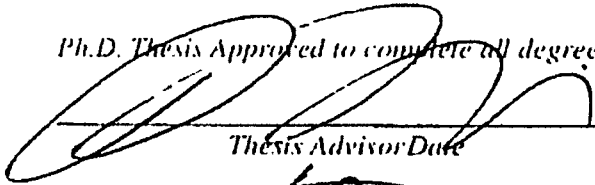


NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS APPROVAL FORM


THESIS TITLE: Mathematical Programming Modulo Theories

AUTHOR: Vasilis Papavasileiou

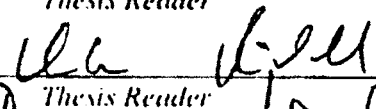
Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.


Thesis Advisor

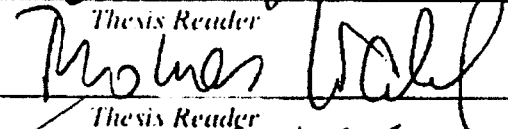
5/14/2015
Date


Thesis Reader

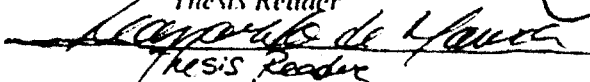
5/14/2015
Date


Thesis Reader

5/14/2015
Date

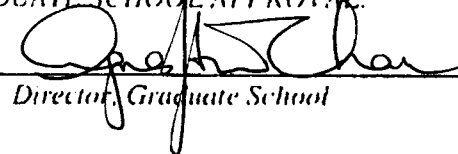

Thesis Reader

5/14/2015
Date


Thesis Reader

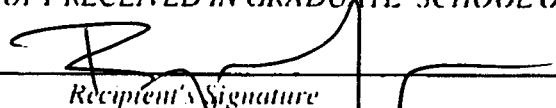
5/14/2015
Date

GRADUATE SCHOOL APPROVAL:


Director, Graduate School

5/26/2015
Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:


Recipient's Signature

5/27/2015
Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

Mathematical Programming Modulo Theories

A dissertation presented by

Vasilis Papavasileiou

to the College of Computer and Information Science,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy.

Northeastern University

Boston, Massachusetts

May, 2015

Abstract

We present the Mathematical Programming Modulo Theories (MPMT) constraint solving framework. MPMT enhances Mathematical Programming by integrating techniques from the field of Automated Reasoning, *e.g.*, solvers for first-order theories. We develop the Branch and Cut Modulo T (BC(T)) architecture for MPMT solvers. BC(T) is formalized as a family of transition systems. We outline classes of first-order theories that BC(T) accommodates. We finally discuss the algorithmic aspects of our BC(T) implementation, and explore potential applications.

Mathematical Programming Modulo Theories

Contents

Introduction	1
1 (Mixed) Integer Linear Programming	7
1.1 Integer Linear Programming	8
1.2 Simplex for Linear Programming	11
1.3 Cutting Planes	12
1.4 Branch-and-Cut	14
1.5 An Application in Aerospace	16
1.6 Experiments	18
2 Automated Reasoning	21
2.1 First-Order Theories	22
2.2 Linear Integer Arithmetic	25
2.3 Combining Theories	26
2.4 Satisfiability Modulo Theories	29
2.5 Related Work	34
3 (M)ILP Meets Theories	37
3.1 MPMT by Example	38
3.2 MPMT Instances and Solvers	40
4 BC(T): An Abstract View	45
4.1 Preliminaries	46
4.2 The Transition System \mathcal{G}	48
4.3 The Transition System \mathcal{T}	51
4.4 Soundness	54

5	Nelson-Oppen via $BC(T)$	59
5.1	Arrangement Search and Optimization	60
5.2	Formal Preliminaries	61
5.3	The Transition System \mathcal{A}	63
5.4	Termination	70
6	$BC(T)$: An Algorithmic View	79
6.1	Preliminaries	80
6.2	High-Level Functions	81
6.3	Propagation	83
6.4	Enforcing Continuous Relaxations	85
6.5	Branching	87
6.6	Discussion	89
7	Propositional Structure	91
7.1	Flattening Propositional Structure	92
7.2	Encoding Indicators	93
7.3	Symbolically Handling Indicators	99
7.4	DPLL(T)-Style Solving	100
7.5	Experiments	104
8	Local Theory Extensions	109
8.1	Preliminaries	110
8.2	Axiom Instantiation	112
8.3	Implementation	116
8.4	Experiments	117
9	Data	121
9.1	Motivating Example	122
9.2	The Logic Δ	124
9.3	The Existential Fragment of Δ	128
9.4	$BC(T)$ for Δ	131
9.5	Applications and Experiments	138
9.6	Related Work	142

CONTENTS

Conclusions and Future Work	143
------------------------------------	------------

List of Transition Rules

Branch	48
Learn	49
Forget	49
Drop	49
Prune	50
Improve	50
Unbounded	50
\mathcal{T} -Learn	51
\mathcal{T} -Improve	52
\mathcal{T} -Unbounded	52
\mathcal{A} -Propagate ⁺	63
\mathcal{A} -Propagate ⁻	64
\mathcal{A} -Forget	64
\mathcal{A} -Branch	65
\mathcal{A} -Unsat	65
\mathcal{B} -Propagate ⁺	66
\mathcal{B} -Propagate ⁻	66
\mathcal{B} -Branch	66
\mathcal{C} -Learn	102
\mathcal{L} -Instantiate	113
\mathcal{L} -Branch	114

Acknowledgments

I would like to thank my advisor, Pete Manolios, for his support and guidance over the years. Thanks to Eugene Goldberg, Leonardo de Moura, Mirek Riedewald, and Thomas Wahl for serving on my committee. Thanks to my undergraduate advisor, Nikos Papaspyrou, who encouraged me to continue my studies. Thanks to my labmates, Konstantinos Athanasiou, Harsh Raju Chamarthi, Mitesh Jain, Peizun Liu, Jorge Pais, and Jaideep Ramachandran, for their helpful suggestions and encouragement. Last but not least, I am grateful to my parents, Aleka and Pavlos, and my sister, Despoina, for their love and support.

Mathematical Programming Modulo Theories

Introduction

This dissertation describes a *constraint solving* framework called *Mathematical Programming Modulo Theories (MPMT)*. Constraint solving is the process of automatically finding a solution to a set of requirements (constraints), expressed in a formal language. The framework that we describe provides optimization capabilities, *i.e.*, it is possible to express an *objective function* that measures the quality of solutions, and to ask for the best possible solution that satisfies all constraints. Our framework incorporates techniques from two distinct research fields that encompass constraint solving and optimization: *Mathematical Programming (MP)* and *Automated Reasoning (AR)*.

Mathematical Programming emphasizes optimization problems. In MP, the constraints typically involve arithmetic over real or integer variables. The emergence of MP can be pinpointed to Dantzig’s work on the *Simplex* algorithm for *Linear Programming (LP)* [21]. In LP, the constraints are linear inequalities over real variables, while the objective functions are linear combinations of the variables. While work on LP predates Dantzig’s Simplex [32, 23, 49], it was Dantzig’s work that initiated the wide-spread use of LP for real-world optimization problems. The Simplex algorithm was extended by Gomory (with what is known as *cutting planes*) to deal with integer variables [38]. Gomory’s techniques give rise to *Integer Linear Programming (ILP)*, which allows only integer variables, and *Mixed Integer Linear Programming (MILP)*, which allows both real and integer variables. LP and (M)ILP remain the most established forms of MP. The discipline of *Operations Research (OR)* applies MP solvers [1, 2, 3] to a variety of problems, including production planning [46], scheduling [11], network design [43], and vehicle routing [33].

First-Order Logic (FOL) [30] (which is the most dominant system of formal logic) provides a general framework for expressing constraints. FOL is undecidable, *i.e.*, it is theoretically impossible to build complete solvers for the whole logic. In recent decades, the field of Automated Reasoning emerged, providing decision procedures for fragments of FOL [72, 82, 73]. The advent of the *Conflict-Driven Clause Learning (CDCL)* scheme [83] for propositional satisfiability (SAT) in the mid-nineties, and its subsequent extension for problems with non-propositional atoms [7, 26, 77], *i.e.*, for *Satisfiability Modulo Theories (SMT)* problems, enabled a new generation of solvers that serve as the underlying technology for static analysis [5], test-case generation [37], and other applications. Much of the work on AR nowadays happens under the SMT banner.

The separation between AR and MP is not absolute, *e.g.*, the fragments of arithmetic that arise in SMT (and the algorithms that tackle them) [27, 28, 12, 41] are closely related to LP and (M)ILP. More generally, the AR and MP families of solvers share key operations, *e.g.*, search, learning, and propagation. The AR and MP families of solvers nevertheless target largely disjoint sets of applications, and reflect different design goals. We believe that there is fertile ground for solvers that combine the strengths of AR and MP. This possibility has so far received some attention, *e.g.*, in the direction of improving the performance of SMT solvers by integrating MP engines [31, 69, 10, 53]. We are nevertheless not aware of any effort towards comprehensively integrating AR and MP techniques.

MPMT aims to fill this gap. In MPMT, a core MILP solver (hence the MP part) communicates with satellite solvers for first-order *theories* (*i.e.*, the kind of procedures studied in AR). A theory is a set of *axioms* that assign meaning to an alphabet (*signature*), thus introducing a custom constraint language. MPMT targets applications for which an MP-based approach seems appropriate, but part of the requirements are hard or impossible to encode by means of linear (in)equalities. MPMT accommodates such problems by allowing non-arithmetic requirements to be modeled with the help of non-arithmetic, theory-constrained symbols. Conversely, MPMT does not simply tackle the union of the sets of problems that MP and AR al-

ready tackle. The synergy of MP and AR rather allows us to tackle problems that were previously beyond the reach of either family of solvers.

The $BC(T)$ Architecture for MPMT

Our work provides the general $BC(T)$ (*Branch and Cut Modulo T*) architecture for solving MPMT instances. $BC(T)$ builds upon the *Branch-and-Cut* (B&C) family of algorithms [68]. B&C algorithms combine *Branch-and-Bound* (B&B) search with Gomory-style cutting planes, and thus indirectly with Simplex. This scheme is very general and matches the architecture of most MILP solvers that are potentially applicable for MPMT. $BC(T)$ extends the established variants of B&C by plugging in theory solvers.

We describe $BC(T)$ as a family of *transition systems*. In our context, a transition system is a set of transition *rules* that relate *states*. A state abstractly describes a solver’s knowledge at a particular point in time, and the solver’s progress towards a solution. It is common to describe solver architectures as transition systems. The $DPLL(T)$ transition system for SMT is a notable example [77]. A transition system facilitates theoretical analysis by abstracting away implementation details of particular solvers.

$BC(T)$ strives for generality. In fact, with any first order theory as T , $BC(T)$ guarantees soundness, *i.e.*, $BC(T)$ cannot lead to wrong answers. However, obtaining completeness (*i.e.*, the guarantee of terminating with a solution for every instance in a fragment) and efficiency introduces requirements on T . We study classes of theories for which the $BC(T)$ architecture can be instantiated in a complete way. Building on the established *Nelson-Oppen* (NO) scheme for combining decision procedures [73, 86, 58], we demonstrate that $BC(T)$ can be applied in a complete way for background theories that (a) accept interpretations with infinite domains (*i.e.*, *stably-infinite* theories), and (b) whose vocabularies do not overlap with that of Linear Integer Arithmetic (LIA). Our NO-based techniques provide support for a broad range of relevant theories. We extend the reach of MPMT to certain theories that do not meet the requirements of NO, by adapting *local theory extensions* [84] to our framework.

Applications

The development of MPMT is strongly tied to applications. Our original motivation for MPMT comes from our work on a class of design problems [62, 59, 44] arising during the development of the Boeing 787 Dreamliner. The problems involve many classes of requirements which can be expressed with integer linear constraints, thus pointing us towards an ILP-centric approach. A class of hard real-time constraints requires specialized non-ILP techniques, and thus necessitates extending the ILP core. The combination of ILP with a custom decision procedure for real-time constraints [44] allowed us to algorithmically create architectural models, a task that according to Boeing engineers previously required “*the cooperation of multiple teams of engineers working over long periods of time.*” This combination of procedures amounts to an early instance of MPMT that predates $BC(T)$. This dissertation reviews our work with Boeing in order to provide motivation for the subsequent development of MPMT.

More recently, we have applied MPMT and $BC(T)$ to database analysis. By integrating a procedure that performs database-like queries as the backend procedure in $BC(T)$, our work enables a reasoning paradigm that takes into account both symbolic constraints (e.g., linear inequalities as in MP) and data organized in tables. In addition to enabling database analysis per se, the combination of MP and data is also of interest from an AR point of view. Namely, we outline a class of constraints that are not formalized as a first-order theory, which we can nevertheless treat much like a theory and thus accommodate in a theory combination mechanism. We also demonstrate that our techniques allow for domain-specific extensions, thus enabling advances in different fields. Finally, we demonstrate how a B&C-based solver can be customized for the kind of search required in a specific domain, and thus serve as a reusable core for diverse solvers.

The publicly available¹ Inez constraint solver implements the $BC(T)$ architecture, and most of the techniques described in this dissertation. Inez is built on top of the SCIP MP framework [3], and is mostly written in OCaml.

¹<https://github.com/vasilisp/inez>

Dissertation Structure

The first few chapters of this dissertation provide the necessary background. We clearly draw from background work both in MP, as discussed in Chapter 1, and in AR, as reviewed in Chapter 2. Chapter 3 provides a bridge between AR and MP by defining the MPMT formalism, thus laying the groundwork for the subsequent development of $BC(T)$.

We discuss $BC(T)$ from different perspectives. Chapters 4 and 5 discuss $BC(T)$ as a transition system, while Chapter 6 follows a more algorithmic view. Chapter 4 provides our most general transition system, while Chapter 5 provides one specialized for stably-infinite theories. The latter transition system can be thought of as a *strategy* that the former and more general system has sufficient power to follow. We also discuss transition systems in Chapters 7 and 8. The more algorithmically-minded reader may read Chapter 6 before (and independently of) Chapters 4 and 5.

Chapters 7 to 9 describe extensions on top of the core $BC(T)$ architecture. Specifically, Chapter 7 discusses how we handle propositional structure; Chapter 8 describes how local theory extensions fit in $BC(T)$; finally, Chapter 9 explains how we integrate database-like operations in $BC(T)$. What these extensions have in common is that they all involve only unobtrusive changes to the $BC(T)$ architecture, or no modifications at all, *i.e.*, only appropriate background procedures.

We provide a range of experiments. Chapter 1 experimentally demonstrates that our Boeing-provided problems necessitate extending MILP. Chapter 7 provides an experimental comparison against SMT solvers on SMT-LIB [8] instances. Chapter 8 reports on experiments with local theory extensions. Finally, Chapter 9 experimentally evaluates our techniques for constraint solving in the presence of data. The experiments in Chapters 7, 8, and 9 rely on lnez.

Chapter 1

(Mixed) Integer Linear Programming

This chapter provides the necessary background in Mathematical Programming (MP), and also discusses our motivation for developing a MP framework that can be extended with other procedures.

MPMT is architected around a MILP core. This is a decision driven by the primacy of linear constraints in the applications that MPMT targets. The Branch and Cut (B&C) algorithmic framework empowers most practical MILP solvers (*e.g.*, CPLEX [1], Gurobi [2], and SCIP [3]), and also constitutes the core of the $BC(T)$ framework that we develop in subsequent chapters. We discuss the algorithms behind B&C as an intermediate step towards exposition of $BC(T)$. B&C builds upon fundamental techniques for LP (notably, the Simplex algorithm [21]) and MILP (notably, cutting planes [38]), which we discuss at a high level.

The driving force behind the development of powerful MILP solvers is their usefulness in Operations Research. Our motivation for MPMT also comes from applications, and in particular from our collaboration with Boeing on design problems arising during the development of the Boeing 787 Dreamliner. We describe these problems, in order to provide an example of a practical application of MILP, and also to motivate our effort to extend MP. As we demonstrate, most of the constraints that arise are natural to encode in MILP. In fact, the structure of the resulting MILP instances is

close to OR-style problems that MILP solvers are optimized for. Nevertheless, the instances involve additional requirements that necessitate non-MP techniques, which led us to introduce a combination of procedures.

1.1 Integer Linear Programming

ILP encompasses problems where the requirements are expressed as linear inequalities over integer variables, and the objective function is linear. ILP instances are of the form

$$\begin{aligned}
& \textbf{minimize} \\
& o_0 \cdot v_0 + o_1 \cdot v_1 + \cdots + o_{n-1} \cdot v_{n-1} \\
& \textbf{subject to} \\
& r_{0,0} \cdot v_0 + \cdots + r_{0,n-1} \cdot v_{n-1} \leq s_0 \\
& r_{1,0} \cdot v_0 + \cdots + r_{1,n-1} \cdot v_{n-1} \leq s_1 \\
& \quad \vdots \\
& r_{m-1,0} \cdot v_0 + \cdots + r_{m-1,n-1} \cdot v_{n-1} \leq s_{m-1} \\
& v_0, \dots, v_{n-1} \in \mathbb{Z},
\end{aligned} \tag{1.1}$$

where v_i are variables, and the coefficients o_i , $r_{j,i}$, and s_j are integer.

An ILP instance in *standard form* is one of the form given in Equation 1.1, with the additional set of constraints $v_i \geq 0$, $i \in [0, n - 1]$. Any ILP instance (as per Equation 1.1) can be translated to one in standard form by replacing each variable v_i with $u_i - u'_i$, where u_i and u'_i are two fresh non-negative integer variables ($u_i, u'_i \geq 0$).

We subsequently formalize the syntactic constructs that are needed to encode instances of the above form. Throughout this dissertation, we assume that the variables that appear in linear constraints belong in a fixed countably infinite set of variable symbols \mathcal{V} . We also assume a strict total order $<$ over \mathcal{V} .

Definition 1 (Integer Linear Sum). *An integer linear sum is an expression of the form*

$$r_0 \cdot v_0 + \cdots + r_{n-1} \cdot v_{n-1},$$

Chapter 1. (Mixed) Integer Linear Programming

where

- r_i are integer constants,
- v_i are variable symbols in \mathcal{V} ,
- $r_i \neq 0$ for every $i \in [0, n-1]$, and
- $v_i \prec v_j$ for every pair i, j such that $0 \leq i < j < n$.

We explicitly allow the case $n = 0$, in which case the sum consists of 0 terms, i.e., it is equal to the constant 0.

Definition 2 (Integer Linear Constraint). *An integer linear constraint is a constraint of the form $e \leq r$, where e is an integer linear sum and r is an integer constant.*

Given the prevalence of integer linear constraints in MPMT, we simply refer to them as *constraints* when this does not cause ambiguity. Whenever we refer to a set of integer linear constraints, we imply that the set is implicitly conjoined. We use propositional connectives over integer linear constraints and sets thereof as appropriate. The duality between sets of integer linear constraints (to which set operators apply) and conjunctions of integer linear constraints (that justify the use of logical operators) also applies to other kinds of constraints that appear later in this dissertation.

We deviate from the strict syntax given in Definitions 1 and 2 as appropriate, e.g., we use operators like $=$ and $<$, we negate integer linear constraints (where the negation is itself an integer linear constraint), and we use terms of the form $r \cdot v$ (where r is an integer constant and v is a variable symbol) both left and right of a comparison operator. In any context where we use such forms, it is clear that we can easily obtain integer linear sums and constraints in the strict sense. An equality (which we are careful not to negate) clearly translates to a pair of inequalities.

Definition 3 (ILP Instance). *An Integer Linear Programming (ILP) instance is a pair of the form C, O , where C is a finite set of integer linear constraints, and the objective function O is an integer linear sum.*

Solutions to ILP instances come in the form of *assignments* that map the variables in \mathcal{V} to integer values.

Definition 4 (Integer Assignment). *An integer assignment A is a function $\mathcal{V} \rightarrow \mathbb{Z}$, where \mathbb{Z} is the set of all integers.*

Note that while ILP instances only contain finitely many variables, an assignment A assigns values to the infinitely many variables in \mathcal{V} . This is to keep the domain of assignments (and thus Definition 4) independent of any specific ILP instance. When we use an assignment A in connection to a specific instance C, O , the variables in \mathcal{V} that do not appear in either C or O may take arbitrary values (e.g., a default value that could be 0) without consequences.

We say that an assignment A *satisfies* the constraint $c = [r_0 \cdot v_0 + \dots + r_{n-1} \cdot v_{n-1} \leq s]$ (where every v_i is in \mathcal{V}) if $\sum_i r_i \cdot A(v_i) \leq s$. An assignment satisfies a set of constraints C if it satisfies every constraint $c \in C$. A set of constraints C is *integer-satisfiable* or *integer-consistent* if some assignment A satisfies C . Otherwise, C is called *integer-unsatisfiable* or *integer-inconsistent*.

It is sometimes the case that an ILP instance accepts solutions with arbitrarily low values for its objective function; the instance is *unbounded*. Whenever we obtain such a solution to an ILP instance, it is useful to record the information that it is just one among a family of arbitrarily good solutions. We do this by annotating an assignment with the superscript $-\infty$, e.g., $A^{-\infty}$. This amounts to assignments carrying along a binary flag that tells us whether they are unbounded or not.

The notation $O(A)$, where O is an integer linear sum and A is an integer assignment, is defined as follows. First, we define $O(A^{-\infty}) = -\infty$. For A that does not have the superscript $-\infty$ and for O of the form $\sum_i r_i \cdot v_i$, $O(A) = \sum_i r_i \cdot A(v_i)$. We only ever use the notation $O(A)$ for integer linear sums that are used as objective functions.

Definition 5 (Optimal Integer Assignment for ILP). *We say that an integer assignment A is optimal for the ILP instance C, O if A satisfies C , and*

- *if the assignment is annotated with $-\infty$, then for every integer k , there exists an assignment B such that B satisfies C and $O(B) < k$;*

- otherwise (i.e., if the assignment is not annotated with $-\infty$), there exists no integer assignment B such that $O(B) < O(A)$ and B satisfies C .

The task of an ILP solver is to provide an optimal integer assignment for instances that have one, or to report infeasibility otherwise. Sections 1.2 to 1.4 survey the algorithms involved in achieving this.

1.2 Simplex for Linear Programming

In 1947, Dantzig initiated the field of MP by introducing the *Simplex* algorithm for Linear Programming (LP). LP encompasses problems that involve linear inequalities and linear objective functions over real variables, i.e., LP instances look like ILP instances (Equation 1.1), except that the variables do not have to take integer values. Simplex is widely regarded as one of the most influential algorithms of all time [17].

Interestingly, work on LP predates Dantzig's development of Simplex. Notably, Fourier [32], Poussin [23], and Kantorovich [49] all studied LP before Dantzig. Their work did not, however, attract the attention of other researchers, possibly due to lack of interest in optimization [20].

While efficient in practice, the Simplex algorithm has exponential worst-case complexity. The first polynomial-time algorithm for LP was presented by Khachiyan in 1979 [51]. The algorithm was based on the *ellipsoid* method. Karmarkar followed with a polynomial-time *interior-point* algorithm for LP in 1984 [50]. Karmarkar's algorithm generated interest in interior-point methods for LP [78]. As their name implies, these methods reach an optimal solution by traversing the interior of the polytope described by the linear inequalities, in contrast to Simplex, which moves between adjacent edges of the polytope.

LP solving is a key operation in ILP solving (commonly through Simplex), and as such manifests itself in MPMT. However, our work does not provide any advancements in LP. LP appears in our algorithms as a black-box operation. We thus refrain from covering Simplex or any other algorithm for LP. The interested reader may read more in a number of textbooks, e.g., by Papadimitriou and Steiglitz [15].

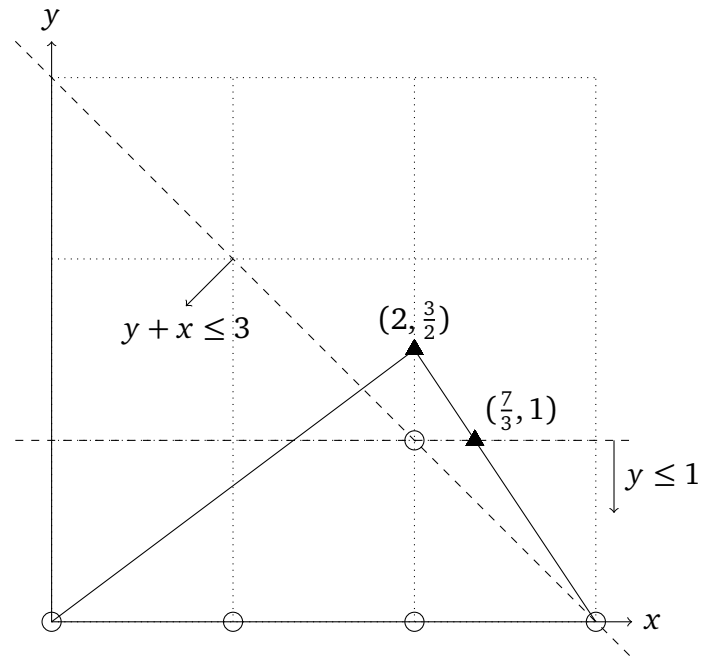


Figure 1.1: Gomory Cuts

1.3 Cutting Planes

ILP was pioneered by Gomory [38], who provided an algorithm based on *cutting planes* (or simply *cuts*). Gomory's algorithm effectively extends Simplex to deal with integer variables. Gomory's algorithm proceeds as follows.

- (a) Simplex is called on a continuous (real) relaxation of the linear constraints. In case Simplex reports infeasibility, or if it provides an assignment that is integer, then the problem has been solved. Otherwise,
- (b) the linear formulation is strengthened with a cut, *i.e.*, with an inequality that is violated by the solution, but implied by the (integer) linear inequalities. Such a cut is guaranteed to exist. Subsequently, Step (a) is executed again.

Example 1. Consider the ILP instance

$$\begin{array}{ll}
 \text{minimize} & \\
 O = -x - y & \\
 \text{subject to} & \\
 \left. \begin{array}{l} -3x + 4y \leq 0 \\ 3x + 2y \leq 9 \\ x, y \geq 0 \end{array} \right\} C & (1.2)
 \end{array}$$

The instance is graphically depicted in Figure 1.1.

Running Simplex on the continuous relaxation of the constraints (i.e., omitting the constraints $x, y \in \mathbb{Z}$) produces the non-integer optimal solution $\{x \mapsto 2, y \mapsto \frac{3}{2}\}$. The linear formulation needs to be strengthened to rule out this solution. Note that the inequalities $-3x + 4y \leq 0$ and $3x + 2y \leq 9$ entail that $(-3 + 3)x + (4 + 2)y \leq (0 + 9)$, which can be simplified to $y \leq \frac{3}{2}$. Since y is integer, it follows that $y \leq 1$. Graphically, the point $(x = 2, y = \frac{3}{2})$ (marked with a triangle) is above the line for $y \leq 1$, while all the integer points (marked with circles) are below or on the line. In other words, $y \leq 1$ separates the problematic point.

Introducing the cut $y \leq 1$ leads to a set of constraints whose continuous relaxation has optimal solution $\{x \mapsto \frac{7}{3}, y \mapsto 1\}$. The linear inequalities entail that $[-3x + 4y] + 7 \cdot [3x + 2y] \leq 0 + 7 \cdot 9$, i.e., $x + y \leq \frac{7}{2}$, which becomes $x + y \leq 3$, given that x and y are integer. This cut can thus strengthen the linear formulation.

Once we add $x + y \leq 3$, the continuous relaxation either leads to the integer assignment $\{x \mapsto 2, y \mapsto 1\}$, or to $\{x \mapsto 3, y \mapsto 0\}$. (Note that the Simplex algorithm moves between vertices, so the family of permissible non-integer solutions on the line $x + y = 3$ is of no consequence.) Either of the aforementioned integer assignments is a solution for the ILP instance of Equation 1.2.

The cuts $y \leq 1$ and $x + y \leq 3$ in our example clearly follow from the given inequalities. What is not explained in the example is how we chose these particular cuts. In fact, cuts (that separate the problematic non-integer points) can always be obtained as a side-product of running the Simplex

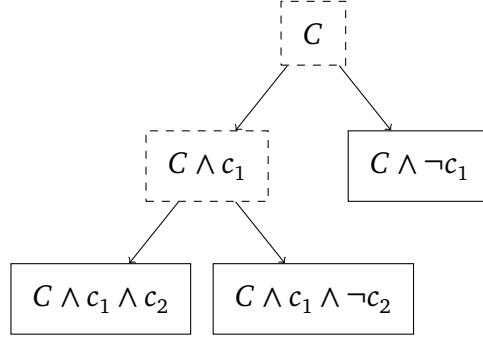


Figure 1.2: B&C Search Tree

algorithm. There are multiple ways to produce cuts. Cornuejols provides a tutorial [18].

Importantly, under certain conditions regarding the way cuts are computed, only finitely many cuts need to be introduced before a solution is reached [39, 15], *i.e.*, the algorithm provides a complete procedure for ILP (which, in fact, also works for MILP).¹

1.4 Branch-and-Cut

Gomory’s algorithm (in its standalone form) is not considered efficient. In fact, significant interest in Gomory’s technique emerged only in recent decades, when cut generation was coupled with Branch-and-Bound (B&B) search. This coupling is known as Branch-and-Cut (B&C) [68, 4, 14].

We describe the B&C solving process aided by Figure 1.2. In Figure 1.2, each *node* is marked with a set of integer linear constraints. The constraints C of the root node correspond to the linear inequalities of an ILP instance. Assume that the instance has objective function O . Each leaf node corresponds to a *subproblem* that needs to be explored, while a node that has children has been branched upon and does not need to be considered further. The ILP instance characterized by C and O can in principle be solved by applying Gomory’s algorithm. What happens instead is splitting C into

¹In this dissertation, completeness is applied to algorithms and means that the algorithm in question always terminates, thus providing an answer. Completeness is used in various senses, *e.g.*, it has different meaning when applied to proof systems.

Chapter 1. (Mixed) Integer Linear Programming

nodes $C \wedge c_1$ and $C \wedge \neg c_1$, where c_1 is an additional inequality. The motivation is that c_1 (or $\neg c_1$) may make solving the set of constraints easier, *e.g.*, by enabling stronger cuts (Section 1.3). What follows is branching on the subproblem $C \wedge c_1$, replacing it with two additional nodes ($C \wedge c_1 \wedge c_2$ and $C \wedge c_1 \wedge \neg c_2$).

The branching steps result in a *front* of three open subproblems, which are drawn with non-dashed frames. All nodes in the front are simultaneously present in the ILP solver's field of view. Any of the nodes in the front can be processed next, either by solving its relaxation and correspondingly generating cuts (Section 1.3), or with further branching. A node may be eliminated, either because it has an infeasible continuous relaxation, or because the solution to its relaxation provides a lower bound for O which guarantees that any solution is at most as good as an already known one. (B&C keeps track of a candidate solution, if there is one.) The process goes on for as long as there are subproblems that can potentially improve upon the known solution.

Example 2 (Example 1 via B&C). *We present a B&C sequence of steps for the MILP instance of Equation 1.2. In addition to leading to the cut $y \leq 1$, the real assignment $(x = 2, y = 1.5)$ highlights the area around $x = 2$ as interesting. It may thus be useful to branch on whether $c \equiv [x \geq 3]$ or $\neg c \equiv [x \leq 2]$.*

- *The relaxation for the subproblem $C \wedge c$ immediately leads to the locally optimal integer solution $\{x \mapsto 3, y \mapsto 0\}$ (with $O = -3$), which is in fact the only permissible solution (even for the continuous relaxation).*
- *The relaxation for the subproblem $C \wedge \neg c$ leads to the solution $\{x \mapsto 2, y \mapsto 1\}$, which is integer and also has $O = -3$.*

Branching thus enabled us to obtain an optimal solution without discovering the cut $x + y \leq 3$.

The termination argument for Gomory's procedure (Section 1.3) extends to a termination argument for the B&C approach. Branching is only an optimization, and in no way required for termination. Branching can thus be restricted, *e.g.*, by permitting only a bounded number of branching steps.

Gomory’s procedure is by itself enough to provide answers for the subproblems generated by branching. Another possibility is to apply branching with decreasing frequency, in a way that Gomory’s technique eventually has time to provide an answer between branching steps.

Note that cut generation can be problem-specific. In fact, Dantzig et al. [19] applied the high-level algorithmic structure of Section 1.3 to the Traveling Salesman Problem (TSP) before Gomory’s work on ILP. Symmetrically, B&C was shown to be computationally effective for TSP [42] before it was demonstrated to be practical for general ILP [4, 14]. Other problems that have been the target of specialized B&C-based approaches include airline crew scheduling [45], network design [43], lot sizing [9], and vehicle routing [33]. A B&C-based framework that integrates first-order theories is presented in subsequent chapters.

1.5 An Application in Aerospace

The application that provided the original motivation for MPMT is an instance of *system assembly*. Given a set of software and hardware components, the system assembly problem is the question of how to connect and integrate these components in a way that satisfies given system requirements. Our group worked on such a problem arising during the development of the Boeing 787 Dreamliner [62, 59, 44]. We briefly review the relevant aspects of this project.

Figure 1.3 provides a scaled-down model which is structurally similar to the Boeing-provided requirements. The components involved are cabinets that provide resources (including CPU time, memory, and bandwidth) and jobs (software components) that consume said resources. We have to map jobs to cabinets subject to a collection of constraints. A cabinet (respectively job) is represented as a record $\{r_1 = v_1; \dots; r_n = v_n\}$, where v_i is the amount of each resource r_i produced (respectively consumed). $x.r$ denotes the amount of resource r produced or consumed by x . Figure 1.3 refers to intuitively named (finite) sets of cabinets, jobs, separated pairs of jobs (*i.e.*, pairs that cannot reside on the same cabinet), and co-located pairs of

Chapter 1. (Mixed) Integer Linear Programming

$$\left. \begin{array}{ll}
 \sum_{c \in \text{cabinets}} m_{j,c} = 1 & \text{for all } j \in \text{jobs} \\
 \sum_{j \in \text{jobs}} j.r \cdot m_{j,c} \leq c.r, & \text{for all } c \in \text{cabinets}, \text{ for all } r \in \text{resources} \\
 m_{j_1,c} + m_{j_2,c} \leq 1, & \text{for all } c \in \text{cabinets}, \text{ for all } (j_1, j_2) \in \text{separated} \\
 m_{j_1,c} - m_{j_2,c} = 0, & \text{for all } c \in \text{cabinets}, \text{ for all } (j_1, j_2) \in \text{colocated} \\
 0 \leq m_{j,c} \leq 1, & \text{for all } c \in \text{cabinets}, \text{ for all } j \in \text{jobs}
 \end{array} \right\} \langle \text{ILP} \rangle$$

$$\text{sched}(\{j \mid j \in \text{jobs}, m_{j,c} \geq 1\}), \text{ for all } c \in \text{cabinets} \quad \langle \text{SCH} \rangle$$

Figure 1.3: ILP Modulo Scheduling

jobs (*i.e.*, pairs that have to reside on the same cabinet). The problem gives rise to integer linear constraints (Figure 1.3:⟨ILP⟩) over the set of Boolean ($\{0, 1\}$ -bounded) variables

$$\{m_{j,c} \mid j \in \text{jobs}, c \in \text{cabinets}\}, \quad (1.3)$$

where $m_{j,c}$ is 1 iff job j resides on cabinet c . Families of (in)equalities encode our allocation constraints (each job resides on exactly one cabinet), resource constraints (the jobs that reside on each cabinet cannot consume more of any resource than the amount provided), separation constraints, co-location constraints, and variable bounds.

The kinds of requirements that we have outlined so far can be solved by $\{0, 1\}$ -ILP alone [62]. In fact, the structure of these requirements (*e.g.*, including resource allocation) is similar to the structure of problems that arise in OR. Therefore, our problems are close to what existing MILP tools are meant for. However, there are additional requirements that necessitate extending MILP technology. Namely, the jobs that co-reside on each cabinet have to meet the scheduling constraints of their joint processor. This requirement is formalized in Figure 1.3:⟨SCH⟩.

