DECISION PROCEDURES FOR PROGRAM SYNTHESIS AND VERIFICATION

Ruzica Piskac

Thèse n. 5220 2011
présenté le 17 Octobre 2011
à la Faculté Informatique et Communications
Laboratoire d'analyse et de raisonnement automatisés (LARA)
programme doctoral en informatique, communications et information
École Polytechnique Fédérale de Lausanne

pour l'obtention du grade de Docteur ès Sciences
par

Ruzica Piskac

acceptée sur proposition du jury:

Prof Arjen Lenstra, président du jury
Prof Viktor Kuncak, directeur de thèse
Dr Nikolaj Bjørner, rapporteur
Prof Rupak Majumdar, rapporteur
Prof Martin Odersky, rapporteur

Lausanne, EPFL, 2011

Za mamu i tatu…

Acknowledgments

First and foremost, I would like to thank my advisor Viktor Kuncak for his support, without which this dissertation would not have been possible. I very much enjoyed our discussions, the challenges he put in front of me, even our little arguments (I still prefer "where" over "let"). Viktor is a person of great knowledge and a brilliant mind, he cares a lot about his students, and I consider myself lucky and privileged for having the chance to work with him.

I am grateful to Arjen Lenstra, Nikolaj Bjørner, Rupak Majumdar, and Martin Odersky, for agreeing to serve on my thesis committee. I thank them for their time, I am aware of their busy schedules, and I thank them for their feedback on my thesis.

I would like to additionally thank Nikolaj Bjørner for being my mentor during a summer internship in 2008 at Microsoft Research, Redmond. Working with Nikolaj in those three summer months helped me to significantly increase my knowledge of the SMT world.

Special thanks go to Tim King. Tim agreed to proofread this thesis and found a number of places where further clarifications were needed. I am thankful to Thomas Wies and Philippe Suter for the discussions on the structure and some of the technical subtleties of my thesis. I am further in debt to Utkarsh Upadhyay for his help on submitting and printing this thesis in time.

During the past four years I had the pleasure to work with a number of amazing people around the world. In particular, I want to thank all my colleagues with whom I co-authored a paper: Nikolaj Bjørner, Tihomir Gvero, Viktor Kuncak, Mikaël Mayer, Leonardo de Moura, Philippe Suter, Thomas Wies, and Kuat Yessenov.

One of the reasons why my PhD studies were so enjoyable is because of the great atmosphere in the LARA group. I thank my colleagues Eva Darulová, Tihomir Gvero, Hossein Hojjat, Swen Jacobs, Giuliano Losa, Andrej Spielmann, and Philippe Suter for creating such a motivating working environment. I thank Danielle Chamberlain, Yvette Gallay, and Fabien Salvi for their help with bureaucratic and technical tasks.

My road towards a PhD degree was long and winding. On this path I had an honor to work in various groups, on various topics. I would therefore like to thank to all the great people with whom I was working: I believe that the experience that I gained in each of those groups helped me in completing this thesis. In particular, I am thankful to Dieter Fensel and his group at the University of Innsbruck, late Harald Ganzinger and his group at the Max-Planck Institute for Computer science in Saarbrücken, Tome Anticic and his group at the Institute Rudjer Boskovic in Zagreb, and Robert Manger and his group at the University of Zagreb.

At the beginning of my studies at EPFL, I was a co-organizer of a workshop at the International

Semantic Web Conference in Busan, Korea. I would like to thank the other co-organizer, Frank van Harmelen, for this unforgettable experience. I am also grateful to Manfred Jeusfeld for introducing me to the world of on-line publishing.
I am grateful for the inspiring discussions with the outstanding people that I met during my PhD studies. I thank Shankar Natarajan for being the kindest host. My first scientific visit was to SRI International, and Shankar also made it one of the most memorable. His understanding and kind attitude encouraged me to speak freely about my research. I met Rupak Majumdar at EPFL, just when I was starting my PhD. I thank him for introducing me to the world of semilinear sets, and without knowing it, helping me to solve some of the research questions that I was working on. During my PhD studies I considered Sabine Süsstrunk as a mentor and role model. I am grateful to her for finding time for me, for her wise advices and her support. Finally, I want to thank my family and my friends for their support and their love through the years. I thank my parents, Bernarda and Josip, and my brother, Tomislav, for being there for me, even when we were physically apart. A very special thanks go to Thomas Wies. This final gratitude also extends to the families Piskac, Slunjski, Puklavec, Wies and Wagner.

Lausanne, November 23, 2011 Ruzica Piskac

Preface

Recent progress in verification technology has started to improve our ability to automatically check the correctness of software systems. Yet an important challenge remains ahead of us: can we verify deeper program properties, beyond the absence of low-level errors? Such deep analysis requires reasoning that is specific to the software application being verified, so analysis techniques that target particular classes of properties stop being sufficient. To introduce application-specific reasoning into verification, it is promising to consider methods that integrate software verification with software development. Among key methodologies supporting this direction is compositional verification based on pre-conditions and post-conditions.

To support such compositional techniques, the ability to reason about logical formulas is essential, because formulas become part of the program itself. Dealing with formulas also arises when modeling program semantics, both in approaches based on verification-condition generation and in more automated approaches, such as counterexample-guided predicate abstraction. Modeling programs with formulas allows us to achieve precision (for example, path-sensitivity), scalability (because we can use efficient algorithms to explore exponentially many program paths), and tool reuse (because we can develop tools largely independently from the programming language semantics).

Due to the need to reason about formulas, theorem proving technology becomes an indispensable component of these verification approaches. The leading automated theorem provers for these tasks are satisfiability modulo theory (SMT) solvers. They are among the most remarkable reasoning tools developed, combining great expressive power with the ability to handle megabyte-sized formulas. The key to this power is specialized reasoning based on decision procedures and a principled technique to combine them, preserving soundness and completeness. Interestingly, despite the great progress in the tools developed, the foundations of decision procedures and their combination techniques has been evolving relatively slowly since the introduction of the approach by Nelson and Oppen in the 1970ies. As a result, verification tools must model many constructs using quantifiers, often resulting in unpredictable reasoning and the inability to generate counterexamples.

This thesis introduces new theorem proving algorithms, qualitatively extending the reach of existing technology. Much of these results is formulated in terms of decision procedures for classes of constraints of interest, and immediately leads to more predictable verification. Moreover, the thesis shows that, in many cases, these algorithms can also be used to directly construct software fragments through a notion of synthesis procedure. In several cases, the

class of logical constraints has not been identified as decidable before, whereas in others the class was known to be decidable but the known algorithms were exponentially worse than the optimal ones introduced in this thesis.

The thesis contributes a number of results in **decision procedures**. Chapter 2 presents decision procedures for multiset constraints, whereas Chapter 3 presents initial implementation results. A number of extensions of this logic are the focus of Chapter 4. This includes relations, resulting in a proper extension of an important description logic. Another direction are collections with fractional membership, which open up the possibility of applications in dealing with uncertainty. A parametrized family of decision procedures for program termination are the subject of Chapter 5. A new, more widely applicable method for composing decision procedures of individual logical theories into a decision procedure for the combined (union) theory is the subject of Chapter 6. We can broadly classify these results into two categories: 1) new decidable logical fragments, and 2) new methods to combine decidable fragments. Both kinds of results are a crucial starting point for the development of SMT solvers.

