_i$ is a positive occurrence of a let variable term l_i, then we get its positive shared replacement s_i through the hash map M. If $t' := (\text{not } l_i)$ is a negative occurrence of a let variable term l_i, then we get its negative shared replacement s'_i through the hash map M. If $t' := y_i$ is an arithmetic or propositional variable, then we lookup the shared replacement term from V. If t' is a truth value, then we also lookup the shared replacement term from V. If $t' := (o\ s_1 \dots s_n)$ is a complex term, then we look through the superterm set of s_1 and check whether there already exists a complex but shared version of the term $(o\ s_1 \dots s_n)$, i.e., a shared term s' with the same operator o and argument pointers $s_1 \dots s_n$, which can be used as the shared replacement term t'. If there exists no such shared term s', then t' itself becomes our shared replacement term, we add it to the superterm sets of $s_1 \dots s_n$, and we map t' in S to a new and empty set.

We continue our traversal in this way, until the original term t is also a shared version of itself. Our transformed formula needs still the same order of size as the original formula, although it has no more let operations, because we combined the let-elimination with the sharing process.

An Alternative Let-Elimination

The above described standard let-elimination causes sometimes an exponential blow-up in the formula size. For the moment, our implementation is not hindered by this blow-up because there are still only linearly many different subterms, which can be represented in a linear amount of memory by our shared term representation. The blow-up will, however, impact the implementation when it has to perform the CNF transformation and the CDCL(LA) algorithm.

We could actually prevent the blow-up of the formula size with an *alternative let-elimination*. For this alternative let-elimination, we traverse the sub-terms in all formulae f from our set of formulae F bottom-up. For each let operator $t' := (\text{let } ((l_1\ t_1) \dots (l_n\ t_n))t)$ that we find in f, (i) we turn the let variables l_i corresponding to formula terms t_i into propositional variables, (ii) we turn the let variables l_i corresponding to arithmetic terms t_i into arithmetic variables, (iii) we replace all t' by t in f, (iv) we extend our formula set to $F := F \cup \{(=\ l_1\ t_1), \dots, (=\ l_1\ t_1)\}$. Finally, we transform the formulae f from our set of formulae F into shared formulae.

The above described alternative let-elimination would prevent the potential exponential blow-up in the formula size. It would, however, also introduce new variables and destroy the structure of the formula. As a result, the alternative let-elimination typically makes problems much harder for the CDCL(LA) implementation than the standard let-elimination when the standard let-elimination does not cause an exponential blow-up. And even if the standard let-elimination causes an exponential blow-up, then it is still debatable whether the ruined formula structure of the alternative let-elimination is not the worse outcome.

It might be the worse outcome because we typically reduce the blow-up caused by the standard let-elimination to a manageable size through some additional preprocessing steps. We start this process with some basic simplifications and an arithmetic flattening procedure, continue it with the elimination of the if-then-else operations, and end it with the CNF transformation.

Basic Simplifications

During our sharing process, we also apply the following series of equivalence preserving transformations to all subterms before we replace them with a shared version:[13]

Reduce to truth value:		
(not false)	$\mapsto$	true
(not true)	$\mapsto$	false
(and)	$\mapsto$	true
(and $t_1 \dots t_k$ false $t'_1 \dots t'_m$)	$\mapsto$	false
(or)	$\mapsto$	false
(or $t_1 \dots t_k$ true $t'_1 \dots t'_m$)	$\mapsto$	true
(ite t_1 t_1 t_2)	$\mapsto$	(ite t_1 true t_2)
(ite t_1 (not t_1) t_2)	$\mapsto$	(ite t_1 false t_2)
(ite t_1 t_2 (not t_1))	$\mapsto$	(ite t_1 t_2 true)
(ite t_1 t_2 t_1)	$\mapsto$	(ite t_1 t_2 false)
Reduce to subterm:		
(not (not t))	$\mapsto$	t
(and $t \dots t$)	$\mapsto$	t
(or $t \dots t$)	$\mapsto$	t
(ite true t_1 t_2)	$\mapsto$	t_1
(ite false t_1 t_2)	$\mapsto$	t_2
(ite t_1 true false)	$\mapsto$	t_1
(ite t_1 t_2 t_2)	$\mapsto$	t_2
(ite (not t_1) t_2 t_3)	$\mapsto$	(ite t_1 t_3 t_2)
Remove arguments:		

[13] Most SMT solvers use similar simplification techniques [9, 27, 40, 49, 56].

$$
\begin{array}{lcl}
(\text{and } t_1 \ldots t_k \text{ true } t'_1 \ldots t'_m) & \mapsto & (\text{and } t_1 \ldots t_k \; t'_1 \ldots t'_m) \\
(\text{or } t_1 \ldots t_k \text{ false } t'_1 \ldots t'_m) & \mapsto & (\text{or } t_1 \ldots t_k \; t'_1 \ldots t'_m) \\
\textbf{Apply associativity:} & & \\
(\text{and } t_1 \ldots t_k \; (\text{and } t'_1 \ldots t'_m) \; \hat{t_1} \ldots \hat{t_n}) & \mapsto & (\text{and } t_1 \ldots t_k \; t'_1 \ldots t'_m \; \hat{t_1} \ldots \hat{t_n}) \\
(\text{or } t_1 \ldots t_k \; (\text{or } t'_1 \ldots t'_m) \; \hat{t_1} \ldots \hat{t_n}) & \mapsto & (\text{or } t_1 \ldots t_k \; t'_1 \ldots t'_m \; \hat{t_1} \ldots \hat{t_n}) \\
\textbf{Replace operators:} & & \\
(\text{ite } t_1 \text{ false true}) & \mapsto & (\text{not } t_1) \\
(< \; t_1 \; t_2) & \mapsto & (\text{not } (>= \; t_1 \; t_2)) \\
(> \; t_1 \; t_2) & \mapsto & (\text{not } (<= \; t_1 \; t_2))
\end{array}
$$

The above simplifications are all rather trivial because they only require a constant number of operator and pointer comparisons.[14] The simplifications do, however, reduce the overall formula size drastically; especially, when our let-elimination led to bad encodings. The effects of these basic simplifications can be divided into five categories: (i) they reduce a complex term to a truth value, (ii) they reduce a complex term to one of its arguments, (iii) they reduce the number of arguments of a complex term, (iv) they apply associativity to a complex term, or (v) they replace a complex operator (ite, $<$, or $>$) with a less complex combination of operators.

At the end of our sharing transformation, we also apply another basic simplification to our whole set of formulae F. We simply split every top-level conjunction in F, i.e., $f \in F \mapsto F \cup \{t_1, \ldots, t_m\} \setminus f$ if $f := (\text{and } t_1 \ldots t_m)$. This transformation is equivalence preserving because the set F is interpreted as a conjunction of the formulae in the set.

Although the effects of these basic simplifications may seem trivial, they are also very beneficial for the other simplifications that we discuss in this section. For instance, they reduce the number of cases that we have to consider for the other simplifications. And since they are so cheap and beneficial, we also apply them for each new shared subterm constructed by our other simplifications.

Arithmetic Flattening

We also use the opportunity of the term sharing conversion to *flatten* our arithmetic terms into a more uniform structure.[15] We do so by applying the following equivalence preserving transformations to all processed arithmetic subterms:

$$
\begin{array}{lcl}
(<= \; a_1 \; t_2) & \mapsto & (<= \; (+ \; a_1) \; t_2) \\
(<= \; t_1 \; a_2) & \mapsto & (<= \; t_1 \; (+ \; a_2)) \\
(<= \; y_1 \; t_2) & \mapsto & (<= \; (+ \; (* \; 1 \; y_1)) \; t_2)
\end{array}
$$

[14] These simplifications require only constant time because the arguments of our subterms are shared terms, i.e., two equivalent arguments point to the same subterm.

[15] Many SMT solvers flatten their arithmetic terms in a similar fashion, e.g., Yices [56].

$$
\begin{array}{lcl}
(<= \ t_1 \ y_2) & \mapsto & (<= \ t_1 \ (+ \ (* \ 1 \ y_2))) \\
(<= \ (* \ q') \ t_1) & \mapsto & (<= \ (+ \ (* \ q')) \ t_1) \\
(<= \ t_1 \ (* \ q')) & \mapsto & (<= \ t_1 \ (+ \ (* \ q'))) \\
(<= \ (\text{ite} \ t_1 \ t_2 \ t_3) \ t_4) & \mapsto & (<= \ (+ \ (* \ 1 \ (\text{ite} \ t_1 \ t_2 \ t_3))) \ t_4) \\
(<= \ t_1 \ (\text{ite} \ t_2 \ t_3 \ t_4)) & \mapsto & (<= \ t_1 \ (+ \ (* \ 1 \ (\text{ite} \ t_2 \ t_3 \ t_4)))) \\
(<= \ (+ \ a_1) \ (+ \ a_2)) & \mapsto & (a_1 \leq a_2) \\
(>= \ a_1 \ t_2) & \mapsto & (>= \ (+ \ a_1) \ t_2) \\
(>= \ t_1 \ a_2) & \mapsto & (>= \ t_1 \ (+ \ a_2)) \\
(>= \ y_1 \ t_2) & \mapsto & (>= \ (+ \ (* \ 1 \ y_1)) \ t_2) \\
(>= \ t_1 \ y_2) & \mapsto & (>= \ t_1 \ (+ \ (* \ 1 \ y_2))) \\
(>= \ (* \ q') \ t_1) & \mapsto & (>= \ (+ \ (* \ q')) \ t_1) \\
(>= \ t_1 \ (* \ q')) & \mapsto & (>= \ t_1 \ (+ \ (* \ q'))) \\
(>= \ (\text{ite} \ t_1 \ t_2 \ t_3) \ t_4) & \mapsto & (>= \ (+ \ (* \ 1 \ (\text{ite} \ t_1 \ t_2 \ t_3))) \ t_4) \\
(>= \ t_1 \ (\text{ite} \ t_2 \ t_3 \ t_4)) & \mapsto & (>= \ t_1 \ (+ \ (* \ 1 \ (\text{ite} \ t_2 \ t_3 \ t_4)))) \\
(>= \ (+ \ a_1) \ (+ \ a_2)) & \mapsto & (a_1 \geq a_2) \\
(= \ a_1 \ t_2) & \mapsto & (= \ (+ \ a_1) \ t_2) \\
(= \ t_1 \ a_2) & \mapsto & (= \ t_1 \ (+ \ a_2)) \\
(= \ y_1 \ t_2) & \mapsto & (= \ (+ \ (* \ 1 \ y_1)) \ t_2) \\
(= \ t_1 \ y_2) & \mapsto & (= \ t_1 \ (+ \ (* \ 1 \ y_2))) \\
(= \ (* \ q') \ t_1) & \mapsto & (= \ (+ \ (* \ q')) \ t_1) \\
(= \ t_1 \ (* \ q')) & \mapsto & (= \ t_1 \ (+ \ (* \ q'))) \\
(= \ (\text{ite} \ t_1 \ t_2 \ t_3) \ t_4) & \mapsto & (= \ (+ \ (* \ 1 \ (\text{ite} \ t_1 \ t_2 \ t_3))) \ t_4) \\
(= \ t_1 \ (\text{ite} \ t_2 \ t_3 \ t_4)) & \mapsto & (= \ t_1 \ (+ \ (* \ 1 \ (\text{ite} \ t_2 \ t_3 \ t_4)))) \\
(= \ (+ \ a_1) \ (+ \ a_2)) & \mapsto & (a_1 = a_2) \\
(\text{ite} \ t_1 \ a_2 \ t_3) & \mapsto & (\text{ite} \ t_1 \ (+ \ a_2) \ t_3) \\
(\text{ite} \ t_1 \ t_2 \ a_3) & \mapsto & (\text{ite} \ t_1 \ t_2 \ (+ \ a_3)) \\
(\text{ite} \ t_1 \ y_2 \ t_3) & \mapsto & (\text{ite} \ t_1 \ (+ \ (* \ 1 \ y_2)) \ t_3) \\
(\text{ite} \ t_1 \ t_2 \ y_3) & \mapsto & (\text{ite} \ t_1 \ t_2 \ (+ \ (* \ 1 \ y_3))) \\
(\text{ite} \ t_1 \ (* \ q') \ t_3) & \mapsto & (\text{ite} \ t_1 \ (+ \ (* \ q')) \ t_3) \\
(\text{ite} \ t_1 \ t_2 \ (* \ q')) & \mapsto & (\text{ite} \ t_1 \ t_2 \ (+ \ (* \ q'))) \\
(\text{ite} \ t_1 \ (\text{ite} \ t_2 \ t_3 \ t_4) \ t_5) & \mapsto & (\text{ite} \ t_1 \ (+ \ (* \ 1 \ (\text{ite} \ t_2 \ t_3 \ t_4))) \ t_5) \\
(\text{ite} \ t_1 \ t_2 \ (\text{ite} \ t_3 \ t_4 \ t_5)) & \mapsto & (\text{ite} \ t_1 \ t_2 \ (+ \ (* \ 1 \ (\text{ite} \ t_3 \ t_4 \ t_5)))) \\
(* \ y_1) & \mapsto & (* \ 1 \ y_1) \\
(* \ y_1 \ a_1) & \mapsto & (* \ a_1 \ y_1) \\
(* \ (\text{ite} \ t_1 \ t_2 \ t_3)) & \mapsto & (* \ 1 \ (\text{ite} \ t_1 \ t_2 \ t_3)) \\
(* \ (\text{ite} \ t_1 \ t_2 \ t_3) \ a_1) & \mapsto & (* \ a_1 \ (\text{ite} \ t_1 \ t_2 \ t_3)) \\
(* \ q \ a_1 \ q' \ a_2 \ \hat{q}) & \mapsto & (* \ q \ (a_1 \cdot a_2) \ q' \ \hat{q}) \\
(* \ q' \ (+ \ t_1) \ \hat{q}) & \mapsto & (* \ q' \ t_1 \ \hat{q}) \\
(* \ q \ (* \ q') \ \hat{q}) & \mapsto & (* \ q \ q' \ \hat{q}) \\
(/ \ t_1 \ (+ \ a_2)) & \mapsto & (* \ (1/a_2) \ t_1) \\
(/ \ t_1 \ a_2) & \mapsto & (* \ (1/a_2) \ t_1) \\
(- \ t_1 \ t_2) & \mapsto & (+ \ t_1 \ (* \ (-1) \ t_2)) \\
(+ \ q \ y_1 \ q') & \mapsto & (+ \ q \ (* \ 1 \ y_1) \ q') \\
(+ \ q' \ (\text{ite} \ t_1 \ t_2 \ t_3) \ \hat{q}) & \mapsto & (+ \ q' \ (* \ 1 \ (\text{ite} \ t_1 \ t_2 \ t_3)) \ \hat{q})
\end{array}
$$