The predicate *sched* concretely corresponds to a hard real-time scheduling discipline called *static-cyclic scheduling*. We are not aware of a way to encode static cyclic scheduling constraints in a way that yields efficient ILP

solving. However, we do have an efficient custom decision procedure for scheduling constraints (a scheduler). Our approach involves integrating an ILP solver and our scheduler [44]. We refer to this scheme as *ILP Modulo Scheduling*. The ILP solver suggests candidate assignments, which are checked by the scheduler. If the scheduler determines that an allocation is not schedulable, it produces a *lemma* that explains why. This lemma strengthens the ILP formulation and triggers a new call to the ILP solver. The two procedures alternate until they agree on a solution that satisfies both kinds of requirements.

The actual Boeing design carries requirements more complicated than what we show in Figure 1.3 [59, 44]. There are additional components that need to be modeled, *e.g.*, memory spaces. (Memory spaces share cabinet-provided resources with jobs.) Applications communicate via a publish-subscribe network. Messages are aggregated into virtual links that are multicast. The network and messages introduce additional families of constraints, *e.g.*, related to bandwidth and buffering. These additional constraints can be encoded as ILP, and thus the problem retains the compositional structure of Figure 1.3.

ILP Modulo Scheduling essentially amounts to an early, domain-specific instance of MPMT. Our successful use of ILP Modulo Scheduling to tackle industrially-relevant design problems and the widespread applicability of MILP and optimization [92, 46, 11, 43, 33] concur that a general MPMT framework has the potential to enable interesting applications in other domains.

1.6 Experiments

We experimentally show that an ILP core is essential for our approach of Section 1.5 to be practical. We thus provide evidence that extending an ILP core with background procedures is useful, thus motivating the development of a more general MPMT framework.

Our experiments rely on a family of 60 synthetic ILP benchmark in-

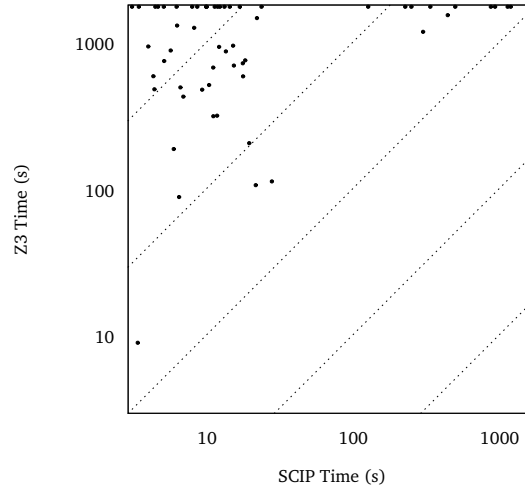


Figure 1.4: SCIP vs. Z3 (System Assembly Instances)

stances that capture the structure of our Boeing problems.² The instances contain (a) integer linear constraints that are similar to the ones given in Figure 1.3, and (b) scheduling lemmas [44] (which are collections of integer linear constraints) recorded by our implementation of ILP Modulo Scheduling in the process of solving problems that involve scheduling. The combination of these two families of constraints thus simulates the kind of reasoning that the core arithmetic engine participating in the ILP Modulo Scheduling scheme needs to perform.

The numbers involved in the instances retain the scale of the actual Boeing problems, with some randomization applied to protect sensitive information and diversify the set of instances. 47 out of 60 instances are unsatisfiable. We compare a leading ILP solver (SCIP [3]) against a leading SMT solver (Z3 [24]). We run both solvers with a timeout of 1800 seconds.

As we have seen (Equation 1.3), all variables involved in the instances are $\{0, 1\}$. There are multiple ways to encode $\{0, 1\}$ -ILP (also known as pseudo-Boolean) problems as SMT instances. A direct translation led to SMT problems that Z3 could not solve, so we tried several encodings, most

²<http://www.ccs.neu.edu/home/vpap/benchmarks.html>

of which yielded similar results. One encoding was significantly better than the rest, and it works as follows. Some of the linear constraints are clauses, *i.e.*, of the form $\sum l_i \geq 1$ for literals l_i . It makes sense to help SMT solvers by encoding such constraints as disjunctions of literals instead of inequalities. To do this, we declare all variables to be Boolean. Since almost all variables also appear in arithmetic contexts where they are multiplied by constants greater than 1, we translate such constraints as demonstrated by the following example: the linear constraint $x + y + 2z \geq 2$ becomes $(\geq (+ (\text{ite } x \ 1 \ 0) (\text{ite } y \ 1 \ 0) (\text{ite } z \ 2 \ 0)) \ 2)$.

Figure 1.4 visualizes the behavior of SCIP versus Z3. The figure is a scatter plot. A mark located at (x, y) corresponds to an instance such that SCIP takes x seconds to solve it, while Z3 takes y seconds. A point that lies on the northern border demonstrates that Z3 times out for the corresponding instance, while a point on the eastern border corresponds to a timeout for SCIP. The scale for both dimensions is logarithmic. SCIP solves all but one instance, while Z3 solves 3 out of the 13 satisfiable and 29 out of the 47 unsatisfiable instances. A clear majority of the instances is above the diagonal, *i.e.*, SCIP significantly outperforms Z3. It is thus clear that an ILP solver is more appropriate as the core of the combination framework that we outlined in Section 1.5.

In addition to SMT, we explored propositional satisfiability as an alternative to ILP for solving our system assembly problems. We notably also experimented with SAT. Obtaining a SAT-based solver able to tackle the Boeing-provided resource and structural constraints required a few months of work. This effort led to a solver for $\{0, 1\}$ -ILP (also known as Pseudo-Boolean Satisfiability) that performs incremental translation to SAT [60]. While in principle our SAT-based solver can replace the ILP solver in our implementation of ILP Modulo Scheduling, such a design choice would be limiting in a more general MPMT setting. Notably, it is not clear how to generalize our incremental scheme to non-Boolean variables, or even how to combine this scheme with theory solvers. The shortcomings revealed by our study of SAT-based approaches thus reinforce our choice of proceeding with an MP-based framework.

Chapter 2

Automated Reasoning

First-Order Logic (FOL) provides a general framework that encompasses diverse classes of constraints. Specifically, FOL allows a certain collection of symbols to obtain meaning via a set of *axioms*, *i.e.*, via a *theory*. In effect, the symbols become a custom constraint language.

AR deals with fragments of FOL (where a fragment may be defined by the applicable theory, or otherwise) that admit efficient procedures. The field encompasses reasoning techniques for theories of particular interest (*e.g.*, fragments of arithmetic), but also general deduction mechanisms that apply to any theory, as well as frameworks for combining theories and their solvers.

AR's flexibility with respect to the families of constraints that its techniques encompass, and the field's emphasis on modularly combining procedures, can help us provide an extensible variant of MP. This section surveys ideas, techniques, and algorithms from AR that we can utilize to achieve this goal.

While we necessarily introduce some first-order concepts, this chapter cannot possibly serve as a satisfactory introduction to FOL. For more details, the interested reader may refer to books on Mathematical Logic [30, among others]. Additionally, Manna and Zarba provide a very helpful introduction to theories and their combination [58].

$$\begin{aligned}
 T &::= V \mid C \mid F(T_1, T_2, \dots) \\
 A &::= T_1 = T_2 \mid P(T_1, T_2, \dots) \\
 G &::= \neg G \mid G_1 \vee G_2 \mid \forall V. G \mid A
 \end{aligned}$$

Figure 2.1: First-Order Syntax

2.1 First-Order Theories

We discuss first-order theories with the specific intent of integrating them within an MP-based framework. Our goal is not to optimize solving in the presence of specific theories, or to develop theory solvers per se. Our discussion thus focuses on the aspects of theories that are relevant at the framework level of abstraction, *i.e.*, we focus on the interface provided by theories and their solvers.

The abstract grammar of Figure 2.1 produces the standard syntactic objects of FOL. The non-terminal symbols V , C , F , and P stand for *variable*, *constant*, *function*, and *predicate symbols*, respectively. These symbols come from sets that will be clear from the context. The non-terminal symbols T , A , and G stand for *terms*, *atomic formulas* (or simply *atoms*), and *formulas*, respectively. Each subscripted letter is of the same kind of object as the non-subscripted non-terminal symbol. In what follows, we assume that all formulas and all other syntactic objects are well formed, *i.e.*, function and predicate symbols are not applied with conflicting arities.

A *literal* is either an atomic formula, or the negation thereof. An occurrence of a variable v is called *free* if v is not bound by the quantifier \forall (in the context of the occurrence). Standard scoping rules apply for \forall . A formula that has no free variables is called a *sentence*. A formula is called *quantifier-free* if it does not contain occurrences of the quantifier \forall .

AR algorithms and techniques routinely target formulas and terms over a limited alphabet, *i.e.*, over specified sets of function, predicate, and constant symbols. These sets are specified by a *signature*.

Definition 6 (Signature). A signature is a tuple (F, P, C, ar) , where F is a set of function symbols, P is a set of predicate symbols, C is a set of constant symbols,

Chapter 2. Automated Reasoning

and $ar : (F \cup P) \rightarrow \mathbb{N}^+$ is an arity function that assigns a non-zero natural number (the arity) to every function and predicate symbol.

As an example, we define the signature Σ_f that contains nothing but a single unary function symbol f . Σ_f appears in examples throughout this dissertation.

Definition 7. $\Sigma_f = (\{f\}, \emptyset, \emptyset, \{f \mapsto 1\})$

We use set operators over signatures whenever appropriate. For example, given signatures $\Sigma_i = (F_i, P_i, C_i, ar_i)$, for $i = 1, 2$, the union $\Sigma_1 \cup \Sigma_2$ is the signature: $(F_1 \cup F_2, P_1 \cup P_2, C_1 \cup C_2, ar)$, where ar is defined over $F_1 \cup P_1 \cup F_2 \cup P_2$ as follows: $ar(x) = ar_1(x)$ if $x \in F_1$ or $x \in P_1$; $ar(x) = ar_2(x)$ otherwise. We do not concern ourselves with the possibility that ar_1 and ar_2 provide conflicting arities.

A formula is called a Σ -formula if all function, predicate, and constant symbols (non-terminal symbols F , P , and C in Figure 2.1) that appear in it are in the signature Σ . We use the prefix Σ with the same meaning for other kinds of syntactic objects, e.g., Σ -term. We use the signature as a prefix whenever the applicable signature is not clear from the context.

A theory is a set of axioms over a given signature.

Definition 8 (Σ -Theory). A Σ -theory T is a set of Σ -sentences.

As an example, we provide the standard definition of the theory of arrays, which arises frequently in software and hardware verification.

Example 3 (Theory of Arrays [65]). Let

$$\Sigma_{\mathcal{A}} = (\{\text{read}, \text{write}\}, \emptyset, \emptyset, \{\text{read} \mapsto 2, \text{write} \mapsto 3\}).$$

The $\Sigma_{\mathcal{A}}$ -theory of arrays (without extensionality) consists of the following sentences:

$$\begin{aligned} & \forall a. \forall i. \forall e. [\text{read}(\text{write}(a, i, e), i) = e] \\ & \forall a. \forall i. \forall j. \forall e. [i \neq j \Rightarrow \text{read}(\text{write}(a, i, e), j) = \text{read}(a, j)] \end{aligned}$$

Example 4 (\emptyset as a Σ_f -Theory). *The set of sentences \emptyset is a Σ -theory for any signature Σ , e.g., \emptyset is a Σ_f -theory.*

The Σ_f -theory \emptyset does not tell us anything about Σ_f beyond what the signature itself already tells us, *i.e.*, that we are dealing with a unary function symbol f . When solving constraints that are Σ_f -formulas, in the absence of any theory other than \emptyset , all that we can assume about f is that it respects Leibniz's law, *i.e.*, that for any $x = y$, $f(x) = f(y)$ holds. Unconstrained functions like f are called *uninterpreted*. Even determining that the functions that appear in a formula satisfy Leibniz's law does however require some work. This most commonly happens via the procedure known as *congruence closure* [72, 76].

Definition 9 (Σ -Interpretation). *Let $\Sigma = (F, P, C, \text{ar})$ be a signature. A Σ -interpretation \mathcal{A} with domain A over a set of variables V is a map which interprets*

- each $v \in V$ as an element $v^{\mathcal{A}} \in A$,
- each $c \in C$ as an element $c^{\mathcal{A}} \in A$,
- each $f \in F$ as a function $f^{\mathcal{A}} : A^{\text{ar}(f)} \rightarrow A$, and
- each $p \in P$ as a subset $p^{\mathcal{A}}$ of $A^{\text{ar}(p)}$ (*i.e.*, as a relation over $A^{\text{ar}(p)}$).

As per the semantics of FOL [30], a Σ -interpretation \mathcal{A} over variable symbols V (indirectly) assigns a Boolean $F^{\mathcal{A}}$ to each Σ -formula F whose free variables are in V . We say that the interpretation \mathcal{A} *satisfies* F , in symbols $\mathcal{A} \models F$, if $F^{\mathcal{A}}$ is true. For formulas F and G , F *entails* G (which we denote by $F \models G$, *i.e.*, we overload \models) means that any interpretation that satisfies F also satisfies G . If, for formulas F and G , $F \models G$ and $G \models F$, then F and G are called (*logically*) *equivalent*. A Σ -formula F with free variable symbols V is called

- *satisfiable*, if there exists a Σ -interpretation \mathcal{A} over V such that $F^{\mathcal{A}}$ is true;
- *valid*, if for every Σ -interpretation \mathcal{A} over V , $F^{\mathcal{A}}$ is true; and

- unsatisfiable, if it is not satisfiable.

For a Σ_1 -formula F and a Σ_2 -theory T , F is *T-satisfiable* if $F \wedge T$ is satisfiable, *i.e.*, if there exists a $(\Sigma_1 \cup \Sigma_2)$ -interpretation over the free variable symbols V of F that satisfies $F \wedge T$. (Note that a theory T does not have free variable symbols.) We use the notions of satisfiability and consistency (which are equivalent in FOL) largely interchangeably, *e.g.*, we also refer to *T-consistency*. A formula F is called *T-unsatisfiable* (or *T-inconsistent*) if it is not *T-satisfiable*. For formulas F and G , F *T-entails* G (in symbols $F \models_T G$) if $F \wedge \neg G$ is *T-unsatisfiable*. A Σ -theory T is *decidable* if there exists a decision procedure that can determine whether a Σ -formula is *T-satisfiable*. This dissertation only deals with the quantifier-free fragments of theories, *i.e.*, with the collections of quantifier-free formulas in the respective signatures. A (*theory*) *solver* for some Σ -theory T is a decision procedure that is able to determine whether a conjunction of Σ -literals is *T-satisfiable*. (The quantifier-free fragment of a theory is decidable iff such a solver exists.)

2.2 Linear Integer Arithmetic

Definition 10. $\Sigma_{\mathbb{Z}} = (\{+, -\}, \{\leq\}, \{0, \pm 1, \pm 2, \dots\}, \{+ \mapsto 2, - \mapsto 1, \leq \mapsto 2\})$.

The signature $\Sigma_{\mathbb{Z}}$ contains the standard symbols of Linear Integer Arithmetic. A multiplication operator would allow non-linear terms, and is thus not provided. We use the operators in $\Sigma_{\mathbb{Z}}$ in the standard infix fashion.

Definition 11 ($\Sigma_{\mathbb{Z}}$ -Theory \mathcal{Z} of Linear Integer Arithmetic). *The theory of Linear Integer Arithmetic, which we denote by \mathcal{Z} , is the $\Sigma_{\mathbb{Z}}$ -theory defined by the set of $\Sigma_{\mathbb{Z}}$ -sentences that are true in the standard $\Sigma_{\mathbb{Z}}$ -interpretation, *i.e.*, the interpretation whose domain is \mathbb{Z} , and which interprets the symbols in $\Sigma_{\mathbb{Z}}$ according to their standard meaning over \mathbb{Z} .*

\mathcal{Z} is closely related to ILP, and thus appears frequently in this dissertation. The inequalities of ILP can be viewed as atomic formulas in $\Sigma_{\mathbb{Z}}$. In our description of ILP, we have used multiplication with a constant, which becomes repeated addition on the $\Sigma_{\mathbb{Z}}$ side. \mathcal{Z} is a decidable theory. The

quantifier-free fragment of \mathcal{Z} (for which we use the abbreviation QFLIA) is NP-complete and of particular importance. As a matter of fact, we have already outlined a theory solver for QFLIA (Section 1.3), which happens to also work for optimization problems.

We can view an integer assignment A (Definition 4) as the set of $\Sigma_{\mathcal{Z}}$ -formulas $\{v = A(v) \mid v \in \mathcal{V}\}$, where $A(v)$ is viewed as a $\Sigma_{\mathcal{Z}}$ -term. An integer assignment A viewed as a set of formulas is always \mathcal{Z} -consistent. If A is an integer assignment and A satisfies a set of integer linear constraints C (with respect to the notion of satisfiability that we defined in Section 1.1), it is also the case that $A \models_{\mathcal{Z}} C$.

Whenever convenient, we use relation symbols like $<$ that do not appear in $\Sigma_{\mathcal{Z}}$, and also multiplication by a constant, which is to be interpreted as repeated addition. These are only syntactic shorthands. Similar syntactic shorthands apply to integer linear constraints (Definition 2).

2.3 Combining Theories

As already noted, integer linear constraints can be viewed as atomic formulas in $\Sigma_{\mathcal{Z}}$ to be solved modulo \mathcal{Z} . Our goal is to combine these constraints with others in a different signature and constrained by some other theory. At its theoretical core, MPMT is thus an instance of theory combination, which is a well-studied topic in AR.

We discuss theory combination with an emphasis on the established (non-deterministic) *Nelson-Oppen (NO)* framework [73, 86]. Note that NO is not the only theory combination method. Shostak provides an influential alternative [82]. Nevertheless, we consider NO to be more promising for our purposes.

Constraint solving modulo a combination of theories naturally involves formulas that mix symbols from multiple signatures. Let T_i be a Σ_i -theory, for $i = 1, 2$, and let $\Sigma_1 \cap \Sigma_2 = \emptyset$. It is common practice to split a formula (or set of formulas) S over $\Sigma_1 \cup \Sigma_2$ into a pair of sets of formulas S_i , where S_i is over Σ_i , for $i = 1, 2$. This happens via *variable abstraction*. We explain the variable abstraction process through the following example.

Chapter 2. Automated Reasoning

Example 5 (Variable Abstraction). Recall the signatures Σ_f (Definition 7) and Σ_Z (Definition 10). Clearly, $\Sigma_Z \cap \Sigma_f = \emptyset$. Let

$$G = f(f(x) + 1) + f(y + 2) \leq 3,$$

where x and y are variable symbols. G is a $(\Sigma_Z \cup \Sigma_f)$ -formula. We introduce a fresh variable for every occurrence of f that appears under either $+$ or \leq , and for every occurrence of $+$ that appears under f :

$$G = [f(\underbrace{f(x)+1}_{v_1}) + \underbrace{f(y+2)}_{v_3} \leq 3]$$

$$\underbrace{\underbrace{}_{v_2}}_{v_4} \quad \underbrace{}_{v_5}$$

G is logically equivalent to $C \cup I$, where

$$C = \{ \ v_4 + v_5 \leq 3, \ v_2 = v_1 + 1, \ v_3 = y + 2 \ \}, \text{ and}$$

$$I = \{ \ v_1 = f(x), \ v_4 = f(v_2), \ v_5 = f(v_3) \ \}.$$

C is a set of Σ_Z -literals and I is a set of Σ -literals. Variable abstraction introduced new variables, v_1, \dots, v_5 . C and I only share variable symbols.

We introduce stably-infinite theories, which are a broad class of theories that can be combined based on the NO scheme. While NO has been extended to accommodate theories that are not stably-infinite [87], stably-infinite theories remain the most relevant family.

Definition 12 (Stably-Infinite Theory). A Σ -theory T is called stably-infinite if for every T -satisfiable quantifier-free Σ -formula F , there exists an interpretation satisfying $F \wedge T$ whose domain is infinite.

\mathbb{Z} is a stably-infinite theory. (Every quantifier-free \mathbb{Z} -satisfiable $\Sigma_{\mathbb{Z}}$ -formula is satisfied by the standard $\Sigma_{\mathbb{Z}}$ -interpretation of \mathbb{Z} , whose domain is the infinite set \mathbb{Z} .)

An *equivalence relation* is a binary relation that is reflexive, symmetric, and transitive. A common way to describe an equivalence relation is through

its set of *equivalence classes*. For example, consider the equivalence relation $E = \{\{1, 2\}, \{3\}\}$ over the finite set $\{1, 2, 3\}$; $1E2$, but neither $1E3$, nor $2E3$.

Definition 13 (Arrangement). *Let E be an equivalence relation over a set of variables V . The set*

$$\alpha(V, E) = \{x = y \mid xEy\} \cup \{x \neq y \mid x, y \in V \text{ and not } xEy\}$$

is the arrangement of V induced by E .

Given a set of variables V and an equivalence relation E over V , $\alpha(V, E)$ entails, for any pair of variables $x, y \in V$, either $x = y$ or $x \neq y$. The equalities and dis-equalities in an arrangement can be viewed as Σ -literals for any signature Σ . The literals in an arrangement are thus understood by all first-order theory solvers. These literals form a kind of information that theory solvers are able to exchange. This idea is central to NO.

Fact 1 (Nelson-Oppen for Stably-Infinite Theories [73, 86, 58]). *Let T_i be a stably-infinite Σ_i -theory, for $i = 1, 2$, and let $\Sigma_1 \cap \Sigma_2 = \emptyset$. Also, let Γ_i be a conjunction of Σ_i -literals. $\Gamma_1 \cup \Gamma_2$ is $(T_1 \cup T_2)$ -satisfiable iff there exists an equivalence relation E of the variables V shared by Γ_1 and Γ_2 such that $\Gamma_i \cup \alpha(V, E)$ is T_i -satisfiable, for $i = 1, 2$.*

Example 6 (Example 5 Continued). *Recall the sets C and I from Example 5. Let V be the set of variables that C and I share; $V = \{v_1, v_2, v_3, v_4, v_5\}$. Consider the equivalence relation $E = \{\{v_1\}, \{v_2, v_3\}, \{v_4, v_5\}\}$. $C \cup \alpha(V, E)$ is \mathcal{Z} -satisfiable, while $I \cup \alpha(V, E)$ is \emptyset -satisfiable. Both \mathcal{Z} and \emptyset are stably-infinite. It follows from Fact 1 that $C \cup I$ (and thus G) is $(\mathcal{Z} \cup \emptyset)$ -satisfiable.*

Corollary 1. *Let T_i be a stably-infinite Σ_i -theory whose quantifier-free fragment is decidable, for $i = 1, 2$, and let $\Sigma_1 \cap \Sigma_2 = \emptyset$. The quantifier-free fragment of $T_1 \cup T_2$ is decidable.*

Proof. We provide a theory solver for $T_1 \cup T_2$, i.e., a decision procedure that is able to determine whether a given set of $(\Sigma_1 \cup \Sigma_2)$ -literals S is $(T_1 \cup T_2)$ -satisfiable. We first perform variable abstraction (Example 5), and obtain sets of Σ_i -literals S_i ($i = 1, 2$) such that S is logically equivalent to $S_1 \cup S_2$. Let

V be the set of variables shared between S_1 and S_2 . We non-deterministically pick an equivalence relation E over V , and perform T_i -satisfiability queries for $S_i \cup \alpha(V, E)$. (The quantifier-free fragments of T_i are decidable, which implies the existence of applicable decision procedures.) From Fact 1 and the logical equivalence between S and $S_1 \cup S_2$, it follows that S is $(T_1 \cup T_2)$ -satisfiable if both T_i -satisfiability queries return true. \square

For a Σ -theory T (where $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$) whose quantifier-free fragment is decidable, Corollary 1 implies that the quantifier-free fragment of $\mathcal{Z} \cup T$ is decidable. Corollary 1 thus provides theoretical basis for our attempt to combine ILP with stably-infinite background theories. However, we have not yet discussed the practical aspects of such a combination. Notably, in the context of practical constraint solving, non-determinism translates to search. Conversely, in any NO-based method, the concrete strategy applied to search over all possible arrangements is crucial. Subsequent chapters discuss such a search strategy, implemented within the MP-focused B&C framework (Section 1.4) in an optimization-aware fashion.

2.4 Satisfiability Modulo Theories

Our discussion has so far focused on problems that do not involve propositional structure. As a matter of fact, recall that a theory solver only deals with a conjunction of literals in the applicable signature. The *Satisfiability Modulo Theories (SMT)* framework encompasses problems where theory atoms are combined in arbitrary ways, e.g., with the disjunction operator. SMT clearly influences MPMT, as even the name of the latter framework implies.

Example 7 (Satisfiability Modulo \mathcal{Z}). *Consider the following instance:*

$$\underbrace{x + y < -2}_p \wedge [\underbrace{(x + 2 \cdot y - z \geq 0)}_q \wedge \underbrace{z - y > 0}_r] \vee (\underbrace{x > 3}_s \wedge \underbrace{y \geq -5}_t). \quad (2.1)$$

It is common in SMT to deal with a propositional abstraction of the problem. This involves introducing propositional variables (in our example, p, q, r, s , and t)

for the theory clauses, thus producing an abstraction of the problem that is purely propositional. Introducing such propositional variables in order to separate first-order atomic formulas from the propositional skeleton is conceptually similar to separating signatures via variable abstraction (Example 5).

The propositional abstraction of our instance can be satisfied either by setting p , q , and r to true, or by setting p , s , and t to true. $p \wedge q \wedge r$ is \mathcal{Z} -inconsistent: $q \wedge r$ implies that $(x + 2 \cdot y - z) + (z - y) > 0$, which implies that $x + y > 0$, which is \mathcal{Z} -inconsistent with p . $p \wedge s \wedge t$ is similarly \mathcal{Z} -inconsistent. The formula is thus \mathcal{Z} -inconsistent, given that no permissible propositional assignment is \mathcal{Z} -consistent.

As the example demonstrates, SMT solving involves propositional reasoning, coupled with checking candidate propositional solutions for theory consistency.

SMT solvers can be broadly categorized as *lazy* and *eager*. A Lazy SMT solver [7, 26, 77, 24] utilizes a propositional satisfiability (SAT) solver (usually based on the CDCL scheme [83, 70], which is an evolution of the classical DPLL scheme [22]) in synergistic combination with theory solvers. In Example 7, the SAT solver would deal with the propositional abstraction $p \wedge [(q \wedge r) \vee (s \wedge t)]$. An Eager SMT solver [56, 81, 34, 61] reduces its input SMT instance to a pure SAT instance, and subsequently calls a SAT solver.

Eager SMT generally emphasizes applications that differ in character from the ones that MPMT targets, e.g., bit-precise reasoning enabled by the theory of bitvectors [56, 34, 61], with the notable exception of an eager SMT approach to \mathcal{Z} [81]. Additionally, an eager solver has no algorithmic similarity with the architecture that we are developing for MPMT. The MPMT counterpart of the eager SMT approach would be to reduce MPMT instances to just MP, which is a possibility that this dissertation does not explore. Conversely, our discussion of SMT focuses on the lazy approach.

We discuss the DPLL(T) framework [35, 77], which describes Lazy SMT in an abstract way. Through DPLL(T), we indirectly provide an introduction to Lazy SMT in general. DPLL(T) is described as a *transition system*, i.e., a set of rules that relate *states*. A DPLL(T) state corresponds to the SMT solver's progress towards a solution. We formally define DPLL(T) states, but only

informally explain some of the DPLL(T) transitions (defined formally in the DPLL(T) journal paper [77]) through an example.

Definition 14 (DPLL(T) Trail). A DPLL(T) trail is a sequence of propositional literals (i.e., propositional variables or negations thereof), some of which are marked with the superscript d . The superscripted literals are called decision literals.

Definition 15 (DPLL(T) State). A DPLL(T) state is either the special state `FailState`, or it has the form $M \parallel F$, where

- M is a DPLL(T) trail, and
- F is a set of propositional clauses (i.e., disjunctions of propositional literals).

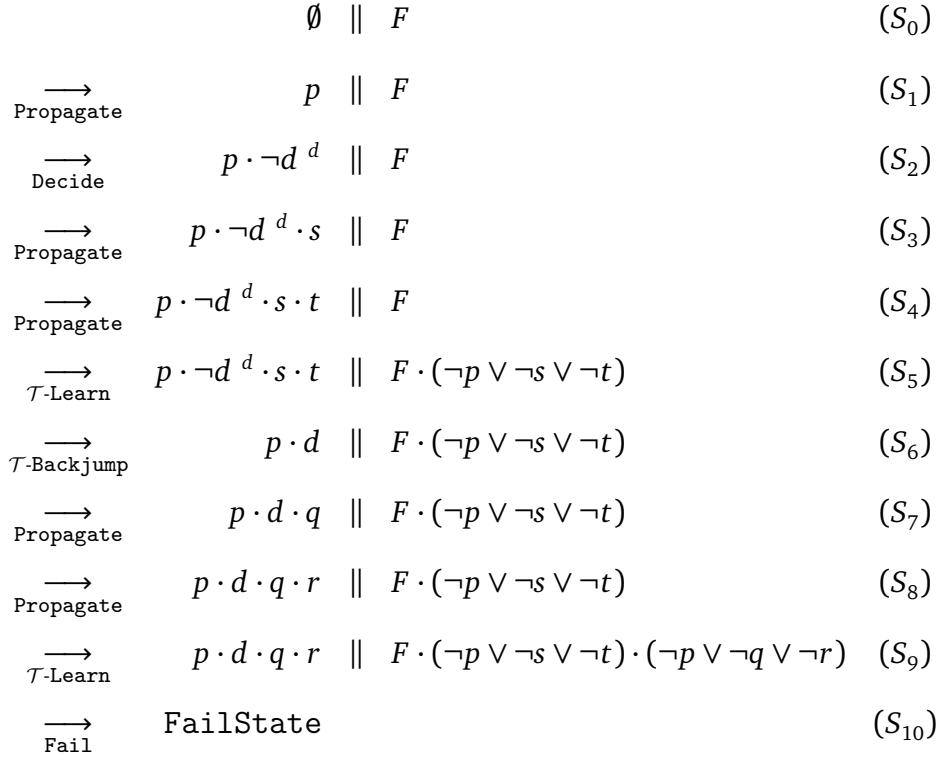
In a DPLL(T) state $M \parallel F$, some of the literals in the trail M are decisions, while the rest have been produced by *propagating* based on previously known literals. Propagation is either purely propositional (in practical implementations based on the two-watched-literal scheme [70]), or based on the theory. The clauses in F are either part of the original formula, or produced by *clause learning*. Just like propagation, clause learning happens based on both propositional and theory techniques. The example that follows covers propagation and clause learning.

Example 8 (Example 7 via DPLL(T)). DPLL(T), instantiated with \mathcal{Z} as the background theory, can solve Example 7. One possible encoding of the propositional abstraction (Equation 2.1) as CNF is

$$F = \{p, \neg d \vee q, \neg d \vee r, d \vee s, d \vee t\},$$

where the auxiliary variable d forces us to pick one of the disjuncts $q \wedge r$ and $s \wedge t$. Figure 2.2 provides a possible sequence of DPLL(T) transition steps. The transition steps are marked with the DPLL(T) rule that applies.

F clearly implies p . Thus, the originally empty assignment can be extended with p (transition $S_0 \rightarrow S_1$ in Figure 2.2). No other literals can be propagated. The CDCL component resorts to the decision $\neg d$ ($S_1 \rightarrow S_2$). Subsequently, the literal s can be propagated ($S_2 \rightarrow S_3$), based on the clause $d \vee s$. Similarly,


 Figure 2.2: DPLL(T) Transitions (Examples 7 and 8)

t can be propagated (transition $S_3 \rightarrow S_4$) based on $d \vee t$. The \mathcal{Z} solver then reports that $p \wedge s \wedge t$ (implied by the current assignment) is \mathcal{Z} -inconsistent (as explained in Example 7) by producing the clause $\neg p \vee \neg s \vee \neg t$ ($S_4 \rightarrow S_5$). The decision $\neg d$ then has to be flipped ($S_5 \rightarrow S_6$). The literals q and r are propagated ($S_6 \rightarrow S_7 \rightarrow S_8$). The \mathcal{Z} -solver then takes action to report the entailed clause $\neg p \vee \neg q \vee \neg r$ ($S_8 \rightarrow S_9$). This last clause is violated by the candidate assignment, but the assignment contains no decision that can be retracted. The instance is thus \mathcal{Z} -unsatisfiable, which is described in DPLL(T) by transitioning ($S_9 \rightarrow S_{10}$) to FailState.

Figure 2.3 depicts the DPLL(T) sequence of Example 8 as a search tree. Node labels correspond to the assignment at different points in time, and additionally denote the states when this assignment is a current one; a node may correspond to multiple states, because the search tree does not provide

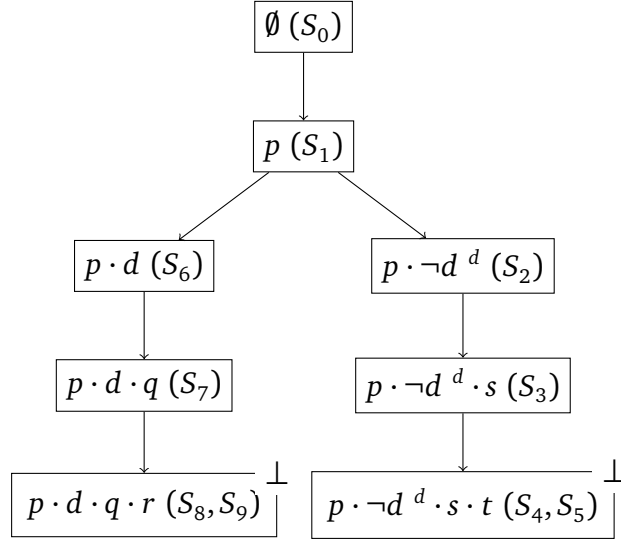


Figure 2.3: DPLL(T) Search Tree

separate nodes for modifications in the formula component of states. Two nodes with the same parent denote a decision, while a node with no siblings corresponds to a propagation step. The tag \perp denotes that the local assignment has been demonstrated to be theory-inconsistent. The children of the node p correspond to the decision $\neg d$ and its negation. Note that the node $p \cdot d$ only becomes visible after the transition to S_6 . This happens only after $p \cdot \neg d$ (the sibling of $p \cdot d$) has led to a contradiction.

The behavior described by Figure 2.3 highlights an important difference between B&C-style search (Section 1.4) and DPLL-style search, as in DPLL(T). In DPLL, only one node is explicitly represented and actively manipulated at any point in time. The negations of the decisions taken (*i.e.*, literals superscripted with d in DPLL(T)) correspond to implicit nodes that may be examined in the future. In contrast, B&C maintains a front of nodes that are simultaneously under examination. A B&C solver may process these nodes in an interleaved fashion.

Either solver design has benefits. DPLL(T) has the benefit of compact data structures that can be manipulated efficiently. Concretely, (a) all that needs to be represented is a (global) collection of clauses, and a (current) assignment that is a simple sequence of literals; also, (b) undoing a decision

is simply a matter of dropping a suffix of the assignment. B&C emphasizes flexibility over efficient low-level search primitives. Namely, (a) branching is not just on pre-existing Boolean variables, and it does not have to be binary; (b) extensive modifications can happen in each node in the front, independently of the other nodes; finally, (c) the solver may interleave processing of the nodes in the front, *e.g.*, driven by heuristics for the possible values of the objective function.

2.5 Related Work

Several techniques have been proposed in the AR literature (and, in recent times, implemented inside the Lazy SMT framework) in pursuit of goals related to ours. We discuss the relevant directions.