An example of contributions of the first kind are the results on decision procedures for multiset constraints with the cardinality operator. This is a very natural fragment, because multisets describe data structures while preserving element multiplicity. They are needed to precisely describe, e.g., a sorting algorithm, or a precise external behavior of a data structure such as red-black tree. The cardinality operator on multisets is a natural measure to describe many data structure invariants. Yet, the very basic questions about the logics of multisets with cardinality constraints were unanswered before this work. The work shows that quantified multisets constraints with cardinalities are undecidable (in contrast to quantified set constraints with cardinalities). The most technically involved were the results on the complexity of the quantifier-free constraints. These constraints previously had a decision procedure giving NEXPTIME upper bound on the decision problem. Despite the exponential lower bound on the sizes of explicit models, the thesis shows the decision problem to be in NP (using, among others, a technique for proving the existence of sparse solutions of compactly represented integer linear programming problems with bounded coefficients, and bounds on the sizes of vectors in semilinear set representations).

An example of contributions of the second kind (complete combination methods), is the result on combining theories that share set operations, presented in Chapter 6. The state-of-the art method implemented in modern SMT solvers goes back to Nelson and Oppen, and could be explained as (demand-driven) reduction of theories to the pure theory of equality. This method works only if the equality is the sole shared symbol between the combined theories. Chapter 6 shows how to reduce theories not to equality but to a richer logic, set algebra with linear arithmetic (BAPA). Among the remarkable observations is that such reduction is possible in many cases, including such important decidable logics as monadic second-order logic of trees, as well as the two-variable logic with counting. As a consequence of this reduction approach, we obtain decidability of a fairly rich specification language, supporting quantifier-free combinations of two-variable logic and WSkS, as well as many other useful logics that can express constraints on sets.

In addition to developing the algorithms and the foundations for constraint solving and

synthesis, the thesis describes the implemention of the underlying automated reasoning and synthesis tools. Chapter 3 shows the design and the development of the multiset reasoner MUNCH; this is the only theorem proving tool capable of effectively handling multisets in the presence of cardinality constraints.

Furthermore, the thesis makes substantial contributions to **software synthesis**. First explored long ago by some of the greatest pioneers of computer science, software synthesis has received increased attention in recent years thanks to new algorithms, new applications, and better understanding of the boundary between tractable and intractable synthesis problems. For synthesis of software it is particularly important to support specifications over domains such as integers and collections of objects, because software implementations almost invariably rely on such unbounded data types, in contrast to finite-state reactive systems. Manna and Waldinger have already identified theorem proving technology as the main bottleneck for future progress of software synthesis. Since then, software verification and advanced type systems such as refinement types have experienced a revolution. This is in part thanks to increasingly efficient SMT solvers. The idea behind the complete functional synthesis approach, described in Chapter 7, is to extend the use of this successful SMT technology to software synthesis. For this to happen, we must generalize decision procedures (which give yes/no answers for formula satisfiability) into algorithms that produce actual satisfying assignments. Moreover, if we wish our synthesized code to be efficient, we need procedures that accept parametrized input and produce an entire family of parametrized solutions, in the form of an efficiently computable function. Chapter 7 presents such procedure for the logic that combines integer linear arithmetic and the set algebra with cardinality operators (we call this logic Boolean Algebra with Presburger Arithmetic, or BAPA for short). The resulting system rewrites the given specification of program fragments into a solved form that, given inputs, compute the outputs that are guaranteed to satisfy the specification.

In addition to a sequence of results centered around decision procedures, the thesis presents a glimpse of a fresh research direction: synthesis of code that combines method calls from existing libraries. This synthesis approach, described in Chapter 8, is driven by type constraints of an expressive type system, including generic types. Types are an abstraction of code which is essential for the synthesized code to compile. The presented results suggest that synthesis based on generic types, even though undecidable in general, has a practical solution that can be deployed in the context of integrated development environments. A crucial part of this solution is an approach to guide the search process using weights derived from a corpus of code. The resulting approach promises development environments that deliver qualitatively more than what we can expect today. Like the previous chapters, it presents algorithmic advances with the potential to greatly improve programmer productivity in developing reliable software systems.

Lausanne, November 2011

Viktor Kuncak
Assistant Professor, EPFL
PhD MIT, 2007

Abstract

Decision procedures are widely used in software development and verification. The goal of this dissertation is to increase the scope of properties that can be verified using decision procedures. To achieve this goal, we identify three improvements over the state of the art in decision procedures, and their use in software reliability tools.

First, we observe that developing *new decision procedures* increases the range of properties and programs that are amenable to automated verification. In this thesis, we are particularly interested in the verification of container data structures. Existing verification tools use set abstractions to reason about the contents of data structures. However, set abstraction loses any information about duplicate occurrences of elements in a container. We therefore propose a new logic for reasoning about multisets with cardinality constraints. This logic subsumes reasoning about sets and enables reasoning about duplicate elements in containers. Cardinality constraints are useful for reasoning about the number of elements stored in a data structure. Based on an extension of linear arithmetic (which we call LIA^*), we describe a decision procedure for the logic of multisets with cardinalities. By investigating properties of LIA^*, we prove that the satisfiability of multisets with cardinality constraints is an NP-complete problem.

Second, we notice that verification conditions expressing properties of data structures often can be decomposed into several well-understood logics. If the signatures of the component theories are not disjoint (i.e., they share more than equality) then it is often unclear whether such a reduction is possible, even if individual decision procedures for all component theories are known to exist. We investigate how to combine non-disjoint theories that share set symbols and operators. We *state and prove a new combination theorem* for such theories. Our theorem states that the combination is possible if each component theory can be reduced to the common theory, the theory of sets with cardinality constraints. We prove that many theories satisfy this property. The resulting combined logic enables reasoning about complex properties of data structure implementations that could not be expressed in any previously known decidable logic.

Finally, we identify *new applications of decision procedures* in software reliability tools. We describe how a model-producing decision procedure can be generalized into a predictable and complete synthesis procedure. Given a specification, a synthesis procedure is an algorithm that outputs the code that meets this specification. We demonstrate this approach in detail for the concrete case of linear integer arithmetic. We further develop an orthogonal approach to use decision procedure for program synthesis: we show how to reconstruct code snippets that

satisfy given type constraints from a proof of unsatisfiability that was computed by a theorem prover. The programmer then interactively selects the desired code snippet from a choice of code snippets generated by the synthesis engine.
Together, our results provide the foundations of sound and predictable verification and synthesis tools for integer arithmetic and container data structures.

Keywords: decision procedure, program verification, software synthesis, combination procedure, automated reasoner for set and multisets, linear integer arithmetic, data structures

Zusammenfassung

Entscheidungsverfahren haben vielfältige Anwendungen in der Software-Entwicklung und Verifikation. Das Ziel dieser Dissertation ist es, die Bandbreite der Eigenschaften zu erhöhen, die mit Hilfe von Entscheidungsverfahren verifiziert werden können. Um dieses Ziel zu erreichen, entwickeln wir drei Neuerungen in der Erforschung von Entscheidungsverfahren und deren Anwendung in Werkzeugen, die die Zuverlässigkeit von Software sicherstellen.

Als erstes stellen wir fest, dass die Entwicklung *neuer Entscheidungsverfahren* die Bandbreite der Eigenschaften und Programme erweitert, die der automatischen Verifikation zugänglich sind. In dieser Dissertation befassen wir uns speziell mit der Verifikation von Container-Datenstrukturen. Existierende Verifikationswerkzeuge verwenden Mengenabstraktionen, um über den Inhalt von Datenstrukturen logische Schlußfolgerungen ziehen zu können. Jedoch verlieren Mengenabstraktionen jegliche Information über Mehrfachvorkommen von Elementen in einem Container. Daher schlagen wir eine neue Logik für die automatische Deduktion von Aussagen über Multimengen mit Kardinalitätsprädikaten vor. Diese Logik subsumiert logisches Schlußfolgern über Mengen, aber ermöglicht darüber hinaus die präzise Behandlung von mehrfach vorkommenden Elementen in einem Container. Kardinalitätsprädikate dienen dazu, Aussagen über die Anzahl der Elemente beweisen zu können, die in einer Datenstruktur gespeichert sind. Basierend auf einer Erweiterung der linearen Arithmetik (die wir LIA^* nennen), beschreiben wir ein Entscheidungsverfahren für die Logik der Multimengen mit Kardinalitätsprädikaten. Durch eine genaue Untersuchung der Eigenschaften von LIA^* gelingt es uns zu beweisen, dass Erfüllbarkeit von Multimengen mit Kardinalitätsprädikaten ein NP-vollständiges Problem ist.