$$
\begin{array}{lcl}
(+\ q\ (+\ q')\ \hat{q}) & \mapsto & (+\ q\ q'\ \hat{q}) \\
(+\ q\ (*\ a_1\ t_{k+1})\ q'\ (*\ a_2\ t_{k+1})\ \hat{q}) & \mapsto & (+\ q\ (*\ (a_1 + a_2)\ t_{k+1})\ q'\ \hat{q}) \\
(+\ q\ a_1\ q'\ a_2\ \hat{q}) & \mapsto & (+\ (a_1 + a_2)\ q\ q'\ \hat{q})
\end{array}
$$

where q, q', $\hat{q}$ abbreviate sequences of term arguments, i.e., $q := t_1 \ldots t_k$, $q' := t'_1 \ldots t'_m$, and $\hat{q} := \hat{t_1} \ldots \hat{t_n}$, and infix operations are immediately evaluated, e.g., (a_1/a_2) is evaluated (and replaced) by the actual result of a_1/a_2. As a result, our terms are now all constructed according to the following language:

$$
\begin{array}{ll}
\langle formula \rangle ::= & (\text{and } \langle formula \rangle^*) \mid (\text{or } \langle formula \rangle^*) \mid \\
& (\text{not } \langle formula \rangle) \mid (=> \ \langle formula \rangle\ \langle formula \rangle) \mid \\
& (= \ \langle formula \rangle\ \langle formula \rangle^+) \mid \\
& (\text{ite } \langle formula \rangle\ \langle formula \rangle\ \langle formula \rangle) \mid \\
& \langle prop\_var \rangle \mid \text{true} \mid \text{false} \mid \\
& (<= \ \langle la\_sum \rangle\ \langle la\_sum \rangle) \mid (>= \ \langle la\_sum \rangle\ \langle la\_sum \rangle) \mid \\
& (= \ \langle la\_sum \rangle\ \langle la\_sum \rangle) \\
\langle la\_sum \rangle ::= & (+\ \langle la\_monom \rangle^+) \\
\langle la\_monom \rangle ::= & (*\ \langle number \rangle\ \langle la\_var \rangle) \mid \\
& (*\ \langle number \rangle\ \langle la\_ite \rangle) \mid \\
& \langle number \rangle \\
\langle la\_ite \rangle ::= & (\text{ite } \langle formula \rangle\ \langle la\_sum \rangle\ \langle la\_sum \rangle) \\
\langle la\_var \rangle ::= & \langle int\_var \rangle \mid \langle rat\_var \rangle \\
\langle number \rangle ::= & \langle integer \rangle \mid \langle rational \rangle
\end{array}
$$

This means that we now only have one case for our arithmetic terms: they always are linear arithmetic sums, i.e., sums of linear arithmetic monomials, where each of the linear arithmetic monomials is either a constant monomial, i.e., (i) a constant number; (ii) a variable monomial, i.e., a constant number (coefficient) multiplied with a variable; or (iii) a complex monomial, i.e., a constant number (coefficient) multiplied with an if-then-else operation (complex term). Moreover, the linear arithmetic sums fulfill the following four conditions: (i) a linear arithmetic sum may only contain one constant monomial, (ii) if the sum contains a constant monomial, then it is the first argument of the sum, (ii) a linear arithmetic sum may only contain one variable monomial with the same variable, and (iii) a linear arithmetic sum may only contain one complex monomial with the same complex term. This has the effect that all linear arithmetic sums contain at most one constant monomial plus one monomial for every arithmetic variable plus one monomial for every arithmetic if-then-else expression. This again reduces the overall formula size drastically; especially, when our let-elimination led to bad encodings.

Moreover, the above conditions allow us to treat any linear arithmetic sum as a map that maps each arithmetic variable and complex term to their coefficient value (if they actually appear in a monomial) or to zero. This is also one of the reason why we represent these sums internally as ordered tree maps with shared terms (variables or complex terms) as keys. The other reason is that this representation allows us to efficiently perform the arithmetic flattening. For instance, we can add/subtract two sums in linear time, add/subtract a single monomial to/from a sum in logarithmic time, and multiply a sum with a constant factor in linear time.[16]

Another added benefit of the arithmetic flattening is that it reduces the number of cases that we have to consider for our remaining preprocessing steps. For instance, we assign now with only five rules an arithmetic type to all arithmetic terms:

$$\begin{aligned}
x_i &: \texttt{Integer} && \text{if } x_i \in \text{Zvars}(F) \\
a_i &: \texttt{Integer} && \text{if } a_i \in \mathbb{Z} \\
(* \; a_i \; t_i) &: \texttt{Integer} && \text{if } a_i \in \mathbb{Z} \text{ and } t_i : \texttt{Integer} \\
(+ \; t_1 \ldots t_n) &: \texttt{Integer} && \text{if } t_1 : \texttt{Integer}, \ldots, t_n : \texttt{Integer} \\
(\text{ite} \; t_1 \; t_2 \; t_3) &: \texttt{Integer} && \text{if } t_2 : \texttt{Integer}, t_3 : \texttt{Integer}
\end{aligned}$$

And since the arithmetic flattening transformations are so cheap and beneficial, we also apply them for each new shared subterm constructed by our other simplifications.

8.2.3 If-Then-Else Preprocessing

Our next series of preprocessing steps eliminates all if-then-else operations without increasing the formula size and if possible even reducing it.[17] One way of eliminating arithmetic if-then-else operators is to simply lift them over the comparators ($<=$, $>=$, $=$) so they become formula if-then-else operators. Then, we can eliminate the formula if-then-else operators based on the following equivalence derived from the semantic interpretation of the if-then-else operator:

$$(\text{ite} \; t_1 \; t_2 \; t_3) \equiv (\text{and} \; (\text{or} \; t_1 \; t_3) \; (\text{or} \; (\text{not} \; t_1) \; t_2)).$$

However, this semantic replacement for the formula if-then-else operation duplicates the condition t_1. This means that replacing nested if-then-else operations in the condition t_1 causes an exponential blow-up in the formula size. Currently, this blow-up would not be a problem because our terms are shared, but the CNF transformation and the CDCL(LA) algorithm would again be impacted by it. Instead, we choose a *standard if-then-else-elimination* that works similarly to the alternative let-elimination.

[16] Linear in the number of monomials that are part of the operation.

[17] All of our if-then-else preprocessing techniques are based on CVC4's if-then-else preprocessing techniques [9].

Standard If-Then-Else Elimination

As for the alternative let-elimination, we traverse all subterms of the form $t := (\text{ite } t_1\ t_2\ t_3)$ in our set of formulae F. For each such subterm t, (i) we create a new propositional variable y_i if the consequence t_2 and the alternative t_3 are formulae, (ii) we create a new integer variable y_i if the consequence t_2 and the alternative t_3 are linear arithmetic sums and both of the `Integer` type, or (iii) we create a new rational variable y_i if the consequence t_2 and the alternative t_3 are linear arithmetic sums and either is of the `Rational` type. Then, we replace all occurrences of t in F with y_i and extend F by $(\text{or } t_1\ (=\ y_i\ t_3))$ and $(\text{or } (\text{not } t_1)\ (=\ y_i\ t_2))$, i.e., $F := F\{t \mapsto y_i\} \cup \{(\text{or } t_1\ (=\ y_i\ t_3)), (\text{or } (\text{not } t_1)\ (=\ y_i\ t_2))\}$.

Since we base our standard if-then-else-elimination on the alternative let-elimination, it is no wonder that it also has the same disadvantages: (i) it has to introduce new variables; and (ii) it destroys the structure of the formula. Both are factors that decrease the efficiency of our CDCL(LA) implementation. For this reason, SPASS-SATT combines both approaches. During its optimal CNF transformation (Subsection 8.2.4) it decides for each formula if-then-else whether the semantic replacement or the standard if-then-else elimination leads to a better result.

However, this is only an alternative for formula if-then-else operators. For arithmetic if-then-else operators, we first try to eliminate some if-then-else operators with a series of additional preprocessing steps before we eliminate the remaining if-then-else operators with the standard elimination. These preprocessing steps include, if-then-else reconstruction, if-then-else simplifications over constants, boolean if-then-else compressions, selective if-then-else liftings, and bounding of arithmetic if-then-else operations.

If-Then-Else Reconstruction

We start our series of if-then-else preprocessing steps with the *reconstruction of if-then-else operations*. This is necessary because some input files in the SMT-LIB are already preprocessed and not always in a way that is advantageous for SPASS-SATT. For instance, the `rings_preprocessed` benchmark family is equivalent to the `rings` benchmark family except that all if-then-else operations have been eliminated with standard if-then-else elimination [76]. However, the if-then-else operations in the `rings` benchmark family can be processed much more efficiently with different techniques. It is, therefore, a disadvantage that the standard if-then-else elimination was applied to the `rings_preprocessed` benchmark family and we want to reverse it.

To do so, we check whether the set of formulae F contains any clauses that match the clauses added by the standard if-then-else elimination, i.e., whether there exists a set $T_i \subseteq \{(\text{or } t_{i1} \ (= \ y_i \ t_{i3})), (\text{or } (\text{not } t_{i1}) \ (= \ y_i \ t_{i2})), (\text{or } (= \ y_i \ t_{i2}) \ (= \ y_i \ t_{i3}))\}$ with at least two elements (i.e., $|T_i| \geq 2$) that is contained in the set of formulae F (i.e., $T_i \subseteq F$). We then remove all such sets T_i that we find from F and replace all occurrences of y_i in F with $t_i := (\text{ite } t_{i1} \ t_{i2} \ t_{i3})$.

This form of if-then-else reconstruction was not invented by us and it is implemented in several state-of-the-art SMT solvers, e.g., CVC4 [9]. However, there does not yet exist any publication describing this process.

Constant If-Then-Else Simplifications

Our next step of if-then-else preprocessing handles so-called *constant if-then-else expressions* (CITEs). A CITE $\langle cite \rangle$ is defined inductively according to the following language:

$$\begin{aligned} \langle cite \rangle ::= \quad & (+ \ \langle cleaf \rangle) \mid \\ & (+ \ \langle number \rangle \ (* \ \langle number \rangle \ (\text{ite } \langle formula \rangle \ \langle cite \rangle \ \langle cite \rangle))) \\ \langle cleaf \rangle ::= \quad & \langle number \rangle \end{aligned}$$

This means a CITE t is either (i) a *leaf* $t := (+ \ a_0)$, which is a sum containing only a constant monomial, or (ii) a *branch* $t := (+ \ a_0 \ (* \ a_1 \ (\text{ite } t_1 \ t_2 \ t_3)))$, which is a sum containing only a constant monomial a_0 and a complex monomial $(* \ a_1 \ t')$ such that the consequence t_2 and the alternative t_3 of the complex term $t' := (\text{ite } t_1 \ t_2 \ t_3)$ are also CITEs.

Now our first preprocessing step for CITEs pushes all constant monomials and constant coefficients to the leaves:

$$\begin{array}{c} (+ \ a_0 \ (* \ a_1 \ (\text{ite } t_1 \ t_2 \ t_3))) \\ \downarrow \\ (+ \ (* \ 1 \ (\text{ite } t_1 \ (+ \ a_0 \ (* \ a_1 \ t_2)) \ (+ \ a_0 \ (* \ a_1 \ t_3))))) \end{array}$$

This means all occurrences of CITEs simplify and conform now to the following language:

$$\begin{aligned} \langle cite \rangle ::= \quad & (+ \ \langle cleaf \rangle) \mid \\ & (+ \ (* \ 1 \ (\text{ite } \langle formula \rangle \ \langle cite \rangle \ \langle cite \rangle))) \\ \langle cleaf \rangle ::= \quad & \langle number \rangle \end{aligned}$$

This means a CITE t is now either (i) a constant *leaf* a_0 represented through the sum $t := (+ \ a_0)$ containing only a_0, or (ii) a *branch* $t' := (\text{ite } t_1 \ t_2 \ t_3)$ represented through $t := (+ \ (* \ 1 \ (\text{ite } t_1 \ t_2 \ t_3)))$, where t_2 and t_3 are also CITEs.