2.5.1 Arithmetic

Dutertre and de Moura describe an efficient implementation of Simplex, specialized for participating as a theory solver in the DPLL(T) scheme [27]. The approach is implemented in the Yices solver [29]. The authors briefly describe how they apply B&C and Gomory cuts to provide support for the integers, and elaborate on integer support in a related technical report [28].

Bozzanno et al. [12] present a layered approach (implemented in a version of MathSAT) that combines (Real and Integer) Linear Arithmetic engines of different power, *e.g.*, equational reasoning over atoms of the form $v = w$ (for integer or real variables v and w) happens more frequently than more involved (and more powerful) arithmetic reasoning. Griggio presents a similarly layered implementation of (Real and Integer) Linear Arithmetic within a state-of-the-art solver (MathSAT 5), notably implementing B&C by combining branching internal to the theory solver with branching performed by the SAT solver [41].

In addition to general Linear Arithmetic, SMT solvers employ techniques that target the Difference Logic [74] and UTVPI [55] fragments. Both of these fragments are of polynomial complexity, but have practical relevance.

The work on Linear Arithmetic in SMT clearly differs from MPMT, in that we employ a Mathematical Programming solver as the core solver (as opposed to a CDCL solver). The expectations that an SMT solver has from its arithmetic engine are different from ours, *e.g.*, incremental operation is important, while individual arithmetic literals tend to be simpler, with hard problems arising because of the way these simple atoms are combined at the Boolean level.

2.5.2 MP Solvers as Theory Solvers

The arithmetic procedures in SMT are generally implemented with exact arithmetic, thus providing the correctness crucially needed for applications where unsatisfiable instances occur frequently, *e.g.*, formal verification. In contrast, MP solvers generally perform floating-point arithmetic, emphasizing large-scale satisfiable instances. (Note that the correctness of a satisfying assignment is easy to validate, no matter how it has been produced.)

Faure et al. [31] experiment with using inexact MP solvers as theory solvers in $DPLL(T)$, and as expected, they come across wrong answers. More interestingly, the authors also discover performance issues, which they attribute to the different emphasis of MP solvers. More recent work [69, 10, 53] studies combinations of exact and inexact solvers that aim to inherit both the soundness of the former solver family and the performance of the latter. The emphasis of this body of work remains on SMT-style applications, as opposed to OR-like MILP (or MILP-heavy) instances.

2.5.3 Optimization

The Lazy SMT scheme has been extended to support optimization. Nieuwenhuis and Oliveras suggest using a *progressively stronger* theory to encode search for progressively better solutions [75]; their concrete implementation targets Max-SMT (*i.e.*, satisfying as many constraints as possible) where the background theory is Integer Difference Logic. Cimatti et al. [16] support optimization modulo a special *theory of costs* (that can express resource and $\{0, 1\}$ -ILP constraints) by integrating either linear search or bi-

nary search in an SMT solver. Sebastiani and Tomasi [80] describe optimization modulo Linear Real Arithmetic by similarly performing linear or branch-and-bound search.

In all these approaches, the search needed for optimizing the objective function is separate from the search happening inside the core DPLL solver. In contrast, the MP core that we employ supports optimization natively.

2.5.4 Generalized CDCL

The CDCL paradigm has been applied to problems that involve domains other than the Booleans, or constraints that are not propositional CNF. Manquinho and Marques-Silva modify CDCL for $\{0, 1\}$ -ILP, also known as pseudo-Boolean solving [63]. Cotton [79] and McMillan et al. [54] perform CDCL-like search over the reals to solve problems in Linear Real Arithmetic. Jovanovic and de Moura [48] provide an ILP solver that follows the same paradigm. The Model Constructing Satisfiability Calculus (MCSat) [25, 47] can be thought of as a $DPLL(T)$ alternative whose trail also contains non-propositional atoms, *i.e.*, MCSat provides direct support for non-propositional branching and learning.

This research direction can be viewed as progress towards SMT solvers where the core procedure handles constraints more expressive than CNF. MPMT is related in that it can also provide an SMT-like framework, while also relying on a non-propositional core solver. However, our emphasis is on supporting theories with a core solver that applies standard MP technology, as opposed to providing a new kind of core solver specifically aimed for SMT-like applications.

Chapter 3

(M)ILP Meets Theories

This chapter introduces the MPMT (Mathematical Programming Modulo Theories) formalism. MPMT encompasses problems that can be mostly, but not fully, described in terms of integer linear constraints. In addition to the linear constraints, MPMT instances involve first-order terms and possibly a background theory. Our introduction to MPMT relies on both examples and formal definitions.

Our first example covers the empty theory (introduced through Example 4), which is the simplest possible theory, and at the same time (arguably) the most prevalent one. We subsequently demonstrate the impact of a stronger theory (*monotonicity*) on the optimal solution. The algorithmic techniques needed to support these examples are developed in subsequent chapters.

We subsequently formally define MPMT, which serves as a unifying formalism for both MP-like (Chapter 1) and AR-like (Chapter 2) constraints. We thus concretize the kinds of instances that we target, and lay the groundwork for the $BC(T)$ solver architecture presented in subsequent chapters.

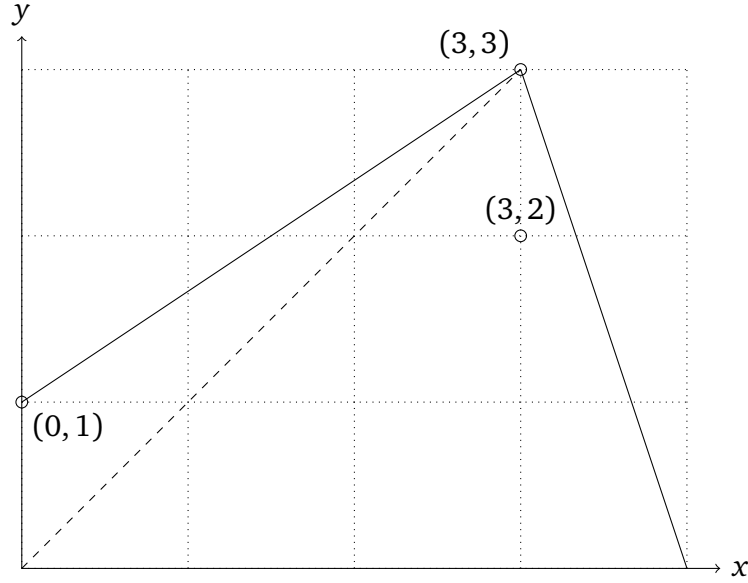


Figure 3.1: Linear Constraints C of Example 9 (xy -Plane)

3.1 MPMT by Example

Example 9 (ILP Modulo Uninterpreted Functions). *Consider the following instance:*

$$\begin{aligned}
 &\textbf{minimize} \\
 &\quad -y \\
 &\textbf{subject to} \\
 &\quad 3 \cdot y - 2 \cdot x \leq 3 \\
 &\quad 3 \cdot x + y \leq 12 \\
 &\quad x \geq 0 \\
 &\quad y \geq 0 \\
 &\quad f(x) - f(y) \geq 1 \\
 &\quad x, y, f(x), f(y) : \mathbb{Z}
 \end{aligned} \tag{3.1}$$

Equation 3.1 involves linear constraints over integer variables x and y . The reason that Equation 3.1 is not a MILP instance is the function f . We do not constrain f in any way, except through its appearance in Equation 3.1. Being a function, f has to satisfy Leibniz's law, but this does not need to be stated in any way. To produce a problem in the language of MILP solvers, we perform variable abstraction (similar to Example 5), which introduces auxiliary

variables f_x for $f(x)$ and f_y for $f(y)$. We obtain Equation 3.2.

$$\begin{array}{ll}
 \text{minimize} & -y \\
 \text{subject to} & \\
 & \left. \begin{array}{ll} 3 \cdot y - 2 \cdot x & \leq 3 \\ 3 \cdot x + y & \leq 12 \\ x & \geq 0 \\ y & \geq 0 \\ f_x - f_y & \geq 1 \end{array} \right\} C \\
 & \left. \begin{array}{ll} f_x & = f(x) \\ f_y & = f(y) \end{array} \right\} I \\
 & x, y, f_x, f_y : \mathbb{Z}
 \end{array} \tag{3.2}$$

Recall the signature Σ_f (Definition 7); I is a set of Σ_f -literals.

Figure 3.1 visualizes how C constraints the variables x and y . To minimize $-y$ (i.e., to maximize y) subject to C , we can look for the uppermost point in the space defined by the inequalities. C thus admits a family of optimal solutions with $x = 3$ and $y = 3$, e.g., $\{x \mapsto 3, y \mapsto 3, f_x \mapsto 1, f_y \mapsto 0\}$. However, I prevents solutions with $x = 3 = y$; $x = y \wedge I$ implies that $f_x = f(x) = f(y) = f_y$, which is \mathcal{Z} -inconsistent with the inequality $f_x - f_y \geq 1$.

Conversely, a MILP solver by itself cannot tackle our example. What is needed is a synergistic combination of ILP solving and theory solving, such that theory input can steer the ILP solver away from the problematic point ($x = 3, y = 3$), and towards the next-best family of solutions, which is represented in Figure 3.1 by the point ($x = 3, y = 2$). One such solution is

$$A = \{x \mapsto 3, y \mapsto 2, f_x \mapsto 1, f_y \mapsto 0\}.$$

A satisfies C , and it is consistent with I . A is thus an optimal solution for our original instance of Equation 3.1.

Example 10 (ILP Modulo Monotonicity). The constraints of Equation 3.1 also have meaning in the presence of theories other than \emptyset . Constraining f by means of a non-empty theory can rule out the solution A . For example, consider

the $(\Sigma_Z \cup \Sigma_f)$ -theory

$$T = \forall x. \forall y. [y \leq x \Rightarrow f(x) \leq f(y)]. \quad (3.3)$$

T consists of a single axiom that forces the function f to be monotonically decreasing. The solution A for Equation 3.1 is not even permissible in the presence of T . $x = 3 > 2 = y$ implies that $f_y = f(y) \geq f(x) = f_x$, which cannot be Z -satisfied simultaneously with $f_x - f_y \geq 1$. Solutions with either $x = y$ or $x > y$ are ruled out. The optimal solution is thus an extension of the point $(x = 0, y = 1)$, e.g.,

$$B = \{x \mapsto 0, y \mapsto 1, f_x \mapsto 1, f_y \mapsto 0\}.$$

$O(B) = -1 > 2 = O(A)$. The theory T (which is clearly stronger than \emptyset) led to a solution that is worse than the solution for Example 9.

3.2 MPMT Instances and Solvers

Definition 16 (Σ -(Interface) Definition). Let Σ be a signature such that $\Sigma \cap \Sigma_Z = \emptyset$. A Σ -interface definition, or simply Σ -definition when this does not cause ambiguity, is a formula of one of the following forms:

- (a) $v = f(e_0, \dots, e_{n-1})$, where $v \in \mathcal{V}$, f is a Σ -function symbol, and e_i are Σ -terms, or
- (b) $v > 0 \Leftrightarrow p(e_0, \dots, e_{n-1})$, where $v \in \mathcal{V}$, p is a Σ -predicate symbol, and e_i are Σ -terms.

Σ -definitions (where Σ is a signature such that $\Sigma \cap \Sigma_Z = \emptyset$) are a restricted kind of first-order formulas. Σ -definitions associate certain variable symbols in \mathcal{V} with Σ -terms (Case (a)) and (non-equational) Σ -atomic formulas (Case (b)). The thus constrained variable symbols serve as references to first-order terms and atoms. Such variables can subsequently appear in integer linear constraints. As an example, notice that the constraints I involving f in Example 9 are Σ_f -definitions.

We expect variable symbols associated with Σ -atomic formulas (Case (b)) to be constrained as Boolean (*i.e.*, bounded in $[0, 1]$) on the ILP side. We do not allow associating variables with non-atomic formulas, because we intend to encode propositional structure only by means of integer linear constraints. We explain how to encode propositional structure in Chapter 7. Importantly, Case (b) of Definition 16 can also express (non-equational) Σ -atoms that hold (or not) unconditionally, by using an auxiliary variable fixed to true (or false). In pursuit of uniformity, we do not provide a specialized case for fixed Σ -literals.

We have not specified where the variable symbols that appear in Σ -definitions (for some signature Σ) come from. Indeed, these variable symbols may belong in \mathcal{V} (and thus also appear in linear constraints) or not (thus being exclusively of theory interest). Note that, as per Definitions 2 and 16, all variables that appear in integer linear constraints are in \mathcal{V} , but the variables that appear in first-order terms inside interface definitions may or may not be in \mathcal{V} .

Definition 17 (MPMT Instance). *An MPMT instance is a quadruple of the form C, O, I, T , where*

- C is a set of integer linear constraints,
- O is an objective function,
- I is a set of Σ -definitions over a signature Σ such that $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$, and
- T is a theory.

An MPMT instance C, O, I, T can be thought of as an extension of the ILP instance C, O with the set of interface definitions I , which are constrained by the theory T . The notion of an MPMT instance (Definition 17) clearly subsumes that of an ILP instance (Definition 3); an ILP instance C, O can be viewed as the MPMT instance $C, O, \emptyset, \emptyset$. We refer to I as the *background theory constraints* in C, O, I, T . Additionally, we refer to the set of MPMT instances that have the same (given) T as their theory as *MP Modulo T instances*.

As explained already, every set of quantifier-free $(\Sigma_Z \cup \Sigma)$ -formulas can be written in separate form, by performing variable abstraction (Example 5). Our syntactic separation between integer linear constraints and Σ -interface definitions is thus not a restriction.

Importantly, Definition 17 does not impose restrictions on the signature (say Σ) of the theory T involved in an MPMT instance C, O, I, T . In the general case, Σ may overlap with Σ_Z , *i.e.*, the axioms in T may use symbols like $+$ and \leq . We study theories whose signatures do not overlap with Σ_Z in Chapter 5, and theories whose signatures do overlap with Σ_Z in Chapter 8. Additionally, there is no fixed relationship between the signatures of I and T , *e.g.*, I may refer to only a subset of the symbols in T , or I may refer to a superset of the symbols in T (*e.g.*, for $T = \emptyset$).

Our setup does not allow variables that appear in linear constraints to take non-integer values. Mixed instances (that involve both integers and reals) are common in MP theory and practice. Our restriction to integer instances is in place to facilitate theoretical analysis. Given that the MP technology (*e.g.*, the B&C framework) that we build upon generally supports mixed instances, it is feasible to extend our techniques to also provide such support.

We subsequently define the notion of a T -model (where T is a theory), which captures what it means for a solution (integer assignment) to be T -consistent.

Definition 18 (*T*-Model). *Let T be a theory, A be an integer assignment (Definition 4), and F be a quantifier-free first-order formula. We say that A is a T -model of F if $\{F\} \cup A$ is $(Z \cup T)$ -satisfiable.*

A T -model A of a formula F (where F can be over any signature) is not a first-order model, but only an integer assignment. While the definition does not introduce restrictions on the signature of F , our intention is to consider T -models in contexts where part of the formula F is integer linear constraints. A provides all the information needed to satisfy these constraints (given that it assigns values to all variables in \mathcal{V}), while guaranteeing the existence of a first-order model that $(Z \cup T)$ -satisfies F . By simply guaranteeing T -satisfiability without providing concrete interpretations of

the non-arithmetic first-order symbols, a T -model abstracts away the first-order part of a solution, which is of no interest to the core ILP solver.

Definition 19 (Optimal Integer Assignment for MPMT). *We say that an integer assignment A is optimal for the instance C, O, I, T if A is a T -model of $C \wedge I$, and*

- *if A is annotated with $-\infty$, then for every integer k , there exists a T -model B of $C \wedge I$ such that $O(B) < k$;*
- *otherwise (i.e., if A is not annotated with $-\infty$), there exists no T -model B of $C \wedge I$ such that $O(B) < O(A)$.*

Definition 20 (Mathematical Programming (MP) Modulo T Problem). *Let T be a first-order theory. The MP Modulo T problem is, for every set of constraints C , objective function O , and set of definitions I ,*

- *to provide an optimal integer assignment A for the MPMT instance C, O, I, T , if an optimal integer assignment exists, or*
- *to report unsatisfiability otherwise.*

As per our problem statement of Definition 20, which is parametric with respect to T , we study solvers that target MPMT instances which have a fixed theory T . It would not be meaningful to define the notion of a solver that solves instances C, O, I, T , with (unrestricted) T comprising part of the user input. That would allow arbitrary first-order sentences as input (as part of T), i.e., the problem formulation would be undecidable.

Chapter 4

BC(T): An Abstract View

This chapter introduces the BC(T) (*Branch and Cut Modulo T*) architecture for MPMT solvers. BC(T) integrates B&C MILP solving and theory reasoning. In BC(T), T is parenthesized to denote that the framework is parametric with respect to the background theory T . BC(T) is meant to be for MPMT what DPLL(T) [77] is for SMT (Section 2.4).

We describe BC(T) as a family of *transition systems*. A transition system is a set of (*transition*) *rules* that relate *states*. A BC(T) state describes a snapshot of the knowledge available to the B&C solver for a specific point in time. Transition rules correspond to the possible computational steps that the solver can take. The rules are non-deterministic, *i.e.*, multiple of them may be applicable on a given state, and each rule may be applicable in multiple ways. A transition system characterizes a family of algorithms, where each algorithm permits only a subset of the transition system’s possible sequences of transitions.

We demonstrate the functionality that different families of background theories require by providing corresponding transition systems. All our transition systems operate on the same kind of states, but differ in their sets of rules. We first provide the \mathcal{G} transition system (where \mathcal{G} stands for *ground*) that does not allow any background constraints. \mathcal{G} is only capable of solving MPMT instances (Definition 17) of the form $C, O, \emptyset, \emptyset$, *i.e.*, ILP instances (Definition 3). We then modify \mathcal{G} to support background theory constraints, and obtain the \mathcal{T} transition system (where \mathcal{T} stands for *theory*). Contrasting

\mathcal{G} and \mathcal{T} highlights the kinds of modifications that a B&C-based solver requires in order to support theories. Subsequent chapters provide transition systems aimed for specific classes of background theories.

4.1 Preliminaries

Definition 21 (State). *A state of $\text{BC}(T)$ is a tuple $P \parallel A$, where P is a set of sets of integer linear constraints, and A is either the constant `None`, or an assignment. If A is an assignment, it can optionally be annotated with the superscript $-\infty$.*

In a state $P \parallel A$, each set of constraints $C \in P$ describes a *subproblem* to be explored. Subproblems are produced by branching (Section 1.4). Our abstract framework maintains a list of open subproblems, in contrast to the implicit subproblems in $\text{DPLL}(T)$ (Section 2.4). There, subproblems are implicit, *i.e.*, backtracking can reconstruct them. B&C-based solvers branch over non-Boolean variables in arbitrary ways. Supporting a sufficiently broad range of B&C strategies mandates that we explicitly record subproblems.

The assignment A kept in a state $P \parallel A$ is the best known (theory-consistent) solution so far, if any. In accordance with Section 1.1, the assignment has a superscript $-\infty$ if it satisfies all the constraints, but is not optimal because the MPMT instance admits solutions with arbitrarily low objective values. If this is the case, it is useful to provide an assignment and to also report that no optimal assignment exists. We use the special constant `None` for the case when no assignment is known. We extend the notation $O(A)$ (defined in Section 1.1) for $A = \text{None}$; $O(\text{None}) = +\infty$.

We define transition systems over states of the form given in Definition 21. Each transition relation is defined as a set of transition rules, which we describe with the following format:

Chapter 4. $BC(T)$: An Abstract View

$$T, I, O \Vdash S \longrightarrow S'$$

if [Conditions under which the rule applies]

[RuleName]: [Description]

This notation means that under the rule RuleName, the transition from S to S' is only permissible in a problem with background theory T , definitions I , and objective function O , where T , I , O , S , and S' combined meet certain conditions. A transition rule thus defines a quinary relation permitting a subset of the tuples (T, I, O, S, S') , where the letters represent the same kinds of objects as above. T , I , and O are not part of the states because they do not change over time. We nevertheless make T , I , and O explicit in transition rules, because certain rules involve restrictions on T , I , and O .

$lb(C, O)$ returns a lower bound for the possible values of the objective function O , subject to the constraints C . lb is a function such that for any A that satisfies C , $O(A) \geq lb(C, O)$. Our transition rules assume a globally defined lb function. In the instantiations that we anticipate, the lower bound comes from solving continuous relaxations of the linear constraints.

In our rules, c always denotes integer linear constraints. C denotes a set of integer linear constraints. $C \cdot c$ stands for the set union $C \cup \{c\}$, under the implicit assumption that $c \notin C$. P stands for a set of subproblems, while A and B are integer assignments. $P \uplus Q$ denotes the union $P \cup Q$, under the implicit assumption that the two sets are disjoint. Primed and subscripted letters are used for the same kinds of objects as their non-primed or non-subscripted counterparts.

A *transition system* is a set of transition rules. A transition system viewed as a quinary relation is the union of the quinary relations corresponding to its rules, *i.e.*, a transition system $R = \{\text{Rule}_0, \text{Rule}_1, \dots, \text{Rule}_{n-1}\}$ enables a transition if one of the rules Rule_i enables it. Formally,

$$T, I, O \Vdash_R S \longrightarrow S' \quad \text{iff} \quad T, I, O \Vdash_{\text{Rule}_i} S \longrightarrow S' \text{ for some } i \in [0, n-1].$$

We use the shorthand $S \xrightarrow{x} S'$ for $T, I, O \Vdash_x S \longrightarrow S'$, where x is either

a transition system or a transition rule, whenever T , I , and O are clear from the context. We also use non-annotated arrows, e.g., $S \longrightarrow S'$, in contexts where the transition rule or system relating the states is either clear or irrelevant. Furthermore, we chain transition arrows where appropriate, i.e., $S_1 \xrightarrow{x} S_2 \xrightarrow{y} S_3$ is shorthand for $T, I, O \Vdash S_1 \xrightarrow{x} S_2$ and $T, I, O \Vdash S_2 \xrightarrow{y} S_3$, for T , I , and O that can be inferred from the context. We finally make extensive use of *transitive closure* (formally defined below) over transition rules and systems.

Definition 22 (Transitive Closure (\longrightarrow^*)). *Let x be a transition system or a transition rule. The transitive closure of x is defined as follows.*

$$T, I, O \Vdash S \xrightarrow{x}^* S'' \text{ iff } S = S'', \text{ or } \begin{cases} T, I, O \Vdash S \xrightarrow{x} S', \text{ and} \\ T, I, O \Vdash S' \xrightarrow{x}^* S'' \end{cases}$$

4.2 The Transition System \mathcal{G}

The transition system \mathcal{G} that we provide in this section only supports MPMT instances of the form $C, O, \emptyset, \emptyset$. In other words, \mathcal{G} serves as an abstract formalization of the B&C framework as it applies to ILP. Some of the rules enable transitions of the form $T, I, O \Vdash S \longrightarrow S'$, with non-empty T and I . The reason is that these rules are re-used by subsequent transition systems that support background theories.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C_i \mid 0 \leq i < n\} \parallel A$$

$$\text{if } \begin{cases} n > 1 \\ \models_{\mathcal{Z}} (C \Leftrightarrow \bigvee_{0 \leq i < n} C_i) \\ C_i \text{ are syntactically distinct} \end{cases}$$

Rule Branch : Case-split on a subproblem C , by replacing it with two or more subproblems C_i .

The rule Branch captures B&C-style branching (Section 1.4). Branch does not enforce a specific branching policy, but rather allows any kind of

Chapter 4. $BC(T)$: An Abstract View

branching such that the disjunction of the resulting subproblems allows the same integer assignments as the original subproblem.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot c\} \parallel A$$

$$\text{if } \begin{cases} C \models_Z c \\ \text{all variable symbols in } c \text{ appear in } C \end{cases}$$

Rule Learn : Add an entailed constraint to a subproblem.

The rule **Learn** enables cut generation (Section 1.3). The introduced cut (inequality) must be Z -entailed by the already existing inequalities. The cut is only added to a single subproblem. Multiple invocations of the rule can add a cut to every subproblem, if it is sound to do so.

$$T, I, O \Vdash P \uplus \{C \cdot c\} \parallel A \longrightarrow P \cup \{C\} \parallel A$$

$$\text{if } C \models_Z c$$

Rule Forget : Remove a constraint entailed by the remaining constraints of a subproblem.

The rule **Forget** is symmetric to **Learn**, *i.e.*, **Forget** removes an inequality. The inequality needs to be Z -entailed by the remaining ones, in order not to underconstrain the subproblem. **Forget** enables garbage collection for problems with prohibitively many cuts.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \parallel A$$

$$\text{if } C \text{ is integer-inconsistent}$$

Rule Drop : Eliminate a subproblem that cannot lead to any solution.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \parallel A$$

$$\text{if } \begin{cases} A \neq \text{None} \\ \text{lb}(C, O) \geq O(A) \end{cases}$$

Rule Prune : Eliminate a subproblem that cannot lead to any solution better than the one already known.

The rules Drop and Prune eliminate subproblems that need not be considered further. While this is not explicit in the transition system, both detecting integer-inconsistency (for Drop) and obtaining a bound for all possible assignments (Prune) can be implemented by solving the continuous relaxation of the subproblem in question (e.g., via Simplex, as described in Section 1.2).

$$\emptyset, \emptyset, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C\} \parallel A'$$

$$\text{if } \begin{cases} A' \text{ satisfies } C \\ O(A') < O(A) \end{cases}$$

Rule Improve : A solution to a subproblem becomes the new incumbent solution, as long as it improves upon the objective value of the previous solution.

$$\emptyset, \emptyset, O \Vdash P \uplus \{C\} \parallel A \longrightarrow \emptyset \parallel A'^{-\infty}$$

$$\text{if } \begin{cases} A' \text{ satisfies } C \\ \text{for any } k, \text{ there exists } B \text{ such that} \\ \quad - B \text{ satisfies } C \\ \quad - O(B) < k \end{cases}$$

Rule Unbounded : A subproblem has solutions with arbitrarily low objective values. Record one of them. There is no need to consider other subproblems.

The rules Improve and Unbounded record integer assignments. For Improve, the subproblem that produced the assignment remains intact, be-

cause we do not require that *Improve* only records the subproblem's best possible assignment. Subsequent application of *Prune* can eliminate the subproblem if it has been solved to optimality. Application of the *Unbounded* rule means immediate termination. In the anticipated implementations, integer linear solutions (which can be recorded via *Improve*) are obtained from sufficiently strengthened continuous relaxations. Continuous relaxations are also tied to *Unbounded*; the Simplex algorithm extended with integer techniques [13] can detect unbounded objective functions.

Definition 23 (The Transition System \mathcal{G}).

$$\mathcal{G} = \{\text{Branch, Learn, Forget, Drop, Prune, Improve, Unbounded}\}$$

4.3 The Transition System \mathcal{T}

In this section, we modify the \mathcal{G} transition system to accommodate background theories, *i.e.*, to support MPMT instances (Definition 17) of the form C, O, I, T , for arbitrary I and T . The resulting transition system is called \mathcal{T} .

The prefix \mathcal{T} in rule names (*e.g.*, in \mathcal{T} -Learn) denotes theory integration. The prefix also denotes the association of such rules with the transition system \mathcal{T} . \mathcal{T} does not stand for some specific background theory T , hence the typographic distinction. The theory involved is explicit in our rules.

$$\begin{aligned} T, I, O \quad &\Vdash \quad P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot c\} \parallel A \\ \text{if } &\left\{ \begin{array}{l} C \wedge I \models_{\mathcal{Z} \cup T} c \\ \text{all variable symbols in } c \text{ appear in } C \text{ or in } I \end{array} \right. \end{aligned}$$

Rule \mathcal{T} -Learn : Theory-aware counterpart of *Learn*.

\mathcal{T} -Learn is a very powerful rule. It allows strengthening a subproblem with any integer linear constraint entailed by the already-existing constraints, in conjunction with the combined theory $\mathcal{Z} \cup T$. \mathcal{T} -Learn thus permits combined $(\mathcal{Z} \cup T)$ -reasoning. This capability by no means implies that a solver for $\mathcal{Z} \cup T$ is a prerequisite for implementing \mathcal{T} . Rather, an

implementation of \mathcal{T} may apply \mathcal{T} -Learn in a very restricted and targeted fashion. The rule \mathcal{T} -Learn is meant to capture a wide range of possible learning techniques. We prove soundness of a transition system that includes the general \mathcal{T} -Learn rule, thus justifying any restriction thereof. The completeness results provided in subsequent chapters rely on restricted versions of \mathcal{T} -Learn.

Importantly, \mathcal{T} -Learn is powerful enough to report $(\mathcal{Z} \cup T)$ -infeasibility (where T is the applicable theory) of a subproblem. If $C \wedge I$ (where I is the set of interface definitions that apply) is $(\mathcal{Z} \cup T)$ -inconsistent, then $C \wedge I \models_{\mathcal{Z} \cup T} 0 \leq -1$, which means that Learn can render the set of integer linear constraints infeasible, and can thus be followed by Drop.

The rules Improve and Unbounded present in the transition system \mathcal{G} cannot be applied for a non-empty theory T and a non-empty set of interface definitions. If we were to enable them, they would record T -inconsistent solutions. Improve and Unbounded are adapted to ensure that we only record T -models (Definition 18). We obtain the rules \mathcal{T} -Improve and \mathcal{T} -Unbounded.

$$\begin{array}{l} T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C\} \parallel A' \\ \text{if } \begin{cases} A' \text{ is a } T\text{-model of } C \wedge I \\ O(A') < O(A) \end{cases} \end{array}$$

Rule \mathcal{T} -Improve : Theory-aware counterpart of Improve.

$$\begin{array}{l} T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow \emptyset \parallel A'^{-\infty} \\ \text{if } \begin{cases} A' \text{ is a } T\text{-model of } C \wedge I \\ \text{for any } k, \text{ there exists } B \text{ such that} \\ \quad - B \text{ is a } T\text{-model of } C \wedge I \\ \quad - O(B) < k \end{cases} \end{array}$$

Rule \mathcal{T} -Unbounded : Theory-aware counterpart of Unbounded.

Definition 24 (The Transition System \mathcal{T}).

$$\mathcal{T} = \{\text{Branch, Forget, Drop, Prune,} \\ \mathcal{T}\text{-Learn, } \mathcal{T}\text{-Improve, } \mathcal{T}\text{-Unbounded}\}$$

The transition system \mathcal{T} is derived from \mathcal{G} by replacing the rules Learn, Improve, and Unbounded with their theory-aware counterparts. We proceed to prove that \mathcal{G} and \mathcal{T} enable the same transitions for MPMT instances of the form $C, O, \emptyset, \emptyset$, i.e., for the instances that both systems support.

Lemma 1. $\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{G}]{} P' \parallel A'$ iff $\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{T}]{} P' \parallel A'$.

Proof. Assume that $\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{G}]{} P' \parallel A'$. It is the case that $\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[x]{} P' \parallel A'$, where x is one of the transition rules of \mathcal{G} . We case-split on x .

Branch, Forget, Drop, Prune:

Clearly $\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{T}]{} P' \parallel A'$, because these rules are also provided by \mathcal{T} .

Learn:

$\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{T}\text{-Learn}]{} P' \parallel A'$, which entails our proof obligation.

Improve:

$\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{T}\text{-Improve}]{} P' \parallel A'$, which entails our proof obligation.

Unbounded:

$\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{T}\text{-Unbounded}]{} P' \parallel A'$, which entails our proof obligation.

The opposite direction is very similar. □

Lemma 2. $\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{G}]{}^* P' \parallel A'$ iff $\emptyset, \emptyset, O \Vdash P \parallel A \xrightarrow[\mathcal{T}]{}^* P' \parallel A'$.

Proof. Induction on the length of the sequence of transitions relating $P \parallel A$ and $P' \parallel A'$. □

4.4 Soundness

A *starting state* for $\text{BC}(T)$ is a state of the form $C \parallel \text{None}$, where C is the set of integer linear constraints of an MP Modulo T instance. A state of the form $\emptyset \parallel A$ is called *final*.

As we prove in this section, our transition systems are *sound*. Consider an MPMT instance of the form C, O, I, T , and a transition system X . Soundness of X means that any solution reachable through X is correct, *i.e.*, if $T, I, O \Vdash \{C\} \parallel \text{None} \xrightarrow[X]^* \emptyset \parallel A$, then if $A \neq \text{None}$, A is an optimal T -model (Theorem 1), otherwise the instance is unsatisfiable (Theorem 2).

Lemma 3. *If*

$$T, I, O \Vdash P \parallel A \xrightarrow[\tau]{} P' \parallel A',$$

and for some $C \in P$, there exists a T -model B of $C \wedge I$, then one of the following conditions holds:

- (a) *either $O(A') \leq O(B)$, or*
- (b) *B is a T -model of $C' \wedge I$ for some $C' \in P'$.*

Proof. Assume that there exists an assignment B such that B is a T -model of $C \wedge I$ for some $C \in P$. If $C \in P'$, then (b) holds trivially. If $C \notin P'$, then the transition cannot possibly be Drop; we cannot possibly apply Drop on C because B is an assignment that satisfies C . For the other rules:

Branch:

There exist $C_0, \dots, C_{n-1} \in P'$ such that $\models_{\mathcal{Z}} (C \Leftrightarrow \bigvee_{0 \leq i < n} C_i)$. Thus, B is a T -model of $C_i \wedge I$ for some i ($0 \leq i < n$). Condition (b) holds.

Prune:

$O(A') = O(A) \leq \text{lb}(C, O) \leq O(B)$. Condition (a) holds.

\mathcal{T} -Learn, Forget:

There exists a subproblem $C' \in P'$ such that $C \wedge I \models_{\mathcal{Z} \cup T} C' \wedge I$. B is a T -model of $C' \wedge I$. Condition (b) holds.

\mathcal{T} -Improve, \mathcal{T} -Unbounded:

$O(A') \leq O(B)$. Condition (a) holds.

□

Lemma 4. *If $T, I, O \Vdash P \parallel A \xrightarrow{\mathcal{T}} P' \parallel A'$, then $O(A') \leq O(A)$.*

Proof. The conditions under which \mathcal{T} -Improve and \mathcal{T} -Unbounded are applicable imply that $O(A') \leq O(A)$. For any other rule, $A = A'$. □

Lemma 5. *If*

$$T, I, O \Vdash P \parallel A \xrightarrow{\mathcal{T}}^* P' \parallel A',$$

and for some $C \in P$, there exists a T -model B of $C \wedge I$, then one of the following conditions holds:

- (a) *either $O(A') \leq O(B)$, or*
- (b) *B is a T -model of $C' \wedge I$ for some $C' \in P'$.*

Proof. We induct on the length n of the sequence of transitions.

Induction Base:

$n = 0$. $P \parallel A = P' \parallel A'$; obvious.

Induction Step:

Assume that the property holds for any sequence of $n - 1$ transitions, where $n \geq 1$. We prove that it holds for any sequence of transitions $P_0 \parallel A_0 \xrightarrow{\mathcal{T}} \dots \xrightarrow{\mathcal{T}} P_{n-1} \parallel A_{n-1} \xrightarrow{\mathcal{T}} P_n \parallel A_n$. Assume there is an assignment B such that B is a T -model of $C_0 \wedge I$ for some $C_0 \in P_0$. By the induction hypothesis, one of the two following conditions holds:

- $O(A_{n-1}) \leq O(B)$: then $O(A_n) \leq O(A_{n-1}) \leq O(B)$ from Lemma 4.
- B is a T -model of $C_{n-1} \wedge I$ for some subproblem $C_{n-1} \in P_{n-1}$: our proof obligation follows from Lemma 3 applied to the transition

$$T, I, O \Vdash P_{n-1} \parallel A_{n-1} \xrightarrow{\mathcal{T}} P_n \parallel A_n.$$

□

Lemma 6. *If $T, I, O \Vdash P \parallel A \xrightarrow{\mathcal{T}} P' \parallel A'$, then $\bigvee_{C \in P'} C \models_{\mathcal{Z}} \bigvee_{C \in P} C$.*

Proof. We case-split on the \mathcal{T} transitions.