Zweitens beobachten wir, dass Verifikationsbedingungen, die Eigenschaften von Datenstrukturen ausdrücken, sich oft in Teileigenschaften aufspalten lassen, die in wohlverstandene Logiken fallen. Wenn die Signaturen dieser Komponententheorien nicht disjunkt sind (d.h. sie teilen mehr als nur das Gleichheitssymbol), dann ist es häufig unklar, ob sich eine solche Reduktion ausnutzen läßt, um die automatische Deduktion von Aussagen in der kombinierten Theorie zu ermöglichen, selbst dann, wenn die individuellen Komponententheorien alle entscheidbar sind. Wir untersuchen den Fall der nicht disjunkten Kombination von Theorien die Mengen und Mengenoperationen teilen. Wir formulieren und beweisen ein neues Kombinationstheorem für solche Theorien. Unser Theorem besagt, dass die Kombination der Theorien möglich ist, wenn sich jede Komponenttheorie auf eine gemeinsame Theorie reduzieren läßt, nämlich die Theorie der Mengen mit Kardinalitätsprädikaten. Wir zeigen, dass viele Theorien diese Eigenschaft erfüllen. Die resultierende kombinierte Logik ermöglicht es

komplexe Aussagen über Datenstrukturen automatisch zu beweisen, die sich in keiner vorher bekannten entscheidbaren Logik ausdrücken ließen.
Schließlich identifizieren wir *neue Anwendungsfelder von Entscheidungsverfahren* in Software-Verifikationswerkzeugen. Wir beschreiben, wie sich modellerzeugende Entscheidungsverfahren zu vollständigen Syntheseverfahren generalisieren lassen. Der Programmierer stellt eine formale Spezifikation zur Verfügung und unser Synthesewerkzeug berechnet den Code, der diese Spezifikation erfüllt. Wir demonstrieren diesen Ansatz im Detail für den konkreten Fall der linearen, ganzzahligen Arithmetik. Des Weiteren entwickeln wir einen orthogonalen Ansatz zur Verwendung von Entscheidungsverfahren in der Programmsynthese: wir zeigen wie sich Code-Schnipsel, die bestimmte Typvorgaben erfüllen, aus einem Unerfüllbarkeitsbeweis rekonstruieren lassen, der von einem Theorembeweiser erbracht wurde. Der Programmierer kann dann interaktiv den gewünschten Code-Schnipsel aus einer vom Synthesewerkzeug generierten Auswahl von Code-Schnipseln wählen.
Zusammengenommen bilden unsere Resultate die Grundlage für fehlerfreie und berechenbare Verifikations- und Syntheseverfahren für ganzzahlige Arithmetik und Container-Datenstrukturen.

Schlagworte: Entscheidungsverfahren, Programmverifikation, Software-Synthese, Kombinationsverfahren, automatische Beweiser für Mengen und Multimengen, sowie lineare ganzzahlige Arithmetik, Datenstrukturen

Résumé

Les procédures de décision sont couramment utilisées dans le cadre du développement et de la vérification logicielle. La présente thèse s'intéresse à la question de l'extension du domaine d'application des procédures de décision, à la vérification de propriétés complexes ainsi qu'à la synthèse de programmes. Nous présentons trois améliorations par rapport à l'état de l'art en matière de procédures de décision et faisons la démonstration de leur applicabilité en tant qu'outils pour améliorer la fiabilité logicielle.

Premièrement, nous observons que le développement de *nouvelles procédures de décision* entraîne une augmentation du nombre de programmes et de propriétés qui peuvent être traités par des techniques de vérification automatisée. Dans cette thèse, nous nous intéressons plus particulièrement à la vérification de structures de données représentant des collections. Les outils existant utilisent typiquement des ensembles comme représentation abstraite des éléments d'une collection. Cependant, une abstraction basée sur des ensembles ne permet pas de tenir compte d'éléments dupliqués. Pour palier à ce problème, nous proposons une nouvelle logique pour raisonner sur les multiensembles et leur cardinalité. Cette logique est suffisamment expressive pour raisonner sur les ensembles, mais permet en plus d'encoder correctement la multiplicité des éléments d'une collection. Les contraintes sur la cardinalité des multiensembles sont utiles pour représenter le nombre d'éléments stockés dans les structures de données. En nous basant sur une extension de l'arithmétique linéaire –que nous appelons LIA*– nous présentons une procédure de décision pour notre logique des multiensembles avec l'opérateur de cardinalité. Une étude des propriétés de LIA* nous permet de prouver que le problème de la satisfiabilité d'une formule dans cette logique est NP-complet.

Deuxièmement, nous remarquons que les formules, souvent complexes, exprimant des conditions de vérification pour des propriétés de structures de données peuvent souvent être décomposées en plusieurs formules dans diverses théories logiques pour lesquelles une procédure de décision est disponible. Si les signatures de ces théories ne sont pas disjointes (c'est-à-dire, si elles partagent d'autres symboles que l'égalité), alors la question de la satisfiabilité de la combinaison des théories est souvent ouverte. Dans cette thèse, nous étudions la combinaison de théories non-disjointes qui partagent des symboles et des opérateurs se rapportant aux ensembles. Nous *formulons et prouvons un nouveau théorème pour la combinaison* de telles théories. Notre théorème montre que la combinaison est possible si chacune des théories peut individuellement être réduite à une théorie commune, dans notre cas, une logique d'ensembles avec l'opérateur de cardinalité. Nous prouvons également que de nom-

breuses théories connues remplissent cette condition. La logique résultant de la combinaison de ces théories nous permet de raisonner au sujet propriétés complexes sur les structures de données qui ne pouvaient être exprimées auparavant dans aucune logique décidable connue. Finalement, nous identifions de *nouvelles applications des procédures de décision* pour la fiabilité logicielle. Nous montrons comment une procédure de décision capable de produire des modèles peut être transformée en une procédure de synthèse, prévisible et complète. Une procédure de synthèse est un algorithme qui, à partir d'une formule exprimant une relation entre des variables d'entrée et de sortie, produit du code qui calcule les variables de sortie en fonction de celles d'entrée de telle sorte que la relation soit satisfaite. Nous présentons cette approche en détails pour le cas de l'arithmétique linéaire des nombres entiers. Nous présentons également une approche orthogonale de l'utilisation des procédures de décision dans le cadre de la synthèse de programmes : nous montrons comment construire des fragments de code qui satisfont certaines contraintes de typage en partant d'une preuve d'insatisfiabilité produite par un prouveur de théorèmes automatisé. Le programmeur peut ensuite choisir interactivement parmi une liste de fragments de code produits par notre outil de synthèse.
Pris ensemble, nos résultats posent les fondations pour le développement d'outils robustes et fiables pour la vérification et la synthèse, pour l'arithmétique des nombres entiers et les structures de données.