Or second preprocessing step for CITEs uses the function $\texttt{csimp}(o\ t_1\ t_2)$, where t_1 and t_2 are CITEs and o is one of the operators $<=$, $=$, $>=$, to simplify all linear arithmetic atoms over at most two CITEs, i.e., the atoms defined by the following language:

$$\begin{aligned}\langle \mathit{catom} \rangle ::= \ & (<= \ \langle \mathit{cite} \rangle \ \langle \mathit{cite} \rangle) \mid \\ & (= \ \langle \mathit{cite} \rangle \ \langle \mathit{cite} \rangle) \mid \\ & (>= \ \langle \mathit{cite} \rangle \ \langle \mathit{cite} \rangle)\end{aligned}$$

To this end, $\texttt{csimp}(o\ t_1\ t_2)$ is applied to all atoms $\langle \mathit{catom} \rangle$ in our formulae set F bottom-up. The function $\texttt{csimp}(o\ t_1\ t_2)$ is defined as follows:

- $\texttt{csimp}(o\ t_1\ t_2)$ returns *true* if t_1 is a constant leaf $(+\ a_1)$ and t_2 a CITE branch such that all leaves a_2 in t_2 evaluate $(o\ a_1\ a_2)$ to *true*.
- $\texttt{csimp}(o\ t_1\ t_2)$ returns *true* if t_2 is a constant leaf $(+\ a_2)$ and t_1 a CITE branch such that all leaves a_1 in t_1 evaluate $(o\ a_1\ a_2)$ to *true*.
- $\texttt{csimp}(o\ t_1\ t_2)$ returns *false* if t_1 is a constant leaf $(+\ a_1)$ and t_2 a CITE branch such that all leaves a_2 in t_2 evaluate $(o\ a_1\ a_2)$ to *false*.
- $\texttt{csimp}(o\ t_1\ t_2)$ returns *false* if t_2 is a constant leaf $(+\ a_2)$ and t_1 a CITE branch such that all leaves a_1 in t_1 evaluate $(o\ a_1\ a_2)$ to *false*.
- $\texttt{csimp}(o\ t_1\ t_2)$ returns $(\text{ite}\ t'_1\ \texttt{csimp}(o\ t_1\ t'_2)\ \texttt{csimp}(o\ t_1\ t'_3))$ if t_1 is a constant leaf $(+\ a_1)$ and $t_2 := (+\ (*\ 1\ (\text{ite}\ t'_1\ t'_2\ t'_3)))$ a CITE branch such that at least one leaf a_2 in t_2 evaluates $(o\ a_1\ a_2)$ to *true* and another leaf a'_2 in t_2 evaluates $(o\ a_1\ a'_2)$ to *false*.
- $\texttt{csimp}(o\ t_1\ t_2)$ returns $(\text{ite}\ t'_1\ \texttt{csimp}(o\ t'_2\ t_2)\ \texttt{csimp}(o\ t'_3\ t_2))$ if t_2 is a constant leaf $(+\ a_2)$ and $t_1 := (+\ (*\ 1\ (\text{ite}\ t'_1\ t'_2\ t'_3)))$ a CITE branch such that at least one leaf a_1 in t_1 evaluates $(o\ a_1\ a_2)$ to *true* and another leaf a'_1 in t_1 evaluates $(o\ a'_1\ a_2)$ to *false*.
- $\texttt{csimp}(o\ t_1\ t_2)$ returns $(\text{or}\ t'_1 \ldots t'_2)$ if o is the equality operator $=$, if t_1 and t_2 are both CITE branches that have the leaves $a_1, \ldots, a_n$ in common, and if $t'_i := (\text{and}\ \texttt{csimp}(=\ t_1\ a_i)\ \texttt{csimp}(=\ t_2\ a_i))$.
- In all other cases, $\texttt{csimp}(o\ t_1\ t_2)$ returns just $(o\ t_1\ t_2)$.

To summarize, $\texttt{csimp}(o\ t_1\ t_2)$ pushes the comparison operator o recursively down the CITE branches and greedily simplifies any branch to *true* or *false* if possible.

The above constant if-then-else simplifications are a subset of the techniques presented by Kim et al. in [96]. The same subset of constant if-then-else simplifications is also used by the SMT solver CVC4 [9]. The above simplifications can only be performed efficiently if the intermediate results are computed once and not recomputed again. To this end, we cache all intermediate results in hash maps and check our cache before we recompute the results. For the constant if-then-else simplifications, we use two caches: (i) we cache the computed leaves of a CITE subterm t in a hash map pointing from shared terms to leaf sets; and (ii) we cache the recursive $\texttt{csimp}(o\ t_1\ t_2)$ calls in a hash map pointing from the triple (o, t_1, t_2) to the return value of $\texttt{csimp}(o\ t_1\ t_2)$.

If-Then-Else Compression

Successful applications of the constant if-then-else simplifications, i.e., calls to `csimp`$(o\ t_1\ t_2)$ that actually simplify branches to *true* or *false*, often lead to nested if-then-else terms of the following form:

$$\begin{array}{l} t_i ::= (\text{ite } t'_i\ t_{i+1}\ \mathit{false}) \mid (\text{ite } t'_i\ \mathit{false}\ t_{i+1}) \quad \text{for } i = 1, \ldots, n-1 \\ t_n \text{ can be any formula term.} \end{array} \tag{8.1}$$

We are interested in terms of this form because we can efficiently eliminate if-then-else operations, where either the consequence or alternative is *false*.[18] For instance, if we look at the semantic interpretation of the term (ite $t'_i\ t_{i+1}$ *false*), then we see that it evaluates only to *true* if (and $t'_i\ t_{i+1}$) also evaluates to *true*. The term (ite $t'_i\ t_{i+1}$ *false*) is, therefore, equivalent to the conjunction (and $t'_i\ t_{i+1}$) and, symmetrically, (ite t'_i *false* t_{i+1}) is equivalent to the conjunction (and (not t'_i) t_{i+1}). We can also apply this equivalence iteratively to the sequence of nested if-then-else terms $t_1, \ldots, t_n$ and combine the resulting conjunctions. The result is the equivalent conjunction (and $t^*_1 \ldots t^*_{n-1}\ t_n$), where $t^*_i := t'_i$ if $t_i :=$ (ite $t'_i\ t_{i+1}$ *false*) and $t^*_i :=$ (not t'_i) if $t_i :=$ (ite t'_i *false* t_{i+1}).

In SPASS-SATT, we also transform nested if-then-else expressions of the form (8.1) into conjunctions. However, we perform the transformation more selectively than described above. We do so by traversing the subterms in our formulae $f \in F$ in top-down order and by looking for any occurrence of a subterm t_1 of the form (8.1). Then we add all nested if-then-else terms t_i to our sequence that also have the form (8.1) and that only appear once as a subterm in F. This means t_n is either appearing more than once in F or it is not an if-then-else operation where the consequence or alternative is *false*. We then create a new propositional variable p_j, replace all occurrences of t_1 in F with p_j, and extend F by the equivalence (= p_j (and $t^*_1 \ldots t^*_{n-1}\ t_n$)). We do this transformation so selectively in order to strengthen the connection of the shared subterms t_n that have multiple occurrences in F. So the introduction of the new propositional variable p_j actually strengthens the structure of our formulae instead of destroying it.

The above described technique is called if-then-else compression and it was first presented by Burch in [37]. However, Burch did not present it with SMT solving in mind but for an alternative application, verification of control circuits with the help of circuit simplifications. We are unaware of any publication describing if-then-else compression in the context of SMT solving although the SMT solver CVC4 has used this technique for quite some time [9].

[18] We can do a similar elimination for if-then-else operations, where either the consequence or alternative is *true*. However, SPASS-SATT did not gain any advantage on the SMT-LIB benchmarks for also treating these cases.

Lifting Shared Monomials

We have already shown how to simplify linear arithmetic atoms $(o\ t_1\ t_2)$, where t_1 and t_2 are constant if-then-else expressions (CITEs). However, other occurrences of CITEs, i.e., $t := (\text{ite}\ t_1\ t_2\ t_3)$ in the linear arithmetic sum $t' := (+\ q\ (*\ a_i\ t)\ q')$ (where t_2 and t_3 are CITEs), are also easy to resolve. This is possible because we can optimize the standard if-then-else elimination for CITEs with additional techniques, e.g., lifting of the least common multiple in constant if-then-else expressions and bounding of constant if-then-else expressions, which we explain later in this section.

Since CITEs are easier to handle than regular if-then-else expressions, we also want to simplify as many regular if-then-else expressions to constant if-then-else expressions as possible. This means we want to lift monomials shared by the consequence and the alternative of an if-then-else expression on top of the if-then-else expression; especially, if this leads to new constant if-then-else expressions. Formally, this means that we traverse the subterms in our formulae $f \in F$ in bottom-up order and transform all subterms $t := (\text{ite}\ t_1\ (+\ q'\ q)\ (+\ \hat{q}\ q))$ that we encounter into subterms $(+\ q\ (*\ 1\ (\text{ite}\ t_1\ (+\ q')\ (+\ \hat{q}))))$, where q, q', and $\hat{q}$ are a series of monomials.[19]

In SPASS-SATT we call the above described technique *lifting of shared monomials*. We are unaware of any publication that describes lifting of shared monomials in the context of SMT solving. We know, however, that the SMT solver CVC4 has used this technique for quite some time [9].

Bounding Constant If-Then-Else Expressions

We can actually combine the standard if-then-else elimination and the if-then-else lifting for constant if-then-else expressions t. We simply create one new linear arithmetic variable x_j for the whole constant if-then-else expression, replace all occurences of t with x_j in F, and extend F by the formula f, which is equivalent to t except that all leaves a_i of t are replaced by the equations $(=\ x_j\ a_i)$. The added formula if-then-else operations f are then handled by our optimal CNF transformation.

Although this combination reduces the number of newly created arithmetic variables, it still destroys a lot of structure that is now no longer accessible from the theory side. For instance, our theory solver does not know that the only legitimate values for x_j correspond to the leaves $\{a_1, \ldots, a_n\} \in \mathbb{Q}$ of t. Unfortunately, our arithmetic constraints are not expressive enough to efficiently limit x_j to an arbitrary set of values $\{a_1, \ldots, a_n\} \in \mathbb{Q}$. Ho-

[19] This definition assumes for simplicity that the shared monomials q appear after the unshared monomials q' and $\hat{q}$. In reality this is not always the case and SPASS-SATT has to find and extract the shared monomials explicitly.

wever, we can at least reduce the number of values x_j can take to a finite interval $[a_{\min}, a_{\max}]$. If the values $\{a_1, \ldots, a_n\}$ are actually integer values, then we can reduce the number of values x_j can take even further until they correspond to a finite superset of $\{a_1, \ldots, a_n\}$.

For the reduction, we preform the following steps: As our first step, we bound x_j by $a_{\max} := \max\{a_1, \ldots, a_n\}$ and $a_{\min} := \min\{a_1, \ldots, a_n\}$, i.e., we add the inequalities $(<= \ x_j \ a_{\max})$ and $(>= \ x_j \ a_{\min})$ to F. This means that x_j can only only be assigned to values from the interval $[a_{\min}, a_{\max}]$. As our second step, we turn the variable x_j into an integer variable if the leaves $\{a_1, \ldots, a_n\}$ of t are all integer values. As a result, x_j can only only be assigned to integer values from the interval $[a_{\min}, a_{\max}]$, which is a finite superset of $\{a_1, \ldots, a_n\}$. Next, we try to shrink the distance between our leaves $\{a_1, \ldots, a_n\}$ and, therefore, the interval $[a_{\min}, a_{\max}]$ over which x_j ranges. Note, however, that this step only works and is, therefore, only applied if the leaves $\{a_1, \ldots, a_n\}$ are all integer values. We perform the interval shrinking by computing the gcd $a_0 := \gcd\{a_1, \ldots, a_n\}$ of our leaves and by replacing all occurrences of x_j in F with $(* \ a_0 \ x_j)$. This means the equations $(= \ x_j \ a_i)$ in f become $(= \ (* \ a_0 \ x_j) \ a_i)$, which is equivalent to $(= \ x_j \ a'_i)$, where $a'_i := (a_i / a_0) \in \mathbb{Z}$. The bounds $(<= \ x_j \ a_{\max})$ and $(>= \ x_j \ a_{\min})$ in F also change symmetrically, and are now equivalent to $(<= \ x_j \ a'_{\max})$ and $(>= \ x_j \ a'_{\min})$, where $a'_{\max} := \max\{a'_1, \ldots, a'_n\}$ and $a'_{\min} := \min\{a'_1, \ldots, a'_n\}$. As a result, x_j can only be assigned to the integer values from the interval $[a'_{\min}, a'_{\max}]$, which contains only $\frac{1}{a_0}$-th of the integer values that$[a_{\min}, a_{\max}]$ contained.

In SPASS-SATT we call the above described technique *bounding constant if-then-else expressions*. We are unaware of any publication that describes this bounding process in the context of SMT solving. We know, however, that the SMT solver CVC4 has used this technique for quite some time [9].