Branch:

The rule replaces a subproblem C with a set of subproblems whose disjunction is \mathcal{Z} -equivalent to C .

Forget:

The rule substitutes a subproblem for a \mathcal{Z} -equivalent one.

Drop, Prune:

The disjunction of the subproblems in P' can only be stronger, because a subproblem is eliminated and the rest of the subproblems remain intact.

\mathcal{T} -Learn:

The rule adds a constraint to a subproblem, and therefore makes the disjunction in P' stronger.

\mathcal{T} -Improve:

None of the subproblems changes.

\mathcal{T} -Unbounded:

$$\bigvee_{C \in P'} C \equiv \text{false}; \text{false} \models_{\mathcal{Z}} \bigvee_{C \in P} C.$$

□

Lemma 7. *If $T, I, O \Vdash P \parallel A \xrightarrow[\mathcal{T}]^* P' \parallel A'$, then $\bigvee_{C \in P'} C \models_{\mathcal{Z}} \bigvee_{C \in P} C$.*

Proof. Induction on the length of the sequence of transitions and application of Lemma 6. □

Theorem 1 (Soundness of \mathcal{T} (Satisfiable Instances)). *If*

$$T, I, O \Vdash \{C\} \parallel \text{None} \xrightarrow[\mathcal{T}]^* \emptyset \parallel A$$

where $A \neq \text{None}$, then

(a) *A is a T -model of $C \wedge I$, and*

(b) *there exists no T -model B of $C \wedge I$ such that $O(B) < O(A)$.*

Chapter 4. $BC(T)$: An Abstract View

Proof.

- (a) The sequence of transitions from $\{C\} \parallel \text{None}$ to $\emptyset \parallel A$ has to be of the form

$$\{C\} \parallel \text{None} \xrightarrow[\mathcal{T}]^* P_1 \parallel A_1 \xrightarrow[\mathcal{T}\text{-Improve}/\mathcal{T}\text{-Unbounded}]{} P \parallel A \xrightarrow[\mathcal{T}]^* \emptyset \parallel A.$$

The sequence contains at least one \mathcal{T} -Improve or \mathcal{T} -Unbounded step, as these are the only rules that modify the assignment. Consider the last such step. The conditions on \mathcal{T} -Improve and \mathcal{T} -Unbounded imply that A is a T -model of $I \wedge \bigvee_{C_1 \in P_1} C_1$. From Lemma 7, $\bigvee_{C_1 \in P_1} C_1 \models_{\mathcal{Z}} C$. Therefore, A is a T -model of $C \wedge I$.

- (b) Follows from Lemma 5.

□

Corollary 2 (Soundness of \mathcal{G} (Satisfiable Instances)). *If*

$$\emptyset, \emptyset, O \Vdash \{C\} \parallel \text{None} \xrightarrow[\mathcal{G}]^* \emptyset \parallel A$$

where $A \neq \text{None}$, then

- (a) A satisfies C , and
(b) there exists no integer assignment B satisfying C such that $O(B) < O(A)$.

Theorem 2 (Soundness of \mathcal{T} (Unsatisfiable Instances)). *If*

$$T, I, O \Vdash \{C\} \parallel \text{None} \xrightarrow[\mathcal{T}]^* \emptyset \parallel \text{None},$$

then $C \wedge I$ is $(\mathcal{Z} \cup T)$ -unsatisfiable.

Proof. Assume that there exists a T -model A of $C \wedge I$. Then, from Lemma 5 either there exists $C' \in \emptyset$ such that A is a T -model of $C' \wedge I$, which cannot possibly be true, or $+\infty = O(\text{None}) \leq O(A)$, which cannot be true either, because $O(A)$ is finite. □

Corollary 3 (Soundness of \mathcal{G} (Unsatisfiable Instances)). *If*

$$\emptyset, \emptyset, O \Vdash \{C\} \parallel \text{None} \xrightarrow[\mathcal{G}]{}^* \emptyset \parallel \text{None},$$

then C is integer-unsatisfiable.

Chapter 5

Nelson-Oppen via $\text{BC}(T)$

The theory-aware transition system \mathcal{T} is sound, *i.e.*, its rules are so designed that they cannot lead to wrong answers (Theorems 1 and 2). Termination is however not guaranteed, *i.e.*, there can be infinite sequences of transitions. In this chapter, we study how $\text{BC}(T)$ can be applied in a complete way. Obtaining completeness involves assumptions about the background theory. We focus on stably-infinite theories (Definition 12). As discussed in Section 2.3, stably-infinite theories enable the NO scheme for combining decision procedures [73, 86, 58].

The NO scheme is non-deterministic and can be implemented in multiple ways. We provide a B&C-based implementation, which we formalize by means of the transition system \mathcal{A} . \mathcal{A} performs the kind of search that NO mandates to achieve completeness. We prove that \mathcal{A} provides a complete optimization procedure for the MP Modulo T problem, under the conditions that

- (a) T is stably-infinite, and
- (b) the signature of T is disjoint from $\Sigma_{\mathcal{Z}}$ (Definition 11).

Non-stably-infinite theories have little practical relevance in the context of MPMT, *i.e.*, Condition (a) is not a significant restriction. The reason is that \mathcal{Z} (the underlying theory of integer linear constraints) only allows interpretations with infinite domains. It follows that for every background theory T , the combined theory $\mathcal{Z} \cup T$ also only allows interpretations with infinite

domains. Solving $\mathcal{Z} \cup T$ (and by extension, MP Modulo T) thus involves determining whether the background constraints can be satisfied modulo T by an interpretation with an infinite domain. This is a problem orthogonal to MPMT, and thus a problem that we do not study.

In contrast, Condition (b) that requires signature disjointness does have practical implications. In Chapter 8, we overcome Condition (b) by supporting within $\text{BC}(T)$ a class of theories whose signatures overlap with $\Sigma_{\mathcal{Z}}$.

5.1 Arrangement Search and Optimization

We provide an example that highlights how NO-style non-deterministic choice of an arrangement (Fact 1) applies to MPMT.

Example 11. *We return to the constraints C and I of Example 9.*

$$\begin{aligned} C &= \{ \quad 3 \cdot y - 2 \cdot x \leq 3, \quad 3 \cdot x + y \leq 12, \quad x \geq 0, \quad y \geq 0, \quad f_x - f_y \geq 1 \} \\ I &= \{ \quad \quad \quad f_x = f(x), \quad \quad \quad f_y = f(y) \quad \quad \quad \} \end{aligned}$$

The atomic formulas in C are in the signature $\Sigma_{\mathcal{Z}}$ (Definition 10). The atomic formulas in I are in the signature Σ_f (Definition 7). The set of variables shared by C and I is $V = \{x, y, f_x, f_y\}$. Let $O = -y$. As in Example 9, we discuss the MP Modulo \emptyset instance C, O, I, \emptyset .

We consider the following equivalence relations over V :

$$\begin{aligned} E_1 &= \{\{x, y\}, \quad \{f_x\}, \quad \{f_y\}\} \\ E_2 &= \{\{x, f_x\}, \quad \{y, f_y\} \quad \} \\ E_3 &= \{\{x, f_y\}, \quad \{y, f_x\} \quad \} \end{aligned}$$

We examine the corresponding arrangements:

$\alpha(V, E_1) = \{x = y, x \neq f_x, x \neq f_y, y \neq f_x, y \neq f_y, f_x \neq f_y\}$:
 $\alpha(V, E_1) \wedge C$ is \mathcal{Z} -satisfiable, with the assignment $A_1 = \{x \mapsto 3, y \mapsto 3, f_x \mapsto 1, f_y \mapsto 0\}$ as a witness. However, $\alpha(V, E_1) \wedge I$ is \emptyset -unsatisfiable, because $x = y \wedge f_x \neq f_y \wedge f_x = f(x) \wedge f_y = f(y)$ is \emptyset -unsatisfiable.

$\alpha(V, E_2) = \{x \neq y, x = f_x, x \neq f_y, y \neq f_x, y = f_y, f_x \neq f_y\}$:

$\alpha(V, E_2) \wedge C$ is \mathcal{Z} -satisfiable, with the assignment $A_2 = \{x \mapsto 3, y \mapsto 2, f_x \mapsto 3, f_y \mapsto 2\}$ as a witness. At the same time, $\alpha(V, E_2) \wedge I$ is \emptyset -satisfiable. A_2 is optimal for the MP Modulo \emptyset instance.

$\alpha(V, E_3) = \{x \neq y, x \neq f_x, x = f_y, y = f_x, y \neq f_y, f_x \neq f_y\}$:

From the integer constraints $\alpha(V, E_3) \wedge C$, it follows that $y = f_x > f_y = x$. Optimal solutions thus lie in the area above the diagonal (i.e., the area $y > x$) in Figure 3.1. It is visually evident that the family of optimal assignments have $x = 0$ and $y = 1$. One such assignment is $A_3 = \{x \mapsto 0, y \mapsto 1, f_x \mapsto 1, f_y \mapsto 0\}$.

From Fact 1, with either of E_2 and E_3 as the choice of equivalence relation, it follows that $C \wedge I$ is $(\mathcal{Z} \cup \emptyset)$ -satisfiable. Importantly, neither $\alpha(V, E_2)$, nor $\alpha(V, E_3)$ is $(\mathcal{Z} \cup \emptyset)$ -entailed by $C \wedge I$, i.e., solving the MPMT instance inherently involves search. However, only E_2 can lead to an optimal assignment ($y = 2$). Conversely, arrangement search within $BC(T)$ needs to be optimization-aware.

Arrangements (Definition 13) like $\alpha(V, E_2)$ and $\alpha(V, E_3)$ in Example 11 contain disequalities, which are not directly encodable as integer linear constraints. Nevertheless, a disequality of the form $x \neq y$ is \mathcal{Z} -entailed by either of the inequalities $x - y \leq -1$ and $y - x \leq -1$. Such inequalities (which we call *difference constraints*) are the machinery for forming arrangements that our NO-implementing variant of $BC(T)$ (presented in this chapter) employs.

5.2 Formal Preliminaries

We formally define difference constraints, and provide the notation and concepts needed to facilitate our presentation of a $BC(T)$ -based implementation of the NO scheme.

Definition 25 (Difference Constraint). A difference constraint is an integer linear constraint of the form $v \leq c$, $-v \leq c$, or $v - w \leq c$, where v and w are distinct variable symbols in \mathcal{V} and c is an integer constant.

Just like we do for general integer linear constraints (Definition 2), we syntactically diverge from Definition 25 (e.g., with operators like $>$ and $=$) in contexts where it is clear that we can obtain difference constraints. Given a set of integer linear constraints C , we denote by $\text{diffs}(C)$ the subset of the constraints in C that are difference constraints:

$$\text{diffs}(C) = \{c \mid c \in C, c \text{ is a difference constraint}\}$$

Difference constraints merit special treatment because they are easier to interpret by procedures that do not understand full \mathcal{Z} , but only limited fragments thereof. Conversely, difference constraints allow for finer-grained interaction between the core solver and theory solvers.

Given a set of integer linear constraints C and a finite set of variables $V \subseteq \mathcal{V}$, let $\text{arrange}(C, V)$ be the set of (dis)equalities entailed by $\text{diffs}(C)$:

$$\begin{aligned} \text{arrange}(C, V) = & \{x = y \mid x, y \in V, x \prec y, \text{diffs}(C) \models_{\mathcal{Z}} x = y\} \cup \\ & \{x \neq y \mid x, y \in V, x \prec y, \text{diffs}(C) \models_{\mathcal{Z}} x \neq y\} \end{aligned}$$

For any C and any V , $\text{arrange}(C, V)$ is in the empty signature, i.e., any theory solver can interpret it. By construction, $C \models_{\mathcal{Z}} \text{arrange}(C, V)$. Clearly, $\text{diffs}(C) \models_{\mathcal{Z}} x = y$ iff $\text{diffs}(C) \models_{\mathcal{Z}} x - y \leq 0$ and $\text{diffs}(C) \models_{\mathcal{Z}} y - x \leq 0$; $\text{diffs}(C) \models_{\mathcal{Z}} x \neq y$ iff $\text{diffs}(C) \models_{\mathcal{Z}} x - y \leq k$ for some $k < 0$, or $\text{diffs}(C) \models_{\mathcal{Z}} y - x \leq k$ for some $k < 0$. $\text{arrange}(C, V)$ can be constructed by representing difference constraints as a weighted graph [74], producing the set of difference constraints corresponding to paths (and not just edges) in this graph, and by looking in the set for difference constraints that \mathcal{Z} -entail either $x = y$ or $x \neq y$, for every $x, y \in V$.

Given an MPMT instance C, O, I, T , and a subproblem C' that is produced in the process of solving the instance, $\text{diffs}(C')$ additionally determines the set of background theory constraints that hold for C' ; let

$$\begin{aligned} \theta(C, I) = & \{F \mid [\nu > 0 \Leftrightarrow F] \in I, \text{diffs}(C) \models_{\mathcal{Z}} \nu > 0\} \cup \\ & \{\neg F \mid [\nu > 0 \Leftrightarrow F] \in I, \text{diffs}(C) \models_{\mathcal{Z}} \nu \leq 0\} \cup \\ & \{\nu = t \mid [\nu = t] \in I\} \end{aligned} \quad (5.1)$$

$\theta(C, I)$ is in the same signature as I . By construction, $C \wedge I \models_{\Sigma} \theta(C, I)$.

Furthermore, for a set of integer linear constraints C and a set of Σ -interface definitions I , we denote by $\text{shared}(C, I)$ the set of variables which appear both in some $\Sigma_{\mathcal{Z}}$ -atomic formula in $C \wedge I$, and in some Σ -atomic formula in I . $\text{shared}(C, I)$ is meant to be the set of variables that our arrangement search needs to consider. Note that even I contains $\Sigma_{\mathcal{Z}}$ -atomic formulas (of the form $v > 0$, where some interface definition $v > 0 \Leftrightarrow F$ is in I), which do need to be taken into account by shared for the $\Sigma_{\mathcal{Z}}$ -side.

5.3 The Transition System \mathcal{A}

We define a transition system that has just enough power to accommodate a stably-infinite background theory. To this end, we replace some of the theory integration rules provided in Section 4.3 with alternative rules that are meant to facilitate exchange of (dis)equalities. (Dis)equalities are the building block of arrangements (Definition 13), and are thus central to NO.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot x - y \leq 0 \cdot y - x \leq 0\} \parallel A$$

$$\text{if } \begin{cases} x, y \in \text{shared}(C, I) \\ \theta(C, I) \wedge \text{arrange}(C, \text{shared}(C, I)) \models_T x = y \\ \text{arrange}(C, \text{shared}(C, I)) \not\models x = y \\ \text{arrange}(C, \text{shared}(C, I)) \not\models x \neq y \end{cases}$$

Rule \mathcal{A} -Propagate⁺ : Learn a T -entailed equality between two variables.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot x - y \leq -1, C \cdot y - x \leq -1\} \parallel A$$

$$\text{if } \begin{cases} x, y \in \text{shared}(C, I) \\ \theta(C, I) \wedge \text{arrange}(C, \text{shared}(C, I)) \models_T x \neq y \\ \text{arrange}(C, \text{shared}(C, I)) \not\models x = y \\ \text{arrange}(C, \text{shared}(C, I)) \not\models x \neq y \end{cases}$$

Rule $\mathcal{A}\text{-Propagate}^-$: Learn a T -entailed disequality between two variables.

The rules $\mathcal{A}\text{-Propagate}^+$ and $\mathcal{A}\text{-Propagate}^-$ only allow the theory solver to access the theory atoms $\theta(C, I)$ that hold locally, and the partial knowledge of an arrangement ($\text{arrange}(C, \text{shared}(C, I))$) available locally. The theory solver in turn notifies the ILP core about T -entailed (dis)equalities. A T -entailed equality $x = y$ can be encoded as the pair of difference constraints $x - y \leq 0$ and $y - x \leq 0$ (rule $\mathcal{A}\text{-Propagate}^+$). $\mathcal{A}\text{-Propagate}^-$ is less straightforward, because disequalities are not directly expressible in ILP. By replacing a subproblem C with the pair of subproblems $C \cdot x - y \leq -1$ and $C \cdot y - x \leq -1$, we rule out $x = y$. $\mathcal{A}\text{-Propagate}^-$ thus forces branching.

$$T, I, O \Vdash P \uplus \{C \cdot c\} \parallel A \longrightarrow P \cup \{C\} \parallel A$$

$$\text{if } \begin{cases} C \models_{\mathcal{Z}} c \\ \text{if } c \text{ is a difference constraint, then } \text{diffs}(C) \models_{\mathcal{Z}} c \end{cases}$$

Rule $\mathcal{A}\text{-Forget}$: Remove a constraint from a subproblem, without weakening its set of difference constraints.

The general Forget rule does not weaken the subproblem C it modifies. However, it may weaken $\text{diffs}(C)$. This is undesirable, given that we rely on $\text{diffs}(C)$ to obtain an arrangement. We provide a specialized rule that is guaranteed to not weaken $\text{diffs}(C)$.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot x < y, C \cdot x = y, C \cdot x > y\} \parallel A$$

$$\text{if } \begin{cases} x, y \in \text{shared}(C, I) \\ \text{arrange}(C, \text{shared}(C, I)) \not\models x = y \\ \text{arrange}(C, \text{shared}(C, I)) \not\models x \neq y \end{cases}$$

Rule \mathcal{A} -Branch : Branch on a pair of shared variables.

The rule \mathcal{A} -Branch is a specialized version of Branch (present in the transition systems \mathcal{G} and \mathcal{T}). \mathcal{A} -Branch is aimed for branching on variables shared between the integer linear constraints and the theory constraints. Our motivation for providing \mathcal{A} -Branch is that its application is guaranteed to make progress towards an arrangement, *i.e.*, towards a state that enables NO-style compositional reasoning.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \parallel A$$

$$\text{if } \theta(C, I) \wedge \text{arrange}(C, \text{shared}(C, I)) \text{ is } T\text{-inconsistent}$$

Rule \mathcal{A} -Unsat : Eliminate a theory-inconsistent subproblem.

The rule \mathcal{A} -Unsat can be thought of as the theory counterpart of Drop, *i.e.*, it eliminates a subproblem whose theory constraints we cannot possibly satisfy. Remember that the \mathcal{T} transition system can carry out the same task via a combination of \mathcal{T} -Learn and Drop (Section 4.3). Therefore, a rule like \mathcal{A} -Unsat would be redundant in \mathcal{T} .

Non-equational interface definitions (Definition 16) amount to a form of propositional structure. Such structure necessitates propositional search and propagation, which can be captured by the powerful rules present in the \mathcal{T} transition system, but not by the restricted \mathcal{A} -prefixed rules. We conversely define rules specialized for the propositional structure that arises in MPMT.

$$\begin{aligned}
 T, I, O \quad \Vdash \quad & P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot v > 0\} \parallel A \\
 & \text{if for some } F, \begin{cases} \text{diffs}(C) \not\models_{\mathcal{Z}} v > 0 \\ [v > 0 \Leftrightarrow F] \in I \\ \theta(C, I) \wedge \text{arrange}(C, \text{shared}(C, I)) \models_T F \end{cases}
 \end{aligned}$$

Rule \mathcal{B} -Propagate⁺ : Given an interface definition $v > 0 \Leftrightarrow F$, if F holds, propagate $v > 0$.

$$\begin{aligned}
 T, I, O \quad \Vdash \quad & P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot v \leq 0\} \parallel A \\
 & \text{if for some } F, \begin{cases} \text{diffs}(C) \not\models_{\mathcal{Z}} v \leq 0 \\ [v > 0 \Leftrightarrow F] \in I \\ \theta(C, I) \wedge \text{arrange}(C, \text{shared}(C, I)) \models_T \neg F \end{cases}
 \end{aligned}$$

Rule \mathcal{B} -Propagate⁻ : Given an interface definition $v > 0 \Leftrightarrow F$, if $\neg F$ holds, propagate $v \leq 0$.

An interface definition of the form $v > 0 \Leftrightarrow F$ establishes a correspondence between the variable v (which we expect to be bounded in $[0, 1]$ by integer linear constraints) and the non-arithmetic atomic formula F . Based on the use of the θ operator (Equation 5.1) in our \mathcal{A} -prefixed transition rules, the theory solver gets access to F (respectively $\neg F$) if $v > 0$ (respectively $v \leq 0$) holds. The rules \mathcal{B} -Propagate⁺ and \mathcal{B} -Propagate⁻ propagate information in the opposite direction, *i.e.*, from the theory solver to the ILP solver.

$$\begin{aligned}
 T, I, O \quad \Vdash \quad & P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot v \leq 0, C \cdot v > 0\} \parallel A \\
 & \text{if } \begin{cases} \text{diffs}(C) \not\models_{\mathcal{Z}} v \leq 0 \\ \text{diffs}(C) \not\models_{\mathcal{Z}} v > 0 \\ \text{for some } F, [v > 0 \Leftrightarrow F] \in I \end{cases}
 \end{aligned}$$

Rule \mathcal{B} -Branch : Given an interface definition $v > 0 \Leftrightarrow F$, obtain a truth value for F by branching on $v > 0$.

\mathcal{B} -Branch branches on literals $v > 0$ such that a relevant interface definition $v > 0 \Leftrightarrow F$ exists. In the resulting subproblems, the theory solver has access to an additional theory atom, obtained through θ in our rules. Clearly, applying \mathcal{B} -Branch eventually exhausts all possibilities with respect to the finitely many interesting inequalities $v > 0$.

Definition 26 (The Transition System \mathcal{A}).

$$\begin{aligned} \mathcal{A} = \{ & \text{Branch, Learn, Drop, Prune,} \\ & \mathcal{T}\text{-Improve, } \mathcal{T}\text{-Unbounded,} \\ & \mathcal{A}\text{-Propagate}^+, \mathcal{A}\text{-Propagate}^-, \\ & \mathcal{A}\text{-Forget, } \mathcal{A}\text{-Branch, } \mathcal{A}\text{-Unsat,} \\ & \mathcal{B}\text{-Propagate}^+, \mathcal{B}\text{-Propagate}^-, \mathcal{B}\text{-Branch} \} \end{aligned}$$

\mathcal{A} provides the general Branch and Learn rules to capture ILP branching and learning strategies that the \mathcal{A} -prefixed and \mathcal{B} -prefixed rules may not be powerful enough for. Additionally, this separation between ILP steps and other steps helps us provide a fine-grained theoretical analysis of the interaction between the solvers involved.

Example 12. *We demonstrate how Example 9 can be solved with the \mathcal{A} transition system. Let*

$$A = \{x \mapsto 3, y \mapsto 2, f_x \mapsto 1, f_y \mapsto 0\}.$$

Figure 5.1 provides a possible sequence of \mathcal{A} transitions for solving the MP Modulo \emptyset instance. Note that the continuous relaxation of the constraints C provides an integer solution with $x = y$. Such a solution cannot be theory consistent, and thus cannot be recorded via \mathcal{T} -Improve. From an ILP perspective, we are stuck with a problem that has known solutions that cannot be recorded, for reasons unknown to the ILP solver. This necessitates input from the background solver. x and y are variables shared between C and I , i.e., x, y is a valid pair for \mathcal{A} -Branch to act on (transition $S_0 \longrightarrow S_1$). Any optimal solution to the subproblem $C \cdot x > y$ (e.g., the assignment A) is a \emptyset -model of $C \wedge I$. Therefore, such a solution can be recorded via \mathcal{T} -Improve ($S_1 \longrightarrow S_2$).

	$\{C\}$	\parallel	None	(S_0)
$\xrightarrow{\mathcal{A}\text{-Branch}}$	$\{C \cdot x < y, C \cdot x = y, C \cdot x > y\}$	\parallel	None	(S_1)
$\xrightarrow{\mathcal{T}\text{-Improve}}$	$\{C \cdot x < y, C \cdot x = y, C \cdot x > y\}$	\parallel	A	(S_2)
$\xrightarrow{\text{Prune}}$	$\{C \cdot x < y, C \cdot x = y\}$	\parallel	A	(S_3)
$\xrightarrow{\mathcal{A}\text{-Propagate}^+}$	$\{C \cdot x < y, C \cdot x = y \cdot f_x = f_y\}$	\parallel	A	(S_4)
$\xrightarrow{\text{Drop}}$	$\{C \cdot x < y\}$	\parallel	A	(S_5)
$\xrightarrow{\text{Learn}}$	$\{C \cdot x < y \cdot y \leq 1\}$	\parallel	A	(S_6)
$\xrightarrow{\text{Prune}}$	\emptyset	\parallel	A	(S_7)

 Figure 5.1: \mathcal{A} Transitions (Examples 9 and 12)

The subproblem can then be pruned, because $O(A)$ is a lower bound for its possible solutions ($S_2 \longrightarrow S_3$). For the subproblem $C \cdot x = y$, $x = y$ is useful information exported via `arrange(C, shared(C, I))` to the theory solver, based on which the theory solver produces the entailed equality $f_x = f_y$ ($S_3 \longrightarrow S_4$). $f_x = f_y$ leads to a linear formulation that is integer-inconsistent, and can thus be dropped ($S_4 \longrightarrow S_5$). For the subproblem $C \cdot x < y$, it holds that $y \leq 1$ ($S_5 \longrightarrow S_6$). It follows that the subproblem does not admit solutions better than the one already known, which has objective value $-y = -2$. The last remaining subproblem can thus be pruned ($S_6 \longrightarrow S_7$), i.e., we have solved the problem to optimality.

Lemma 8. *If*

$$T, I, O \Vdash P \parallel A \xrightarrow{\mathcal{A}} P' \parallel A',$$

then

$$T, I, O \Vdash P \parallel A \xrightarrow{\mathcal{T}}^* P' \parallel A'.$$

Proof. We demonstrate that the rules specific to \mathcal{A} can be simulated with the rules of \mathcal{T} .

Learn, $\mathcal{A}\text{-Propagate}^+$, $\mathcal{B}\text{-Propagate}^+$, $\mathcal{B}\text{-Propagate}^-$:

These rules are restricted forms of learning. They are special cases of $\mathcal{T}\text{-Learn}$.

\mathcal{A} -Propagate⁻:

Suppose that \mathcal{A} -Propagate⁻ is applied to learn the disequality $x \neq y$. This can be simulated with \mathcal{T} as follows. We apply Branch to produce subproblems for (a) $x < y$, (b) $x = y$, and (c) $x > y$. For the subproblem (b), the constraints $C \wedge I \wedge x = y$ are $(\mathcal{Z} \cup \mathcal{T})$ -inconsistent, therefore they entail anything, *e.g.*, the inequality $0 \leq -1$, which can be introduced via the powerful \mathcal{T} -Learn rule. It is then possible to apply Drop and end up with the subproblems (a) and (c) that \mathcal{A} -Propagate⁻ produces.

\mathcal{A} -Unsat:

The \mathcal{T} -unsatisfiable subproblem can be eliminated by introducing the inequality $0 \leq -1$ (via \mathcal{T} -Learn) and subsequently applying Drop.

\mathcal{A} -Branch, \mathcal{B} -Branch:

The general Branch rule (present in \mathcal{T}) captures these specialized kinds of branching.

□

Lemma 9. *If*

$$T, I, O \Vdash P \parallel A \xrightarrow[\mathcal{A}]{}^* P' \parallel A',$$

then

$$T, I, O \Vdash P \parallel A \xrightarrow[\mathcal{T}]{}^* P' \parallel A'.$$

Proof. Induction on the length of the sequence of transitions relating $P \parallel A$ and $P' \parallel A'$ and application of Lemma 8. □

Corollary 4 (Soundness of \mathcal{A} (Satisfiable Instances)). *If*

$$T, I, O \Vdash \{C\} \parallel \text{None} \xrightarrow[\mathcal{A}]{}^* \emptyset \parallel A$$

where $A \neq \text{None}$, then

(a) *A is a \mathcal{T} -model of $C \wedge I$, and*

(b) *there exists no \mathcal{T} -model B of $C \wedge I$ such that $O(B) < O(A)$.*

Corollary 5 (Soundness of \mathcal{A} (Unsatisfiable Instances)). *If*

$$T, I, O \Vdash \{C\} \parallel \text{None} \xrightarrow[\mathcal{A}]^* \emptyset \parallel \text{None},$$

then $C \wedge I$ is $(\mathcal{Z} \cup T)$ -unsatisfiable.

5.4 Termination

We demonstrate that (under reasonable assumptions) the transition system \mathcal{A} terminates. Our termination analysis relies on ranking functions that operate over subproblems and states.

Definition 27 (Subproblem Rank (rankp)). *Let I be a set of interface definitions. The rank of a subproblem C (modulo I), which we denote by $\text{rankp}(C, I)$, is an integer computed as follows:*

- *if $\text{diffs}(C, I)$ is \mathcal{Z} -consistent, then*

$$\begin{aligned} \text{rankp}(C, I) = & \\ & |\text{shared}(C, I)| \cdot (|\text{shared}(C, I)| - 1)/2 \\ & - |\text{arrange}(C, \text{shared}(C, I))| \\ & + |\{l \mid l = [\nu > 0 \Leftrightarrow F] \in I, \text{diffs}(C) \not\models_{\mathcal{Z}} \nu > 0, \text{diffs}(C) \not\models_{\mathcal{Z}} \nu \leq 0\}|. \end{aligned}$$

- *otherwise, $\text{rankp}(C, I) = 0$.*

$\text{rankp}(C, I) \geq 0$ for any C and any I . Even in the case of \mathcal{Z} -consistent $\text{diffs}(C)$, observe that $|\text{shared}(C, I)| \cdot (|\text{shared}(C, I)| - 1)/2$ is the cardinality of an arrangement over $\text{shared}(C, I)$. Thus,

$$|\text{arrange}(C, \text{shared}(C, I))| \leq |\text{shared}(C, I)| \cdot (|\text{shared}(C, I)| - 1)/2.$$

For \mathcal{Z} -consistent $\text{diffs}(C)$, $\text{rankp}(C, I) = 0$ if (a) $\text{arrange}(C, \text{shared}(C, I))$ is an arrangement over $\text{shared}(C, I)$, and (b) $\text{diffs}(C)$ provides truth values for all F that appear in interface atoms $[\nu > 0 \Leftrightarrow F] \in I$.

Definition 28 (State Rank (rank)). *Let I be a set of interface definitions. The rank of a state $P \parallel A$ (modulo I), which we denote by $\text{rank}(P \parallel A, I)$, is an integer computed as follows:*

$$\text{rank}(P \parallel A, I) = \sum_{C \in P} 4^{\text{rankp}(C, I)}.$$

Clearly, $\text{rank}(P \parallel A, I) \geq |P| \geq 0$; $\text{rank}(P \parallel A, I) = |P|$ if $\text{rankp}(C, I) = 0$ for every $C \in P$.

Lemma 10. *Let Σ be a signature such that $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$. Furthermore, let C be a set of integer linear constraints, T be a stably-infinite Σ -theory, and I be a set of Σ -interface definitions. If $\text{rankp}(C, I) = 0$, then $C \wedge I$ is $(\mathcal{Z} \cup T)$ -satisfiable iff*

- (a) $\theta(C, I) \wedge \text{arrange}(C, \text{shared}(C, I))$ is T -satisfiable, and
- (b) C is integer-satisfiable.

Proof.

(\Rightarrow):

If $C \wedge I$ is $(\mathcal{Z} \cup T)$ -satisfiable, given that $C \models_{\mathcal{Z}} \text{arrange}(C, \text{shared}(C, I))$ and that $C \wedge I \models_{\mathcal{Z}} \theta(C, I)$, we obtain that

$$C \wedge I \wedge \theta(C, I) \wedge \text{arrange}(C, \text{shared}(C, I))$$

is $(\mathcal{Z} \cup T)$ -satisfiable. The latter fact implies both conditions (a) and (b).

(\Leftarrow):

C is integer-satisfiable, and thus its subset $\text{diffs}(C)$ is also integer-satisfiable.

Let $V = \text{shared}(C, I)$. From $\text{rankp}(C, I) = 0$ and the \mathcal{Z} -consistency of $\text{diffs}(C)$, it follows that $\text{arrange}(C, V)$ is an arrangement over V , i.e., there exists an equivalence relation E over V such that $\text{arrange}(C, V) = \alpha(V, E)$.

$C \models_{\mathcal{Z}} \alpha(V, E)$, which (given that C is \mathcal{Z} -satisfiable) implies that $C \wedge \alpha(V, E)$ is \mathcal{Z} -satisfiable. Also, $\theta(C, I) \wedge \alpha(V, E)$ is T -satisfiable. By applying Fact 1, we obtain that $C \wedge \theta(C, I)$ is $(\mathcal{Z} \cup T)$ -satisfiable.

But $C \wedge \theta(C, I) \models_{\mathcal{Z} \cup T} C \wedge I$. This entailment holds because (given $\text{rankp}(C, I) = 0$) for every definition $[v > 0 \Leftrightarrow F] \in I$, $\theta(C, I)$ contains F with the same polarity as the truth value that $v > 0$ has according to $\text{diffs}(C) \subseteq C$, hence the bidirectional implication is entailed, while all equational interface definitions are also in $\theta(C, I)$. $C \wedge I$ is thus $(\mathcal{Z} \cup T)$ -satisfiable.

□

According to Lemma 10, $\text{rankp}(C, I) = 0$ means that $C \wedge I$ can be solved compositionally, by separately solving C modulo \mathcal{Z} and $\theta(C, I)$ (strengthened with an arrangement that we can efficiently extract) modulo T . For a subproblem satisfying this condition, the theory solver has all the information it needs to either determine T -unsatisfiability and report it via \mathcal{A} -Unsat, or to sign off on the subproblem and let the core solver discover an optimal integer assignment, if one exists. The core solver on the other hand can apply known complete algorithms (Sections 1.3 and 1.4) to obtain optimal integer assignments. The basic steps of such algorithms correspond to \mathcal{A} transitions.

Definition 29 (Last-Mile State). *Let I be a set of interface definitions. A $\text{BC}(T)$ state $P \parallel A$ is called a last-mile state (modulo I) if $\text{rank}(P \parallel A, I) = |P|$, i.e., if for every $C \in P$, $\text{rankp}(C, I) = 0$.*

Assuming some signature Σ with $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$, and a stably-infinite Σ -theory T whose quantifier-free fragment is decidable, all subproblems in a last-mile state meet the requirements of Lemma 10, and can thus be solved compositionally. Each subproblem can be de-composed into two independent parts that are both decidable. If all the subproblems are $(\mathcal{Z} \cup T)$ -unsatisfiable, then the instance is unsatisfiable; otherwise, the best among the solutions of the subproblems is a solution to the MP Modulo T instance. Solving all subproblems (with techniques that correspond to \mathcal{A} transitions)

allows us to reach a final state. Given that \mathcal{A} is sound (Corollaries 4 and 5), we are guaranteed to obtain an optimal T -model, iff one exists.

A strategy for applying \mathcal{A} that is guaranteed to reach a last-mile state therefore amounts to a complete optimization procedure for MP Modulo T . We subsequently provide such a strategy.