Mots-clés : procédures de décision, vérification logicielle, synthèse de programmes, procédures de combinaison, raisonnement automatisé pour ensembles et multiensembles, arithmétique linéaire

Contents

List of Figures

1 Introduction

Software correctness research has a long history. In the last decade we have witnessed a significant progress in verification technologies, leading to tools that are applied to large software applications of industrial relevance. The following are examples of tools that are successfully used for verification and finding bugs in software:

- ARMC [Podelski and Rybalchenko(2007a)] is a model checker based on abstraction refinement and Constraint Logic Programmingm and is used for reachability and termination properties. It was applied to verify hardware design, as well as model checking real-time properties of the European train control system.
- BLAST [Beyer et al.(2007)Beyer, Henzinger, Jhala, and Majumdar] is a software model checker for C programs, based on lazy abstraction. Using Blast, memory-safety properties of various benchmarks of C programs were proved. In addition, it was also used as a testing framework: tests were derived from counter-examples.
- CBMC [Clarke et al.(2004)Clarke, Kroening, and Lerda] is a bounded model checker for C and C++ programs. CBMC was used in various applications to increase software reliability: by appplying CBMC, it is possible to detect the cause of errors and the worst-case number of loop iterations. It was also used to verify Linux Device Drivers, as well as detect security-relevant bugs in WIN32 binaries.
- HAVOC [Lahiri et al.(2009)Lahiri, Qadeer, Galeotti, Voung, and Wies] is a tool for specifying and verifying properties of programs written in C. It is based on a logic that allows reasoning about lists and arrays. HAVOC was used to check properties in the Microsoft Windows operating system on more than 300 thousand lines of code and 1500 procedures.
- Jahob [Zee et al.(2008)Zee, Kuncak, and Rinard] is a verification system for programs written in a subset of Java. Jahob is primarily used for verification of the container data structures since it relies on decision procedures that can reason automatically about collections [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard, Kuncak et al.(2005)Kuncak,

Nguyen, and Rinard]. Jahob relies on the tool Bohne [Podelski and Wies(2010)] to automatically infer loop invariants for heap-manipulating programs.

- SLAM [Ball et al.(2004)Ball, Cook, Levin, and Rajamani] is a tool for static verification of device drivers. Verification of C programs using the SLAM toolkit [Ball et al.(2001)Ball, Majumdar, Millstein, and Rajamani] has recently won the Most Influential PLDI Paper award. SLAM is based on predicate abstraction and it is integrated in the Static Driver Verifier Research Platform, which is shipped with the Windows Driver Kit.

- SLAyer [Berdine et al.(2011)Berdine, Cook, and Ishtiaq] is a tool for proving memory-safety properties about linked data structures, based on separation logic [Reynolds(2002)]. It has been applied to industrial software components of up to 100,000 lines of code.

- Spec# [Barnett et al.(2004b)Barnett, Leino, and Schulte] is is an extension of the C# programming language. It is integrated into the Microsoft Visual Studio development environment. It checks method contracts in the form of pre- and postconditions at run-time and emits warnings. Spec# is also a static program verifier—it generates verification conditions from a program and then invokes a solver to verify them.

A common aspect of these tools is that they encode the verification task into the problem of reasoning about logical formulas. They translate both the desired properties and the program semantics into a formula F, using techniques such as verification condition generation, symbolic execution, or predicate abstraction. Once a formula F is obtained, there are two questions in which we are usually interested. The first question is whether the formula F is *satisfiable*. That means that we are asking whether there is a model in which F evaluates to true. For example, formula $x \leq y$ is clearly satisfiable, by letting x to be 1 and y to be 2. In contrast, the formula $x \leq y \wedge x + z > y + z$ is never satisfiable—we call it unsatisfiable. The other question that we are interested in is *validity*. A formula is valid if it evaluates to true in every model. Formula $x \leq y$ is not valid, but formula $x \leq y \Rightarrow x + z \leq y + z$ is a valid formula. Validity and satisfiability are related: a formula is valid iff its negation is unsatisfiable.

We believe that the mentioned verification tools are successful in part because they use automated reasoners to automatically answer questions about satisfiability and validity. We call such reasoners *provers*, or *solvers*. If a tool takes a formula in a certain logic, and answers the satisfiability question, we usually call such tool a solver. The current state-of-the-art satisfiability modulo theories (SMT) solvers are specialized for the satisfiability problem for the particular logics (often defined by first-order theories) [de Moura and Bjørner(2008a), Barrett and Tinelli(2007), Bruttomesso et al.(2010)Bruttomesso, Pek, Sharygina, and Tsitovich, Bruttomesso et al.(2008)Bruttomesso, Cimatti, Franzén, Griggio, and Sebastiani]. In addition, these solvers are also efficient in combining the theories, assuming that the requirements of the Nelson-Oppen combination procedure are met. However, solvers face challenges when they need to reason about quantified formulas. The other type of tools are so-called provers. They are mostly optimized to find a proof of unsatisfiability fast. The current provers [Riazanov and Voronkov(2002), Weidenbach et al.(2009)Weidenbach, Dimova, Fietzke, Kumar, Suda, and

Wischnewski, Schulz(2002), Korovin(2009)] are mostly based on resolution [Robinson(1965)], and they are general purpose tools, i.e. not theory-specific. To reason about a certain theory, one needs to add the theory axioms. Provers can also efficiently handle quantifiers.

Solvers are based on *decision procedures*. A decision procedure is an algorithm that takes a formula in a certain logic and then checks whether the formula is satisfiable. Decision procedures are a rich field of study [Bradley and Manna(2007), Kroening and Strichman(2008)]. There are several logics of particular interest for proving software correctness. Standard propositional logic is a logic that does not contain any functions or quantifiers. The formulas in propositional logic are formed from boolean variables and boolean connectives. If we allow quantifiers and functions, we obtain a more expressive logic. However, this logic is undecidable, even if we restrict quantification to range only over first-order variables. In that case the logic is called first-order logic. There are various fragments of first-order logic that are decidable, as for example, linear integer arithmetic. It is a logic of the natural numbers with addition, which was proved to be decidable already in [Presburger(1929)]. In his honor this logic is sometimes also called Presburger arithmetic. The satisfiability problem in many other logics can be reduced to reasoning in Presburger arithmetic. For example, for every formula expressing properties about sets in the presence of the cardinality operator there exists an equisatisfiable Presburger arithmetic formula. Extending linear integer arithmetic with unrestricted multiplication results in an undecidable logic, even for quantifier-free formulas [Matiyasevich(1970)].

Despite the success of the above mentioned tools, there are still certain restrictions that limit their applicability:

1. All these tools reason about an abstraction of the system and therefore simplified properties. There is a trend in more recent tools, like Boogie [Barnett et al.(2005)Barnett, Chang, DeLine, Jacobs, and Leino] or Jahob [Zee et al.(2008)Zee, Kuncak, and Rinard], to reason about more complex properties. However, reasoning about more complex properties requires more complex theories, which are either undecidable, or of a very high complexity, or their decidability is not even known. To tackle that problem, tools like Boogie use an incomplete axiomatization to reason about data structures. Because of this solution, we can see that there is certainly a need for new decision procedures. We believe that having a decision procedure for a problem helps to better understand the problem and it gives a better insight into its structure, even if the decision procedure is of a very high complexity. As shown in Chapter 3, by analysis of the decision procedure and the structure of the input problems, we can develop a more efficient, even if incomplete, technique. In our current experience, this techniques scales better than the complete algorithm based on a decision procedure.