Combination of If-Then-Else Preprocessing Steps

Bounding constant if-then-else expressions concludes the if-then-else preprocessing steps performed by SPASS-SATT. For a better overview, we summarize here the whole if-then-else preprocessing process: SPASS-SATT starts by reconstructing if-then-else operations that were naively eliminated from the input-file itself. Then, SPASS-SATT simplifies atoms over constant if-then-else expressions. As its next step, SPASS-SATT compresses if-then-else expressions that are equivalent to conjunctions. After that, SPASS-SATT shifts its focus back to constant if-then-else expressions. It first tries to create more of them by lifting shared monomials over the if-then-else operations. The resulting constant if-then-else expressions are then bounded

and eliminated by a combination of standard if-then-else elimination and if-then-else lifting. Next, all remaining arithmetic if-then-else expressions are eliminated through standard if-then-else elimination. The formula if-then-else expression are later handled as part of the optimal CNF transformation.

8.2.4 Other Simplifications

We have already presented several of SPASS-SATT's preprocessing techniques. Most of them were focused on reducing the formula size, on eliminating special operators, or on simplifying the overall formula structure. Our next series of preprocessing techniques focuses instead on optimizing the formula structure for our decision procedures.

Handling Variable Definitions

For instance, SPASS-SATT tries to remove some trivial variables, which we call *defined variables*. A *defined variable* y_j has an equation $f \in F$ that either assigns it to a truth value, e.g., $f := (= \ y_j \ \mathit{true})$, a constant a_i, e.g., $f := (= \ y_j \ a_i)$, another variable y_i, e.g., $f := (= \ y_j \ y_i)$, a negated version of another variable (not y_i), e.g., $f := (= \ y_j \ (\text{not } y_i))$, or a negativ version of another variable $(- \ y_i)$, e.g., $f := (= \ y_j \ (- \ y_i))$. This means it is quite easy to eliminate defined variables y_j by simply replacing them with their assigned values.

Removing variables is beneficial to most of our decision procedures because their runtimes are dependent on the number of variables. We restrict ourselves to variables assigned to small terms because the runtimes of our decision procedures are also dependent on the number of monomials in each inequality and the number of literals in each clause.

Pseudo-Boolean Preprocessing

SPASS-SATT also optimizes the formula structure when it encounters so-called *linear pseudo-boolean* problems [78]. Linear pseudo-boolean problems are ground and conjunctive linear arithmetic problems, where all arithmetic variables x_j are actually *pseudo-boolean variables*, i.e., integer variables with bounds $(>= \ x_j \ 0)$ and $(<= \ x_j \ 1)$ in F ($\{(>= \ x_j \ 0), (<= \ x_j \ 1)\} \subseteq F$). There actually exist many real world applications that can be encoded as linear pseudo-boolean problems [78]. As a result, a specialized research community, the pseudo-boolean optimization community, emerged that focuses solely on these kinds of problems. And naturally, they have also developed their own specialized decision (and optimization) procedures for pseudo-boolean problems.

Currently, the pseudo-boolean solvers with their specialized procedures still perform far better on pseudo-boolean problems than SPASS-SATT with its very general branch-and-bound approach. And this will not change until we have the time to reimplement some of their specialized procedures in SPASS-SATT. In the meantime, we have at least implemented a preprocessing technique that helps SPASS-SATT in closing the gap. To be more specific, our preprocessing technique turns some linear *pseudo-boolean inequalities* (i.e., inequalities containing just pseudo-boolean variables) into equivalent disjunctions and conjunctions of literals. For instance, the inequality $(>=\ (+\ t_1 \ldots t_n)\ 1)$, where t_i is either x_i or $(-\ 1\ x_i)$, is semantically equivalent to the disjunction $(\text{or}\ l_1 \ldots l_n)$, where l_i is $(>=\ x_i\ 1)$ if $t_i = x_i$ or l_i is $(\textit{not}\ (>=\ x_i\ 1))$ if $t_i = (-\ 1\ x_i)$. Symmetrically, the inequality $(>=\ (+\ t_1 \ldots t_n)\ n)$, where t_i is either x_i or $(-\ 1\ x_i)$, is semantically equivalent to the conjunction $(\text{and}\ l_1 \ldots l_n)$, where l_i is $(>=\ x_i\ 1)$ if $t_i = x_i$ or l_i is $(\text{not}\ (>=\ x_i\ 1))$ if $t_i = (-\ 1\ x_i)$. SPASS-SATT recognizes all similar cases of linear pseudo-boolean inequalities that are equivalent to a single disjunction or a single conjunction with at most three literals. Moreover, SPASS-SATT replaces these inequalities with their equivalent disjunctions and conjunctions. We do so because the SAT solver seems to reason more efficiently over the disjunctions and conjunctions than our theory solver over the equivalent linear pseudo-boolean inequalities. However, this holds only for linear pseudo-boolean inequalities containing at most three variables. If SPASS-SATT replaces all linear pseudo-boolean inequalities that are equivalent to a single disjunction or a single conjunction, then it fails to solve some of the problems from the pigeonhole benchmark family.

The above described transformations go back to the NP-hardness proof of 0-1 programming [94]. Moreover, they are used by at least one other SMT solver (CVC4) [9].

Small CNF Construction

The final preprocessing step of SPASS-SATT is the *CNF transformation*, i.e., the transformation of any general first-order formula into an equisatisfiable first-order formula in clause normal form (CNF). This step is necessary because our CDCL(LA) implementation, which combines most of SPASS-SATT's decision procedures, can only handle formulae in CNF. The *standard CNF transformation* works as follows [6]. It first replaces all equivalences, implications, and if-then-else expressions with equivalent expressions consisting only of conjunctions, disjunctions, and negations, e.g., $(=\ t_1\ t_2)$ is equivalent to $(\text{or}\ (\text{and}\ t_1\ t_2)\ (\text{and}\ (\text{not}\ t_1)\ (\text{not}\ t_2)))$. Then, it pushes all negations down until there are only negations on top of the atoms (see also Section 8.2.2). Finally, it distributes all disjunctions inwards over conjunctions, i.e., it replaces all subterms $(\text{or}\ t_1\ (\text{and}\ t_2\ t_3))$ with $(\text{and}\ (\text{or}\ t_1\ t_2)\ (\text{or}\ t_1\ t_3))$. The advantage of the standard CNF transforma-

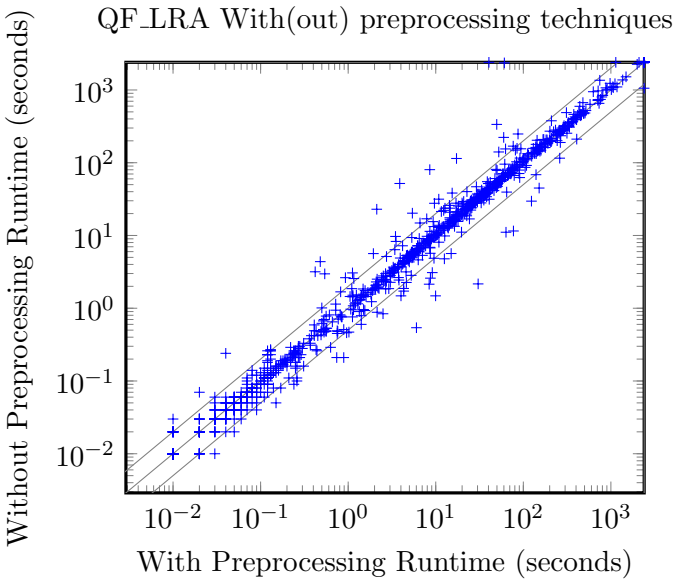
tion is that it preserves not only equisatisfiability but also equivalence. This comes, however, at a cost: in the worst case, a formula grows during the standard CNF transformation by an exponential factor. The exponential growth is caused by nested applications of those parts of the transformation that duplicate subterms, i.e, the elimination of equivalences and if-then-else expressions and the distribution of disjunctions over conjunctions.

An alternative to the standard CNF transformation is the *greedy renaming CNF transformation* [133]. Renaming replaces a subterm t in a formula f with a new predicate p_i, which is then defined as equivalent to t over the whole formula f by conjunctively adding the equivalence $(= \ p_i \ t)$ to f, i.e., f is transformed into (and $f\{t \mapsto p_i\} \ (= \ p_i \ t)$). The CNF transformation with greedy renaming now uses this technique to rename those subterms in f that would be duplicated by the standard CNF transformation. As a result, the renamed formula f' contains no more nested subterms that would cause an exponential growth during the standard CNF transformation. Therefore, we can apply the standard CNF transformation after the remaining without encountering an exponential growth in the formula size.

Although the greedy renaming CNF transformation prevents the exponential growth of the standard CNF transformation, it also introduces four new problems. Firstly, the renaming adds new propositional variables and the runtime of CDCL(LA) is in the worst case exponentially proportional to the number of propositional variables. Secondly, the renaming also destroys connections in the original problem structure, which we could otherwise exploit in our decision procedures. Thirdly, the resulting formula is only equisatisfiable and not equivalent. [20] Finally, renaming sometimes creates a larger formula (or at least larger subformulae) then the standard CNF. But despite these new disadvantages, the greedy renaming CNF transformation typically performs much better in practice than the standard CNF transformation.

We can, however, get rid of the fourth disadvantage and lessen the first two disadvantages by renaming only those parts of the formula, where renaming leads to a smaller subformula than standard CNF. We call this CNF transformation the *small CNF transformation* and it was first presented by Nonnengart and Weidenbach [115]. The small CNF transformation always produces the smallest (sub)formula with respect to the standard CNF transformation and the greedy renaming CNF transformation. Moreover, the small CNF transformation always introduces at most as many new propositional variables as greedy renaming does. This means the small CNF transformation typically introduces less new propositional variables and destroys less structural connections.

[20] This is actually not a major problem because we can easily transform any model for the CNF formula into a model for the original formula by just ignoring the newly added propositional variables.



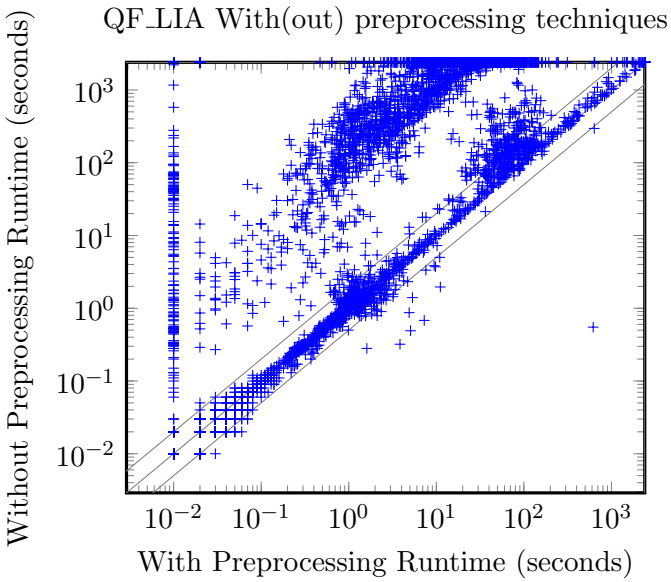
SPASS-SATT with preprocessing techniques solves 1607 instances
SPASS-SATT without preprocessing techniques solves 1605 instances

Figure 8.12: With(out) preprocessing techniques on the QF_LRA SMT-LIB benchmarks

The implementation of the small CNF transformation was taken from the SPASS Workbench [3]. It replaced a previous implementation of the greedy renaming CNF transformation. Since we have both transformations available, we were also able to confirm that the small CNF transformation helps in practice. One example where the small CNF transformation performs much better than the greedy renaming CNF transformation is the `convert` benchmark family from the QF_LIA category of the SMT-LIB. More details are provided in the preprocessing experiments at the end of this section.

8.2.5 Preprocessing Experiments

In this section, we described several preprocessing techniques that simplify input problems so they become easier to solve for SPASS-SATT. Techniques of particular interest are SPASS-SATT's preprocessing techniques for if-then-else expressions (constant if-then-else simplification, if-then-else compression, if-then-else reconstruction, shared monomial lifting, constant if-then-else bounding), as well as SPASS-SATT's preprocessing techniques for variable definitions and pseudo-boolean inequalities. In order to measure the impact of these preprocessing techniques, we performed a series of benchmark experiments with SPASS-SATT.



SPASS-SATT with preprocessing techniques solves 6806 instances
SPASS-SATT without preprocessing techniques solves 5113 instances

Figure 8.13: With(out) preprocessing techniques on the QF_LIA SMT-LIB benchmarks

Our first two figures (Figure 8.12 & 8.13) examine the combined impact that SPASS-SATT's preprocessing techniques (for if-then-else expressions, defined variables, and pseudo-boolean inequalities) have on SPASS-SATT's performance on the QF_LRA and QF_LIA benchmarks. To this end, we compare the default version of SPASS-SATT with all preprocessing techniques turned on (horizontal axis) and the version of SPASS-SATT without the preprocessing techniques for if-then-else expressions, defined variables, and pseudo-boolean inequalities (vertical axis).