Lemma 11. *If $T, I, O \Vdash S \xrightarrow{\mathcal{A}\text{-Branch}} S'$, then $\text{rank}(S', I) < \text{rank}(S, I)$.*

Proof. The transition is necessarily of the form

$$P \uplus \{C\} \parallel A \xrightarrow{\mathcal{A}\text{-Branch}} P \cup \{C \cdot x < y, C \cdot x = y, C \cdot x > y\} \parallel A.$$

$\text{rankp}(C, I) > 0$, because $\text{diffs}(C)$ \mathcal{Z} -entails neither $x = y$ nor $x \neq y$, which means that $\text{diffs}(C)$ is \mathcal{Z} -consistent and that the truth value for one equality between shared variables is unknown. Thus, $\text{rank}(P \uplus \{C\} \parallel A, I) > 0$, i.e., rank can possibly decrease. In fact rank does decrease:

$$\begin{aligned} & \text{rank}(P \cup \{C \cdot x < y, C \cdot x = y, C \cdot x > y\} \parallel A, I) \\ & \leq \sum_{C' \in P} [4^{\text{rankp}(C', I)}] + 3 \cdot 4^{\text{rankp}(C, I) - 1} \\ & < \sum_{C' \in P} [4^{\text{rankp}(C', I)}] + 4^{\text{rankp}(C, I)} \\ & = \text{rank}(P \uplus \{C\} \parallel A, I). \end{aligned}$$

For each of the subproblems $C \cdot x < y$, $C \cdot x = y$, and $C \cdot x > y$ resulting from the branching step, the value of rankp is at most $\text{rankp}(C, I) - 1$, because at least one extra (dis)equality relating the variables $x, y \in \text{shared}(C, I)$ is known. \square

Lemma 12. *If $T, I, O \Vdash S \xrightarrow{\mathcal{A}\text{-Propagate}^-} S'$, then $\text{rank}(S', I) < \text{rank}(S, I)$.*

Proof. The transition is necessarily of the form

$$P \uplus \{C\} \parallel A \xrightarrow{\mathcal{A}\text{-Propagate}^-} P \cup \{C \cdot x < y, C \cdot x > y\} \parallel A.$$

$\text{rankp}(C, I) > 0$, because $\text{diffs}(C)$ \mathcal{Z} -entails neither $x = y$ nor $x \neq y$, which means that $\text{diffs}(C)$ is \mathcal{Z} -consistent and that the truth value for one equality

between shared variables is unknown. Thus, $\text{rank}(P \uplus \{C\} \parallel A, I) > 0$, i.e., rank can possibly decrease. rank decreases as follows:

$$\begin{aligned}
 & \text{rank}(P \cup \{C \cdot x < y, C \cdot x > y\} \parallel A, I) \\
 & \leq \sum_{C' \in P} [4^{\text{rankp}(C', I)}] + 2 \cdot 4^{\text{rankp}(C, I) - 1} \\
 & < \sum_{C' \in P} [4^{\text{rankp}(C', I)}] + 4^{\text{rankp}(C, I)} \\
 & = \text{rank}(P \uplus \{C\} \parallel A, I).
 \end{aligned}$$

For each of the subproblems $C \cdot x < y$ and $C \cdot x > y$, the value of rankp is at most $\text{rankp}(C, I) - 1$, because at least one extra (dis)equality relating the variables $x, y \in \text{shared}(C, I)$ is known. \square

Lemma 13. *If $T, I, O \Vdash S \xrightarrow{\mathcal{B}\text{-Branch}} S'$, then $\text{rank}(S', I) < \text{rank}(S, I)$.*

Proof. The transition is necessarily of the form

$$P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot v \leq 0, C \cdot v > 0\} \parallel A.$$

$\text{rankp}(C, I) > 0$, because $\text{diffs}(C)$ \mathcal{Z} -entails neither $v \leq 0$ nor $v > 0$, which means that $\text{diffs}(C)$ is \mathcal{Z} -consistent and that the truth value for the left-hand side of at least one interface definition is unknown. Thus, $\text{rank}(P \uplus \{C\} \parallel A, I) > 0$, i.e., rank can possibly decrease. rank decreases as follows:

$$\begin{aligned}
 & P \cup \{C \cdot v \leq 0, C \cdot v > 0\} \parallel A \\
 & \leq \sum_{C' \in P} [4^{\text{rankp}(C', I)}] + 2 \cdot 4^{\text{rankp}(C, I) - 1} \\
 & < \sum_{C' \in P} [4^{\text{rankp}(C', I)}] + 4^{\text{rankp}(C, I)} \\
 & = \text{rank}(P \uplus \{C\} \parallel A, I).
 \end{aligned}$$

For each of the subproblems $C \cdot v \leq 0$ and $C \cdot v > 0$, the value of rankp is at most $\text{rankp}(C, I) - 1$, because the truth value of (at least) one atom $v > 0$ appearing in an interface definition $[v > 0 \Leftrightarrow F] \in I$ is known. \square

Lemma 14. *If*

$$T, I, O \Vdash S \xrightarrow{\mathcal{A}} S'$$

but not

$$T, I, O \Vdash S \xrightarrow{\text{Branch}} S',$$

then $\text{rank}(S', I) \leq \text{rank}(S, I)$.

Proof. We case-split on the rule that relates S and S' . As per our assumptions, the transition cannot possibly be a Branch step.

\mathcal{A} -Branch:

Our proof obligation follows from Lemma 11.

\mathcal{A} -Propagate⁻:

Our proof obligation follows from Lemma 12.

\mathcal{B} -Branch:

Our proof obligation follows from Lemma 13.

Learn, \mathcal{A} -Propagate⁺, \mathcal{A} -Forget, \mathcal{B} -Propagate⁺, \mathcal{B} -Propagate⁻:

Each of these rules modifies one subproblem without weakening its set of difference constraints, and thus without increasing its rank. Given that the rest of the subproblems do not change, the summation in the definition of rank does not increase.

\mathcal{A} -Unsat, Drop, Prune, \mathcal{T} -Unbounded:

These rules eliminate one subproblem, and do not modify the rest of the subproblems. Thus, rank decreases.

\mathcal{T} -Improve:

Only the assignment part of the state changes. rank does not take the assignment into account. Therefore, rank does not change.

□

Definition 30 (*k*-Fair \mathcal{A} Sequence). *Let k be a non-negative integer. A sequence of \mathcal{A} transitions of the form*

$$\begin{aligned}
 T, I, O \Vdash P_0 \parallel A_0 &\xrightarrow{\mathcal{A}} P_1 \parallel A_1 & (R_0) \\
 T, I, O \Vdash P_1 \parallel A_1 &\xrightarrow{\mathcal{A}} P_2 \parallel A_2 & (R_1) \\
 &\dots & \\
 T, I, O \Vdash P_i \parallel A_i &\xrightarrow{\mathcal{A}} P_{i+1} \parallel A_{i+1} & (R_i) \\
 &\dots &
 \end{aligned} \tag{5.2}$$

is called k -fair, if for every subsequence of transitions $R_i, R_{i+1}, \dots, R_{i+k-1}$, there exists some integer j , $i \leq j < i + k$, such that either

- (a) $P_j \parallel A_j$ is a last-mile state (modulo I), or
- (b) R_j is an application of \mathcal{A} -Branch, or
- (c) R_j is an application of \mathcal{B} -Branch.

It is easy for an implementation of the transition system \mathcal{A} to guarantee k -fairness (Definition 30) for some k . For as long as we have not reached a last-mile state, the branching rules \mathcal{A} -Branch and \mathcal{B} -Branch need to be applied with the appropriate frequency. The fact that we are not yet confronted with a last-mile state means that there exists some condition for one of these rules to branch on, *i.e.*, it is within the control of the solver to apply either \mathcal{A} -Branch or \mathcal{B} -Branch. We subsequently prove that k -fairness (coupled with a non-divergent ILP branching strategy) leads to last-mile states.

Theorem 3. *Let k be a positive integer. Consider a k -fair sequence of \mathcal{A} transitions σ of the form given in Equation 5.2. If σ contains only finitely many Branch steps, then there exists a positive integer l such that $P_l \parallel A_l$ is a last-mile state (modulo I).*

Proof. Because only finitely many transitions are Branch, there exists a suffix τ of the sequence σ that doesn't have any Branch steps; the suffix τ is of the form $P_m \parallel A_m \xrightarrow{\mathcal{A}} P_{m+1} \parallel A_{m+1} \xrightarrow{\mathcal{A}} \dots$ for some $m \geq 0$. For the suffix τ ,

every application of \mathcal{A} -Branch or \mathcal{B} -Branch decreases rank, while the remaining rules do not increase rank. Let $n = \text{rank}(P_m \parallel A_m, I)$. The sequence τ is k -fair, because a suffix of a k -fair sequence is clearly also k -fair. For as long as we don't get to a last-mile state, the maximum possible interval between successive \mathcal{A} -Branch and \mathcal{B} -Branch steps is k . It can take at most $k \cdot n$ transitions for rank to reach the minimum possible value of 0 (which implies that we reach a state $\emptyset \parallel A$ that is clearly a last-mile state), or until we otherwise get to a last-mile state $P_l \parallel A_l$. \square

The significance of Theorem 3 (combined with Lemma 10) is as follows. Let T be a stably-infinite Σ -theory, with $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$, where the quantifier-free fragment of T is decidable. Consider an MP Modulo T instance C, O, I, T , where I is a set of Σ -literals. If the ILP solver does not venture into a non-terminating branching strategy, and if the rules \mathcal{A} -Branch and \mathcal{B} -Branch are applied in a fair way, then we are guaranteed to reach a last-mile state. As we explained, this means that \mathcal{A} (applied as per our restrictions) provides a complete optimization procedure for MP Modulo T .

Explicitly assuming that the ILP solver does not perform infinitely many Branch steps (as opposed to proving this fact) may seem like a limitation of our analysis. This limitation is justified as follows. We chose a general Branch rule, as opposed to providing a less powerful rule that would enable us to guarantee finite behavior. ILP solvers employ a diverse set of branching strategies. Simultaneously capturing a broad enough subset of them while guaranteeing termination would be infeasible. In any case, failing to meet the condition on Branch would indicate a misbehaving ILP solver. A more sophisticated analysis of ILP branching is beyond the scope of our work, which does not focus on core ILP techniques, but rather on the interaction of B&C-based solvers with other procedures.

The strategy outlined in this section provides more than a decision procedure for MP Modulo T (where T is stably infinite and signature-disjoint with \mathcal{Z}). Remember that our instances (Definition 17) contain objective functions. The strategy outlined is guaranteed to produce *optimal* answers, thus offering a complete *optimization* procedure.

Chapter 6

$BC(T)$: An Algorithmic View

In Chapters 4 and 5, we formalized the $BC(T)$ architecture for MPMT as a family of non-deterministic transition systems. By abstracting away solver implementation details, our transition systems capture a wide range of possible implementations, and facilitate theoretical analysis. $BC(T)$ can be thought of as a design space for MPMT solvers. Implementing an MPMT solver involves zooming in on a region of this space, with assorted performance trade-offs.

To inform efficient solver design, this chapter provides an algorithmic (and more deterministic) view of $BC(T)$. There are multiple ways to implement $BC(T)$. For instance, it is possible to implement $BC(T)$ from scratch, with efficient theory integration as a top priority. We have not gone through the potentially multiple man-years involved in such an implementation effort. Instead, our implementation of $BC(T)$ (the `lnez` solver) extends a pre-existing B&C-based MP solver (`SCIP` [3]) to support theories.

We describe the key ideas behind our implementation. We explain as much of the operation of a B&C-based solver as needed to demonstrate where theory solvers fit, with an emphasis on the interface between theory solvers and the B&C procedure. We do not cover purely internal operations of either side. For example, we treat Simplex (which handles continuous relaxations within B&C) purely as a black box.

We use congruence closure (CC) as an example of a background procedure. Given that we inherit the core operations of CC from related work [76],

our discussion only covers the $BC(T)$ -specific aspects. Our choice of CC is motivated by its wide applicability and by the relatively simple (but not trivial) constraints and algorithms involved.

6.1 Preliminaries

In alignment with $BC(T)$, as presented in Chapters 4 and 5, the algorithms operate upon subproblems and sets thereof. A subproblem is described by a set of integer linear constraints (Definition 2). Subproblems also carry metadata, like known variable bounds.

In addition to the integer linear constraints, the input to the solver contains uninterpreted function (UF) constraints. We assume that variable abstraction (Example 5) has happened as a preprocessing step, resulting in linear constraints that do not involve UF terms. The definitions symbol that is used in the pseudocode stands for a collection of atomic formulas of the form $v = f(l)$, where v is a variable symbol, f is a UF symbol, and l is a list of arguments of the form $w + k$, where w is a variable symbol and k is an integer constant. Entirely concrete terms are a special case that can be encoded with a single integer variable fixed to zero. UF terms thus involve limited arithmetic, as is common practice [76]. definitions is an immutable global constant.

Our pseudocode uses *sum types* (also known as *tagged unions*) for some of the variables. Sum types have multiple *constructors* that correspond to different cases for the values carried. The constructor of a particular element serves as a tag denoting which case the element belongs in. Furthermore, the magic constant $*$ stands for non-deterministic Boolean choice. $*$ is used in conditionals where heuristics apply. $\langle e \rangle$ denotes that standard operators within e are to be interpreted over syntactic objects, e.g., $\langle v - w \rangle$ is not a concrete integer or real, but a term representing the subtraction of w from v . We follow a generally applicative style, e.g., operations that modify a node (by producing new linear constraints and bounds) produce a new node. Our presentation is top-down.

Our CC solver is implemented by the functions with suffix `_cc`, i.e., by

```

function bc( $p$  : node) : (Unsat|Optimal(assignment))
   $P \leftarrow \{p\}$ 
   $\alpha \leftarrow \text{None}$ 
  while  $P \neq \emptyset$  do
     $q \leftarrow \text{pick}(P)$ 
     $P \leftarrow P \setminus \{q\}$ 
    match solve_node( $q$ , obj( $\alpha$ )) with
      case Unsat
      | noop() // do nothing
      case Solved( $\beta$ )
      |  $\alpha \leftarrow \text{Some}(\beta)$ 
      case Branched( $Q$ )
      |  $P \leftarrow P \cup Q$ 
    match  $\alpha$  with
      case None
      | return Unsat
      case Some( $\beta$ )
      | return Optimal( $\beta$ )

```

propagate_cc, enforce_cc, check_cc, and branch_cc. These are the functions that need to be replaced (or enhanced) for supporting a theory other than \emptyset .

6.2 High-Level Functions

The top-level B&C procedure, bc, accepts as its argument a set of integer linear constraints, p . p corresponds to the root node of the B&C search tree. bc keeps track of a set P of nodes to be examined (initialized with $\{p\}$). α carries a candidate satisfying (integer) assignment. α , belonging in a sum type, is of the form $\text{Some}(\beta)$ if an assignment β is known; α is None otherwise. The loop body in bc picks one of the remaining nodes in P and processes it by calling solve_node. The implementation of pick (not provided) may involve sophisticated heuristics for the choice of next node to be examined. It is common for this choice to be biased towards the children of the node that was more recently branched upon (*i.e.*, depth-first

```

function solve_node( $p$  : node,  $l$  : int) :
(Unsat|Solved(assignment)|Branched({node}))
  match propagate( $p$ ) with
    case Unsat
    | return Unsat
    case Modified( $p'$ )
    |  $p \leftarrow p'$ 
    case Unmodified
    | noop()

  match solve_relaxation( $p$ ) with
    case Unsat
    | return Unsat
    case Modified( $p'$ )
    | return solve_node( $p'$ ,  $l$ )
    case Solved( $\alpha$ )
    | if obj( $\alpha$ )  $\geq l$  then
      | return Unsat
    | else
      | match enforce( $p$ ,  $\alpha$ ) with
        case Sat
        | return Solved( $\alpha$ )
        case Unsat
        | return Unsat
        case Modified( $p'$ )
        | return solve_node( $p'$ ,  $l$ )
        case Branched( $P$ )
        | return Branched( $P$ )

```

search).

`solve_node` receives as arguments the node p to be processed, in addition to an upper bound l for the objective values of the solutions of interest. l corresponds to an already-known solution. (We assume that the function `obj` that computes l and our comparisons with l take care of the possibility of no known solution or unbounded solution, by supporting the special constants $+\infty, -\infty$.) `solve_node` performs three processing stages:

(a) propagation (Section 6.3);

- (b) solving a continuous relaxation of the linear constraints; and
- (c) enforcing constraints against a relaxation-obtained solution (Section 6.4).
Enforcing may result in branching (Section 6.5).

The aforementioned stages operate on one node at a time, always called p in the respective functions, while their output may be multiple nodes, as a result of branching. In the event of branching, the new nodes are passed to the top-level procedure (response `Branched` from `solve_node` to `bc`) and added to the set of nodes that need to be examined.

6.3 Propagation

```

function propagate( $p$  : node) : (Unsat| Modified(node)|Unmodified)
   $m \leftarrow$  false
  while * do
    if * then
      match propagate_ilp( $p$ ) with
        case Unsat
           $\sqsubset$  return Unsat
        case Modified( $p'$ )
           $p \leftarrow p'$ 
           $m \leftarrow$  true
        case Unmodified
           $\sqsubset$  noop()
    if * then
      match propagate_cc( $p$ ) with
        case Unsat
           $\sqsubset$  return Unsat
        case Modified( $p'$ )
           $p \leftarrow p'$ 
           $m \leftarrow$  true
        case Unmodified
           $\sqsubset$  noop()
  return  $m ?$  Modified( $p$ ) : Unmodified

```

```

function propagate_cc( $p$  : node) : (Unsat|Modified(node)|Unmodified)
   $m \leftarrow \text{false}$ 
  forall  $v, w \in \text{variables}(p)$  do
    forall  $k \in \mathbb{Z}$  such that  $\text{equalities}(p) \wedge \text{definitions} \models_{\mathcal{Z}} v = w + k$  do
      match set_lb( $p, \langle v - w \rangle, k$ ) with
        case Unsat
        | return Unsat
        case Modified( $p'$ )
        |  $m \leftarrow \text{true}$ 
        |  $p \leftarrow p'$ 
        case Unmodified
        | noop()
      match set_ub( $p, \langle v - w \rangle, k$ ) with
        case Unsat
        | return Unsat
        case Modified( $p'$ )
        |  $m \leftarrow \text{true}$ 
        |  $p \leftarrow p'$ 
        case Unmodified
        | noop()
    return  $m ? \text{Modified}(p) : \text{Unmodified}$ 

```

The function `propagate` attempts to reduce the domain of variables. In the process of doing so, it may detect infeasibility (response `Unsat`); if it succeeds, `propagate` returns a version p' of the original node p modified with new bounds (`Modified(p')`); `Unmodified` means that no propagation was possible, neither was the function able to detect infeasibility. The implementation we provide combines ILP (`propagate_ilp`) and CC (`propagate_cc`) propagation techniques. We do not explain `propagate_ilp`, which is internal to the ILP solver, and as such orthogonal to $\text{BC}(T)$. Either kind of propagation can be skipped. We repeatedly perform propagation, until a fixpoint is reached, or until a heuristic for termination returns true, e.g., after a fixed number of rounds. In practice, SCIP employs various *constraint handlers* that provide propagation procedures of different *priority*. The top-level propagation procedure takes into account priorities to combine the sub-procedures.

```

function enforce( $p$  : node,  $\alpha$  : assignment) :
  (Sat|Unsat|Modified(node)|Branched({node}))
  match enforce_ilp( $p, \alpha$ ) with
    case Sat
    | return enforce_cc( $p, \alpha$ )
    case Unsat
    | return Unsat
    case Modified( $p'$ )
    | return Modified( $p'$ )
    case Branched( $P$ )
    | return Branched( $P$ )

```

`propagate_cc` is described in a declarative way. Our concrete implementation is similar to the CC procedures in SMT, and thus takes offsets into account [76]. `equalities(p)` stands for known equalities of the form $v = w + k$, where v and w are integer variables and k is an integer constant. Such equalities can be obtained via the known difference constraints, for which we use the auxiliary function `diffs` in Chapter 5. For any equality $v = w + k$ implied by the already known equalities (conjoined with definitions), we try to fix the upper and lower bound of $v - w$ to k (via the functions `set_lb` and `set_ub` that provide an interface to the ILP solver), and report unsatisfiability if this is impossible. The `forall` statements should be read as a declarative specification (*i.e.*, we range over all relevant v, w, k), not as a suggestion for (in)efficient implementation.

6.4 Enforcing Continuous Relaxations

In `solve_node`, propagation is followed by solving a continuous relaxation (`solve_relaxation`). A response `Unsat` for the relaxation implies that the integer constraints of the node (which are strictly harder than the real constraints of the relaxation) are also unsatisfiable. If `solve_relaxation` returns an assignment α (case `Solved(α)`), `solve_node` first checks whether α is better than the already known solution ($\text{obj}(\alpha) < l$), and does not further process the node if not; integer solutions can be at most as good as

```

function enforce_cc(p : node,  $\alpha$  : assignment) :
(Sat|Unsat|Modified(node)|Branched({node}))
  match check_cc(p,  $\alpha$ ) with
    case Sat
    | return Sat
    case Conflict(c)
    | match propagate_cc(p) with
      case Unsat
      | return Unsat
      case Modified(p')
      | return Modified(p')
      case Unmodified
      | return Branched(branch_cc(p, c))

```

```

function check_cc(p : node,  $\alpha$  : assignment) : (Sat|Conflict(conflict))
  m  $\leftarrow$  {} // m is a map
  foreach  $\langle v = f(l) \rangle \in$  definitions do
    c  $\leftarrow$  [ $\alpha(w) + k | \langle w + k \rangle$  in l] // list comprehension over l
    if (f, c)  $\in$  keys(m) then
      (v', l')  $\leftarrow$  m[(f, c)]
      if  $\alpha(v) \neq \alpha(v')$  then
        | return Conflict(f, v, v', l, l')
    else
      | m[(f, c)]  $\leftarrow$  (v, l)
  return Sat

```

the solution to the relaxation. Otherwise, `enforce` is executed. If α is not integer, or if it is theory-inconsistent, `enforce` is responsible for explaining why, e.g., by introducing implied linear constraints violated by α . `enforce` may determine that α satisfies all constraints (response `Sat`), or that the node (and not just α) is infeasible (response `Unsat`). In either of these cases, `solve_node` has solved p . Enforcing may result in learning new linear constraints or bounds (case `Modified(p')`), in which case `solve_node` needs to process the node again.

`enforce` combines different kinds of enforcement, in much the same

way that `propagate` combines different kinds of propagation. The part of enforcement that is related to integrality (`enforce_ilp`) may branch around a real solution, or apply cut generation techniques [38, 18]. ILP cut generation techniques are beyond the scope of this dissertation. Conversely, the implementation of `enforce_ilp` is not shown. We proceed to describe CC enforcement (`enforce_cc`). First, `enforce_cc` calls `check_cc` to check whether α is theory-consistent. `check_cc` reports that α does not satisfy the UF constraints if there exist calls $v = f(l)$ and $v' = f(l')$ of some function f , such that all arguments in the respective positions of the lists of arguments l and l' have the same value under α , but $\alpha(v) \neq \alpha(v')$. `check_cc` then returns the *conflict* (f, v, v', l, l') to explain what is wrong with α . If no conflict is found, `bc` receives α , and α becomes the new candidate solution.

In case `check_cc` returns a conflict, `enforce_cc` ensures that propagation has happened by calling `propagate_cc` again. The latter function may have been skipped during the propagation stage. `enforce_cc` only needs to act further if propagation can neither detect unsatisfiability, nor produce new information. In this case, `enforce_cc` proceeds by branching.

Note that CC enforcement happens after the corresponding method for the ILP constraints. CC enforcement thus only ever deals with integer assignments, which yields cleaner implementation. Additionally, this design prioritizes ILP-related over theory-related operations, thus emphasizing ILP-heavy problems.

6.5 Branching

Branching is what our CC implementation performs when all else fails. Concretely, the following invariant holds when we get to `branch_cc`. There exists an integer solution for the non-theory constraints of p (given that integrality enforcement has succeeded) which violates the theory constraints, but the integer bounds that hold for p do not allow any information to be propagated, neither can we deduce unsatisfiability of p .

When we call `branch_cc` from `enforce_cc`, we have access to a con-

```

function branch_cc( $p$  : node, ( $f, v, v', l, l'$ ) : conflict) : {node}
  if  $\ast \wedge \text{lb}(p, \langle v - v' \rangle) \leq 0 \wedge \text{ub}(p, \langle v - v' \rangle) \geq 0$  then
     $P \leftarrow \{ \langle p \wedge v = v' \rangle \}$ 
    if  $\text{lb}(p, \langle v - v' \rangle) < 0$  then
       $P \leftarrow P \cup \{ \langle p \wedge v < v' \rangle \}$ 
    if  $\text{ub}(p, \langle v - v' \rangle) > 0$  then
       $P \leftarrow P \cup \{ \langle p \wedge v > v' \rangle \}$ 
    return  $P$ 
  for  $i \in [0, \text{arity}(f) - 1]$  do
    if  $\alpha(l[i]) = \alpha(l'[i])$  then
      if  $\text{lb}(p, \langle l[i] - l'[i] \rangle) < 0 \vee \text{ub}(p, \langle l[i] - l'[i] \rangle) > 0$  then
         $P \leftarrow \{ \langle p \wedge l[i] = l'[i] \rangle \}$ 
        if  $\text{lb}(p, \langle l[i] - l'[i] \rangle) < 0$  then
           $P \leftarrow P \cup \{ \langle p \wedge l[i] < l'[i] \rangle \}$ 
        if  $\text{ub}(p, \langle l[i] - l'[i] \rangle) > 0$  then
           $P \leftarrow P \cup \{ \langle p \wedge l[i] > l'[i] \rangle \}$ 
        return  $P$ 
  assert(false) // unreachable

```

flict (f, v, v', l, l') . Note that there must be some position $i \in [0, \text{arity}(f) - 1]$ such that the equality $l[i] = l'[i]$ is not implied by the bounds visible to `propagate_cc`. Otherwise, all arguments would have been equal, and `propagate_cc` would have produced the equality $v = v'$, which is violated. It is always possible to branch on whether $l[i] < l'[i]$, $l[i] = l'[i]$, or $l[i] > l'[i]$. If, according to the bounds on $v - v'$, $v = v'$ is a possibility, then we may instead choose to branch on whether $v < v'$, $v = v'$, or $v > v'$.

The branching strategy that we have outlined involves very little guesswork. A conflict (f, v, v', l, l') provides a witness for the gap between the assignment under examination α (which is not feasible with respect to UF) and the more limited information that is available as bounds in p (which do not entail infeasibility). In order to steer the ILP solver away from the problematic assignment α (and other assignments similar to it), we have to examine the aforementioned gap. We do so by branching driven by the conflict.

Additionally, our branching is in alignment with our termination argu-

ment of Section 5.4. We branch on pairs of variables that are shared between ILP and UF, *i.e.*, we make progress towards an *arrangement* of the shared variables. As explained in Section 5.4, such branching eventually produces subproblems for which CC has all the information on the shared variables that it needs to determine (in)feasibility of the UF constraints (in definitions); similarly, for the UF-feasible subproblems, the ILP solver (with no more input possible from CC) has all the information it needs to apply complete techniques and determine feasibility. We thus guarantee termination of the combination.

6.6 Discussion

lnez implements querying over the difference of two variables x, y by defining an auxiliary variable $d_{x,y}$, and by introducing the constraint $d_{x,y} = x - y$. The core B&C solver has to maintain known bounds about $d_{x,y}$, and respect the constraint, which implies that the bounds on $d_{x,y}$ have to be compatible with the bounds on x and y . Obtaining bounds on $x - y$ (*e.g.*, to determine whether x and y are equal) amounts to inspecting the bounds on $d_{x,y}$. While in the general case we need quadratically many auxiliary variables $d_{x,y}$, this is not necessarily a practical issue. In practice, the set of pairs that matter is fairly small, and a good over-approximation thereof can be obtained by syntactic analysis of the input constraints.

In addition to standard CC-based propagation [76], lnez applies techniques specific to the integer domain. Consider variables x and y , and a context such that their difference is locally bounded between integers m and n , *i.e.*, $m \leq x - y \leq n$. Furthermore, assume that for every integer $i \in [m, n]$, $f(y + i)$ is (locally) bounded between l and u . Given that $y + m \leq x \leq y + n$, it follows that $l \leq f(x) \leq u$. (The bounds l and u hold for $f(x)$ in every possible case.)

More generally, a practical implementation of B&C needs to strike a suitable balance with respect to modularity. Clear separation between the tasks of different solvers has engineering benefits, and also facilitates theoretical analysis. At the same time, techniques that cross boundaries (*e.g.*, by

simultaneously exploiting the integer nature of variables and the behavior of uninterpreted functions) may offer a performance boost. Conversely, the techniques described in this chapter (which involve crossing boundaries) are complementary to the abstract view of our NO-implementing transition system of Chapter 5.

Chapter 7

Propositional Structure

This chapter studies problems that have arbitrary propositional structure, *e.g.*, disjunctions of inequalities. In light of Definition 17 (which formalizes the structure of MPMT instances), it is not entirely evident that such structure can be supported. Namely, the constraints of an MPMT instance belong in

- (a) a conjunction of integer linear constraints (Definition 2), which have no propositional structure, and
- (b) a conjunction of interface definitions (Definition 16), which do have (very specialized) propositional structure.

We focus on stably-infinite background theories (Definition 12) whose signatures do not overlap with that of \mathcal{Z} . We demonstrate that for such theories, interface definitions and integer linear constraints combined provide sufficient power to encode propositional structure. Our sound (Theorems 1 and 2) and complete (Section 5.4) $BC(T)$ framework can thus be used for problems with propositional structure, without any modification to the techniques described in Chapters 4, 5, and 6. We nevertheless describe small-scale enhancements that can improve the handling of propositional structure.

We finally experimentally evaluate our implementation of $BC(T)$ (Inez) on input with propositional structure, namely benchmarks from the SMT-LIB [8]. These are the benchmarks that the SMT solvers are tuned against.

We compare Inez against leading SMT solvers, namely Z3 [24] and MathSAT [41].

7.1 Flattening Propositional Structure

Throughout this section, we assume a stably-infinite Σ -theory T , where $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$. We outline $\text{BC}(T)$ -based techniques for determining the T -satisfiability of quantifier-free $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formulas.

Without loss of generality, we assume that each atomic formula that appears in our input is either a Σ -atomic formula, or a $\Sigma_{\mathcal{Z}}$ -atomic formula. It is always possible to obtain such a form. The propositional operators connecting the atomic formulas in question do not prevent us in any way from performing variable abstraction, as in Example 5. We can further assume that the only occurrence of the equality operator is in Σ -atomic formulas of the form $v = f(t_0, \dots, t_{n-1})$ that appear at the top level; any equality that appears under propositional structure can be viewed as a conjunction of integer inequalities. While purifying such inequalities may necessitate auxiliary equalities, these are only top-level Σ -atomic formulas.

Modern MILP solvers provide *indicator constraints* (or simply *indicators*), which we can utilize to handle $\Sigma_{\mathcal{Z}}$ -atomic formulas appearing under propositional operators.

Definition 31 (Indicator Constraint). *An indicator constraint is a constraint of the form $l \Rightarrow c$, where*

- l is a Boolean literal, i.e., a Boolean variable or the negation thereof, and
- c is an integer linear constraint (Definition 2).

A Boolean variable, as noted elsewhere, is an integer variable (in \mathcal{V}) bounded (by integer linear constraints) in $[0, 1]$. Indicator constraints can establish equivalence between a Boolean variable b and an inequality $\Sigma_{0 \leq i < n} [r_i \cdot x_i] \leq r$ via the constraints

$$\begin{aligned} b &\Rightarrow \quad \Sigma_{0 \leq i < n} [r_i \cdot x_i] \leq r, & \text{and} \\ \neg b &\Rightarrow \quad - \Sigma_{0 \leq i < n} [r_i \cdot x_i] \leq -r - 1. \end{aligned}$$

Formula	Literal	ILP & Indicator Constraints	Σ -definitions
$\Sigma_i[r_i \cdot x_i] \leq r$	\diamond	$\diamond \Rightarrow \Sigma_i[r_i \cdot x_i] \leq r$ $\neg \diamond \Rightarrow -\Sigma_i[r_i \cdot x_i] \leq -r - 1$	—
$p(t_0, \dots, t_{n-1})$	\diamond	—	$\diamond \Leftrightarrow p(t_0, \dots, t_{n-1})$
$F \vee G$	\diamond	$\diamond - \llbracket F \rrbracket - \llbracket G \rrbracket \leq 0$ $\llbracket F \rrbracket - \diamond \leq 0$ $\llbracket G \rrbracket - \diamond \leq 0$	—
$\neg F$	$1 - \llbracket F \rrbracket$	—	—
$v = f(x)$	—	—	$v = f(x)$

Figure 7.1: Flattening Formulas

Figure 7.1 describes a transformation that flattens a $(\Sigma \cup \Sigma_{\mathcal{Z}})$ -formula (with the restrictions we outlined) into a conjunction of constraints, which are integer linear constraints, indicator constraints, and Σ -definitions. The main idea is to recursively introduce Boolean literals that correspond to subformulas. This is very similar to the Tseitin transformation [89], which is commonly employed in applications of SAT and SMT. In Figure 7.1, the symbol \diamond stands for a fresh Boolean variable (if one is needed), while $\llbracket F \rrbracket$ stands for the Boolean literal produced by recursively applying the procedure on F . The transformation is completed by asserting the top-level literal produced, *i.e.*, by introducing the inequality $v \geq 1$ if the top-level input formula is represented by the variable v , or the inequality $v \leq 0$ if the formula is represented by $\neg v$.

We have already developed techniques for handling instances in a separate form that involves integer linear constraints and Σ -definitions (Definition 17). All that we additionally need in order to support propositional structure is a mechanism for supporting indicator constraints.

7.2 Encoding Indicators

It is possible to encode indicator constraints in terms of integer linear constraints, based on a technique that is known as Big-M. As a prerequisite for applying the Big-M technique, we show how to bound integer terms in

quantifier-free formulas while preserving $(\mathcal{Z} \cup T)$ -satisfiability. We build upon well-known results for ILP [15]. Similar ideas have been applied to \mathcal{Z} [81]. Our results go beyond the bounds for \mathcal{Z} , in that we take into account background theories and objective functions.

Our starting point is known bounds for ILP instances of the form

$$\begin{aligned} & \min c \mathbf{x} \\ & \text{subject to } A\mathbf{x} = b \\ & \mathbf{x} \geq 0 \end{aligned} \tag{7.1}$$

where A , b , and c are matrices of integers ($m \times n$, $m \times 1$, and $1 \times n$ respectively), and \mathbf{x} is an n -vector of integer variables. It is common to describe ILP instances by using matrix operations, as we do here.

Fact 2 ([15, Corollary of Theorem 13.5]). *If an ILP instance of the form (7.1) has a finite optimum, then it has an optimal solution x such that*

$$x_j \leq n^3[(m+2)d]^{4m+12}$$

for $1 \leq j \leq n$, where

$$d = \max(\max_{i,j} |A_{i,j}|, \max_i |b_i|, \max_i |c_i|).$$

In what follows, it is more convenient to work with the following form:

$$\begin{aligned} & \min c \mathbf{x} \\ & \text{subject to } A\mathbf{x} = b \\ & D\mathbf{x} \leq h \end{aligned} \tag{7.2}$$

where A , D , b , h and c are matrices of integers ($m \times n$, $k \times n$, $m \times 1$, $k \times 1$ and $1 \times n$ respectively), and \mathbf{x} is an n -vector of integer variables.

Lemma 15. *If an ILP instance I of the form (7.2) has a finite optimum, then it has an optimal solution \mathbf{x} such that*

$$|x_j| \leq (2n+k)^3[(m+k+2)d]^{4m+4k+12}$$

Chapter 7. Propositional Structure

for $1 \leq j \leq n$, where

$$d = \max(\max_{i,j} |A_{i,j}|, \max_i |b_i|, \max_i |c_i|, \max_{i,j} |D_{i,j}|, \max_i |h_i|).$$

Proof. We reduce I to an equisatisfiable instance over a vector \mathbf{x}' of $2n$ variables constrained to be non-negative ($\mathbf{x}' \geq 0$). We achieve this by replacing each variable x_i with $x'_i - x'_{n+i}$. In the resulting matrices A' , D' and c' , x'_i appears with the same coefficients as x_i ; x'_{n+i} appears with the coefficients multiplied by -1 . The resulting ILP instance I' has $m + k$ constraints and $2n$ variables. We replace inequalities with equalities by introducing k slack variables and obtain an equisatisfiable instance I'' over a vector \mathbf{x}'' of $2n + k$ variables (the last k of which are the slack variables) and $m + k$ constraints.