2. When combining several theories, most tools use the Nelson-Oppen combination procedure [Nelson and Oppen(1980)], which has strong restrictions. It requires that the theories are stably-infinite: if a formula is satisfiable, then it also must have an infinite model. In addition, the theories must be disjoint, i.e. their signatures can share only

the equality symbol. In the Nelson-Oppen procedure we cannot combine theories that allow only finite models, as well as theories that share more than only equality. Recent work has shown how to relax the requirement about the stably-infiniteness [Tinelli and Zarba(2003), Fontaine(2009), Jovanovic and Barrett(2010)]. However, signatures still need to be disjoint. This is a problem, for instance, when proving properties about container data structures. As we demonstrate in Chapter 6, for many verification tasks we generate formulas that, after purification, still share the set operators, so we cannot apply the standard procedure. The general problem of combining non-disjoint theories was also studied [Tinelli and Ringeissen(2003)]. However, there is still less research in this direction than in tools based on Nelson-Oppen approach.

3. In many cases, new decision procedures are motivated by program verification problems. When proving code correctness, one usually identifies a theory that is the most suitable for describing the needed properties. If the decidability of this theory is not known, or the existing tools do not scale, then the focus of research moves to developing or improving a decision procedure for the given logic. The resulting algorithm is usually non-trivial. We believe that it is therefore worthwhile considering additional areas where we can leverage these algorithms. In Chapters 7 and 8 we show how a decision procedure can be modified to not only prove formulas, but also output the code to compute the values that satisfy given constraints, or to have the expected type. This results in an approach for synthesis of code based on solvers and provers.

1.1 Contributions

We next summarize the contributions of this dissertation in automated reasoning about collections, combining non-disjoint theories, and software synthesis.

1.1.1 Reasoning about Collections

When reasoning about container data structures that can hold duplicate elements, multisets are the obvious choice of abstraction. In this approach, the need for cardinality constraints naturally arises in order to reason about the number of elements in the data structure. However, before this dissertation has started, the decidability and the complexity of multisets constraints with cardinalities was not known. The contributions of our work on reasoning about the collections are the following:

1. We defined a highly expressive language that allows reasoning about multisets and cardinalities. Because sets are a special case of multisets, this language subsumes languages for reasoning about sets with cardinality constraints.

2. We showed the decidability of this language in [Piskac and Kuncak(2008a)]. We described a reduction to an extension of linear integer arithmetic.

3. We defined and further studied this new extension of linear integer arithmetic (so-called LIA^* logic). In [Piskac and Kuncak(2008c)] we developed an algorithm for checking the satisfiability of formulas that belong to LIA^*. We proved that the satisfiability question for LIA^* is an NP-complete problem.

4. We also gave the answer to the open problem stated by Lugiez [Lugiez(2005)] whether the logic containing quantified multiset constraints with cardinalities is decidable. We proved that adding quantifiers yields undecidability.

5. We developed a tool for reasoning about multisets with cardinality constraints called Munch [Piskac and Kuncak(2010)] and tested it on formulas derived from a software analysis tool. We were able to write simple and more precise specifications using multisets instead of sets.

6. Common to all previously described results is that we did not place any restrictions on the domain of the multisets. However, in practice the domain set (i.e. the set used for populating collections) is often known and fixed. We showed that the reasoning about collections defined over totally ordered sets (specifically, the integers) still remains in the class of the NP-complete problems [Kuncak et al.(2010c)Kuncak, Piskac, and Suter].

7. To support reasoning about further data structures, we introduced an extension of the logic of sets and multisets with the ability to compute direct and inverse relations and function images. We established decidability and complexity bounds for these extended logics in [Yessenov et al.(2010)Yessenov, Piskac, and Kuncak].

8. We developed a new decision procedure for reasoning about multiset orderings [Piskac and Wies(2011)], which are among the most powerful orderings used for proving termination of programs. We considered multiset orderings defined over an arbitrary preodered set. The decidability of this logic was not previously known.

1.1.2 Combining Non-disjoint Theories

It is often the case that only one theory is not expressive enough on its own to accurately express the required verification conditions. For example, to describe the insertion of an element into an imperative linked list data structure, we need transitive closure, unconstrained functions defined on sets, and the cardinality operator. As we argued before, the Nelson-Oppen combination procedure is too restrictive if the theories share more than only the equality as a common operator. In this particular example, after purification, we obtain a conjunction of formulas belonging to logics WS1S, C^2 and BAPA. In addition to the equality, these logics share the set operators as well. In [Wies et al.(2009)Wies, Piskac, and Kuncak] we have presented a new combination technique for theories that share sets. The combination procedure reduces them to a common shared theory, to the BAPA logic [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard]. BAPA is a logic of sets with cardinality constraints. We call such theories BAPA-reducible. We showed that the logics

1. Boolean Algebra with Presburger Arithmetic [Kuncak and Rinard(2007)],
2. weak monadic second-order logic of two successors WS2S [Thatcher and Wright(1968)],
3. two-variable logic with counting C^2 [Pratt-Hartmann(2005)],
4. Bernays-Schönfinkel-Ramsey class [Börger et al.(1997)Börger, Grädel, and Gurevich], and
5. quantifier-free multisets with cardinality constraints [Piskac and Kuncak(2008c), Piskac and Kuncak(2008a)]

all meet the conditions of our combination technique. Consequently, we obtain the decidability of quantifier-free combination of formulas in these logics.

1.1.3 Software Synthesis

Software synthesis aims to generate software satisfying a given specification. Instead of writing the software directly, the programmer provides a specification, from which a synthesis tool then automatically generates code. Consequently, this code is correct by construction and there is no need to verify it. Moreover, this way programmers can also be more productive. The downside of software synthesis is that it is difficult to write complete specifications, maybe even harder than to write the code itself. Therefore, code synthesis should be used as a help for programmers to write code more efficiently rather than as a stand-alone tool.

The use of formal techniques for software synthesis was suggested already earlier [Manna and Waldinger(1980)]. However, not until recently the idea could be efficiently implemented. In the last decade we have witnessed a breakthrough in the research on decision procedures and automated reasoning. Applying our insights from decision procedures, we have developed a Scala plug-in called Comfusy [Kuncak et al.(2010b)Kuncak, Mayer, Piskac, and Suter, Kuncak et al.(2010a)Kuncak, Mayer, Piskac, and Suter]. Given a specification for a code fragment, Comfusy generates the code satisfying it, together with the preconditions required for the existence of the solution.

We have also developed a tool called InSynth [Gvero et al.(2011)Gvero, Kuncak, and Piskac], which generates code fragments based on type constraints. While Comfusy constructs code based on a model for the specification, InSynth derives a proof of unsatisfiability of the constraints. Based on that proof, InSynth generates a code snippet and repeats the process. InSynth is an interactive tool in the sense that outputs several snippets and the user can choose the desired one.

All together, our contributions to software synthesis are the following:

1. We describe an approach for deploying algorithms for synthesis within programming

languages. Given a specification and a separation of variables into output variables and parameters, our procedure constructs

   (a) a program that computes the values of outputs given the values of inputs

   (b) the weakest among the conditions on inputs that guarantees the existence of outputs (the domain of the given relation between inputs and outputs).

2. We describe a methodology to convert decision procedures for a class of formulas into synthesis procedures that can rewrite the corresponding class of expressions into efficient executable code. Most existing procedures based on quantifier elimination are directly amenable to our approach.

3. We describe synthesis procedures for propositional logic, rational arithmetic and linear integer arithmetic. We developed an algorithm that efficiently handles equalities in linear integer arithmetic.

4. We show that the synthesis for integer arithmetic can be extended to the non-linear case where coefficients multiplying output variables are expressions over parameters that are known only at run-time.

5. We also described and implemented a synthesis procedure for Boolean Algebra with Presburger Arithmetic (BAPA), a logic of constraints on sets and their sizes.