In Figure 8.12, we observe that our preprocessing techniques have only a minor impact on the QF_LRA benchmarks. SPASS-SATT with our preprocessing techniques solves only two more QF_LRA benchmark instances (both from the `latendresse` benchmark family). However, we can at least conclude that these additions are independent of any cluster fluctuations since they occur far enough away from our timeout limit. Otherwise, we observe that SPASS-SATT with our preprocessing instances is significantly faster (i.e., it is more than twice as fast) as often as it is significantly slower (i.e., it is more than twice as slow). We were unable to discern a clear pattern that would allow us to predict when our preprocessing techniques are advantageous on the QF_LRA benchmarks.

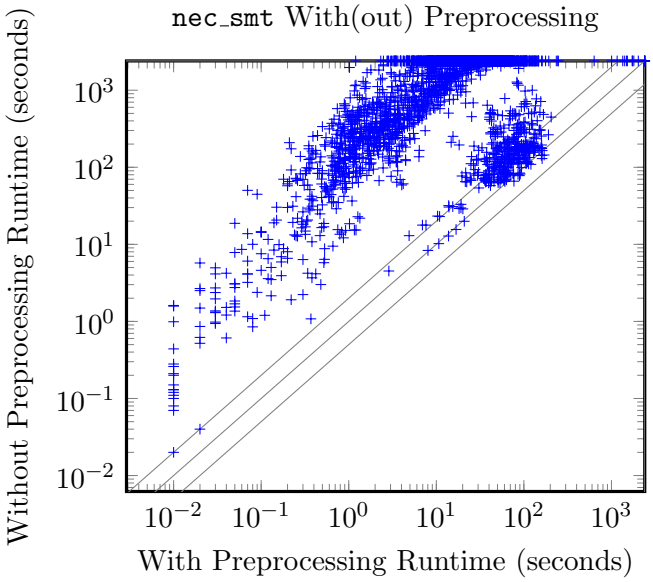
In Figure 8.13, we observe that our preprocessing techniques have a major impact on the QF_LIA benchmarks. SPASS-SATT with our preprocessing techniques solves 1693 additional QF_LIA benchmark instances and is significantly faster (i.e., it is more than twice as fast) on 3146 QF_LIA benchmark instances than SPASS-SATT without our preprocessing techniques. Moreover, instances where our preprocessing techniques cause a significant slowdown occur only rarely. Also in contrast to the QF_LRA results, we can clearly determine four benchmark families, on which our preprocessing techniques make this positive impact. These families are `nec_smt`, `rings`, `rings_preprocessed`, and `pb2010`; we are going to analyze them in more detail as part of this section.

The `nec_smt` Benchmark Family

The instances from the `nec_smt` benchmark family were generated by the SMT-based boolean model checking engine of F-Soft, which is a software verification platform [85]. All instances from `nec_smt` have in common that they contain many nested if-then-else and let expressions. As a result of this nesting, we cannot just remove the let operations or else we are creating formulas that are too big to be handled within reasonable memory and time limits. We can, however, handle them if we additionally simplify the nested if-then-else expressions. The two preprocessing techniques that make this possible are our constant if-then-else simplifications and the boolean if-then-else compression (Subsection 8.2.3).

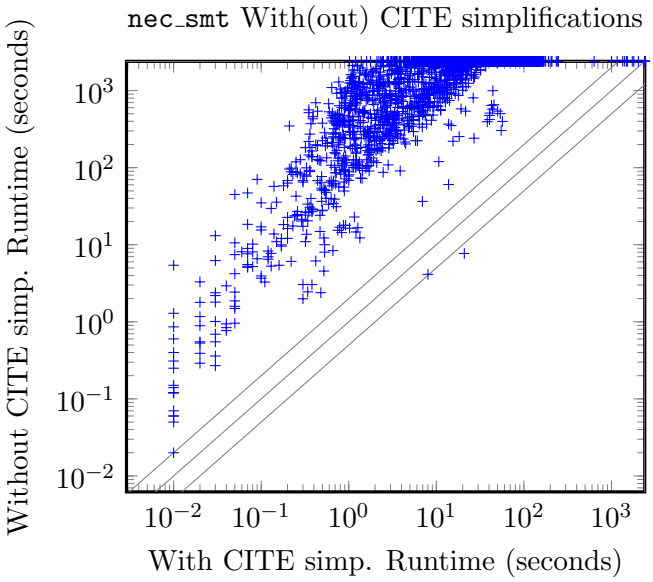
In Figure 8.14, we see that SPASS-SATT without our preprocessing techniques solves only 1422 out of the 2800 `nec_smt` benchmark instances and is by far slower on the instances it can solve. In Figures 8.15 & 8.16, we see that SPASS-SATT performs even worse if we only perform one of our two preprocessing techniques.[21] However, SPASS-SATT with both constant if-then-else simplifications and the boolean if-then-else compression solves 2782 out of the 2800 benchmark instances. This means that only the combination of the two techniques produces an actually beneficial result. Moreover, the two preprocessing techniques simplify the input problems so much that SPASS-SATT does not need branch-and-bound or one of its extensions to solve these instances. SPASS-SATT's simplex implementation on its own is sufficient enough and detects for all solved instances either a rational conflict or a rational solution that is also an integer solution.

[21] Other preprocessing techniques, e.g., constant if-then-else bounding, also have a negative effect on the `nec_smt` benchmarks if we do not first apply the constant if-then-else simplifications and the boolean if-then-else compression.



SPASS-SATT with preprocessing solves 2772 instances
SPASS-SATT without preprocessing solves 1422 instances

Figure 8.14: With(out) extensive preprocessing on the QF_LIA `nec_smt` benchmark family



SPASS-SATT with CITE simplifications solves 2772 instances
SPASS-SATT without CITE simplifications solves 1016 instances

Figure 8.15: With(out) constant if-then-else (CITE) simplifications on the QF_LIA `nec_smt` benchmark family

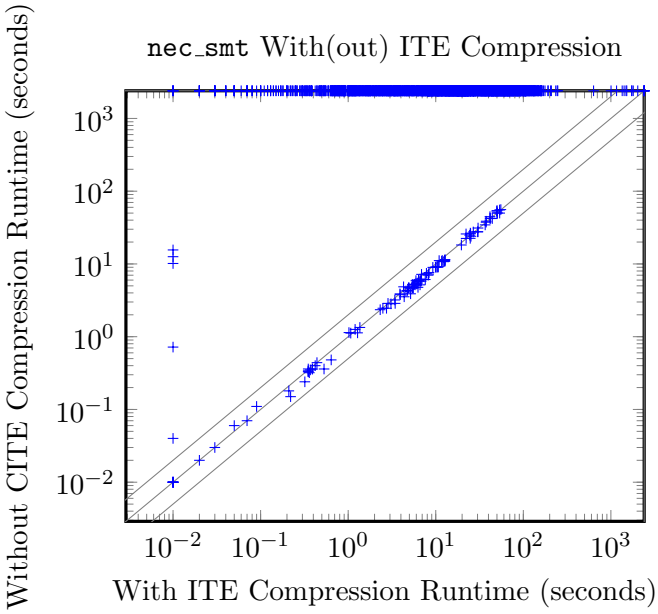
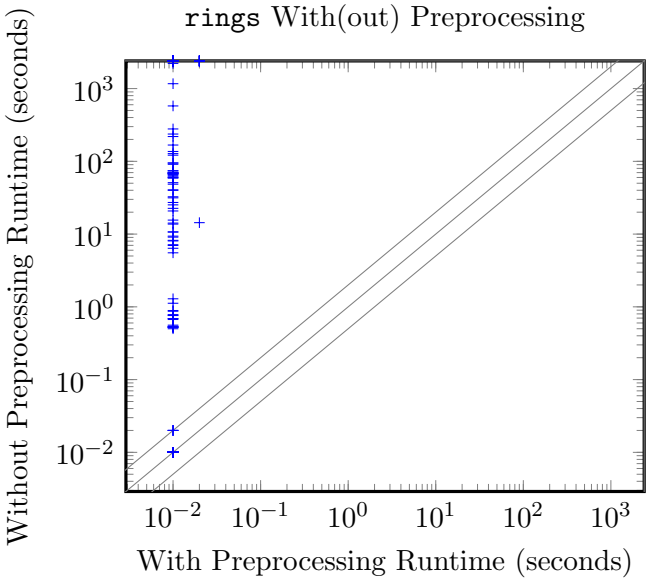


Figure 8.16: With(out) if-then-else (ITE) compression on the QF_LIA `nec_smt` benchmark family

The `rings` Benchmark Family

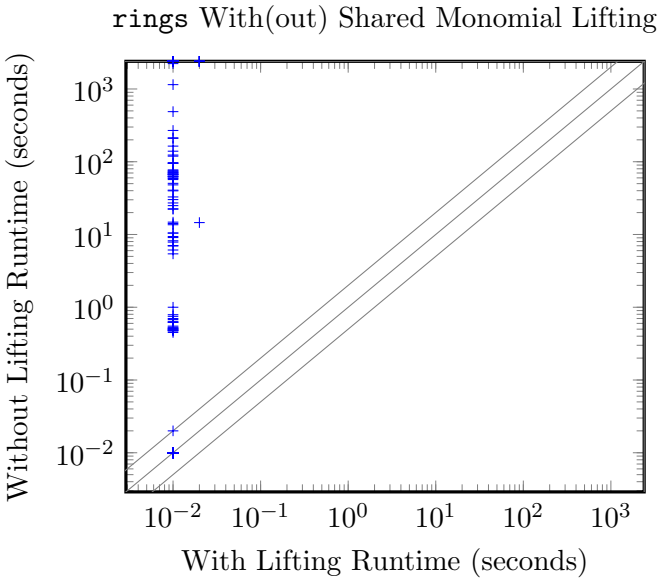
The `rings` benchmark family was constructed in order to add some hard QF_LIA problems to the SMT-LIB. To be more precise, the goal was to create integer problems that cannot be solved with only techniques for linear rational arithmetic [76]. This goal was achieved by encoding associative properties on modular arithmetic with the help of if-then-else expressions. However, with the right preprocessing techniques, e.g., a combination of shared monomial lifting and constant if-then-else bounding (Subsection 8.2.3), these problems become almost trivial to solve. In fact, SPASS-SATT needs less than one second for each problem instance and needs only techniques for linear rational arithmetic to solve each of them.

In Figure 8.17, we see that SPASS-SATT without our preprocessing techniques solves only 136 out of the 294 `rings` benchmark instances and is by far slower on the instances it can solve. In Figures 8.18 & 8.19, we see that SPASS-SATT performs not much better if we just perform either shared monomial lifting or constant if-then-else bounding. However, SPASS-SATT with both shared monomial lifting and constant if-then-else bounding solves all of the 294 benchmark instances in no time. This means that only the combination of the two techniques produces an actually beneficial result.



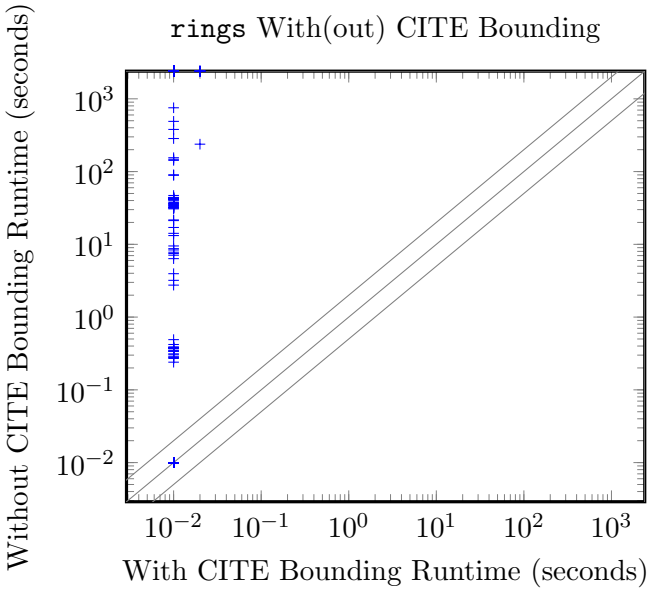
SPASS-SATT with preprocessing solves 294 instances
SPASS-SATT without preprocessing solves 136 instances

Figure 8.17: With(out) extensive preprocessing on the QF_LIA `rings` benchmark family



SPASS-SATT with shared monomial lifting solves 294 instances
SPASS-SATT without shared monomial lifting solves 136 instances

Figure 8.18: With(out) shared monomial lifting on the QF_LIA `rings` benchmark family



SPASS-SATT with CITE bounding solves 294 instances
SPASS-SATT without CITE bounding solves 138 instances

Figure 8.19: With(out) constant if-then-else (CITE) bounding on the QF_LIA `rings` benchmark family

The `rings_preprocessed` Benchmark Family

The `rings_preprocessed` benchmark family is equivalent to the `rings` benchmark family except that all if-then-else operations were eliminated with standard if-then-else elimination [76]. It was constructed in order to prevent the previously explained preprocessing trick for the `rings` benchmark family [76]. However, we can use the same trick that we used for the `rings` benchmark family also for the `rings_preprocessed` benchmark family if we just use our if-then-else reconstruction technique to reverses the standard if-then-else elimination (Subsection 8.2.3). In Figure 8.20, we see that SPASS-SATT without if-then-else reconstruction performs as poorly on the `rings_preprocessed` benchmark family as it performs on the `rings` benchmark family without any preprocessing techniques. However, SPASS-SATT with if-then-else reconstruction solves again all of the 294 benchmark instances in no time.