The chain of transformations preserves the maximum absolute coefficients. We can translate solutions of I to solutions of I'' and vice versa via the equalities $x_i = x''_i - x''_{n+i}$, for $1 \leq i \leq n$; an optimal assignment on either side corresponds to an optimum on the other. I'' has a finite optimum because I does. By Lemma 2, I'' has an optimal solution \mathbf{y}'' such that $y''_j \leq (2n + k)^3[(m + k + 2)d]^{4m+4k+12}$. For the corresponding solution y to I , we have $|y_i| = |y''_i - y''_{n+1}| \leq (2n + k)^3[(m + k + 2)d]^{4m+4k+12}$, which concludes our proof. \square

We denote by $\text{vars}(x)$ the set of variables that appears in x , where x can be a formula or term (or set of formulas, or set of terms). Furthermore, we define

$$\text{bounds}(V, \rho) = \{-\rho \leq v \wedge v \leq \rho \mid v \in V\},$$

where V is a set of variable symbols and ρ is a positive integer.

Lemma 16. *Let*

- Σ be a signature such that $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$,
- T be a stably-infinite Σ -theory,
- L be a finite set of $\Sigma_{\mathcal{Z}}$ -literals that are not (dis)equalities,
- U be a finite set of Σ -literals,
- O be an integer linear sum (an objective function),
- V be the set of variables shared by L and U ,

- n be the number of variables that appear in L ,
- d be the maximum coefficient that appears in L ,
- $k = |L| + |V| - 1$, and
- $\rho = (2n + k)^3[(k + 2)(d + 1)]^{4k+12}$.

If there exists a $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -interpretation M such that $M \models L \cup U \cup \mathcal{Z} \cup T$ and M is a finite optimum for $L \cup U$ with respect to O (i.e., there exists some integer constant c such that $M \models O = c$ and there exists no $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -interpretation M' such that $M' \models L \cup U \cup \mathcal{Z} \cup T$ and $M' \models O < c$), then $L \cup U \cup \text{bounds}(\text{vars}(L), \rho) \cup \{O = O(M)\}$ is $(\mathcal{Z} \cup T)$ -satisfiable.

Proof. Let E be the equivalence relation on V induced by M . Clearly, $L \cup \alpha(V, E)$ is \mathcal{Z} -satisfiable and $U \cup \alpha(V, E)$ is T -satisfiable. L contains $|L|$ possibly negated inequalities, and has n variables. We eliminate any negations in L by rewriting $\neg(S \leq r)$ to $-S \leq -r - 1$ and obtain a set of inequalities L' .

We order the variables V so that v is before u if $M \models v < u$. We obtain a sequence of variables $v_0, v_1, \dots, v_{|V|-1}, v_i \in |V|$. Let

$$\begin{aligned} \zeta &= \{v_i - v_{i+1} = 0 \mid 0 \leq i < |V| - 1 \text{ and } M \models v_i = v_{i+1}\}, \text{ and} \\ \eta &= \{v_i - v_{i+1} < 0 \mid 0 \leq i < |V| - 1 \text{ and } M \models v_i < v_{i+1}\}. \end{aligned}$$

Minimizing O subject to $L' \cup \zeta \cup \eta$ is an ILP instance N of the form (7.2). The maximum absolute value among coefficients in the matrices representing N is $d + 1$. Lemma 15 applies to N . In the worst case, ζ is empty and $|\eta| = |V| - 1$, i.e., we have 0 equalities and k inequalities. Thus, N has an optimal assignment A such that for every variable v , $|A(v)| \leq \rho$. Therefore,

$$L' \cup \zeta \cup \eta \cup \text{bounds}(\text{vars}(L'), \rho) \cup \{O = O(M)\}$$

is \mathcal{Z} -satisfiable. Since $\zeta \cup \eta \models_{\mathcal{Z}} \alpha(V, E)$, $L' \models_{\mathcal{Z}} L$, and $\text{vars}(L') = \text{vars}(L)$,

$$L \cup \text{bounds}(\text{vars}(L), \rho) \cup \{O = O(M)\} \cup \alpha(V, E)$$

is \mathcal{Z} -satisfiable. Because it is also the case that $U \cup \alpha(V, E)$ is T -satisfiable,

Chapter 7. Propositional Structure

it follows from Fact 1 that

$$L \cup U \cup \text{bounds}(\text{vars}(L), \rho) \cup \{O = O(M)\}$$

is $(\mathcal{Z} \cup T)$ -satisfiable. □

Theorem 4. *Let*

- Σ be a signature such that $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$,
- T be a stably-infinite Σ -theory,
- F be a quantifier-free $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula, such that no atomic formula in F contains symbols from both Σ and $\Sigma_{\mathcal{Z}}$, and where the $\Sigma_{\mathcal{Z}}$ -atoms are not equalities,
- O be an integer linear sum (an objective function),
- L be the set of $\Sigma_{\mathcal{Z}}$ -atomic formulas in F ,
- U be the set of Σ -atomic formulas in F ,
- V be the set of variables shared between L and U ,
- n be the number of variables that appear in L ,
- d be the maximum coefficient that appears in L ,
- $k = |L| + |V| - 1$, and
- $\rho = (2n + k)^3[(k + 2)(d + 1)]^{4k+12}$.

If there exists a $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -interpretation M such that $M \models \{F\} \cup \mathcal{Z} \cup T$ and M is a finite optimum for F with respect to O (i.e., there exists some integer constant c such that $M \models O = c$ and there exists no $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -interpretation M' such that $M' \models \{F\} \cup \mathcal{Z} \cup T$ and $M' \models O < c$), then $\{F\} \cup \text{bounds}(\text{vars}(L), \rho) \cup \{O = O(M)\}$ is $(\mathcal{Z} \cup T)$ -satisfiable.

Proof. M satisfies some of the atomic formulas in L and U , and falsifies the rest. Let

$$L' = \{l \mid l \in L, M \models l\} \cup \{\neg l \mid l \in L, M \models \neg l\}$$

$$U' = \{l \mid l \in U, M \models l\} \cup \{\neg l \mid l \in U, M \models \neg l\}$$

M is a finite optimum for $L' \cup U'$ with respect to O : if it was not, it would not be a finite optimum for F either, which contradicts our assumptions. For any $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -interpretation M' such that $M' \models L' \cup U' \cup \mathcal{Z} \cup T$, it is also

the case that $M' \models \{F\} \cup \mathcal{Z} \cup T$ (because M and M' assign the same truth values to the literals that appear in F , and the propositional structure does not change). By Lemma 16,

$$L' \cup U' \cup \text{bounds}(\text{vars}(L), \rho) \cup \{O = O(M)\}$$

is $(\mathcal{Z} \cup T)$ -satisfiable, *i.e.*, there exists some $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -interpretation M' that satisfies it, and also satisfies $\mathcal{Z} \cup T$. M' also $(\mathcal{Z} \cup T)$ -satisfies

$$\{F\} \cup \text{bounds}(\text{vars}(L), \rho) \cup \{O = O(M)\},$$

which concludes our proof. \square

Given a quantifier-free $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula F and an objective function O , Theorem 4 allows us to bound the integer terms of F while preserving (finite) optimality. Unbounded instances do not prevent us from utilizing Theorem 4 for bounding variables. A solver can first impose bounds and solve the bounded instance, and subsequently inspect the instance for unboundedness by imposing the additional constraint $O < O(M)$, re-computing bounds, and solving the resulting instance. If the updated instance is satisfiable, the original is unbounded.

Bounding variables (enabled by Theorem 4) allows us to express indicator constraints with just integer linear constraints. Consider an indicator constraint of the form

$$l \Rightarrow \sum_{0 \leq i < n} [r_i \cdot x_i] \leq r.$$

We bound all variables as per Theorem 4, and compute an integer k such that

$$\sum_{0 \leq i < n} [r_i \cdot x_i] \leq k$$

holds for any assignment to the variables x_i that satisfies the bounds. The indicator constraint can then be expressed as

$$\sum_{0 \leq i < n} [r_i \cdot x_i] \leq r + (k - r) \cdot (1 - l).$$

Note that the (possibly astronomical) coefficients we compute only serve the purpose of representing formulas in terms of integer linear constraints. Their magnitude does not necessarily have algorithmic side-effects. In the worst case, the initial continuous relaxation is weak, but relaxations become stronger once we start branching on the Boolean variables. This is no worse than Lazy SMT, where linear constraints are only applicable once the SAT core assigns the corresponding Boolean variables.

7.3 Symbolically Handling Indicators

If one is to use a MILP solver as a black box, handling indicator constraints by encoding them in terms of integer linear constraints is the only option. However, it is also possible to extend a B&C-based solver to deal with indicator constraints symbolically.

The idea is to extend the language accepted by the MILP solver to represent indicator constraints as such, *i.e.*, without translating them to pure integer linear constraints. Subsequently, the solver can follow a branching strategy that exhausts all possibilities with respect to the (finitely many) Boolean literals that appear as antecedents in indicator constraints. This is similar to the exhaustive search performed over the left-hand sides of interface definitions that our \mathcal{B} -Branch rule is meant for (Theorem 3). Whenever, in a particular node in the B&C search tree, the antecedent of an indicator constraint holds, the linear inequality in its consequent strengthens the local linear formulation, and participates in continuous relaxation, cut generation, and the other solver operations just like the original integer linear constraints.

The SCIP MILP framework [3] handles indicator constraints in the way we outlined.¹ With such support, the techniques described in Chapters 4 to 6 automatically apply to problems with propositional structure, given that the internal handling of indicator constraints is entirely orthogonal to the enhancements that a B&C-based solver needs in order to support background theories.

¹http://scip.zib.de/doc-3.1.1/html/cons__indicator_8h.php

7.4 DPLL(T)-Style Solving

The preceding sections demonstrate that Satisfiability Modulo ($\mathcal{Z} \cup T$) instances, where T is stably-infinite and signature-disjoint with \mathcal{Z} , can be encoded as MP Modulo T instances. A $BC(T)$ -based solver for MP Modulo T can thus be used as a drop-in replacement for a DPLL(T)-based SAT Modulo ($\mathcal{Z} \cup T$) solver.

While the families of theories supported by DPLL(T) and $BC(T)$ overlap, the techniques employed by the two frameworks for solving these families differ. Namely, the CDCL core empowering DPLL(T) differs fundamentally from the B&C core empowering $BC(T)$, as discussed in Section 2.4. Nevertheless, part of the techniques present in DPLL(T) can be implemented within $BC(T)$.

7.4.1 Branching and Backtracking

Branching in DPLL(T) amounts to deciding on a (possibly theory constrained) propositional (Boolean) variable. As Example 8 demonstrates, a decision extends the DPLL(T) trail with a literal. The introduced literal is specifically marked as a decision, so that the decision can later be reverted.

Branching on Boolean variables (possibly tied to first-order atoms, as described in Section 7.1) is easy to simulate in $BC(T)$. The Branch rule (present in the transition systems \mathcal{G} , \mathcal{T} , and \mathcal{A}) supports arbitrary integer linear inequalities as branching conditions. Branching on whether $v \geq 1$ or $v \leq 0$ for a Boolean variable v is a straightforward special case. The information about decisions that would be present in the DPLL(T) trail manifests itself as integer linear constraints in $BC(T)$ subproblems. These constraints (corresponding to Boolean literals) fall within the category of difference constraints (Definition 25), possibly admitting specialized techniques.

What is not easy to achieve with $BC(T)$ (and B&C in general) is the implicit nature of DPLL(T) subproblems. Remember that the negation of a DPLL(T) decision can be thought of as a subproblem that can be easily reconstructed when backtracking. Mapping DPLL(T)-style decisions to $BC(T)$ concepts explicitly introduces subproblems, both for the decision and for its

negation. The space needed for this explicit representation may thus be prohibitive for problems that involve deep branching. Additionally, CDCL-style non-chronological backtracking [83] (permitted by $\text{DPLL}(T)$) involves reverting multiple decisions at a time, *i.e.*, it would amount to eliminating multiple $\text{BC}(T)$ subproblems. That would require a version of $\text{BC}(T)$ that keeps track of the relationships between subproblems, *e.g.*, via a tree.

A $\text{BC}(T)$ variant that provides $\text{DPLL}(T)$ -style decisions and backtracking is conceivable, by combining a $\text{DPLL}(T)$ -style trail with $\text{BC}(T)$ -style subproblems, *e.g.*, by embedding a $\text{DPLL}(T)$ -style trail in $\text{BC}(T)$ subproblems. Such an extension of $\text{BC}(T)$ is beyond the scope of this dissertation.

7.4.2 Learning

Learning lemmas is a key operation in constraint solving frameworks, and $\text{DPLL}(T)$ is no exception. The flavor of learning supported by $\text{DPLL}(T)$ is its second defining characteristic, branching and backtracking being the first. We demonstrate that $\text{DPLL}(T)$ -style learning can be meaningfully implemented in $\text{BC}(T)$.

Given that $\text{DPLL}(T)$ is CDCL-based, lemmas take the form of clauses, *i.e.*, disjunctions of literals (that can be theory literals). We quote the $\text{DPLL}(T)$ theory-aware learning rule [77].

$$M \parallel F \longrightarrow M \parallel F, C$$

$$\text{if } \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models_T C \end{cases}$$

$\text{DPLL}(T)$ \mathcal{T} -Learn Rule

In the $\text{DPLL}(T)$ \mathcal{T} -Learn rule, M is a trail (Definition 14), F is a set of clauses, and C is a clause. The clauses involved are over propositional literals representing first-order literals in a signature constrained by the background theory.

For a meaningful comparison between $\text{DPLL}(T)$ and $\text{BC}(T)$, given that \mathcal{Z} is omnipresent in $\text{BC}(T)$, $\text{DPLL}(T)$ needs to be instantiated with $\mathcal{Z} \cup T$ as

the background theory, for some first-order theory T that yields a decidable combined theory $\mathcal{Z} \cup T$. We provide a $\text{BC}(T)$ rule that captures the essence of the $\text{DPLL}(T)$ \mathcal{T} -Learn rule.

$$\begin{array}{l}
 T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot [\Sigma_{0 \leq i < n} u_i - \Sigma_{0 \leq j < k} v_j \geq 1 - k]\} \parallel A \\
 \text{if } \left\{ \begin{array}{l}
 \text{diffs}(C) \models_{\mathcal{Z}} u_i \geq 0 \wedge u_i \leq 1, \text{ for } 0 \leq i < n \\
 \text{diffs}(C) \models_{\mathcal{Z}} v_j \geq 0 \wedge v_j \leq 1, \text{ for } 0 \leq j < k \\
 n + k > 1 \\
 \text{the variable symbols } u_i, v_j \text{ appear in } C \text{ or in } I \\
 \text{there exist atomic formulas} \\
 \text{- } F_i \text{ (} 0 \leq i < n \text{), where } C \wedge I \models_{\mathcal{Z}} [u_i > 0 \Leftrightarrow F_i], \text{ and} \\
 \text{- } G_j \text{ (} 0 \leq j < k \text{), where } C \wedge I \models_{\mathcal{Z}} [v_j > 0 \Leftrightarrow G_j], \\
 \text{such that } C \wedge I \models_{\mathcal{Z} \cup T} [\bigvee_{0 \leq i < n} F_i] \vee [\bigvee_{0 \leq j < k} \neg G_j].
 \end{array} \right.
 \end{array}$$

Rule \mathcal{C} -Learn : Learn a clause.

The atomic formulas F_i and G_j in the \mathcal{C} -Learn setup have the same role as theory atoms in $\text{DPLL}(T)$. F_i are the atoms that appear positively in the resulting clause, while G_j are the literals that appear negatively. The \mathcal{C} -Learn rule presupposes Boolean (*i.e.*, $\{0, 1\}$) variables u_i and v_j , which (viewed as truth values) are equivalent (according to $C \wedge I$) to the atoms F_i and G_j . As Section 7.1 demonstrates, the syntax of MPMT instances is powerful enough to introduce variables (u_i and v_j) corresponding to such atoms. In order for \mathcal{C} -Learn to be impervious to the exact encoding, the rule declaratively assumes some mapping between variables and atoms via constraints $u_i \geq 1 \Leftrightarrow F_i$ and $v_j \geq 1 \Leftrightarrow G_j$ \mathcal{Z} -entailed by $C \wedge I$. Note that this mapping is not achieved through combined $(\mathcal{Z} \cup T)$ -entailment. \mathcal{Z} -entailment is enough, given that the techniques of Section 7.1 do not take T into account. Our statement of the fact that the clause follows from the pre-existing constraints $C \wedge I$ ($C \wedge I \models_{\mathcal{Z} \cup T} [\bigvee_{0 \leq i < n} F_i] \vee [\bigvee_{0 \leq j < k} \neg G_j]$) involves combined $(\mathcal{Z} \cup T)$ -entailment. This is in order to mimic the $\text{DPLL}(T)$ side,

Chapter 7. Propositional Structure

where the background theory is $\mathcal{Z} \cup T$. Also, note that \mathcal{C} -Learn strengthens a single subproblem with a clause, in contrast to $\text{DPLL}(T)$, where learning (\mathcal{T} -Learn) is a global operation. The $\text{DPLL}(T)$ behavior can be simulated by applying \mathcal{C} -Learn on all subproblems.

Lemma 17. *If*

$$T, I, O \Vdash S \xrightarrow[\mathcal{C}\text{-Learn}]{} S',$$

then

$$T, I, O \Vdash S \xrightarrow[\mathcal{T}]{} S'.$$

Proof. The transition that forms our hypothesis is necessarily of the form given in the definition of \mathcal{C} -Learn.

$$C \wedge I \models_{\mathcal{Z} \cup T} [\bigvee_{0 \leq i < n} F_i] \vee [\bigvee_{0 \leq j < k} \neg G_j],$$

thus

$$C \wedge I \models_{\mathcal{Z} \cup T} [\bigvee_{0 \leq i < n} (u_i > 0)] \vee [\bigvee_{0 \leq j < k} \neg(v_j > 0)],$$

thus

$$C \wedge I \models_{\mathcal{Z} \cup T} \Sigma_{0 \leq i < n} u_i + \Sigma_{0 \leq j < k} (1 - v_j) \geq 1,$$

thus

$$C \wedge I \models_{\mathcal{Z} \cup T} \Sigma_{0 \leq i < n} u_i - \Sigma_{0 \leq j < k} v_j \geq 1 - k.$$

The step that relates disjunctions to inequalities is valid due to the $[0, 1]$ -bounds of the variables u_i and v_j . Given that $C \wedge I$ ($\mathcal{Z} \cup T$)-entails the inequality that \mathcal{C} -Learn introduces, this inequality can also be introduced by \mathcal{T} -Learn. \square

Given that \mathcal{C} -Learn does not add any power to the transition system \mathcal{T} , which we have proven to be sound (Theorems 1 and 2), \mathcal{C} -Learn can strengthen any of our transition systems (all of which are at most as powerful as \mathcal{T}), e.g., the transition system \mathcal{A} that leads to a complete procedure for stably-infinite theories. The introduction of \mathcal{C} -Learn does not in any way hinder the completeness strategy informed by Theorem 3. If anything, learning useful clauses via \mathcal{C} -Learn can lead to faster termination.

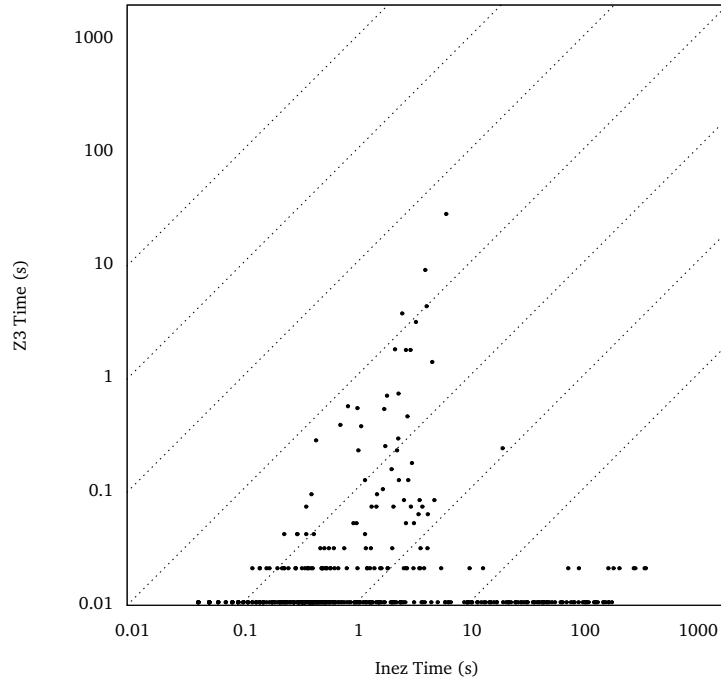


Figure 7.2: Inez vs. Z3 (SMT-LIB Instances)

7.5 Experiments

Inez (our implementation of $BC(T)$) provides support for propositional structure, and thus for SMT-like problems. We are thus able to perform a comparison between Inez and SMT solvers. We choose Z3 [24] and MathSAT [41] as representative modern SMT solvers. The version of Inez that we report on uses SCIP 3.1.1, which is the latest available version at the time of writing. Inez implements congruence closure based on the techniques discussed in Chapter 6. For propositional structure, Inez relies on SCIP-provided symbolic support for indicator constraints (Section 7.3).

Our experiments are based on the SMT-LIB [8], which is the set of benchmarks used for the SMT Competition [6]. SMT solvers are thus tuned against the SMT-LIB benchmarks. We report on the most relevant SMT-LIB category, which is QF_UFLIA (Quantifier-Free Linear Integer Arithmetic with Uninterpreted Functions). We use 564 benchmarks, 396 of which are satis-

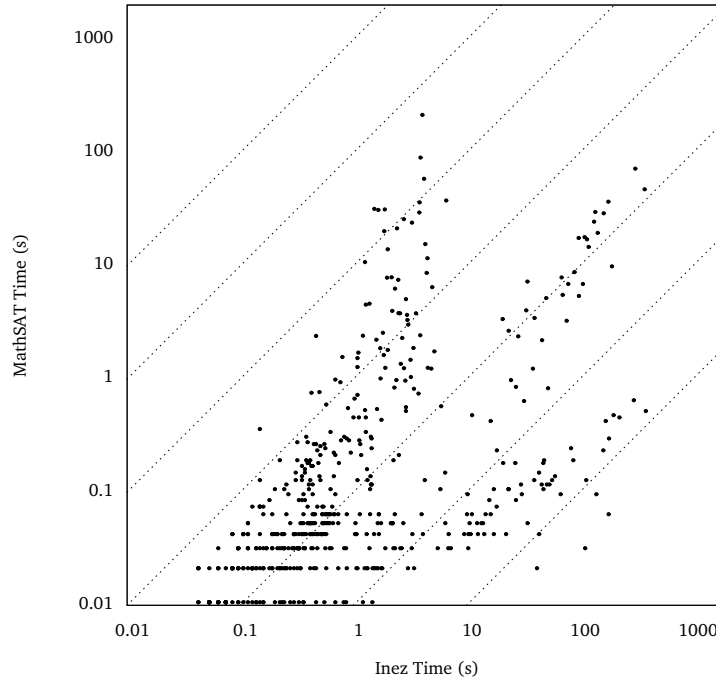


Figure 7.3: Inez vs. MathSAT (SMT-LIB Instances)

fiable and 168 of which are unsatisfiable. Our comparison excludes a set of benchmarks that was recently added to the SMT-LIB, and which uses SMT-LIB language features that Inez does not yet support.

Figure 7.2 provides a comparison between Inez and Z3. Figure 7.3 provides a comparison between Inez and MathSAT. Z3 and MathSAT solve all 564 benchmarks, while Inez solves all but 7 (*i.e.*, more than 98% of the instances). While Inez is generally slower than the more mature SMT solvers, the majority of the benchmarks (333) require less than a second, 473 benchmarks require less than 10 seconds, and 534 less than 100 seconds.

Z3 employs very sophisticated preprocessing techniques that are frequently able to solve the instance by themselves, hence the vast majority of the instances (535) are solved in less than 0.1 seconds. In comparison, Inez only applies very limited preprocessing, so it commonly resorts to time-consuming search where Z3 does not. While not as sophisticated as Z3, MathSAT is a robust SMT solver in its own right. Inez is faster than Math-

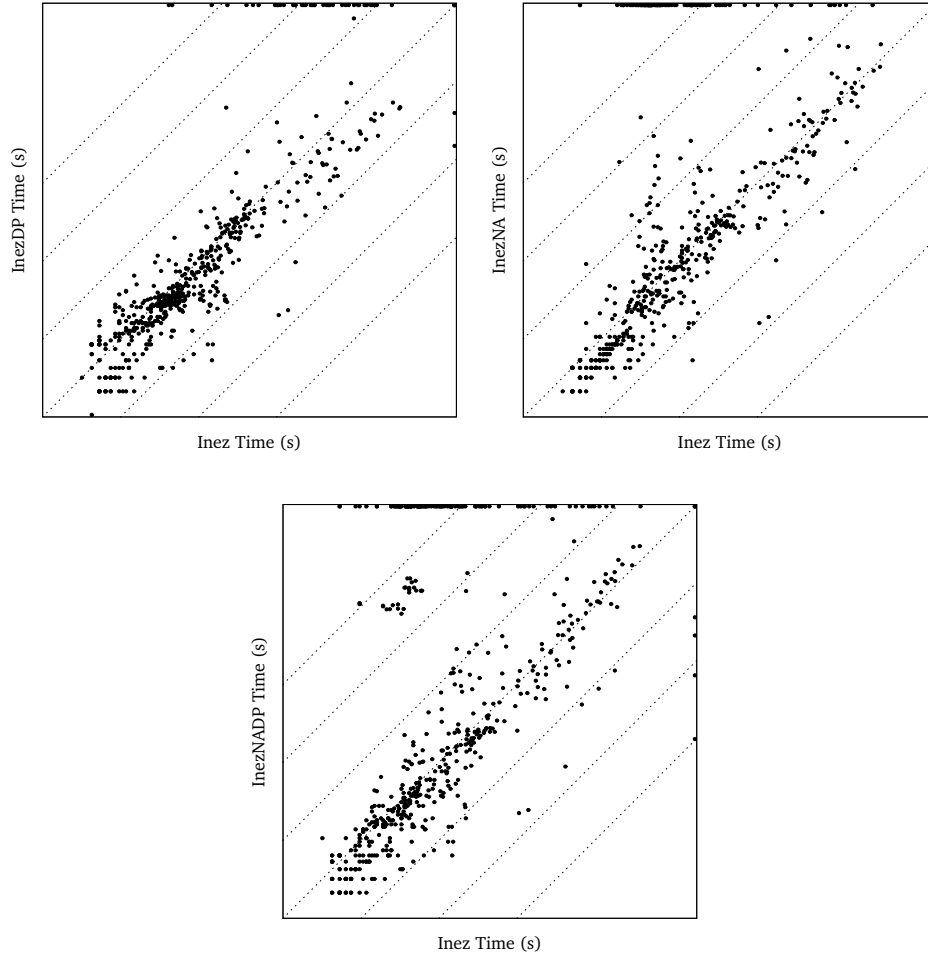


Figure 7.4: Impact of Custom SCIP Settings (SMT-LIB Instances)

SAT for 47 benchmarks, within a factor of 2 for 81 benchmarks, and within a factor of 8 for the majority of the benchmarks (287).

Obtaining the version of Inez that solves almost all of the instances required extensive experimentation with SCIP. SCIP can be customized in multiple ways, drastically altering the behavior of the solver. Figure 7.4 visualizes the impact of the customizations that we discovered to be crucial. What appears as Inez in Figure 7.4 is exactly the same configuration that appears as Inez in Figures 7.2 and 7.3. InezDP stands for a configuration that uses the default set of SCIP plugins, as opposed to Inez that uses a subset of the default plugins. InezNA uses the default settings for the time spent on

heuristics and separation, as opposed to lnez that is more aggressive with (*i.e.*, spends more time on) heuristics and separation. lnezNADP uses the default set of plugins and default settings for heuristics and separation. The most striking difference is the number of instances that lnez is able to solve: 557 versus 519 for lnezDP, 447 for lnezNA, and 450 for lnezNADP. While the plots demonstrate variability for the benchmarks that all configurations solve, most of the instances are close to the diagonals, *i.e.*, lnez is not visibly outperformed for any significant subset of the benchmarks. The need to significantly diverge from the default SCIP configuration indicates significant differences between the SMT-LIB benchmarks and the OR-like problems that SCIP is primarily meant for.

Interestingly, even for our best lnez configuration that is used for Figures 7.2 and 7.3, the underlying SCIP solver learns no cutting planes whatsoever for 377 out of the 557 solved instances. For the remaining instances the number of cuts is limited. Namely, 36 instances lead to a single cut, 50 lead to 2 cuts, 64 lead to 3 cuts, 16 lead to 4 cuts, and the remaining 14 instances lead to 11 cuts or less. Based on this observation, the branching part of lnez’s Branch-and-Cut algorithm is being stressed here. We have not tried to optimize branching heuristics, so there is plenty of room for improvement. More importantly, the instances are not representative of arithmetic-heavy optimization problems, where we would expect more cuts.

A final observation is that SCIP performs floating-point (FP) arithmetic, which may lead to wrong answers. Interestingly, lnez provides no wrong answers for the benchmark set in question, *i.e.*, the instances do not pose numerical difficulties. The fact that we learn very few cutting planes partially explains why. There is little room for learning anything at all, let alone for learning something unsound.

Chapter 8

Local Theory Extensions

We study a family of theories whose signatures are not disjoint from the signature $\Sigma_{\mathcal{Z}}$ (Definition 10) of \mathcal{Z} (Definition 11). Such theories may use arithmetic to constrain new function symbols.

Example 13. *Consider the following monotonicity axiom, which has already appeared in Example 9:*

$$\forall x. \forall y. [y \leq x \Rightarrow f(x) \leq f(y)] \quad (8.1)$$

Equation 3.3 is only meaningful as an extension of \mathcal{Z} . The intended meaning (i.e., that f is monotonically decreasing) is only achieved because \leq is already constrained by \mathcal{Z} .

The signature disjointness requirement of Fact 1 (which manifests itself in our completeness arguments for the transition system \mathcal{A} in Chapter 5) rules out our monotonicity axiom. It is thus meaningful to overcome this requirement. To encompass axioms like the one of Example 13, we adapt the machinery of local theory extensions [84] to the MPMT context.

Theories whose signatures overlap with $\Sigma_{\mathcal{Z}}$ still fit in the general form of MPMT instances (Definition 17), while the transition system \mathcal{T} (Definition 24) is sound even when the background signature overlaps with $\Sigma_{\mathcal{Z}}$. What does change is that a new strategy for applying the $\text{BC}(\mathcal{T})$ rules is required to achieve completeness.

The approach we describe effectively provides some support for quantifiers in MPMT. So far, our input has been quantifier-free, modulo the theory. The theory is thought of as hard-coded, with only the theory solver (a software module external to the core B&C framework) understanding it. The theories we study are comprised of user-provided universally quantified axioms that are handled by $BC(T)$ -provided functionality. Universal quantifiers thus appear in our input and not just in a hard-coded theory.

8.1 Preliminaries

We adapt the concepts involved in local theory extensions for MPMT. In the interest of succinctness, we restrict our attention to axioms that involve no new predicate or constant symbols. (The axioms thus only involve function symbols.) We also focus on axioms that have a clausal form, which clearly encompasses implicative constraints, *e.g.*, the monotonicity axiom of Example 13. \mathcal{Z} is omnipresent as the base theory. Conversely, to keep the naming scheme for new concepts simpler, the prefix Σ doesn't mean that the corresponding object only involves the signature Σ , but rather that it involves $\Sigma_{\mathcal{Z}} \cup \Sigma$.

Definition 32 (Σ -Flat Atomic Formula). *Let Σ be a signature containing only function symbols, such that $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$. A $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -atomic formula C is Σ -flat if*

- (a) *only variables appear in C under function symbols from Σ , and*
- (b) *C is not an equality.*

Definition 33 (Σ -Axiom). *Let Σ be a signature containing only function symbols, such that $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$. A $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula X of the form*

$$\forall x_0. \dots \forall x_{k-1}. \left[\bigvee_{i=0}^{n-1} c_i \right], \quad (8.2)$$

where c_i are Σ -flat atomic formulas, is called a Σ -axiom if

- (a) *X contains no variable symbols other than x_0, \dots, x_{k-1} ;*

Chapter 8. Local Theory Extensions

- (b) no terms other than the variable symbols x_i ($0 \leq i < k$) appear under Σ -function symbols in X ;
- (c) whenever a variable x_i appears in two Σ -terms in X , the two terms are identical;
- (d) no x_i appears twice in the same Σ -term in X ; and
- (e) every x_i appears in some Σ -term in X .

An *instance* of a Σ -axiom $X = [\forall x_0. \dots \forall x_{k-1}. (\bigvee_{i=0}^{n-1} c_i)]$ (for some signature Σ) is a formula produced by substituting the variables x_i with $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -terms t_i in $\bigvee_{i=0}^{n-1} c_i$. We view the literals of an axiom (or instance thereof) as a set whenever convenient. Given a signature Σ and a quantifier-free $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula F , we denote by $\mathcal{U}_{\Sigma}(F)$ the (finite) set of terms in F that start with a function symbol from Σ .

Given a signature Σ , a set of Σ -axioms T and a quantifier-free $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula F , let

$$T[F] = \bigcup_{X \in T} \{x \mid x \text{ is an instance of } X \text{ such that } \mathcal{U}_{\Sigma}(x) \subseteq \mathcal{U}_{\Sigma}(F)\};$$

$T[F]$ is finite. A Σ -extension of our base theory \mathcal{Z} is a theory (set of Σ -axioms) T defined on top of \mathcal{Z} , with the property that for any quantifier-free $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula F , only the finite set of axiom instances $T[F]$ matters. We subsequently formalize the notion of an extension.

Definition 34 (Σ -Extension of \mathcal{Z}). *Let Σ be a signature containing only function symbols, such that $\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$. We say that a set of Σ -axioms T is a local extension of \mathcal{Z} if for every quantifier-free $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula F such that $\mathcal{Z} \cup T \cup \{F\}$ is unsatisfiable, it is also the case that $\mathcal{Z} \cup T[F] \cup \{F\}$ is unsatisfiable.*

We slightly restrict the notion of an interface definition (Definition 16) that was used in previous chapters.

Definition 35 (Flat Σ -(Interface) Definition). *Let Σ be a signature such that*

$\Sigma \cap \Sigma_{\mathcal{Z}} = \emptyset$. A flat Σ -interface definition is a formula of the form

$$v = f(v_0, \dots, v_{n-1}),$$

where $v \in \mathcal{V}$, f is a Σ -function symbol, and v_i are variable symbols.

Flat interface definitions are clearly a special case of interface definitions (Definition 16). Given our restriction on signatures Σ that only contain function symbols, any non-flat interface definition can be encoded as a set of flat interface definitions by appropriately introducing auxiliary variables.

Given a Σ -extension T of \mathcal{Z} , we can determine the satisfiability modulo $\mathcal{Z} \cup T$ of a quantifier-free formula F by substituting T with $T[F]$. With the axioms of T gone, the function symbols from Σ become uninterpreted, and can thus be handled by congruence closure (CC) [72, 76]. The problem becomes solving the (quantifier-free) formula $F \wedge T[F]$ modulo the combination of \mathcal{Z} with uninterpreted functions, which is decidable. From an algorithmic point of view, all that is needed is an instance of $\text{BC}(T)$ with CC as the theory solver. Chapter 6 elaborates on such a combination. Eagerly producing $T[F]$ may however be impractical, due to the sheer number of axiom instances involved.