6. We developed a tool called InSynth, which is an interactive synthesis tool based on parametrized types, test cases, and weights indicating user preferences. Its algorithmic foundation is a variation of ordered resolution and intuitionistic calculus. We have found InSynth to be fast enough for interactive use and helpful in synthesizing meaningful code fragments.

1.2 Outline of the Dissertation

The rest of this thesis is organized as follows:

Chapter 2 describes a decision procedure for reasoning about multisets with cardinality constraints. It is based on the papers [Piskac and Kuncak(2008a)] and [Piskac and Kuncak(2008c)]. This chapter merges the papers, provides a uniform notation and expends all the main proofs. In addition, it also introduces the Hilbert bases and describes a connection between computation of semilinear sets and a Hilbert basis. In Appendix A we also prove the background theorems about the number of generators of an integer cone.

Chapter 3 describes an implementation of a reasoner for sets and multisets with cardinality constraints. This chapter is based on the tool description presented in [Piskac and Kuncak(2010)].

Chapter 4 describes extensions of the logic defined in Chapter 2. First we describe an extension that leads to a generalized framework, as introduced in [Piskac and Kuncak(2008b)]. In this new framework we can also reason about fractional collections. In this chapter we provide additional examples. The other extension was introduced in [Yessenov et al.(2010)Yessenov, Piskac, and Kuncak]. It extends the logic of Chapter 2 with function symbols. Here we also provide an extended proof for the complexity result.

Chapter 5 is based on [Piskac and Wies(2011)]. It defines multiset orderings which are used in proving termination of programs. We consider multiset orderings defined over an arbitrary pre-ordered set. This way we can prove properties, for instance, of multiset orderings defined over a multiset of trees and the subtree relation. This ordering is not total. This chapter contains additional explanations and more detailed proofs of all main theorems from [Piskac and Wies(2011)].

Chapter 6 contains an extended version of [Wies et al.(2009)Wies, Piskac, and Kuncak]. We describe a new combination procedure for non-disjoint theories. The combination procedure is based on a reduction to the theory of sets with cardinality constraints (BAPA). This chapter contains additional examples and the extended proofs for most of the reduction procedures.

Chapter 7 introduces complete functional synthesis. We describe how to convert a decision procedure into a synthesis procedure. Based on a given specification, the described synthesis procedure always finds a corresponding code. In addition, it also outputs the preconditions needed for a solution to exist. This chapter combines [Kuncak et al.(2010b)Kuncak, Mayer, Piskac, and Suter] and [Kuncak et al.(2010a)Kuncak, Mayer, Piskac, and Suter].

Chapter 8 introduces interactive synthesis of code snippets. The chapter is based on [Gvero et al.(2011)Gvero, Kuncak, and Piskac] as well as recent work under submission. Given an incomplete program and a program point, at which we invoke InSynth, a specification is derived based on type constraints and our tool outputs possible code snippets suitable for that program point. The user then interactively selects the desired snippet. This chapter, in addition to [Gvero et al.(2011)Gvero, Kuncak, and Piskac], contains an algorithmic foundations of the calculus used in InSynth.

Chapter 9 concludes the dissertation and highlights selected future work directions.

2 Decision Procedures for Multisets with Cardinality Constraints

This chapter introduces a language for reasoning about collections with cardinality constraints. Motivated by applications in software verification, we consider the standard operators on collections, such as union, intersection, and difference. In addition, we also consider some operators that are specific for reasoning about multisets, for instance the disjoint union ($\uplus$) operator. We present the formal syntax and semantics (Sec. 2.4). In Section 2.6, we show that the satisfiability problem in this logic is decidable, and we construct an algorithm for answering the satisfiability question. In the process, we generalize integer linear arithmetic by adding a star (integer cone) operator into the language. By analyzing the satisfiability problem for integer linear arithmetic with a star operator, Section 2.7 demonstrates that the problem is in NP, and presents the second algorithm for satisfiability of multiset constraints.

2.1 Motivation

Collections of objects are fundamental and ubiquitous concepts in computer science and mathematics. It is therefore not surprising that they often arise in software analysis and verification, as well as in interactive theorem proving. Moreover, such constraints often contain cardinality bounds on collections. There is an extensive work on decision procedures for reasoning about sets of objects, where also cardinality constraints might appear [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard, Kuncak and Rinard(2007)]. In that work, the authors characterzed the complexity of both quantified and quantifier-free constraints.

In many applications [Bouajjani et al.(2011)Bouajjani, Drăgoi, Enea, and Sighireanu], it is more appropriate to use multisets (bags) rather than sets as a way of representing collections of objects. The content of a data structure is abstracted as a multiset. The cardinality constraints in such abstractions may arise if there is a need to count the number of elements in the data structure. It is therefore a natural problem to consider constraints on multisets along with the cardinality bounds. There is a range of useful operations and relations on multisets, beyond the traditional disjoint union and difference. These operations are all definable using quantifier-free Presburger arithmetic (QFPA) formulas on the number of occurrences of each

element in the multiset. This paper describes such a language that admits reasoning about integers, sets and multisets, supports standard set and multiset operations as well as any QFPA-definable operation on multisets (including the conversion of a multiset into a set), and supports a cardinality operator that counts the total number of elements.

Previously, Zarba [Zarba(2002a)] considered decision procedures for quantifier-free multisets but without the cardinality operator, showing that it reduces to quantifier-free pointwise reasoning. However, the cardinality operator makes such reduction impossible.

Lugiez studied multiset constraints in the context of a more general result on multitree automata [Lugiez(2005)] and proved the decidability of quantified constraints with a weaker form of cardinality operator that counts only distinct elements in a multiset. He also established the decidability results of certain quantifier-free expressible constraints with cardinality operator. Regarding quantified constraints with the general cardinality operator, [Lugiez(2005), Section 3.4] states "the status of the complete logic is still an open problem". In this chapter, we resolve this open problem, showing that the quantified constraints with cardinality are undecidable (Section 2.9). The decidable quantified constraints in [Lugiez(2005)] allow quantifier elimination and the resulting formulas are quantifier-free constraints, which can then be expressed using the decidable constraints in this chapter.

2.2 Definition and Applications of Multisets

Definition of multisets. Multisets are collections of objects where an element can occur several times. They can be seen as "sets with counting". For example, on the set level $\{a, a\} = \{a\}$. However, in the multiset interpretation, they are two different multisets. We represent multisets (*bags*) as well as sets with their characteristic functions. A multiset m is a function $\mathbb{E} \to \mathbb{N}$, where $\mathbb{E}$ is the universe and $\mathbb{N}$ is the set of non-negative integers. The value $m(e)$ is the multiplicity (the number of occurrences) of an element e in a multiset m. We assume that the domain $\mathbb{E}$ is fixed and finite but of unknown size. We represent sets within our formulas as special multisets m for which $m(e) = 0 \vee m(e) = 1$ for all elements $e \in \mathbb{E}$.

Applications of set and multiset constraints. Sets and multisets directly arise in verification conditions for proving properties of programs in languages and paradigms such as SETL [Schwartz(1973)] and Gamma [Banâtre and Métayer(1993), Page 103]. In programming languages such as Java, data abstraction can be used to show that data structures satisfy set specifications, and then techniques based on sets become applicable for verifying data structure clients [Kuncak(2007), Nguyen et al.(2007)Nguyen, David, Qin, and Chin]. To validate properties of programs with lists and data, a common approach is to abstract the content of a data structure as a multiset [Bouajjani et al.(2011)Bouajjani, Drăgoi, Enea, and Sighireanu].