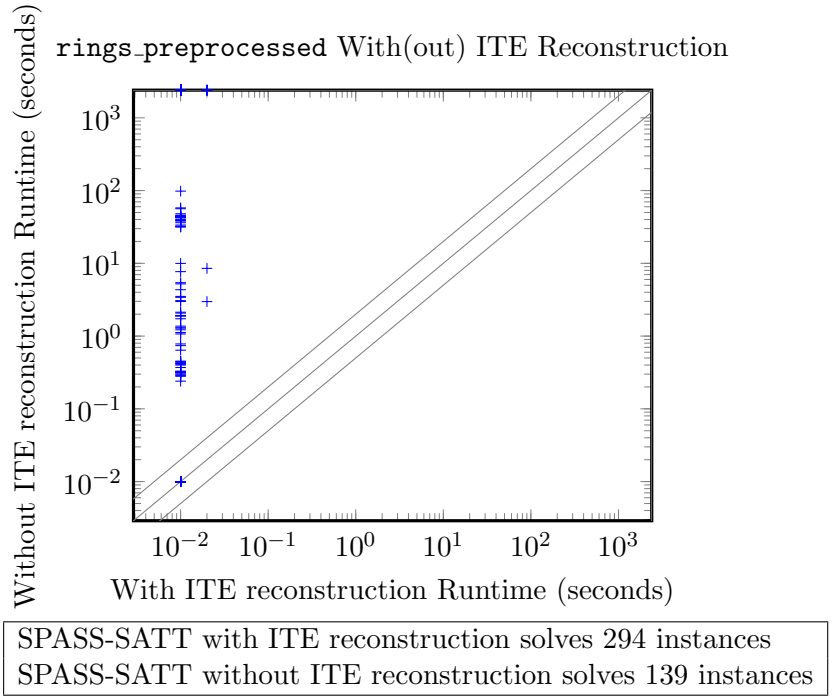


Figure 8.20: With(out) if-then-else (ITE) reconstruction on the QF_LIA `rings_preprocessed` benchmark family

The `pb2010` Benchmark Family

The `pb2010` benchmark family is a set of industrial problems taken from the pseudo-boolean competition 2010. Naturally, all of the contained problems are pseudo-boolean problems and, therefore, rather difficult for solvers without specialized decision procedures for pseudo-boolean problems. This includes SPASS-SATT with its very general branch-and-bound approach. In fact, SPASS-SATT does not use/need the branch-and-bound approach for the pseudo-boolean problems it can solve within the time limit. Instead, SPASS-SATT uses its preprocessing technique for pseudo-boolean inequalities to shift the difficulty of the problem instances from arithmetic to SAT solving. This allows SPASS-SATT to solve 22 more benchmark instances from the 81 instances in the `pb2010` benchmark family (see Figure 8.21) than SPASS-SATT without pseudo-boolean preprocessing.

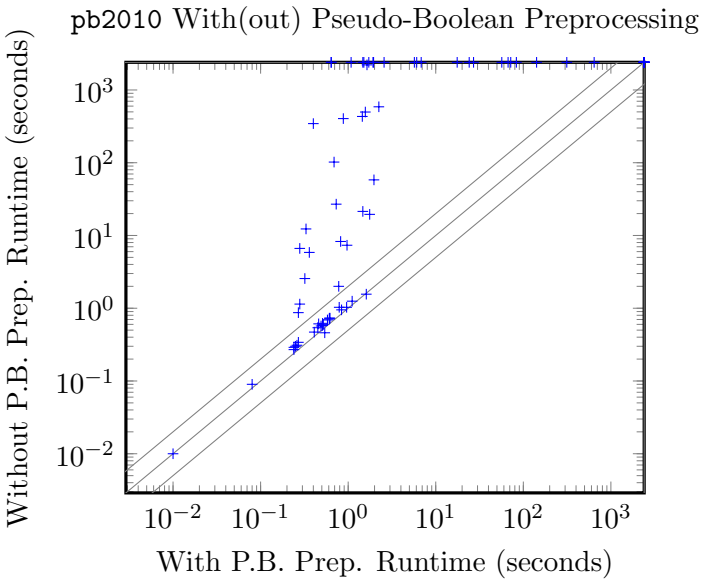
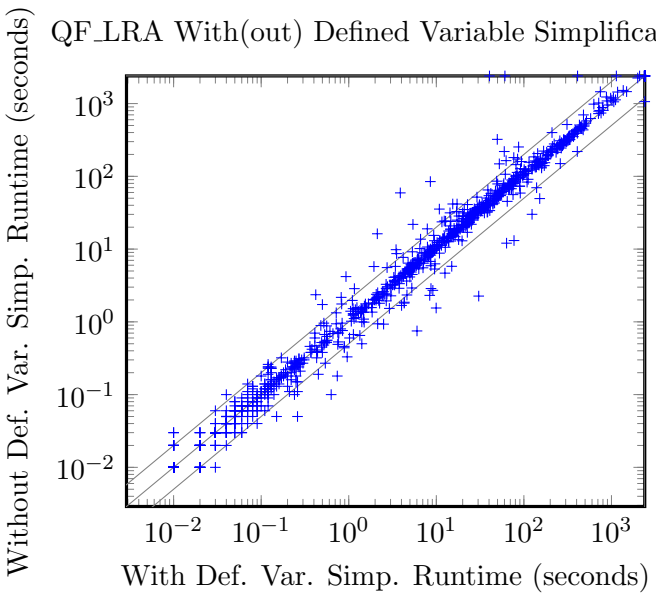


Figure 8.21: With(out) pseudo-boolean preprocessing on the QF_LIA `pb2010` benchmark family

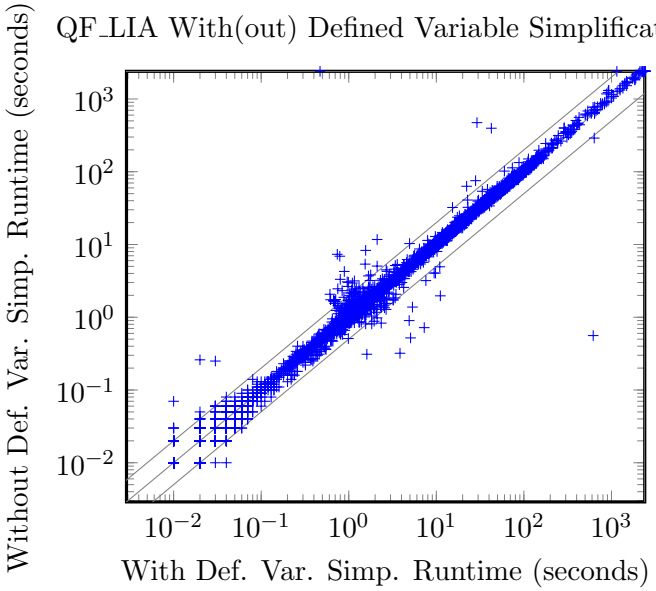
Handling Variable Definitions

Another one of our more specialized preprocessing techniques is the elimination of defined variables. In Figures 8.24 & 8.25, we examine the impact that the elimination of defined variables has on SPASS-SATT's performance on the QF_LRA and QF_LIA benchmarks. In these figures, we can observe that the elimination of defined variables has only a minor impact on SPASS-SATT's performance. However, we can also see that there are at least a few problems from both QF_LRA and QF_LIA that we can only solve thanks to the elimination of defined variables. These problems are 2 problems from the `latendresse` benchmark family and 2 problems from the `bofill-scheduling` benchmark family. The other additionally solved benchmark instances are solved so close to the timeout that they probably occurred due to performance fluctuations of our cluster.



SPASS-SATT with defined var. simplifications solves 1607 instances
SPASS-SATT without defined var. simplifications solves 1603 instances

Figure 8.22: With(out) defined variable simplifications on the QF_LRA SMT-LIB benchmarks



SPASS-SATT with defined var. simplifications solves 6806 instances
SPASS-SATT without defined var. simplifications solves 6799 instances

Figure 8.23: With(out) defined variable simplifications on the QF_LIA SMT-LIB benchmarks

Small CNF Construction

As explained before, SPASS-SATT uses a *small CNF transformation* instead of the commonly used *greedy renaming CNF transformation*. Naturally, this also has an impact on SPASS-SATT's performance on the SMT-LIB benchmarks. In Figures 8.24 & 8.25, we examine the impact that the small CNF transformation has on SPASS-SATT's performance on the QF_LRA and QF_LIA benchmarks compared to the greedy renaming CNF transformation. In these figures, we see that the small CNF transformation leads to overall better results than the greedy renaming CNF transformation. In spite of that, there are many instances where the greedy renaming CNF transformation performs better. We suspect, however, that most of the positive and negative changes to the results are not because one transformation produces a better clause set, but because our new transformation orders the clauses and literals by accident differently than our old transformation.

There is, however, one benchmark family in the QF_LIA division where the small CNF transformation produces a beneficial result because it actually produces a better clause set and not just a better clause order. This benchmark family is called `convert` and Figure 8.26 examines the impact that the small CNF transformation has on SPASS-SATT's performance on the `convert` benchmark family compared to the greedy renaming CNF transformation. In this figure, we can see that SPASS-SATT with the small CNF transformation solves all 319 `convert` instances in a couple of seconds, and SPASS-SATT with the greedy renaming CNF transformation can only solve 218 out of the 319 instances. The reason is that most formulas in the `convert` family look abstractly as follows:

$$F := (\text{and } \ldots\ f_1\ \ldots\ f_n\ \ldots)\ ,$$
$$\text{where } f_i := (\text{or } (=\ t_{i1}\ t_{i2})\ (\text{not } (=\ t_{i3}\ t_{i4})))\ .$$

This formula structure is problematic for our implementation of the greedy renaming CNF transformation because of two implementation choices. Firstly, SPASS-SATT splits all occurring arithmetic equalities into two inequalities in order to handle negated arithmetic equalities. Therefore, SPASS-SATT transforms the subformulas f_i into

$$f_i' := (\text{or } (\text{and } (<=\ t_{i1}\ t_{i2})\ (>=\ t_{i1}\ t_{i2}))\ (<\ t_{i3}\ t_{i4})\ (>\ t_{i3}\ t_{i4}))\ .$$

Secondly, SPASS-SATT applies the renamings in the greedy renaming CNF transformation polarity dependent. This means the greedy renaming CNF transformation transforms the subformulas f_i' into

$$\begin{aligned} f_i^g := (\text{and} \quad & (\text{or } p_i\ (<\ t_{i3}\ t_{i4})\ (>\ t_{i3}\ t_{i4})) \\ & (\text{or } (\text{not } p_i)\ (<=\ t_{i1}\ t_{i2})) \\ & (\text{or } (\text{not } p_i)\ (>=\ t_{i1}\ t_{i2})))\ , \end{aligned}$$

where the p_i are propositional variables freshly introduced for the renamings. For comparison, SPASS-SATT with the small CNF transformation would not apply any renamings to the subformulas f_i', but would apply the standard CNF transformation because it generates a smaller subformula.

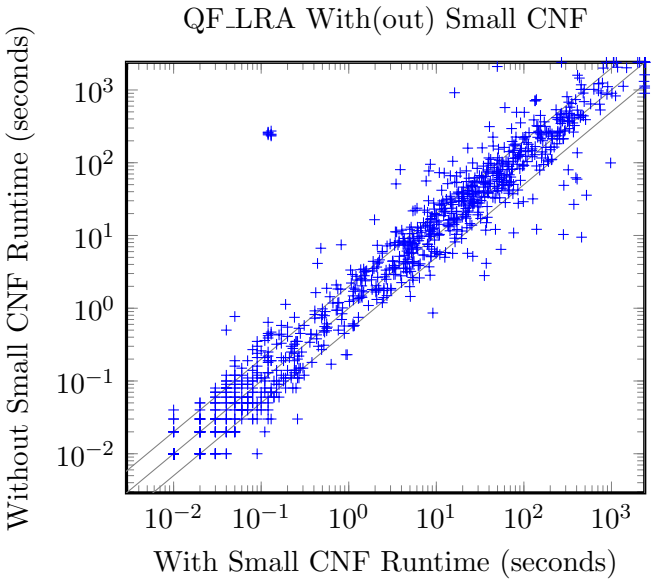


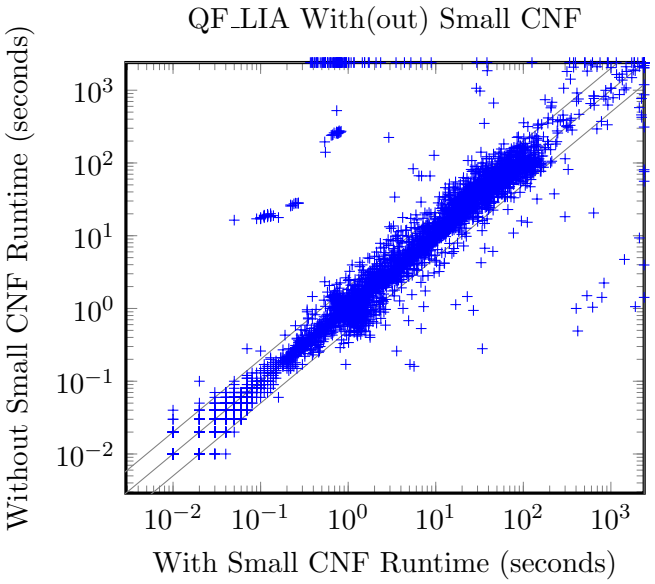
Figure 8.24: With(out) small CNF on the QF_LRA SMT-LIB benchmarks