8.2 Axiom Instantiation

We discuss MPMT instances (Definition 17) of the form C, O, I, T , where (for a signature Σ disjoint from $\Sigma_{\mathcal{Z}}$ that only has function symbols) T is a set of Σ -axioms and I is a set of flat Σ -definitions. The locality notion of Definition 34 translates to instantiating all the axioms in $T[I]$, given that $\mathcal{U}_{\Sigma}(C) = \emptyset$. We describe a scheme that allows us to solve such instances without necessarily producing $T[I]$ in an eager fashion. Our scheme is described by means of new $\text{BC}(T)$ transition rules.

In our context, for any Σ -term that appears in a $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula (e.g., in an axiom instance), the applicable set of flat interface definitions provides a corresponding variable symbol. For a $(\Sigma_{\mathcal{Z}} \cup \Sigma)$ -formula F and a set of Σ -flat definitions I , we denote by $\text{abstract}(F, I)$ the $\Sigma_{\mathcal{Z}}$ -formula obtained from F

by substituting the Σ -terms with corresponding variable symbols obtained from I .

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot \text{abstract}(c, I)\} \parallel A$$

if $\begin{cases} T \text{ is a set of } \Sigma\text{-axioms} \\ I \text{ is a set of flat } \Sigma\text{-interface definitions} \\ \text{for some instance } (L \uplus \{c\}) \in T[I], C \models_{\mathcal{Z}} \neg \text{abstract}(L, I) \end{cases}$

Rule \mathcal{L} -Instantiate : Instantiate an axiom.

The \mathcal{L} -Instantiate rule instantiates axiom instances as follows. Given an instance in $T[I]$ where all the literals (inequalities) but one are falsified, then the remaining literal can be instantiated as a cut. While the literal may contain terms t_i that contain Σ -function symbols, such terms are substituted out via `abstract`. Given that clauses are sets of literals, the order of literals does not matter, so the rule applies for any literal in an instance from $T[I]$.

\mathcal{L} -Instantiate involves entailment of the $\Sigma_{\mathcal{Z}}$ -formula $\neg \text{abstract}(L, I)$ by the applicable set of integer linear constraints C . This seemingly involves \mathcal{Z} -satisfiability queries. However, such queries are neither mandatory nor recommended. First, the entailment trivially holds if $\neg c \in C$ for every $c \in L$, i.e., a simple syntactic (but also weak) check can be used. Also, for any difference constraint (Definition 25) in $\text{abstract}(L, I)$, checking its violation can happen simply based on known bounds; e.g., $v - w \leq k$ is violated if the known lower bound on $v - w$ is greater than k . Relying on nothing but difference constraints suffices for Example 13.

\mathcal{L} -Instantiate makes no assumptions about the representation of $T[I]$. For example, the solver may compute relevant axiom instances incrementally. Only a few of the clauses in $T[I]$ may end up being useful, either because a small set is enough to produce a contradiction, or because for most of the clauses at least one literal is \mathcal{Z} -entailed, and thus \mathcal{L} -Instantiate doesn't apply. Note that even for a simple implementation that precomputes and explicitly represents $T[I]$ (whose size is exponential in the size of the input in the general case), we never resort to encoding the disjunctive propositional structure of its instances in terms of integer linear constraints.

\mathcal{L} -Instantiate resembles *Boolean Constraint Propagation (BCP)* [64]. BCP operates over literals, and propagates a Boolean literal belonging in a clause of length n when the remaining $n - 1$ are falsified. \mathcal{L} -Instantiate operates over clauses that involve arithmetic. Introducing a cut on the $\text{BC}(T)$ side is equivalent to fixing the n -th literal on the BCP side.

$$T, I, O \Vdash P \uplus \{C\} \parallel A \longrightarrow P \cup \{C \cdot \text{abstract}(c, I), C \cdot \neg \text{abstract}(c, I)\} \parallel A$$

$$\text{if } \left\{ \begin{array}{l} T \text{ is a set of } \Sigma\text{-axioms} \\ I \text{ is a set of flat } \Sigma\text{-interface definitions} \\ \text{some instance } (L \uplus \{c\}) \in T[I] \\ \text{abstract}(c, I) \notin C \\ \neg \text{abstract}(c, I) \notin C \end{array} \right.$$

Rule \mathcal{L} -Branch : Branch on literals appearing in axiom instances.

For obtaining truth values for literals in axiom instances (in order to apply \mathcal{L} -Instantiate), branching on these literals may be necessary. The \mathcal{L} -Branch rule is a version of Branch specialized for branching driven by the axioms. By systematically branching based on \mathcal{L} -Branch, and by correspondingly applying \mathcal{L} -Instantiate when truth values for enough literals are obtained, the core ILP solver becomes aware of all applicable consequences of the axioms.

The combination of \mathcal{L} -Instantiate and \mathcal{L} -Branch is responsible for introducing axiom instances. However, these rules cannot enforce that the Σ -function symbols in I meet Leibniz's rule, *i.e.*, CC steps cannot be described in terms of \mathcal{L} -Instantiate and \mathcal{L} -Branch. \mathcal{L} -Instantiate and \mathcal{L} -Branch thus need to be combined with rules from the transition system \mathcal{A} .

Definition 36 (The Transition System \mathcal{L}).

$$\begin{aligned} \mathcal{L} = \{ & \text{Branch, Drop, Prune, Learn,} \\ & \mathcal{T}\text{-Improve, } \mathcal{T}\text{-Unbounded,} \\ & \mathcal{A}\text{-Propagate}^+, \mathcal{A}\text{-Propagate}^-, \\ & \mathcal{A}\text{-Forget, } \mathcal{A}\text{-Branch, } \mathcal{A}\text{-Unsat,} \\ & \mathcal{L}\text{-Instantiate, } \mathcal{L}\text{-Branch} \} \end{aligned}$$

The transition system \mathcal{L} only operates on MPMT instances C, O, I, T where (for some appropriate Σ) T is a set of Σ -axioms and I is a set of Σ -flat interface definitions. The definitions in I are only equational. We thus omit the rules $\mathcal{B}\text{-Propagate}^+$, $\mathcal{B}\text{-Propagate}^-$, and $\mathcal{B}\text{-Branch}$, which would serve no purpose.

Lemma 18. *If $T, I, O \Vdash S \xrightarrow{\mathcal{L}} S'$, then $T, I, O \Vdash S \xrightarrow{\mathcal{T}} S'$.*

Proof. \mathcal{T} -Learn simulates \mathcal{L} -Instantiate. Branch simulates \mathcal{L} -Branch. □

Completeness: From Lemma 18 and the fact that \mathcal{T} is sound (Theorems 1 and 2), it follows that \mathcal{L} is also sound. Additionally, \mathcal{L} provides a complete procedure for MP Modulo T , where T is a Σ -extension of \mathcal{Z} . The completeness strategy for stably-infinite theories (Theorem 3 and associated discussion) needs to be extended with appropriate branching via \mathcal{L} -Branch over the literals $T[I]$, where I is the set of applicable interface definitions. If \mathcal{L} -Branch produces subproblems such that at least $n - 1$ literals are known for every clause of length n in $T[I]$, then for every clause in $T[I]$, either we know that at least one literal is satisfied, or we can apply \mathcal{L} -Instantiate. By applying \mathcal{L} -Instantiate wherever applicable, we guarantee that $T[I]$ is fully enforced, and thus that T is enforced. What remains is solving ILP and UF constraints. Given that the uninterpreted functions are constrained by the empty theory, which is stably-infinite, all that we need is the techniques of Chapters 5 and 6.

8.3 Implementation

lnez provides prototype support for local theory extensions. Our implementation relies on a SCIP-provided handler¹ for *bound disjunction* constraints, which are of the form

$$\bigvee_{i \in [0, n-1]} [v_i \{\leq, \geq\} r_i], \quad (8.3)$$

where v_i are variable symbols and r_i are rational constants. (For our purposes, integer constants suffice.)

We obtain constraints of the form 8.3 out of our axiom instances in $T[I]$ (where T is the applicable set of Σ -axioms, and I the applicable set of flat Σ -interface definitions) by applying the following transformations. First, we eliminate Σ -terms from the instances by substituting based on I (as explained for the function abstract used throughout this chapter). Subsequently, we eliminate inequalities involving multiple variables by appropriately defining auxiliary variables.

For the kinds of axioms that we anticipate, the auxiliary variables needed are introduced for the sake of CC anyway (Section 6.6). For example, for monotonicity (Example 8.1), the axiom antecedents $x = y$ (for relevant x and y) lead to variables $d_{x,y}$ and constraints $d_{x,y} = x - y$, allowing the CC procedure to determine whether $x = y$ by inspecting the bounds of $d_{x,y}$; the axiom consequents $f(x) = f(y)$ lead to variables $d_{f(x),f(y)} = f_x - f_y$ (where the auxiliary variables f_x and f_y stand for $f(x)$ and $f(y)$) that allow the CC procedure to potentially learn $f(x) = f(y)$ by fixing $d_{f(x),f(y)}$ to 0. The same auxiliary variables $d_{x,y}$ and $d_{f(x),f(y)}$ are utilized for encoding axiom instances as bound disjunctions. Axioms thus introduce a second form of reasoning over the same auxiliary variables that CC uses, that operates synergistically with CC. The facts that CC learns benefit axiom-based reasoning, and vice versa.

Bound disjunctions are essentially clauses. Therefore, SCIP's handler for bound disjunctions can (and does) use techniques from SAT, e.g., the two-watched-literal (2WL) scheme [70] for BCP. The difference with SAT is that

¹http://scip.zib.de/doc-3.1.1/html/cons__bounddisjunction_8h.php

the literals involved are non-propositional. Obtaining and fixing truth values amounts to querying over and fixing bounds. The propagation enabled by 2WL is modeled by the \mathcal{L} -Instantiate rule.

In effect, the scheme we outline is a reversal of the DPLL(T) architecture. In DPLL(T), the core solver is CDCL-based, with B&C possibly appearing inside a theory solver for arithmetic (Section 2.5.1). In BC(T), the core solver is B&C-based, with CDCL-like reasoning happening inside a theory solver. The two frameworks thus differ with respect to the kind of reasoning that they prioritize, rather than the kind of reasoning that they theoretically permit.

Clearly, there is room for improvement in our implementation, *e.g.*, the scheme described by the rules \mathcal{L} -Instantiate and \mathcal{L} -Branch can be implemented by lazily producing the set of interesting axiom instances. This possibility is not explored further in this dissertation.

8.4 Experiments

Inez provides a collection of examples that involve local axioms.² Figure 8.1 reports on experiments with a subset of these examples, which rely on the axioms given in Figure 8.2. The purpose of our experiments is to provide some intuition about the performance to be expected from an implementation following the ideas of Section 8.3.

In Figure 8.1, the horizontal axis corresponds to the number of appearances n of the axiom-constrained function symbols involved. The instances are parametric with respect to n , so we are able to run experiments for different n and thus evaluate the scalability of our techniques. Each point (n, t) on the line corresponding to instance x indicates that x with n as the parameter is solved in t seconds.

The instances `f_lt_g` and `choose` are satisfiable, while `bounds` and `mono` are unsatisfiable. For `mono` and `f_lt_g`, the number of axiom instances needed as per the locality criterion of Definition 34 is quadratic in

²<https://github.com/vasilisp/inez/tree/master/frontend/examples/axioms>

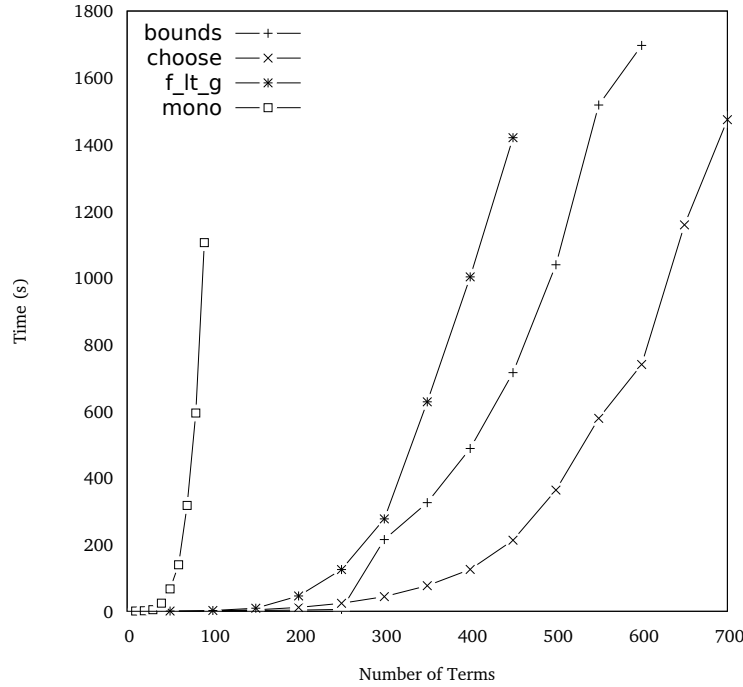


Figure 8.1: `lnez` on Instances with Local Axioms

the number of relevant terms n ; for `bounds` and `choose`, the number of axiom instances is linear in n . In all cases, the constraints are chains of inequalities involving the axiom-constrained function symbols. The constraints are not trivial. Namely, the satisfiable instances involve at least linearly many axiom instances that are not trivially satisfied, while for the unsatisfiable instances at least linearly many axiom instances are involved in deriving a contradiction.

For the easiest instance (`choose`), `lnez` scales to 700 terms. In the other extreme, `mono` scales to only 90 terms. The instances that lead to a linear set of axiom instances scale better than the instances that require quadratically many axioms. The satisfiable instances scale better than the unsatisfiable ones. For all four instances, the time increases roughly exponentially. While the results are not surprising, the behavior could vary based on the nature of the constraints involved.

Chapter 8. Local Theory Extensions

bounds	$\begin{aligned} \forall x.[l \leq x \wedge x \leq u] &\Rightarrow l' \leq f(x), \quad \text{and} \\ \forall x.[l \leq x \wedge x \leq u] &\Rightarrow f(x) \leq u'. \end{aligned}$
choose	$\begin{aligned} \forall x.\forall y.\text{choose}(x, y) < x &\Rightarrow \text{choose}(x, y) \leq y, \\ \forall x.\forall y.\text{choose}(x, y) < x &\Rightarrow \text{choose}(x, y) \geq y, \\ \forall x.\forall y.\text{choose}(x, y) > x &\Rightarrow \text{choose}(x, y) \leq y, \quad \text{and} \\ \forall x.\forall y.\text{choose}(x, y) > x &\Rightarrow \text{choose}(x, y) \geq y. \end{aligned}$
f_lt_g	$\forall x.f(x) < g(x)$
mono	$\forall x.\forall y.x \leq y \Rightarrow f(x) \leq f(y)$

Figure 8.2: Axioms Used for Figure 8.1

Chapter 9

Data

This chapter examines an application of MPMT and $BC(T)$. Namely, we describe how to integrate a procedure that performs queries over database-like tables as a background procedure in $BC(T)$. The result is a kind of constraint solving where logical properties may express the existence of certain kinds of tuples in a database.

Our techniques enhance constraint solvers with data. Dually, our techniques enhance database systems with constraints. Concretely, our approach leads to database systems that (a) store symbolic (in addition to concrete) data, and at the same time (b) allow queries of a symbolic nature, *e.g.*, with free variables. Symbolic data allows us to encode partially specified or entirely speculative information, *e.g.*, database entries that exist for the purpose of what-if analysis. Symbolic queries enable deductive reasoning about data. We outline a range of applications that can benefit from a combination of constraints and data.

Existing relational query languages (*e.g.*, SQL) only allow concrete data and queries. Symbolic enhancements require a formalism that combines constraints and relational queries. We address this need by introducing the Δ logic. Δ extends quantifier-free Linear Integer Arithmetic (QFLIA) with database tables and operators from Relational Algebra, like selection (σ), union (\cup) and cross product (\times).

While Δ is decidable, the logic in its general form gives rise to hard satisfiability problems, primarily because it allows universal quantification

over cross products of big tables. We study unrestricted Δ , for it is a natural umbrella formalism. We subsequently provide restrictions that enable an efficient $BC(T)$ -based procedure, and elaborate on this procedure.

9.1 Motivating Example

Our motivating example (formalized in Figure 9.1) concerns the problem of optimally investing a given amount of capital. This is an appropriate application for our techniques, because

- (a) investments are almost always data-driven as they take historical stock prices into account, and
- (b) financial institutions already rely on Mathematical Programming.

The problem involves investing in a *portfolio* of n publicly traded stocks, with the goal of maximizing profit while following guidelines that minimize risk. A database provides information on these stocks, including stock prices from the New York Stock Exchange (NYSE). We pick the n stocks that would have yielded the highest profit over a period of time in the recent past, *e.g.*, over the preceding year. This optimization problem is subject to risk-mitigation constraints that require us to pick companies from a variety of sectors. While investing in the exact solver-generated portfolio (which relies only on past performance) is not necessarily a good strategy, such a portfolio provides useful information for the analysts who make the final investment decisions.

The data is given in tables `stocks` and `quotes` (Figures 9.1a and 9.1b). Each company in `stocks` is described by a unique ID (with the associated NYSE symbol parenthesized), its capitalization (small, medium, or large), and its sector (*e.g.*, tech, retail, financials, automotive, energy, emerging-markets). While Figure 9.1 uses human-readable names, we can encode these fields with bounded integer quantities. Each entry in `quotes` describes the observed movement of a certain stock in a given timeframe, assuming reinvestment of dividends. For example, the first row describes an increase of 28% in the price of EMC. `quotes` is an application-specific

Id	Cap	Sector	Id	Diff
1 (EMC)	large	tech	1	128
2 (FII)	medium	financials	2	117
3 (AKR)	small	retail	3	89
...

(a) stocks
(b) quotes

maximize

$$\sum_{1 \leq i \leq n} a_i \cdot d_i$$

subject to

$$\begin{aligned}
 (x_i, c_i, s_i) &\in \text{stocks}, & 1 \leq i \leq n \\
 (x_i, d_i) &\in \text{quotes}, & 1 \leq i \leq n \\
 x_i &\neq x_j, & 1 \leq i < j \leq n \\
 \sum_{\{i \mid 1 \leq i \leq n, s_i = s\}} a_i &\leq \sum_{1 \leq i \leq n} a_i / 3, & \text{for every sector } s \\
 \sum_{\{i \mid 1 \leq i \leq n, c_i = \text{small}\}} a_i &\leq \sum_{1 \leq i \leq n} a_i / 4
 \end{aligned}$$

(c) Constraints

Figure 9.1: Portfolio Management

abstraction, *i.e.*, the actual database contains past stock prices and quotes is a *view* produced by comparing data for two time periods.

The i^{th} stock in the portfolio is characterized by a unique ID x_i that corresponds to entries in the dataset, *i.e.*, there exist entries $(x_i, c_i, s_i) \in \text{stocks}$ and $(x_i, d_i) \in \text{quotes}$. To minimize risk, we force the n IDs x_i to be distinct, and allow no single sector to account for more than a third of the total capital. Additionally, no more than a fourth of the capital goes to smallcap companies. The objective function maximizes the capital at the end of the period, and thus the profit.

Note that if the amounts a_i are variables, the objective function is non-linear. The problem can be circumvented by providing integer constants for a_i , *i.e.*, by specifying how the capital is to be partitioned. With constants for a_i , the non-table constraints are essentially in QFLIA. (The summations for i that satisfy conditions like $s_i = s$ and $c_i = \text{small}$ are easy to encode as sums of if-then-else terms.) Conversely, the problem is essentially satisfiability of an arithmetic instance, where certain variables correspond to database contents. This is the kind of problem that our combination of $\text{BC}(T)$ with

$$\begin{aligned}
 F &::= T_1 \leq T_2 \mid \exists D \mid \neg F \mid F_1 \vee F_2 \\
 D &::= \{T^+\} \mid \langle \sigma x : F : D \rangle \mid D_1 \times D_2 \mid D_1 \cup D_2 \\
 T &::= (T_1, T_2) \mid \text{left}(T) \mid \text{right}(T) \mid \\
 &\quad x \mid K \mid K \cdot T \mid T_1 + T_2 \\
 K &::= \dots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots
 \end{aligned}$$

Figure 9.2: Grammar of Δ

data is meant for. We cannot use a standalone DBMS, since DBMS's do not handle constraints and optimization. Neither are existing solvers up to the task, since they do not provide ways of managing data.

The constraints we have described are meant to be representative. Clearly investors also have to consider other options, including investing in index funds, bonds, debt securities and derivative contracts. These financial instruments may have other characteristics that need to be modeled. Our constraints also rely on simplifying assumptions, *e.g.*, that we can invest an arbitrary amount in any given stock at any time. It is not within our scope to model investment problems comprehensively. What matters is that these additional concerns also mix arithmetic with data, thus reinforcing the need for data-aware solving.

9.2 The Logic Δ

This section introduces the logic Δ . Δ combines arithmetic with queries over tabular data. Δ thus encompasses database problems like our motivating example of Section 9.1.

The grammar of Δ is given in Figure 9.2. K , T , D , and F are the non-terminal symbols for integer constants, terms, tables, and formulas, respectively. The first line of productions for T corresponds to pairs and their accessors; the second line is for variable symbols (x) and integer expressions. A table (non-terminal symbol D) is either an *input table*, a *selection*, a *cross-product*, or a *union*. The selection $\langle \sigma x : F : D \rangle$ is a table that con-

sists of only these entries in D that satisfy F , i.e., the variable x ranges over the table entries; σ binds x in F , but not in D . For formulas (non-terminal symbol F), $\exists D$ should be read as “ D is not empty”. All other constructs bear the obvious meaning. We assume that all variables not bound by σ are integer. We freely use derived operators, e.g., conjunction and integer equality.

Δ is typed. Each term is either of type `int` or of type $s * t$, where s and t are types. `left` and `right` are only permissible when applied to a term of type $s * t$ for some type s and some type t ; if x is of type $s * t$, then `left`(x) is of type s and `right`(x) is of type t . The integer constants are of type `int`. The arithmetic operators ($+$, \cdot , and \leq) only apply to terms of type `int`; $+$ and \cdot produce integers. Each table has a *schema*, which is the type of its entries. (Schemas are the table-level counterpart of types.) An input table is comprised of entries of the same type. If table D_1 has schema s_1 and table D_2 has schema s_2 , then $D_1 \times D_2$ has schema $s_1 * s_2$. For $\langle \sigma x : F : D \rangle$ to be properly typed, F should be a properly-typed formula under the assumption that the type of x is the schema of D ; the schema of $\langle \sigma x : F : D \rangle$ is the same as that of D . Union expects tables of the same schema and preserves it.

Clearly, Δ is at least as powerful as QFLIA. At the same time, Δ encompasses most features one would expect from a relational query language. We have left out certain operators usually present in query languages. First, note that projection (π) would not provide additional power, since it is possible to refer to any subset of the columns, without producing an intermediate table that leaves out the irrelevant ones. Also, the set difference $A \setminus B$ can be encoded as $\langle \sigma a : \neg \exists b : a = b : B \rangle : A$, assuming that the schema of A and B has exactly one column; otherwise, in place of $a = b$ we would have a conjunction of equalities over all columns. Additionally, Δ can express many forms of aggregation, including count (when compared to a constant), min, and max.

Example 14. *The portfolio encoded by Figure 9.1 can be represented as the input table*

$$\text{portfolio} = \{(1, (x_1, a_1)), \dots, (n, (x_n, a_n))\}.$$

$$\begin{aligned}
 \llbracket \{r_1, \dots, r_n\} \rrbracket &= \{r_1 \oslash \text{true}, \dots, r_n \oslash \text{true}\} \\
 \llbracket \langle \sigma x : F : D \rangle \rrbracket &= \{r \oslash (b \wedge \llbracket F[x/r] \rrbracket) \mid r \oslash b \in \llbracket D \rrbracket\} \\
 \llbracket D_1 \times D_2 \rrbracket &= \{(r_1, r_2) \oslash (b_1 \wedge b_2) \mid r_i \oslash b_i \in \llbracket D_i \rrbracket, i = 1, 2\} \\
 \llbracket D_1 \cup D_2 \rrbracket &= \llbracket D_1 \rrbracket \cup \llbracket D_2 \rrbracket \\
 \llbracket T_1 \leq T_2 \rrbracket &= \llbracket T_1 \rrbracket \leq \llbracket T_2 \rrbracket \\
 \llbracket \exists D \rrbracket &= \bigvee_{r \oslash b \in \llbracket D \rrbracket} b \\
 \llbracket \neg F \rrbracket &= \neg \llbracket F \rrbracket \\
 \llbracket F_1 \vee F_2 \rrbracket &= \llbracket F_1 \rrbracket \vee \llbracket F_2 \rrbracket
 \end{aligned}$$

 Figure 9.3: Reducing Δ to QFLIA

portfolio contains symbolic data, something which is not allowed by DBMS's. The first column ensures that the n entries are distinct, irrespective of the assignment. portfolio is of schema $\text{int}^*(\text{int}^* \text{int})$. Consider the following constraint:

$$\neg \exists \left[\begin{array}{l} \langle \sigma x : \text{left}(\text{left}(x)) \neq \text{left}(\text{right}(x)) \wedge \\ \text{left}(\text{right}(\text{left}(x))) = \text{left}(\text{right}(\text{right}(x))) \\ : \boxed{\text{portfolio}} \times \boxed{\text{portfolio}} \rangle \end{array} \right]$$

The constraint states that there exist no entries $(i, (x_i, a_i))$ and $(j, (x_j, a_j))$ in portfolio such that $i \neq j$ and $x_i = x_j$, i.e., portfolio references n distinct stocks, as was our intention in Figure 9.1. The constraint essentially involves universal quantification over $\text{portfolio} \times \text{portfolio}$.

9.2.1 Decidability

Δ satisfiability can be reduced to QFLIA satisfiability. We explain the reduction briefly. We represent a table expression D of schema s as a set $\llbracket D \rrbracket$ consisting of pairs $r \oslash b$, where r is a term of type s and b is a QFLIA formula, with the intended meaning that r is present in the table iff b is true. We use the operator \oslash to distinguish the auxiliary pairs used for the reduction from the ones allowed by the syntax of Δ . For a formula F , $\llbracket F \rrbracket$ denotes the cor-

responding formula in QFLIA; similarly for integer terms. $F[x/r]$ stands for substituting x with r in F , with appropriate care for occurrences of the symbol x bound by σ inside F . Figure 9.3 defines $\llbracket \cdot \rrbracket$ for tables and formulas as two mutually recursive functions.

For encoding Δ integer terms as QFLIA terms (e.g., $\llbracket T_i \rrbracket$ in Figure 9.3), all that needs to be done is elimination of pair constructors and accessors via the rules $\text{left}((x, y)) = x$ and $\text{right}((x, y)) = y$. The reduction suffices to establish decidability of Δ . The reduction also provides formal semantics for Δ by specifying its meaning in terms of QFLIA.

9.2.2 Complexity

Theorem 5. *The satisfiability problem for Δ is in NEXPTIME.*

Proof Sketch. The reduction to QFLIA (Figure 9.3) produces a formula exponentially larger than the input. Since QFLIA is in NP, the reduction provides a non-deterministic exponential time procedure for Δ -satisfiability. \square

Theorem 6. *The satisfiability problem for Δ is PSPACE-hard.*

Proof Sketch. We reduce the (PSPACE-complete) QBF problem to Δ satisfiability in polynomial time. We deal with Boolean quantification by quantifying over the input table $\mathcal{B} = \{0, 1\}$. For example, the formula $\forall x \exists y (x \vee \neg y)$ becomes $\neg \exists \boxed{\langle \sigma x : \neg \exists \boxed{\langle \sigma y : x = 1 \vee y = 0 : \mathcal{B} \rangle} : \mathcal{B} \rangle}$. \square

Complexity analysis of unrestricted Δ beyond Theorems 5 and 6 is not within the scope of this chapter, and has mostly theoretical significance. In practice, query size is orders of magnitude smaller than data size. Conversely, it is meaningful to study *data complexity* [90], i.e., complexity where only the amount of data varies. Instead of assuming a query of constant size, we provide a stronger result by limiting the number of tables that can participate in a cross product. (We also limit nested quantifiers, because the latter can simulate cross products.) We define below the rank function that

characterizes this number.

$$\begin{aligned}
 \text{rank}(\{r_1, \dots, r_n\}) &= 1 \\
 \text{rank}(\langle \sigma x : F : D \rangle) &= \text{rank}(F) + \text{rank}(D) \\
 \text{rank}(D_1 \times D_2) &= \text{rank}(D_1) + \text{rank}(D_2) \\
 \text{rank}(D_1 \cup D_2) &= \max(\text{rank}(D_1), \text{rank}(D_2)) \\
 \text{rank}(T_1 \leq T_2) &= 0 \\
 \text{rank}(\exists D) &= \text{rank}(D) \\
 \text{rank}(\neg F) &= \text{rank}(F) \\
 \text{rank}(F_1 \vee F_2) &= \max(\text{rank}(F_1), \text{rank}(F_2))
 \end{aligned} \tag{9.1}$$

Definition 37 ($k\text{-}\Delta$). For any natural number k , $k\text{-}\Delta$ is the set of formulas $\{F \mid F \in \Delta \text{ and } \text{rank}(F) \leq k\}$.

Theorem 7. For any natural number k , $k\text{-}\Delta$ is NP-complete.

Proof Sketch. $k\text{-}\Delta$ is NP-hard, because any QFLIA formula can be reduced to a $0\text{-}\Delta$ formula in polynomial time ($0\text{-}\Delta \subseteq k\text{-}\Delta$). We obtain membership in NP from the reduction defined by Figure 9.3, which produces polynomially-sized QFLIA formulas. \square

Given the class of formulas $k\text{-}\Delta$ for some k , the reduction produces QFLIA formulas of size $O(n^{k+1})$, where n is the input size. While the reduction is polynomial (since k is fixed), it may not be practical even for $k = 2$, given that datasets of millions of entries are common. Conversely, we propose restrictions that yield a lazy solving architecture.

9.3 The Existential Fragment of Δ

We proceed to study the *existential fragment* of Δ , which we denote by $\exists\Delta$.

Definition 38 ($\exists\Delta$). A Δ formula belongs to $\exists\Delta$ if the \exists operator always appears below an even number of negations, i.e., \exists only appears with positive polarity.

Chapter 9. Data

The motivation for studying $\exists\Delta$ is as follows. Universal quantification pushes for an approach similar to quantifier instantiation, *e.g.*, Example 14 (which is not in $\exists\Delta$) inherently requires instantiating a constraint for every element in $\text{portfolio} \times \text{portfolio}$. This can be done incrementally by applying techniques that are standard in Automated Reasoning. In contrast, we are not aware of techniques that would be a good match for the kind of existential quantification that arises in Δ . Therefore, the rest of this chapter focuses on $\exists\Delta$.

Formulas in $\exists\Delta$ can be transformed into formulas in a convenient intermediate logic without cross products, selections, or unions. We rephrase \exists in terms of a new membership operator. Each formula of the form $\exists D$ is viewed as $x \in D$, where \in has the obvious semantics and x is a properly shaped row comprised of fresh integer variables. We refer to rows like x that serve as witnesses for \exists as *witness rows*. (Witness rows are analogous to Skolem constants.) The next step is to translate membership in arbitrary table expressions to membership in input tables. $(x, y) \in D \times E$ becomes $x \in D \wedge y \in E$, while $x \in D \cup E$ becomes $x \in D \vee x \in E$. Finally, $x \in \langle \sigma y : F : D \rangle$ becomes $F[y/x] \wedge x \in D$. We eliminate all cross products, selections, and unions by repeated application of the above transformations.

Example 15. The tables of Figures 9.1a and 9.1b can be easily encoded as Δ input tables of schemas $\text{int} * (\text{int} * \text{int})$ and $\text{int} * \text{int}$. Let small capitalization be represented by the constant 0. Consider the following constraint:

$$\exists \left\langle \sigma x : \begin{array}{l} \text{left}(\text{left}(x)) = \text{left}(\text{right}(x)) \wedge \\ \text{left}(\text{right}(\text{left}(x))) = 0 \wedge \\ \text{right}(\text{right}(x)) \geq 150 \\ : \boxed{\text{stocks}} \times \boxed{\text{quotes}} \end{array} \right\rangle$$

The constraint asserts the existence of some tuple

$$((x_1, (x_2, x_3)), (x_4, x_5)) \in \text{stocks} \times \text{quotes}$$

that satisfies

$$\Phi = [x_1 = x_4 \wedge x_2 = 0 \wedge x_5 \geq 150].$$

(We have eliminated the accessors left and right.) Equivalently, we can assert that

$$(x_1, (x_2, x_3)) \in \text{stocks} \wedge (x_4, x_5) \in \text{quotes} \wedge \Phi.$$

The procedure we outlined produces a *decomposed* formula consisting of a QFLIA part and *membership constraints*. We proceed to define these notions formally.

Definition 39 ((Conditional) Membership Constraint). *A membership constraint is a constraint of the form*

$$(x_1, \dots, x_k) \in \{(y_{1,1}, \dots, y_{1,k}), \dots, (y_{l,1}, \dots, y_{l,k})\} \quad (9.2)$$

for positive integers k and l and variable symbols $x_i, y_{j,i}$. A constraint of the form $b = 1 \Rightarrow m$, where b is a variable symbol and m is a membership constraint, is called a *conditional membership constraint*.

A membership constraint may hold conditionally, either because it arises from an \exists -atom that appears under propositional structure (and therefore holds conditionally), or because of a disjunction introduced by the union operator. We use conditions of the form $b = 1$ because ILP necessitates $[0, 1]$ -bounded integer variables in place of Boolean variables. Implication in the opposite direction is never needed, since \exists always appears with positive polarity (as per Definition 38).

Membership constraints do not contain arbitrary arithmetic expressions, but only variable symbols. Variable abstraction (similar to Example 5) eliminates richer expressions. While variable abstraction allows for compositional reasoning and helps with theoretical analysis, a limited fragment of arithmetic in membership constraints yields more efficient implementation. Part of our discussion involves tables that contain integer constants and terms of the form $v + c$, where v is a variable symbol and c is an integer constant. (Everything we present is easy to generalize for such terms.) For convenience, we flatten out rows constructed using the pair constructor of

Figure 9.2, and instead deal with k -tuples of integers. This is only a matter of presentation and has no impact on the algorithms.

Definition 40. A decomposed formula is a conjunction $F \wedge M$, where (a) F is a QFLIA formula and (b) M is a conjunction of possibly conditional membership constraints.

Theorem 8. $\exists\Delta$ satisfiability is NP-complete.

Proof. $\exists\Delta$ satisfiability is NP-hard, because $\exists\Delta$ is at least as powerful as QFLIA. $\exists\Delta$ satisfiability is in NP, because we can reduce $\exists\Delta$ to QFLIA in polynomial time. The reduction first produces a formula in decomposed form (Definition 40). Equation 9.2 is equivalent to $\bigvee_{j=1,\dots,l} \bigwedge_{i=1,\dots,k} x_i = y_{j,i}$; therefore, the membership operator can be eliminated. The result is a formula in QFLIA. \square

The polynomial size of the reduction relies on the fact that Δ does not allow tables to be named and referenced from multiple places, *i.e.*, table expressions are not DAG-shaped. Despite the polynomial reduction, a lazy scheme remains relevant. The reason is that QFLIA solvers are not meant for long disjunctions that essentially encode database tables.