Isabelle [Nipkow et al.(2005)Nipkow, Wenzel, Paulson, and Voelker], an interactive theorem prover, as well as KIV [Balser et al.(2000)Balser, Reif, Schellhorn, Stenzel, and Thums] (Karl-

sruhe Interactive Verifier) and Why [Filliâtre and Marché(2007)], all contain the multisets and sets libraries. The formulas present there may also have the cardinality constraints. A decision procedure for reasoning about multisets with the cardinality constraints can increase the automation within such systems.

Linear integer arithmetic. Linear integer arithmetic plays a vital role in reasoning about multisets. After several transformation steps, a multiset formula is reduced to an equisatisfiable linear integer arithmetic formula. Sometimes we also use the name Presburger arithmetic. A grammar for linear integer arithmetic is given in Figure 2.1.

$$
\begin{array}{lll}
F & ::= & A \mid F \wedge F \mid \neg F \\
A & ::= & T \leq T \mid T = T \\
T & ::= & x \mid c \mid T + T \mid c \cdot T \mid \mathsf{ite}(F, T, T)
\end{array}
$$

x - integer variable; c - integer constant

Figure 2.1: Linear Integer Arithmetic

Linear integer arithmetic admits the addition of variables and the multiplication of a variable by a constant, but does not allow the multiplication of two variables. The satisfiability question in linear integer arithmetic is decidable. Quantified linear integer arithmetic also admits quantifier elimination [Cooper(1972)].

2.3 Introduction to Logic through an Example

In software analysis and verification it is often desirable to abstract the content of mutable and immutable data structures into collections to raise the level of abstraction when reasoning about programs. Abstracting linked structures as sets and relations enables high-level reasoning in verification systems, such as Jahob [Kuncak(2007)]. For collections that may contain duplicates, abstraction using multisets is more precise than abstraction using sets. Our goal is to investigate the decidability of a logic allowing reasoning about multisets with cardinality constraints. Moreover, if decidable, we wish to describe decision procedures that would enable reasoning about such precise abstractions, analogously to the way current decision procedures enable reasoning with set abstraction.

Figure 2.2 contains a Java code that removes an element from a list. To illustrate the role of cardinality operator, we assume the following declaration:

```
public class Node {
    public Object data;
    public Node next;
}
class SinglyLinkedList  {
    private Node first ;
    int size ;
```

```
public void remove(Object d0) {
  Node f = first ;
  if (f.data == d0) {
    Node second = f.next;
    f.next = null;
    first  = second;
  } else {
    Node prev = first ;
    Node current = prev.next;
    while (current.data != d0) {
      prev = current;
      current = current.next;
    }
    Node nxt = current.next;
    prev.next = nxt;
    current.next = null;
  }
  size = size - 1;
}
```

Figure 2.2: Java code that removes an element from a list

}

Data structure implementations often contain integer size fields. With s we denote a data structure size field and with L an abstract multiset field denoting the data structure content. All data structure operations need to preserve the size invariant $s = |L|$. When verifying an insertion of an element into a container, we therefore obtain verification conditions such as $|L|{=}s \wedge |e| = 1 \rightarrow |L \uplus e|{=}s{+}1$. To show that a deletion also preserves the size invariant, we need to additionally annotate the code in Figure 2.2, by adding preconditions and postconditions:

$$\begin{array}{ll} \text{requires} & d_0 \neq null \wedge d_0 \in L \\ \text{ensures} & L = \text{old } L \setminus \{d_0\} \wedge d_0 \in \text{old } L \end{array}$$

Using those annotations, we generate verification conditions such as

$$D \subseteq L \wedge |D| = 1 \rightarrow |L \setminus D| = |L| - 1 \tag{2.1}$$

Here D denotes a multiset containing only the element d_0. In this chapter, we will describe how one can prove such verification conditions. We will use formula (2.1) as an example for illustrating how our decision procedure works.

To capture operations on data structures, it is useful to have not only operators such as a disjoint union $\uplus$ and a set difference, but also an operator that, given multisets m_1 and m_2, produces a multiset m_0 which is the result of removing from m_1 *all* occurrences of elements that occur m_2. We can specify that requirement, by saying that every element that appears in m_2 cannot appear in m_0, while all other element of m_1 stay unchanged in m_0.

This can be expressed by the formula $\forall e.(m_2(e) = 0 \rightarrow m_0(e) = m_1(e)) \wedge (m_2(e) > 0 \rightarrow m_0(e) = 0)$. We call this operator a multiset difference, denoted by $m_0 = m_1 \setminus\setminus m_2$. Our goal is to define a language that will support any such operation definable pointwise by quantifier-free Presburger arithmetic (QFPA) formulas.

We next outline the main ideas of the first decision procedure for a logic that allows such constrains [Piskac and Kuncak(2008a)]. The running example will be the formula (2.1). Here we demonstrate only the basic ideas of the algorithm; Sections 2.4.1 and 2.6 give the detailed description.

To prove validity of (2.1), we show that its negation,

$$D \subseteq L \wedge |D| = 1 \wedge |L \setminus D| \neq |L| - 1 \tag{2.2}$$

is unsatisfiable.

The main idea of the algorithm is to reduce a given formula into an equisatisfiable Presburger arithmetic formula. The reduction steps rely on the pointwise definitions of multiset operators.

First, by introducing a fresh multiset variable Y for $L \setminus D$, we obtain the formula

$$Y = L \setminus D \wedge D \subseteq L \wedge |D| = 1 \wedge |Y| \neq |L| - 1$$

We next perform a similar step and introduce fresh integer variables k_1 and k_2:

$$k_1 = |Y| \wedge k_2 = |L| \wedge Y = L \setminus D \wedge D \subseteq L \wedge 1 = |D| \wedge k_1 \neq k_2 - 1$$

The number of occurrences of an element e in a multiset M is denoted by $M(e)$. The cardinality of a multiset M is the number of all elements that occur in M: $|M| = \sum_{e \in \mathbb{E}} M(e)$, where $\mathbb{E}$ is some base set used for populating multisets. Applying this definition results in:

$$k_1 \neq k_2 - 1 \wedge k_1 = \sum_{e \in \mathbb{E}} Y(e) \wedge k_2 = \sum_{e \in \mathbb{E}} L(e) \wedge 1 = \sum_{e \in \mathbb{E}} D(e) \wedge Y = L \setminus D \wedge D \subseteq L$$

All the sums range over the set $\mathbb{E}$ so we make this formula more compact by using vectors:

$$k_1 \neq k_2 - 1 \wedge (k_1, k_2, 1) = \sum_{e \in \mathbb{E}} (Y(e), L(e), D(e)) \wedge Y = L \setminus D \wedge D \subseteq L \tag{2.3}$$

We next apply the pointwise definition of multiset inclusion: $D \subseteq L$ iff for every element e appearing in D there are at least as many elements in L:

$$D \subseteq L \Leftrightarrow \forall e.D(e) \leq L(e)$$

Finally, we apply the definition of the set difference operator:

$$Y = L \setminus D \Leftrightarrow \forall e.Y(e) = \mathsf{ite}(L(e) \geq D(e), L(e) - D(e), 0)$$

Given a term $t = \mathsf{ite}(F, t_1, t_2)$, if the formula F evaluates to true, then t has value t_1, otherwise t_2. We combine those definitions and obtain the formula:

$$k_1 \neq k_2 - 1 \wedge (k_1, k_2, 1) = \sum_{e \in \mathbb{E}} (Y(e), L(e), D(e)) \wedge \\ \forall e. Y(e) = \mathsf{ite}(L(e) \geq D(e), L(e) - D(e), 0) \wedge D(e) \leq L(e) \quad (2.4)$$

Formula (2.4) is in a normal form that we call a sum normal form. It is a conjunction of three parts: a part containing a Presburger arithmetic formula, a part containing a sum expression and a part of universally quantified Presburger arithmetic formulas.