This means the standard CNF transformation transforms the subformulas f_i' into

$$f_i^m := (\text{and} \quad (\text{or } (<= \ t_{i1}\ t_{i2})\ (<\ t_{i3}\ t_{i4})\ (>\ t_{i3}\ t_{i4})) \\ (\text{or } (>= \ t_{i1}\ t_{i2})\ (<\ t_{i3}\ t_{i4})\ (>\ t_{i3}\ t_{i4}))) \ .$$

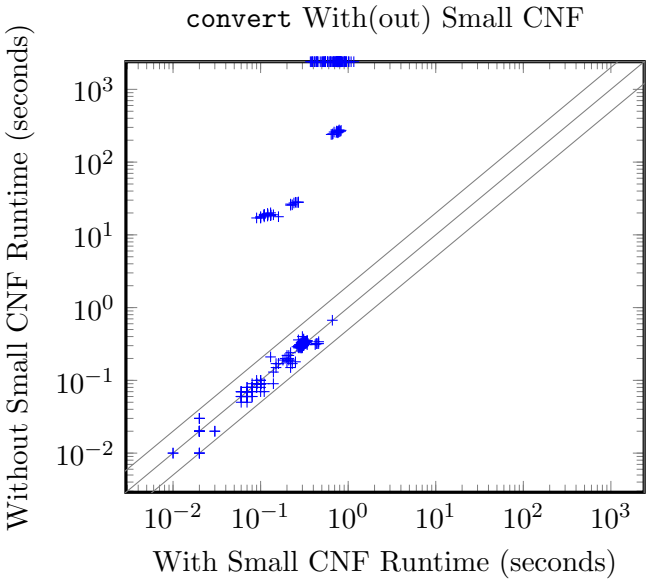
Naturally, f_i^m is better than f_i^g because it is smaller and does not contain any new propositional variables. But due to the polarity dependent renaming in f_i^g, f_i^m is also more expressive, i.e., there are potential unit propagations, which can be performed with f_i^m but that cannot be simulated with f_i^g.[22] We could fix this by performing the renamings in the greedy renaming CNF transformation polarity independent, but this would also mean that the resulting formula gets bigger.

[22]The only unit propagations, which can be performed with f_i^g but that cannot be simulate with f_i^m, propagate p_i or (not p_i), so literals that do not even occur in f_i^m.



SPASS-SATT with small CNF solves 6806 instances
SPASS-SATT without small CNF solves 6678 instances

Figure 8.25: With(out) small CNF on the QF_LIA SMT-LIB benchmarks



SPASS-SATT with small CNF solves 319 instances
SPASS-SATT without small CNF solves 218 instances

Figure 8.26: With(out) small CNF on the QF_LIA `convert` benchmark family

Chapter 9

Conclusion

We started this thesis by extending the incomplete CutSat calculus by Jovanović and de Moura [87]. The result was our CutSat++ calculus (Chapter 3), which is the first sound, complete, and terminating calculus for linear integer problems based on the model assumption and conflict learning approach motivated by CDCL style SAT solving. In addition to the techniques derived from CDCL style SAT solving, CutSat++ also uses a lazy quantifier elimination scheme with which it handles unbounded problems.

There is a reasonable gap between the CutSat++ calculus and an efficient implementation of it. Firstly, we found no intuitive way with which we could extend CutSat++ to linear mixed arithmetic. Secondly, many techniques that make CDCL-based SAT solving efficient are not straightforward to incorporate into CutSat++ due to the combination with the lazy quantifier elimination scheme. For example, the efficiency of CDCL-based SAT relies on a heuristic for picking decision variables that is refined dynamically during the runtime. For CutSat++, a similarly dynamic decision heuristic seems to be impossible because the *a priori* fixed variable order, which is used to guarantee termination, also has a lot of influence on the way CutSat++ picks variables for decisions. We assume that this becomes a problem for an efficient implementation because of observations of other calculi with similar limitations. For instance, we have observed a difference in efficiency between CDCL-based SAT solving and superposition-based SAT solving, which is also based on model assumptions and a fixed *a priori* ordering. Both calculi are guaranteed to terminate on the ground fragment of first-order logic. However, it seems that the lazy, dynamic and problem driven way CDCL-based SAT solving develops the variable order [137] is more efficient, in general. Currently, it is still an open problem whether CutSat++ can be further refined so that the ordering becomes dynamic and

the calculus still guarantees termination. Otherwise, we hope that heuristics motivated by quantifier elimination procedures may yield further insights in how to select the best possible *a priori* fixed variable order for a given input problem.

Alongside CutSat++, we developed additional ideas for handling unbounded problems. These ideas are all based on a deeper analysis of unbounded problems. For instance, unbounded problems can be intuitively partitioned into two categories: absolutely unbounded and partially bounded problems. Partially bounded problems contain, in fact, all of the hard unbounded problems and absolutely unbounded problems are trivial to detect and to solve with the right methods. This led us to the development of the fast cube tests (Chapter 4), which can detect and solve absolutely unbounded problems in polynomial time.

Our two fast cube test are named the *largest cube test* and the *unit cube test*. In contrast to many complete methods that search along the problem surface for a solution, these tests use cubes to explore the interior of the problem. The largest cube test finds a cube with maximum edge length contained in the rational solutions of the input problem, determines its rational valued center, and rounds it to a potential mixed/integer solution. The unit cube test determines instead whether the the rational solutions of the input problem contain a cube with edge length one, which is the minimal edge length that always guarantees that a cube contains a mixed/integer solution.

The tests are especially efficient on constraint systems with a large number of integer solutions, e.g., those that are absolutely unbounded. Inside the SMT-LIB benchmarks, we have found almost one thousand problem instances that are absolutely unbounded. Benchmark evaluations confirm that our tests are superior on these instances compared to several state-of-the-art SMT solvers (Section 4.5). As a result, other SMT solvers (e.g., MATHSAT and Z3) now also employ our unit cube tests [22, 41, 49].

Conceptually, it is also possible to generalize our tests so they search for any d-norm ball (where d is fixed) and not just cubes. However, the resulting d-norm ball tests are no longer guaranteed to be linear programs because they require inequality bounds that may contain roots. This means that we cannot directly solve general d-norm ball tests with our linear arithmetic solver but at best over-approximate them.

One major obstacle for a wider application of our cube tests are equalities. To resolve this obstacle, we developed several techniques for the investigation and removal of equalities (Chapter 5). These methods also have many additional applications besides simplifications for our cube tests. For instance, as a short cut in quantifier elimination procedures (Section 5.6) and as part of the Nelson-Oppen combination of theories (Section 5.4). But the most important application with regard to the main theme of this thesis is their capability to efficiently detect all (un)bounded directions (Section 5.5).

Solver	Rank	Correctly Solved Score	CPU time Score	Solved Instances
CVC4	1	1586.833	69.006	1566
SPASS-SATT	2	1586.396	64.292	1590
Yices 2.6.0	3	1583.186	63.901	1567
veriT	4	1568.212	79.840	1527
SMTInterpol	5	1548.476	102.257	1521
MathSAT	6	1536.458	107.673	1461
z3-4.7.1	7	1527.249	113.154	1435
opensmt2	8	1498.663	131.674	1329
Ctrl-Ergo	9	1450.082	172.097	1354
SMTRAT-Rat	10	1297.891	275.918	984
SMTRAT-MCSAT	11	1090.526	409.015	711

Figure 9.1: SMT-COMP 2018 results for QF_LRA (main track, sequential, benchmarks: 1649, time limit: 1200s) taken from `http://smtcomp.sourceforge.net/2018/results-QF_LRA.shtml`

Thanks to the efficient detection of (un)bounded directions, we were now also able to design a much better bounding transformation (Chapter 6), i.e., a transformation that reduces an unbounded problem to an equisatisfiable bounded problem. Bounding transformations are interesting because most linear mixed decision procedures, e.g., branch-and-bound, become terminating on bounded problems. Bounding transformations are, therefore, extensions that complete other decision procedures. Previous bounding transformations from the literature, e.g., *a priori* bounds [119], are inefficient with respect to actual implementations because they orient themselves only on the easiest to measure structural properties of the problem, e.g., the number of variables and the absolute size of constants. As a result, the transformed problems cannot be solved in reasonable time (e.g. the life time of earth) even for small unbounded problems. This is also the reason why previous bounding transformations cannot be found in any state-of-the-art linear arithmetic solver.

Our transformation orients itself on more significant structures: the bounded and unbounded inequalities in the input problem. To be more precise, our bounding transformation consists of two steps: First, we use the Double-Bounded reduction (Section 6.3) to eliminate all unbounded inequalities from our problem. Then we use the Mixed-Echelon-Hermite transformation (Section 6.2) to shift the variables in our problem to ones that are either bounded or do not appear in the new inequalities and are, therefore, eliminated. The result of these two steps is a bounded problem that is solvable in practice. Benchmark experiments provide further evidence to the efficiency of our bounding transformation in practice (Section 6.5). Moreover, we presented a polynomial method for converting certificates of (un)satisfiability from the transformed to the original problem.

Solver	Rank	Correctly Solved Score	CPU time Score	Solved Instances
SPASS-SATT	1	6587.626	72.048	6744
Ctrl-Ergo	2	6221.467	156.086	6259
MathSAT	3	6135.114	164.626	6528
SMTInterpol	4	5915.623	204.123	6286
CVC4	5	5891.019	194.986	6357
Yices 2.6.0	6	5867.976	209.452	6232
z3-4.7.1	7	5733.374	224.539	6195
SMTRAT-Rat	8	4049.914	515.394	3112
veriT	9	3155.162	295.434	2734

Figure 9.2: SMT-COMP 2018 results for QF_LIA (main track, sequential, benchmarks: 6947, time limit: 1200s) taken from `http://smtcomp.sourceforge.net/2018/results-QF_LIA.shtml`

We also integrated some of our new decision procedures into our linear arithmetic theory solver SPASS-IQ (Chapter 7). To be more precise, we integrated (i) the unit cube test, (ii) our method that efficiently detects all (un)bounded directions, which is based on our methods for equality investigation, and (iii) our bounding transformation consisting of the Double-Bounded reduction and the Mixed-Echelon-Hermite transformation. Moreover, we extended our theory solver SPASS-IQ into the CDCL(LA) implementation SPASS-SATT (Chapter 8).

Our implementations also allow us to confirm the practical efficiency of our methods through various benchmark experiments (Sections 4.5 & 6.5). As a byproduct, we were also able to perform an impact analysis (i) of various other techniques from linear arithmetic for SMT solving and (ii) of the most efficient ways of implementing these methods (Chapters 7 & 8). Moreover, we participated with SPASS-SATT in the 13th International Satisfiability Modulo Theories Competition (SMT-COMP 2018) and ranked first in the category QF_LIA (quantifier free linear integer arithmetic) and ranked second in the category QF_LRA (quantifier free linear rational arithmetic). In both categories, SPASS-SATT solved the largest number of problems in the shortest amount of time. (See Figures 9.1 and 9.2 for more details on the competition results.)

Bibliography

[1] SMT-COMP 2018 results for QF_LIA (main track). `http://smtcomp.sourceforge.net/2018/results-QF_LIA.shtml`.

[2] SMT-COMP 2018 results for QF_LRA (main track). `http://smtcomp.sourceforge.net/2018/results-QF_LRA.shtml`.

[3] The SPASS Workbench. `https://www.spass-prover.org/`.

[4] Gábor Alagi and Christoph Weidenbach. NRCL - A model building approach to the Bernays-Schönfinkel fragment. In *FroCos*, volume 9322 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2015.

[5] Ernst Althaus, Evgeny Kruglov, and Christoph Weidenbach. Superposition modulo linear arithmetic SUP(LA). In *FroCoS*, volume 5749 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2009.

[6] M. Baaz, U. Egly, and A. Leitsch. Normal form transformations. In *Handbook of Automated Reasoning*, pages 275–333. Elsevier and MIT Press, 2001.

[7] Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In *CADE*, volume 449 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 1990.

[8] Erwin H. Bareiss. Sylvester's identity and multistep integer-preserving Gaussian elimination. *Mathematics of Computation*, 22(103):565–578, 1968.

[9] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[11] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, 2006.

[12] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT 2010*, 2010.

[13] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[14] Peter Baumgartner and Uwe Waldmann. Hierarchic superposition with weak abstraction. In *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2013.

[15] E. M. L. Beale. An alternative method for linear programming. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 50, pages 513–523. Cambridge University Press, 1954.

[16] Leonard Berman. Precise bounds for Presburger arithmetic and the reals with addition: Preliminary report. In *FOCS*, pages 95–99. IEEE Computer Society, 1977.

[17] Leonard Berman. The complexity of logical theories. *Theoretical Computer Science*, 11(1):71–77, 1980.

[18] Livio Bertacco, Matteo Fischetti, and Andrea Lodi. A feasibility pump heuristic for general mixed-integer problems. *Discrete Optimization*, 4(1):63–76, 2007.

[19] Etienne Bézout. *Théorie générale des équations algébriques.* de l'imprimerie de Ph.-D. Pierres, rue S. Jacques, 1779.