9.4 BC(T) for Δ

The decomposed form of Definition 40 is particularly suited for a scheme that combines separate procedures for QFLIA and table membership. Given that the QFLIA part can be encoded as a conjunction of integer linear constraints (Chapter 7), it becomes possible to solve instances in decomposed form (and by extension $\exists\Delta$ instances) by instantiating the BC(T) framework. A B&C-based solver deals with integer linear constraints, and exchanges information with a procedure that checks membership in finite sets. Since database queries typically have simple propositional structure, we do not expect encoding the latter with linear constraints to be a bottleneck.

The membership procedure is confronted with a conjunction of membership constraints (Definition 39). Dealing with conditional constraints is

essentially a matter of Boolean search, similar to the search implemented by the transition rules \mathcal{B} -Propagate⁺, \mathcal{B} -Propagate⁻, and \mathcal{B} -Branch for non-equational interface definitions (Definition 16). The membership procedure needs to understand equality atoms, equality being a primitive. (Our setting is standard first-order logic with equality.) In particular, the procedure keeps track of truth assignments to the equalities in:

$$\{x_i = y_{j,i} \mid j \in [1, l], i \in [1, k]\} \quad (9.3)$$

The symbols x_i and $y_{j,i}$ have the same meaning as in Definition 39. In the presence of multiple membership constraints, the union of sets, like in Equation 9.3, is relevant. Given that membership constraints can be checked in isolation, our discussion proceeds with a single constraint. The variables x_i and $y_{j,i}$ also appear in integer linear constraints. It simplifies our design to assume that all of them appear in linear constraints, even if they are unconstrained there. Just like we do for the case of stably-infinite theories (Chapter 5), we can use the mechanism of difference constraints (Definition 25) for notifying the procedure managing data about atoms like the ones in Equation 9.3. Given truth values for these atoms, we check that a membership constraint is satisfied by simply traversing the table and looking for a tuple that is column-wise equal to the witness row. The constraint is violated if for every $j \in [1, l]$, there exists some $i \in [1, k]$ such that $x_i \neq y_{j,i}$, *i.e.*, there is no candidate tuple.

The arithmetic and membership parts share variables. It is vital that we systematically explore the space of (dis)equalities between these variables. This exchange of information resembles the non-deterministic Nelson-Oppen scheme (NO) for combining decision procedures (Fact 1 and Chapter 5). We demonstrate that NO can accommodate membership constraints.

Lemma 19 (Nelson-Oppen with Propositional Structure). *Let T_i be a stably-infinite Σ_i -theory, for $i = 1, 2$, and let $\Sigma_1 \cap \Sigma_2 = \emptyset$. Also, let Φ_i be a quantifier-free Σ_i -formula. $\Phi_1 \wedge \Phi_2$ is $(T_1 \cup T_2)$ -satisfiable iff there exists an equivalence relation E of the variables V shared by Φ_1 and Φ_2 such that $\{\Phi_i\} \cup \alpha(V, E)$ is T_i -satisfiable, for $i = 1, 2$.*

Proof.

(\Rightarrow):

If $\Phi_1 \wedge \Phi_2$ is $(T_1 \cup T_2)$ -satisfiable, there exists a $(\Sigma_1 \cup \Sigma_2)$ -interpretation L that satisfies it. The way L interprets the variables in V gives rise to an equivalence relation E over V such that L satisfies $T_i \cup \{\Phi_i\} \cup \alpha(V, E)$, $i = 1, 2$.

(\Leftarrow):

If there exists an equivalence relation E over V such that $\{\Phi_i\} \cup \alpha(V, E)$ is T_i -satisfiable, then there exists a Σ_i -interpretation L_i that T_i -satisfies Φ_i , $i = 1, 2$. Let

$$\begin{aligned} \Gamma_i = & \{t \mid t \text{ is an atom in } \Phi_i, L_i \models t\} \cup \\ & \{\neg t \mid t \text{ is an atom in } \Phi_i, L_i \models \neg t\}, \quad i = 1, 2. \end{aligned}$$

$\Gamma_i \cup \alpha(V, E)$ is T_i -satisfiable, $i = 1, 2$. By Fact 1, $\Gamma_1 \cup \Gamma_2$ is $(T_1 \cup T_2)$ -satisfiable. But $\Gamma_1 \cup \Gamma_2 \models \Phi_1 \wedge \Phi_2$. Therefore, $\Phi_1 \wedge \Phi_2$ is $(T_1 \cup T_2)$ -satisfiable.

□

Lemma 20 (Nelson-Oppen for Membership Constraints). *Let T be a stably-infinite Σ -theory. Also, let Γ be a conjunction of Σ -literals, and M be a conjunction of possibly negated membership constraints. $\Gamma \cup M$ is T -satisfiable iff there exists an equivalence relation E of the variables V shared by Γ and M such that $\Gamma \cup \alpha(V, E)$ is T -satisfiable and $M \cup \alpha(V, E)$ is satisfiable.*

Proof. Membership constraints can be viewed as disjunctions of conjunctions (Proof of Theorem 8) in which no function, predicate, and constant symbols appear, *i.e.*, in the empty signature. The theory pertaining the membership constraints is the empty theory (\emptyset), since no axioms are needed. \emptyset is trivially stably-infinite. Our proof obligation follows by applying Lemma 19 with $T_1 = T$ and $T_2 = \emptyset$. □

Note that Lemma 20 allows negated membership constraints. While the latter do not pose algorithmic difficulties, our discussion is limited to the

positive occurrences needed for $\exists\Delta$. The statement of Lemma 20 is structurally similar to that of Fact 1, with membership constraints replacing the constraints of some participating stably-infinite theory. It follows that a membership procedure can participate in NO as a black box, much like a theory solver, even though we have not formalized membership constraints by means of a theory. We can thus combine a form of set reasoning with any stably-infinite theory, *e.g.*, with \mathcal{Z} .

$\text{BC}(T)$, by instantiating the NO scheme, guarantees completeness for MP Modulo T , where T is a stably-infinite theory (Chapter 5). We established that membership can be used much like a stably-infinite theory. All that is needed for completeness is a membership procedure capable of checking consistency of its constraints conjoined with a given arrangement (that contains all literals of Equation 9.3). As we have seen, this operation is simple and involves no arithmetic. In pursuit of efficiency, we proceed to describe branching and propagation techniques based on table contents. Meaningful branching and propagation involve the integer bounds of variables, *i.e.*, necessitate limited arithmetic reasoning on the membership side.

9.4.1 Propagation

B&C-based ILP solvers keep track of variable lower and upper bounds, and heavily rely on bounds propagation algorithms. We describe how to enhance such propagation to exploit the structure of membership constraints.

We denote by $\text{lb}(v)$ and $\text{ub}(v)$ the current lower and upper bounds on variable v . $\text{lb}(v)$ (respectively $\text{ub}(v)$) is either an integer constant, or $-\infty$ (resp. $+\infty$) if no bound is known. We use the notation $\text{lb}'(v)$ and $\text{ub}'(v)$ for bounds on v that the membership procedure infers. (The prime symbol indicates “next state.”) We proceed with a membership constraint as per Definition 39. Let $x = (x_1, \dots, x_k)$; similarly, we denote by y_j the tuple $(y_{j,1}, \dots, y_{j,k})$. Let $\text{match}(x, y_j)$ be true if and only if for all $i \in [1, k]$, the

sets $[\text{lb}(x_i), \text{ub}(x_i)]$ and $[\text{lb}(y_{j,i}), \text{ub}(y_{j,i})]$ intersect.

$$\text{lb}'(x_i) = \max(\text{lb}(x_i), \min\{\text{lb}(y_{j,i}) \mid j \in [1, l], \text{match}(x, y_j)\}) \quad (9.4)$$

$$\text{ub}'(x_i) = \min(\text{ub}(x_i), \max\{\text{ub}(y_{j,i}) \mid j \in [1, l], \text{match}(x, y_j)\}) \quad (9.5)$$

We over-approximate the values of the variables x_i by considering all candidate entries (inner min and max). The outer max and min guarantee that we do not weaken bounds. If there exists exactly one value j such that $\text{match}(x, y_j)$, it is sound to deduce the equalities $x_i = y_{j,i}$, for all $i \in [1, k]$. If there is no candidate entry, inconsistency is reported.

Example 16 (Interleaved Propagation). *Consider the decomposed formula $x = y \wedge (x, y) \in \{(1, 2), (2, 4), (3, 6), (4, 8)\}$. The formula corresponds to a query over concrete tuples that any DBMS can evaluate in linear time. It is thus vital that our techniques yield acceptable performance. Equations 9.4 and 9.5 bound x to*

$$[\min\{1, 2, 3, 4\}, \max\{1, 2, 3, 4\}] = [1, 4]$$

and y to

$$[\min\{2, 4, 6, 8\}, \max\{2, 4, 6, 8\}] = [2, 8].$$

Given the equality $x = y$, ILP propagation deduces that $x, y \in [2, 4]$, since $[2, 4]$ is the intersection of permissible ranges for x and y . The membership procedure detects that match now only holds for $(2, 4)$, and fixes x to 2 and y to 4. The ILP solver in turn deduces unsatisfiability, since $x = y$ is violated. No branching was needed. Encoding the formula in QFLIA would hide its structure, leading to search. The example generalizes to other lengths and bounded symbolic data.

9.4.2 Branching and Arrangement Search

It follows from Lemma 20 that a branching strategy which exhaustively explores all possible arrangements of the shared variables guarantees completeness. To achieve better performance, we have to branch with the tabular structure of databases in mind, without overlooking symbolic data.

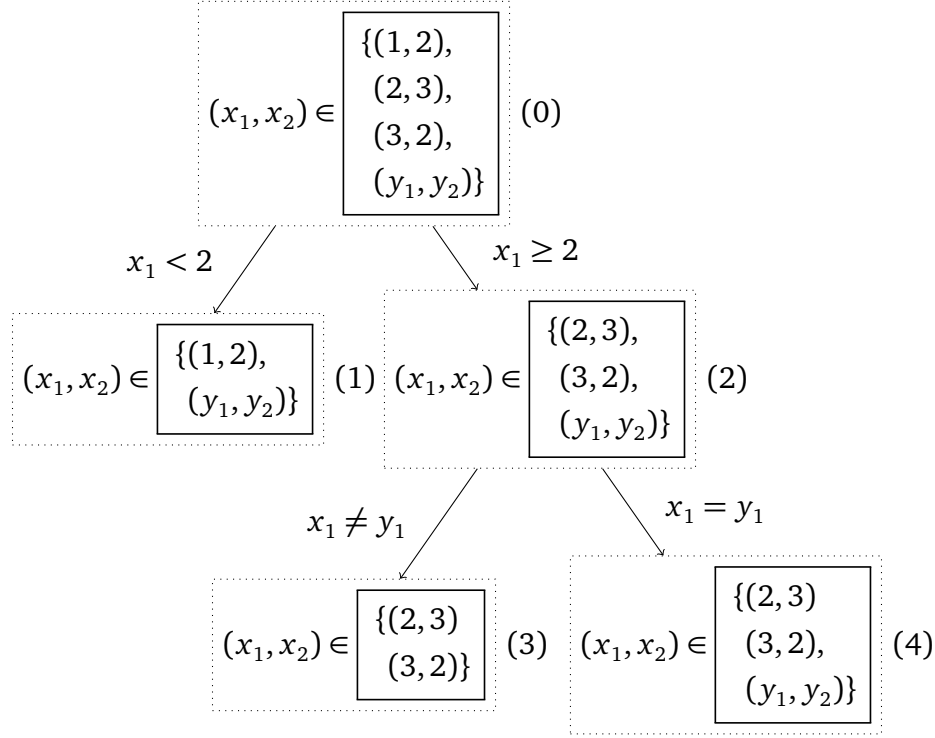


Figure 9.4: Data-Driven Branching

Figure 9.4 provides an example. The root node (Node 0) describes a single membership constraint, which we assume to be part of a larger decomposed formula. We maintain integer constants in the table, instead of performing variable abstraction which would introduce auxiliary variables for them. According to Equation 9.3, the membership procedure needs truth assignments for the equalities in $\{x_1 = 1, x_1 = 2, x_1 = 3, x_1 = y_1, x_2 = 2, x_2 = 3, x_2 = y_2\}$. It would not be wise for the search strategy to overlook that this set originates from a table containing numbers, and treat the set members as if they were atomic propositions unrelated to each other.

In our example, branching on the condition $x_1 < 2$ produces two sub-problems. Node 1 shows only the tuples that still apply under the condition $x_1 < 2$, *i.e.*, the ones that still satisfy the predicate match; similarly for Node 2. $x_1 < 2$ is a choice informed by the tabular structure. Since 2 as the value of the first column is close to the “middle” of the table, branching on $x_1 < 2$ rules out approximately half of the candidates. (y_1, y_2) is present in

both subproblems (Nodes 1 and 2). Branching based on constant bounds is therefore not enough, for we possibly have to deal with symbolic tuples. Figure 9.4 demonstrates further branching on $x_1 = y_1$ to determine whether (y_1, y_2) is a suitable witness for the membership constraint.

The example demonstrates the dual nature of the search strategy needed. The problem naturally pushes towards branch-and-bound (which is a restriction of B&C), *e.g.*, branching on $x_1 < 2$ is meaningful. It remains necessary to also branch on equalities between shared variables (*e.g.*, $x_1 = y_1$), just like in any practical implementation of NO. (To be precise, in ILP we would have two separate nodes for $x_1 < y_1$ and $x_1 > y_1$ in place of $x_1 \neq y_1$.) Implementing NO with B&C enables both kinds of branching.

Branching is organically tied to propagation. Initially (Node 0), assuming no previously known bounds for x_1 , the table contents only allow us to bound x_1 to the range $[\min(\text{lb}(y_1), 1), \max(\text{ub}(y_1), 3)]$; if y_1 is unbounded, x_1 remains unbounded. The decisions $x_1 \geq 2$ and $x_1 \neq y_1$ (*i.e.*, Node 3) tighten x_1 to $[2, 3]$. We also obtain the range $[2, 3]$ for x_2 , *i.e.*, branching on some column potentially leads to propagation across other columns.

9.4.3 Discussion

The analysis of this section indicates that Δ formulas can be decomposed in such a way that a procedure for table lookup assumes part of the workload. $\text{BC}(T)$ is particularly suited for implementing such a combination. $\text{BC}(T)$ can easily accommodate data-aware propagation (Section 9.4.1) and branching (Section 9.4.2). Our techniques would be harder to implement within a $\text{DPLL}(T)$ -style solver [77], given that the toplevel search of $\text{DPLL}(T)$ is over the Booleans (and not the integers). A $\text{DPLL}(T)$ -based implementation of our techniques would essentially require integrating branch-and-bound in $\text{DPLL}(T)$, which is beyond the scope of our work.

The table lookup procedure can be thought of as a small database engine within the solver. The employed database engine can be an actual DBMS, storing the concrete part of tables and possibly bounds on symbolic fields. A DBMS would provide multiple opportunities for improvements. Equations 9.4 and 9.5 essentially describe database aggregation, and thus

provide a starting point for the kinds of queries that apply. DBMS queries can be over multiple tables at a time, and can involve conditions other than bounds. As a matter of fact, the match predicate of Equations 9.4 and 9.5 can be strengthened with any condition on the data that follows from the formula (e.g., $x = y$ in Example 16), thus computing tighter bounds. Different kinds of database optimizations apply, e.g., materializing queries for better incremental behavior and smarter indexing based on user input.

$\exists\Delta$ (and its decomposed form) formally characterizes a relevant class of problems that can be solved by a compositional scheme which employs a database engine. Our scheme may actually apply to a superset of $\exists\Delta$.

9.5 Applications and Experiments

We have implemented support for databases on top of our lnez constraint solver. The data-enabled version of lnez supports existential database constraints by means of the $BC(T)$ -based combination described in Section 9.4, but also universal quantification by eager instantiation. We have produced a collection of lnez input files that have the structure we expect in applications. Our benchmark suite is publicly available.¹ We provide a brief overview of the application areas that inspire our benchmarks.

9.5.1 How-To Analysis

Research in reverse data management [66] proposes ways of obtaining the desired results out of a database query. We outline this class of problems through an example, which gives rise to some of our benchmarks.

Example 17 (`emp_join.ml`). *The management of a company is surprised to find out that (according to the corporate database) there is no employee younger than 30 whose yearly income exceeds \$60000. Why not is not obvious, since income is a complicated function of multiple quantities including a base salary, benefits based on age, employee level (junior, middle, or senior), and bonuses.*

¹<http://www.ccs.neu.edu/home/vpap/fmcad-2014.html>

The management consults the database administrator on how to [67] ameliorate the seeming injustice. Together, they explore bonuses that would allow young employees to exceed the \$60000 limit. This amounts to synthesizing tuples for the table of bonuses. An alternative is to adjust various parameters in the income computation, i.e., to modify the query instead of the data [88]. This can be done by replacing constants with variables, and letting the solver come up with suitable values.

9.5.2 Test-Case Generation

Test case generation is relevant for databases [91]. A family of benchmarks in our collection demonstrate test data generation by concretizing tables initially containing symbolic data.

Example 18 (`emp_keys.ml`). *The problem involves two tables, named `incomes` and `employees`. `incomes` has an ID column constrained to reference existing entries in `employees`, i.e., there is a foreign key constraint. `incomes` contains thousands of tuples with symbolic IDs. A satisfying assignment corresponds to a generated database that meets the foreign key constraint, thus serving as meaningful test input.*

9.5.3 Scientific Applications

Studying big datasets is a key aspect of scientific research in fields ranging from ornithology [85] to astronomy [40]. To demonstrate the applicability of our techniques, we provide benchmarks inspired by queries that ornithologists perform.

Example 19 (`birds_box.ml`). *An ornithologist wants to see a rare species in person, but has not decided on a good location. She has access to a database of observations. Each observation describes a bird and the geographic coordinates where it was seen. An area can be described as a symbolic rectangle*

$$B = [\text{longitude}_{\min}, \text{longitude}_{\max}] \times [\text{latitude}_{\min}, \text{latitude}_{\max}].$$

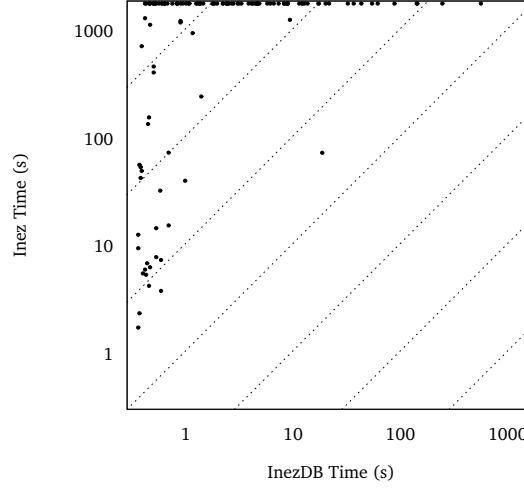


Figure 9.5: InezDB vs. Inez (Database Instances)

Our techniques allow the ornithologist to simply ask for n observations of the species of interest that lie in B . The query effectively concretizes B .

9.5.4 Experiments

We compare InezDB against an Inez frontend that solves Δ formulas by eagerly translating them to QFLIA, via the encoding of Theorem 8. Inez in turn solves QFLIA formulas by reducing them to constraints that SCIP understands. These constraints are not strictly ILP, since we utilize specialized constraint handlers (Chapter 7). We refer to this configuration simply as Inez, since the only addition to Inez is a new frontend. We also produce SMT-LIB versions of our QFLIA formulas, and run them against Z3.

We provide 8 benchmark generators that allow different modes of operation (e.g., some of them are able to produce both satisfiable and unsatisfiable benchmarks), and are able to output benchmarks with different table sizes. Our input table sizes range from 60 tuples to 640000 tuples. In total, our parameters give rise to 134 benchmarks. We run all three solvers with a timeout of 1800 seconds and a memory limit of 12GB. Figures 9.5 and 9.6 visualize our experiments. Inez solves 7 satisfiable and 34 unsatisfi-

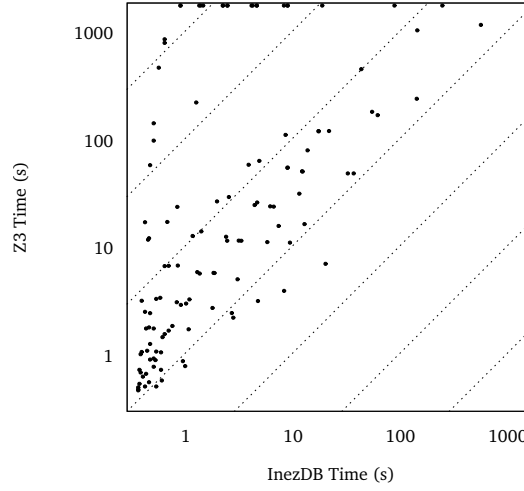


Figure 9.6: InezDB vs. Z3 (Database Instances)

able benchmarks. InezDB solves 44 satisfiable and 80 unsatisfiable benchmarks. Finally, Z3 solves 39 satisfiable and 66 unsatisfiable benchmarks. Among the failures for Inez (respectively, Z3), 20 (respectively, 8) are due to the memory limit. InezDB runs out of memory only once. If we turn off the memory limits, the total numbers of failures don't change much.

Figure 9.5 indicates that InezDB outperforms Inez by a significant margin. This margin can be attributed to two factors. First, InezDB exploits the structure of database problems (e.g., for branching and propagation), while Inez has no knowledge of this structure. Second, our reduction to QFLIA (in the case of Inez) produces patterns that SCIP is not optimized for, since the latter is designed for MILP and not for QFLIA.

Figure 9.6 compares Inez against a leading solver for QFLIA (Z3), and thus characterizes the tool's performance in absolute terms. There is a cluster of 47 benchmarks for which InezDB is 2-8 times faster than Z3. (Note that the scale is logarithmic.) InezDB is at least 8 times faster for 23 of the benchmarks that both tools solve, and solves many benchmarks for which Z3 times out. All failures for InezDB are failures for Z3. Z3 outperforms InezDB for only 9 benchmarks, only 2 of which take InezDB more than 5 seconds to solve. In light of these experiments, our techniques do not just

allow *lnezDB* to outperform *lnez*, but also to cover the gap between *lnez* and *Z3* and surpass *Z3*.

We conclude the evaluation by pointing out that there is significant room for improvement in *lnezDB*. For example, *lnezDB* can benefit from better preprocessing and more sophisticated branching. *lnezDB* can also be improved by adopting database techniques (as we outlined in Section 9.4), or by integrating a DBMS. Our promising experimental results even without such optimizations suggest that our techniques provide a viable design for data-enabled reasoning tools.

9.6 Related Work

“Table constraints” [57, 36], as studied in Constraint Programming, resemble our membership constraints. Such tables are not meant as database tables. Our work differs in significant ways, *e.g.*, our setup allows symbolic table contents. Also, the algorithms presented for table constraints rely on table contents from small domains (*i.e.*, not the reals or the integers). This aligns with the overall emphasis of Constraint Programming, but conflicts with our intended applications.

Veanes et al. describe the Qex technique and tool that uses *Z3* to generate tests for SQL queries [91]. Qex essentially encodes the relational operators via axioms, which are later instantiated via E-matching [71]. E-matching is a generic scheme that is not optimized in any way for database problems. Qex is geared towards relatively small tables that suffice as test cases, while our target applications involve bigger tables.

Other approaches tackle constraints arising in database applications with off-the-shelf generic solvers (via eager reductions). Notably, Khalek et al. use Alloy [52], while Meliou and Suciu use MILP [67]. In neither of these approaches does the core of the solver exploit the structure of database instances, *e.g.*, for branching or propagation.

Conclusions and Future Work

We introduced the Mathematical Programming Modulo Theories (MPMT) constraint solving framework. MPMT tackles problems that consist of linear constraints, along with background theory constraints. We discussed MPMT from both theoretical and practical viewpoints. We presented the Branch and Cut Modulo T (BC(T)) framework for MPMT, and discussed classes of constraints that BC(T) tackles. Our work can be extended across multiple dimensions.

We focused on a particular variant of Mathematical Programming (MP), namely Integer Linear Programming. It would be possible and useful to extend the framework to other kinds of MP, starting from Mixed-Integer Linear Programming (MILP). MILP would be a straightforward generalization, given that the underlying MP technology already supports real variables. Solvable classes of non-linear programming would be possible to support, as long as the solvers involved follow a B&B-like high-level strategy. This dissertation would provide almost no help for extending non-B&B solvers. That would be a radical divergence from the high-level design of BC(T).

It would be interesting to attempt a clean-room implementation of BC(T). The implementation of Inez is ultimately constrained by the design of the underlying MILP solver (SCIP), whose goals are other than theory integration. This approach can potentially produce a solver more competitive with SMT solvers on SMT-like problems. A new implementation of BC(T) can improve the handling of difference constraints, which are crucial for combining procedures in the style of Nelson-Oppen. Furthermore, we can combine exact and inexact (floating-point) operations in a way that provides soundness guarantees, while achieving good performance.

Mathematical Programming Modulo Theories

We anticipate interesting connections between MPMT and the sophisticated techniques employed in Mathematical Programming, *e.g.*, domain-specific cut generation, column generation (Branch-and-Price), and decomposition. We finally anticipate opportunities for cross-pollination between MPMT and paradigms other than SMT and MP, *e.g.*, constraint programming.

Bibliography

- [1] CPLEX. See <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>.
- [2] Gurobi. See <http://www.gurobi.com>.
- [3] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.
- [4] E. Balas, S. Ceria, G. Cornuejols, and N. Natraj. Gomory Cuts Revisited. *Operations Research Letters*, 19:1–9, 1996.
- [5] T. Ball and S. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *POPL*, 2002.
- [6] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2012.
- [7] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *CAV*, 2002.
- [8] C. Barrett, P. Fontaine, A. Stump, and C. Tinelli. The SMT-LIB Standard Version 2.5. 2010.
- [9] G. Belvaux and L. Wolsey. bc-prod: A Specialized Branch-and-Cut System for Lot-Sizing Problems. *Management Science*, 46(5):724–738, 2000.
- [10] F. Besson. On Using an Inexact Floating-Point LP Solver for Deciding Linear Arithmetic in an SMT Solver. In *SMT*, 2010.

- [11] J. Blazewicz, M. Dror, and J. Weglarz. Mathematical Programming Formulations for Machine Scheduling: A Survey. *European Journal of Operational Research*, 51(3):283–300, 1991.
- [12] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An Incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In *TACAS*, 2005.
- [13] R. H. Byrd, A. J. Goldman, and M. Heller. Recognizing Unbounded Integer Programs. *Operations Research*, 35(1):140–142, 1987.
- [14] S. Ceria, C. Cordier, H. Marchand, and L. Wolsey. Cutting Planes for Integer Programs with General Integer Variables. *Mathematical Programming*, 81(2):201–214, 1998.
- [15] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, 2nd edition, 1998.
- [16] A. Cimatti, A. Franzen, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *TACAS*, 2010.
- [17] B. A. Cipra. The Best of the 20th Century: Editors Name Top 10 Algorithms. *SIAM News*, 33(4), 2000.
- [18] G. Cornuejols. Valid Inequalities for Mixed Integer Linear Programs. *Mathematical Programming*, 112(1):3–44, 2008.
- [19] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.
- [20] G. B. Dantzig. Linear Programming. In *History of Mathematical Programming: A Collection of Personal Reminiscences*. Elsevier, 1991.
- [21] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1998. 11th Printing.

BIBLIOGRAPHY

- [22] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *CACM*, 5(7):394–397, 1962.
- [23] C. de la Vallee-Poussin. Sur la methode de l’approximation minimum. *Annales de la Societe de Bruxelles*, 35(2), 1910-1.
- [24] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
- [25] L. de Moura and D. Jovanovic. A Model-Constructing Satisfiability Calculus. In *VMCAI*, 2013.
- [26] L. de Moura and H. Ruess. Lemmas on Demand for Satisfiability Solvers. In *SAT*, 2002.
- [27] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, 2006.
- [28] B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical report, SRI International, 2006.
- [29] B. Dutertre and L. de Moura. The Yices SMT Solver, 2006. Tool paper available at <http://yices.csl.sri.com/tool-paper.pdf>.
- [30] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer, 2nd edition, 1994.
- [31] G. Faure, R. Nieuwenhuis, A. Oliveras, and E. Rodriguez-Carbonell. SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In *SAT*, 2008.
- [32] J. Fourier. Solution d’une question particuliere du calcul des inegalites, 1826.
- [33] R. Fukasawa, H. Longo, J. Lysgaard, M. P. de Aragao, M. Reis, E. Uchoa, and R. Werneck. Robust Branch-and-Cut-and-Price for the Capacitated Vehicle Routing Problem. *Mathematical Programming*, 106(3):491–511, 2006.

- [34] V. Ganesh and D. Dill. A Decision Procedure for Bit-vectors and Arrays. In *CAV*, 2007.
- [35] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *CAV*, 2004.
- [36] I. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data Structures for Generalised Arc Consistency for Extensional Constraints. In *AAAI*, 2007.
- [37] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.
- [38] R. E. Gomory. Outline of an Algorithm for Integer Solutions to Linear Programs. *Bulletin of the AMS*, 64:275–278, 1958.
- [39] R. E. Gomory. An Algorithm for the Mixed Integer Problem. Technical report, The Rand Corporation, 1960.
- [40] J. Gray, A. Szalay, A. Thakar, P. Kunszt, C. Stoughton, D. Slutz, and J. vandenBerg. Data Mining the SDSS SkyServer Database. *arXiv preprint cs/0202014*, 2002.
- [41] A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, 2012.
- [42] M. Grotschel and O. Holland. Solution of Large-Scale Symmetric Travelling Salesman Problems. *Mathematical Programming*, 51:141–202, 1991.
- [43] O. Gunluk. A Branch-and-Cut Algorithm for Capacitated Network Design Problems. *Mathematical Programming*, 86(1):17–39, 1999.
- [44] C. Hang, P. Manolios, and V. Papavasileiou. Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints. In *CAV*, 2011.
- [45] K. Hoffman and M. Padberg. Solving Airline Crew Scheduling Problems by Branch-and-Cut. *Management Science*, 39(6):657–682, 1993.

BIBLIOGRAPHY

- [46] L. Johnson and D. Montgomery. *Operations Research in Production Planning, Scheduling, and Inventory Control*. John Wiley & Sons, Inc., 1974.
- [47] D. Jovanovic, C. Barrett, and L. D. Moura. The Design and Implementation of the Model Constructing Satisfiability Calculus. In *FMCAD*, 2013.
- [48] D. Jovanovic and L. de Moura. Cutting to the Chase: Solving Linear Integer Arithmetic. In *CADE*, 2011.
- [49] L. Kantorovich. Mathematical Methods in the Organization and Planning of Production. Technical report, Leningrad State University, 1939.
- [50] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. In *STOC*, 1984.
- [51] L. Khachiyan. Polynomial Algorithms in Linear Programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.
- [52] S. A. Khalek, B. Elkarablieh, Y. Laleye, and S. Khurshid. Query-Aware Test Generation Using a Relational Constraint Solver. In *ASE*, 2008.
- [53] T. King, C. Barrett, and C. Tinelli. Leveraging Linear and Mixed Integer Programming for SMT. In *FMCAD*, 2014.
- [54] A. Kuehlmann, K. McMillan, and M. Sagiv. Generalizing DPLL to Richer Logics. In *CAV*, 2009.
- [55] S. Lahiri and M. Musuvathi. An Efficient Decision Procedure for UTVPI Constraints. In *FroCoS*, 2005.
- [56] S. Lahiri and S. Seshia. The UCLID Decision Procedure. In *CAV*, 2004.
- [57] C. Lecoutre and R. Szymanek. Generalized Arc Consistency for Positive Table Constraints. In *CP*, 2006.

- [58] Z. Manna and C. Zarba. Combining Decision Procedures. In *10th Anniversary Colloquium of UNU/IIST*, 2002.
- [59] P. Manolios and V. Papavasileiou. Virtual Integration of Cyber-Physical Systems by Verification. In *AVICPS*, 2010.
- [60] P. Manolios and V. Papavasileiou. Pseudo-Boolean Solving by Incremental Translation to SAT. In *FMCAD*, 2011.
- [61] P. Manolios, S. K. Srinivasan, and D. Vroon. BAT: The Bit-Level Analysis Tool. In *CAV*, 2007.
- [62] P. Manolios, G. Subramanian, and D. Vroon. Automating Component-Based System Assembly. In *ISSTA*, 2007.
- [63] V. Manquinho and J. Marques-Silva. Satisfiability-Based Algorithms for Boolean Optimization. *Annals of Mathematics and Artificial Intelligence*, 40(3-4):353–372, 2004.
- [64] D. McAllester. Truth Maintenance. In *AAAI*, 1990.
- [65] J. McCarthy. Towards a Mathematical Science of Computation. In *Congress IFIP-62*, 1962.
- [66] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse Data Management. In *VLDB*, 2011.
- [67] A. Meliou and D. Suciu. Tiresias: The Database Oracle for How-To Queries. In *SIGMOD*, 2012.
- [68] J. E. Mitchell. Branch-and-Cut Algorithms for Combinatorial Optimization Problems. In *Handbook of Applied Optimization*, pages 223–233. Oxford University Press, 2000.
- [69] D. Monniaux. On Using Floating-Point Computations to Help an Exact Linear Arithmetic Decision Procedure. In *CAV*, 2009.
- [70] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, 2001.

BIBLIOGRAPHY

- [71] L. D. Moura and N. Bjorner. Efficient E-matching for SMT Solvers. In *CADE-21*, 2007.
- [72] G. Nelson and D. Oppen. Fast Decision Algorithms Based on Union and Find. In *FOCS*, 1977.
- [73] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *TOPLAS*, 1:245–257, 1979.
- [74] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *CAV*, 2005.
- [75] R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *SAT*, 2006.
- [76] R. Nieuwenhuis and A. Oliveras. Fast Congruence Closure and Extensions. *Information and Computation*, 205:557–580, 2007.
- [77] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.
- [78] K. Roos. Interior-Point Methods for Linear Optimization. Technical report, TU Delft, 2000.
- [79] Scott Cotton. Natural Domain SMT: A Preliminary Assessment. In *FORMATS*, 2010.
- [80] R. Sebastiani and S. Tomasi. Optimization in SMT with LA(Q) Cost Functions. In *IJCAR*, 2012.
- [81] S. A. Seshia and R. E. Bryant. Deciding Quantifier-Free Presburger Formulas Using Parameterized Solution Bounds. In *LICS*, 2004.
- [82] R. Shostak. A Practical Decision Procedure for Arithmetic with Function Symbols. *JACM*, 26(2):351–360, 1979.
- [83] J. M. Silva and K. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, 1996.

- [84] V. Sofronie-Stokkermans. Hierarchic Reasoning in Local Theory Extensions. In *CADE*, 2005.
- [85] D. Sorokina, R. Caruana, M. Riedewald, W. Hochachka, and S. Kelling. Detecting and Interpreting Variable Interactions in Observational Ornithology Data. In *DDDM*, pages 64–69. IEEE, 2009.
- [86] C. Tinelli and M. Harandi. A New Correctness Proof of the Nelson-Oppen Combination Procedure. In *FroCoS*, 1996.
- [87] C. Tinelli and C. Zarba. Combining Non-Stably Infinite Theories. *Journal of Automated Reasoning*, 34(3):209–238, 2005.
- [88] Q. T. Tran and C.-Y. Chan. How to ConQueR Why-Not Questions. In *SIGMOD*, 2010.
- [89] G. Tseitin. On the Complexity of Proof in Propositional Calculus. *Zapiski Nauchnykh Seminarov POMI*, 8:234–259, 1968.
- [90] M. Vardi. The Complexity of Relational Query Languages. In *STOC*, 1982.
- [91] M. Veanes, N. Tillmann, and P. de Halleux. Qex: Symbolic SQL Query Explorer. In *LPAR-16*, 2010.
- [92] Wayne L. Winston. *Operations Research: Applications and Algorithms*. Duxbury Press, 3rd edition, 1994.