[20] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[21] Nikolaj Bjørner. *Integrating Decision Procedures for Temporal Verification.* PhD thesis, Stanford, CA, USA, 1999.

[22] Nikolaj Bjorner and Lev Nachmanson. Theorem recycling for theorem proving. In Laura Kovács and Andrei Voronkov, editors, *Vampire 2017. Proceedings of the 4th Vampire Workshop*, volume 53 of *EPiC Series in Computing*, pages 1–8. EasyChair, 2018.

[23] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[24] Jasmin Christian Blanchette, Andrei Popescu, Daniel Wand, and Christoph Weidenbach. More SPASS with Isabelle - superposition with hard sorts and configurable simplification. In *ITP*, volume 7406 of *Lecture Notes in Computer Science*, pages 345–360. Springer, 2012.

[25] Robert G. Bland. New finite pivoting rules for the simplex method. *Math. Oper. Res.*, 2(2):103–107, 1977.

[26] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic. In *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2012.

[27] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[28] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[29] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free presburger arithmetic. In *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 384–399. Springer, 2010.

[30] Martin Bromberger. A reduction from unbounded linear mixed arithmetic problems into bounded problems. In *IJCAR*, volume 10900 of *Lecture Notes in Computer Science*, pages 329–345. Springer, 2018.

[31] Martin Bromberger, Mathias Fleury, Simon Schwarz, and Christoph Weidenbach. SPASS-SATT a CDCL(LA) solver. In *CADE-27*, Lecture Notes in Computer Science. Springer, 2019. (Forthcoming).

[32] Martin Bromberger, Thomas Sturm, and Christoph Weidenbach. Linear integer arithmetic revisited. In *CADE*, volume 9195 of *Lecture Notes in Computer Science*, pages 623–637. Springer, 2015.

[33] Martin Bromberger and Christoph Weidenbach. Computing a complete basis for equalities implied by a system of LRA constraints. In *SMT@IJCAR*, volume 1617 of *CEUR Workshop Proceedings*, pages 15–30. CEUR-WS.org, 2016.

[34] Martin Bromberger and Christoph Weidenbach. Fast cube tests for LIA constraint solving. In *IJCAR*, volume 9706 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2016.

[35] Martin Bromberger and Christoph Weidenbach. New techniques for linear arithmetic: cubes and equalities. *Formal Methods in System Design*, 51(3):433–461, 2017.

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

[37] Jerry R. Burch. Techniques for verifying superscalar microprocessors. In *DAC*, pages 552–557. ACM Press, 1996.

[38] Ricardo Caferra, Alexander Leitsch, and Nicholas Peltier. *Automated Model Building*, volume 31 of *Applied Logic Series*. Springer Netherlands, 2004.

[39] Jürgen Christ and Jochen Hoenicke. Cutting the mix. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2015.

[40] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012.

[41] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[42] Michael Codish, Yoav Fekete, Carsten Fuhs, Jürgen Giesl, and Johannes Waldmann. Exotic semi-ring constraints. In *SMT@IJCAR*, volume 20 of *EPiC Series in Computing*, pages 88–97. EasyChair, 2012.

[43] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[44] David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.

[45] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 360–368. Springer, 2015.

[46] George B. Dantzig. *Maximization of a Linear Function of Variables Subject to Linear Inequalities, in Activity Analysis of Production and Allocation*, chapter XXI. Wiley, New York, 1951.

[47] George B. Dantzig. A history of scientific computing. In *Origins of the Simplex Method*, pages 141–151. ACM, 1990.

[48] Mahlon M. Day. Normed linear spaces. In *Normed Linear Spaces*, pages 27–52. Springer, 1973.

[49] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[50] Leonardo Mendonça de Moura and Nikolaj Bjørner. Applications and challenges in satisfiability modulo theories. In *WING@ETAPS/IJCAR*, volume 1 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[51] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[52] Isil Dillig, Thomas Dillig, and Alex Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2009.

[53] Yann Disser and Martin Skutella. The simplex algorithm is NP-mighty. *ACM Trans. Algorithms*, 15(1):1–19, 2019.

[54] Andreas Dolzmann, Thomas Sturm, and Volker Weispfenning. Real quantifier elimination in practice. In *Algorithmic Algebra and Number Theory*, pages 221–247. Springer, 1999.

[55] Ioan Dragan, Konstantin Korovin, Laura Kovács, and Andrei Voronkov. Bound propagation for arithmetic reasoning in Vampire. In *SYNASC*, pages 169–176. IEEE Computer Society, 2013.

[56] Bruno Dutertre. Yices 2.2. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[57] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006. Extended version: Integrating simplex with DPLL(T). Tech. rep., CSL, SRI INTERNATIONAL (2006).

[58] Matthias Ehrgott. Scalarization techniques. In *Multicriteria Optimization*, pages 97–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[59] Xin Gui Fang and George Havas. On the worst-case complexity of integer Gaussian elimination. In *ISSAC*, pages 28–31. ACM, 1997.

[60] Julius Farkas. Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik*, 124:1–27, 1902.

[61] Germain Faure, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. SAT modulo the theory of linear arithmetic: Exact, inexact and commercial solvers. In *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2008.

[62] Jeanne Ferrante and Charles Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Comput.*, 4(1):69–76, 1975.

[63] Jeanne Ferrante and Charles W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *LNM*. Springer, 1979.

[64] Arnaud Fietzke and Christoph Weidenbach. Superposition as a decision procedure for timed automata. *Mathematics in Computer Science*, 6(4):409–425, 2012.

[65] Michael J Fischer and Michael O Rabin. Super-exponential complexity of Presburger arithmetic. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 122–135. Springer, 1998.

[66] Matteo Fischetti, Fred W. Glover, and Andrea Lodi. The feasibility pump. *Math. Program.*, 104(1):91–104, 2005.

[67] Martin Fürer. The complexity of Presburger arithmetic with bounded quantifier alternation depth. *Theor. Comput. Sci.*, 18:105–111, 1982.

[68] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[69] Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. In *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2010.

[70] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, July 2018.

[71] Fred Glover and Manuel Laguna. General purpose heuristics for integer programming - part I. *J. Heuristics*, 2(4):343–358, 1997.

[72] Fred Glover and Manuel Laguna. General purpose heuristics for integer programming-part II. *J. Heuristics*, 3(2):161–179, 1997.

[73] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs *and* an algorithm for the mixed integer problem. In *50 Years of Integer Programming*, pages 77–103. Springer, 2010.

[74] Erich Grädel. *The complexity of subclasses of logical theories.* PhD thesis, Univ. Basel, 1987.

[75] Torbjrn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2016. Version 6.1.2, `http://gmplib.org/`.

[76] Alberto Griggio. A practical approach to satisability modulo linear integer arithmetic. *JSAT*, 8(1/2):1–27, 2012.

[77] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*, 2018. `http://www.gurobi.com`.

[78] Peter L Hammer and Sergiu Rudeanu. *Boolean methods in operations research and related areas*, volume 7 of *Econometrics and Operations Research*. Springer Science & Business Media, 2012.

[79] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*, 2013. Version 2.4.0, `http://flintlib.org`.

[80] W. B. Hart. Fast library for number theory: An introduction. In *Proceedings of the Third International Congress on Mathematical Software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[81] Pascal Van Hentenryck and Thomas Graf. Standard forms for rational linear arithmetic in constraint logic programming. *Ann. Math. Artif. Intell.*, 5(2-4):303–319, 1992.

[82] Frederick S. Hillier. Efficient heuristic procedures for integer linear programming with an interior. *Operations Research*, 17(4):600–637, 1969.

[83] Frederick S. Hillier and Gerald J. Lieberman. *Duality Theory and Sensitivity Analysis*. McGraw-Hill Companies, seventh edition, 2001.

[84] Zhi-yue Huang and Bin-wu He. Volume of unit ball in an n-dimensional normed space and its asymptotic properties. *Journal of Shanghai University (English Edition)*, 12(2):107–109, 2008.

[85] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 301–306. Springer, 2005.

[86] Dejan Jovanović and Leonardo Mendonça de Moura. Cutting to the chase solving linear integer arithmetic. In *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2011. Extended version: Cutting to the chase - solving linear integer arithmetic. J. Autom. Reasoning, 51(1), (2013).

[87] Dejan Jovanović and Leonardo Mendonça de Moura. Cutting to the chase - solving linear integer arithmetic. *J. Autom. Reasoning*, 51(1):79–108, 2013.

[88] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *CP*, volume 1118 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 1996.

[89] Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors. *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. Springer, 2010.

[90] Ravi Kannan and László Lovász. Covering minima and lattice point free convex bodies. In *FSTTCS*, volume 241 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 1986.

[91] Ravindran Kannan and Achim Bachem. Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix. *SIAM J. Comput.*, 8(4):499–507, 1979.

[92] Ravindran Kannan and Clyde L. Monma. On the computational complexity of integer programming problems. In *Optimization and Operations Research*, volume 157 of *Lecture Notes in Economics and Mathematical Systems*, pages 161–172. Springer, 1978.

[93] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

[94] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[95] Leonid G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[96] Hyondeuk Kim, Fabio Somenzi, and HoonSang Jin. Efficient term-ite conversion for satisfiability modulo theories. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2009.

[97] Tim King, Clark Barrett, and Bruno Dutertre. Simplex with sum of infeasibilities for SMT. In *FMCAD*, pages 189–196. IEEE, 2013.

[98] Victor Klee and George J. Minty. How good is the simplex algorithm? *Inequalities, III (Proc. Third Sympos., Univ. California, Los Angeles, Calif., 1969; dedicated to the memory of Theodore S. Motzkin)*, pages 159–175, 1972.

[99] J. W. Klop, Marc Bezem, and R. C. De Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, New York, NY, USA, 2001.

[100] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 5th edition, 2012.

[101] Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming*, pages 105–132. Springer, 2010.

[102] Aless Lasaruk and Thomas Sturm. Weak quantifier elimination for the full linear theory of the integers. A uniform generalization of Presburger arithmetic. *Applicable Algebra in Engineering, Communication and Computing*, 18(6):545–574, 2007.

[103] C. E. Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1(1):36–47, 1954.

[104] John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989, 1963.

[105] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *Comput. J.*, 36(5):450–462, 1993.

[106] Andrew Makhorin. *GNU Linear Programming Kit (GLPK)*, 2018. Version 4.6.5, `http://www.gnu.org/software/glpk/glpk.htm`.

[107] Jiří Matoušek and Bernd Gärtner. Duality of linear programming. In *Understanding and Using Linear Programming*, pages 81–104. Springer, 2007.

[108] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[109] John E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. In *Handbook of Applied Optimization*, pages 65–77. Oxford University Press, 2002.

[110] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.

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

[112] Robert Nieuwenhuis. The IntSat method for integer linear programming. In *CP*, volume 8656 of *Lecture Notes in Computer Science*, pages 574–589. Springer, 2014.

[113] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2005.

[114] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[115] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In *Handbook of Automated Reasoning*, pages 335–367. Elsevier and MIT Press, 2001.

[116] Keith B. Oldham, Jan Myland, and Jerome Spanier. *An atlas of functions: with equator, the atlas function calculator*. Springer Science & Business Media, 2010.

[117] Derek C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *J. Comput. Syst. Sci.*, 16(3):323–332, 1978.

[118] Derek C. Oppen. Complexity, convexity and combinations of theories. *Theor. Comput. Sci.*, 12:291–302, 1980.

[119] Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.

[120] Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *J. Autom. Reasoning*, 44(4):401–424, 2010.

[121] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du premier congres de Mathematiciens des Pays Slaves*, pages 92–101, Warsaw, Poland, 1929.

[122] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *SC*, pages 4–13. ACM, 1991.

[123] Philippe Refalo. Approaches to the incremental detection of implicit equalities with the revised simplex method. In *PLILP/ALP*, volume 1490 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 1998.

[124] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett, and Morgan Deters. Refutation-based synthesis in SMT. *Formal Methods in System Design*, pages 1–30, 2017.

[125] Harald Rueß and Natarajan Shankar. Solving linear arithmetic constraints. Technical report, SRI International, Computer Science Laboratory, 2004.

[126] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2008.

[127] Martin W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *INFORMS Journal on Computing*, 6(4):445–454, 1994.

[128] Alexander Schrijver. *Theory of linear and integer programming.* Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[129] Roberto Sebastiani. Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224, 2007.

[130] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[131] Thomas Sturm. Real quadratic quantifier elimination in risa/asir. Technical report, ISIS-RM-5E, Fujitsu Laboratories Ltd., 1996.

[132] Jan Telgen. Identifying redundant constraints and implicit equalities in systems of linear constraints. *Management Science*, 29(10):1209–1222, 1983.

[133] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[134] Joachim von zur Gathen and Malte Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proceedings of the American Mathematical Society*, 72(1):155–158, 1978.

[135] Dominik Wagner. Design and implementation of a CDCL(LA) calculus. Bachelor thesis, Universität des Saarlandes, Saarbrücken, 2017.

[136] Chris Wallace. ZI round, a MIP rounding heuristic. *J. Heuristics*, 16(5):715–722, 2010.

[137] Christoph Weidenbach. Automated reasoning building blocks. In *Correct System Design*, volume 9360 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 2015.

[138] Volker Weispfenning. The complexity of almost linear diophantine problems. *J. Symb. Comput.*, 10(5):395–404, 1990.

[139] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

Index