We can eliminate the multiset indices altogether and reduce the formula to an equisatisfiable formula that belongs to an extension of Presburger arithmetic formula over the integers. In Theorem 2.4 we formally prove the correctness of the reduction and here we just apply the final results. In a sum normal form, multisets occur only in expressions of the form $M(e)$. These expressions denote integer values. Therefore, for every expression $M(e)$ we introduce a fresh integer variable. In the case of formula (2.4) for $L(e)$ we introduce the integer variable l, while for $Y(e)$ and $D(e)$ we use y and d.

The summation in formula (2.4) is unbounded. Let us assume that the set $\mathbb{E}$ has N elements and for every multiset expression, let $m_i = M(e_i)$. Using this new integer notation, the sum can be rewritten as $\exists N \geq 0.(k_1, k_2, 1) = \sum_{i=0}^{N} (y_i, l_i, d_i)$. In addition, all integer variables y_i, l_i and d_i have to satisfy the universally quantified Presburger arithmetic formula in (2.4). Applying these new variables, formula (2.4) reduces to

$$k_1 \neq k_2 - 1 \wedge (k_1, k_2, 1) = \sum_{i=0}^{N} (y_i, l_i, d_i) \wedge \forall i. y_i = \mathsf{ite}(l_i \geq d_i, l_i - d_i, 0) \wedge d_i \leq l_i \quad (2.5)$$

Variables y_i, l_i and d_i all represent the number of occurrences of an element in a multisets, thus they are all implicitly assumed to be non-negative. Using this assumption, the universally quantified part can be further simplified and the formula becomes:

$$k_1 \neq k_2 - 1 \wedge (k_1, k_2, 1) = \sum_{i=0}^{N} (y_i, l_i, d_i) \wedge \forall i. l_i = y_i + d_i$$

We next eliminate the variables l_i entirely from the formula. In this particular example this is easy to do. In fact, it can always be done, independently of how complex is the formula under the universal quantifier. It is possible because all non-negative solutions of a Presburger arithmetic formula have a special form, they are so-called semilinear sets [Ginsburg and Spanier(1966)]. The semilinear sets provide a finite description of the set of solutions of a formula. The semilinear sets are then further used to eliminate the sums, so the final result is again a Presburger arithmetic formula. To go back to our example, after eliminating l_i we obtain:

$$k_1 \neq k_2 - 1 \wedge (k_1, k_2, 1) = \sum_{i=0}^{N} (y_i, y_i + d_i, d_i)$$

Because all variables are non-negative integers, d_i has to be almost all the time zero, except once. Without loss of generality let that be the last value:

$$k_1 \neq k_2 - 1 \wedge (k_1, k_2, 1) = \sum_{i=0}^{N-1} (y_i, y_i, 0) + (y_N, y_N + 1, 1)$$

Let c denote $c = \sum_{i=0}^{N} y_i$. It is an integer value and the only value in the formula that contains a link to unbounded sums (N). Using c we can remove the sum from the formula:

$$k_1 \neq k_2 - 1 \wedge \exists c. k_1 = c \wedge k_2 = c + 1$$

After eliminating c we obtain formula

$$k_1 \neq k_2 - 1 \wedge k_2 = k_1 + 1 \tag{2.6}$$

Because (2.2) and (2.6) are equisatisfiable and (2.6) is unsatisfiable, we conclude that (2.1) is a valid formula.

2.4 Multiset Constraints

Figure 2.3 defines constraints whose satisfiability we study. Our constraints combine multiset expressions and two kinds of QFPA formulas: *outer linear arithmetic formulas*, denoting relationship between top-level integer values in the constraint, and *inner linear arithmetic formulas*, denoting constraints specific to a given index element $e \in \mathbb{E}$. Note that the syntax is not minimal; we subsequently show how many of the constructs are reducible to others.

Formulas (F) are propositional combinations of atomic formulas (A). Atomic formulas can be multiset equality and subset, pointwise linear arithmetic constraint $\forall e.\mathsf{F}^{\mathsf{in}}$, or atomic outer linear arithmetic formulas ($\mathsf{A}^{\mathsf{out}}$). Outer linear arithmetic formulas are equalities and inequalities between outer linear arithmetic terms ($\mathsf{t}^{\mathsf{out}}$), as well as summation constraints of the form $(u_1, \ldots, u_n) = \sum_F (t_1, \ldots, t_n)$, which compute the sum of the vector expression $(t_1, \ldots, t_n)$ over all indices $e \in \mathbb{E}$ that satisfy the formula F. Outer linear arithmetic terms ($\mathsf{t}^{\mathsf{out}}$) are built using standard linear arithmetic operations starting from integer variables (k), cardinality expressions applied to multisets ($|M|$), and integer constants (C). The $\mathsf{ite}(F, t_1, t_2)$ expression is the standard if-then-else construct, whose value is t_1 when F is true and t_2 otherwise. Inner linear arithmetic formulas are linear arithmetic formulas built starting from non-negative integer constants (P) and values $m(e)$ of multiset variables at the current index e. This way inner terms t^{in} are always non-negative. In Section 4 we remove this requirement, and the inner terms can be positive and negative. We will show that the logic is decidable, and of the same complexity (Section 4.5.2). In this chapter, for ease of presentation, we investigate only the positive inner terms. Additionally, this requirement does restrict a construction of the inner formulas. They can contain arbitrary linear arithmetic expressions. As an illustration,

top-level formulas:

$$\begin{array}{rcl} F & ::= & A \mid F \wedge F \mid \neg F \\ A & ::= & M{=}M \mid M \subseteq M \mid \forall e.\mathsf{F}^{\mathsf{in}} \mid \mathsf{A}^{\mathsf{out}} \end{array}$$

outer linear arithmetic formulas:

$$\begin{array}{rcl} \mathsf{F}^{\mathsf{out}} & ::= & \mathsf{A}^{\mathsf{out}} \mid \mathsf{F}^{\mathsf{out}} \wedge \mathsf{F}^{\mathsf{out}} \mid \neg \mathsf{F}^{\mathsf{out}} \\ \mathsf{A}^{\mathsf{out}} & ::= & \mathsf{t}^{\mathsf{out}} \leq \mathsf{t}^{\mathsf{out}} \mid \mathsf{t}^{\mathsf{out}} {=} \mathsf{t}^{\mathsf{out}} \mid (\mathsf{t}^{\mathsf{out}}, \ldots, \mathsf{t}^{\mathsf{out}}) {=} \sum_{\mathsf{F}^{\mathsf{in}}} (\mathsf{t}^{\mathsf{in}}, \ldots, \mathsf{t}^{\mathsf{in}}) \\ \mathsf{t}^{\mathsf{out}} & ::= & k \mid \mid M \mid \mid C \mid \mathsf{t}^{\mathsf{out}} + \mathsf{t}^{\mathsf{out}} \mid C \cdot \mathsf{t}^{\mathsf{out}} \mid \mathsf{ite}(\mathsf{F}^{\mathsf{out}}, \mathsf{t}^{\mathsf{out}}, \mathsf{t}^{\mathsf{out}}) \end{array}$$

inner linear arithmetic formulas:

$$\begin{array}{rcl} \mathsf{F}^{\mathsf{in}} & ::= & \mathsf{A}^{\mathsf{in}} \mid \mathsf{F}^{\mathsf{in}} \wedge \mathsf{F}^{\mathsf{in}} \mid \neg \mathsf{F}^{\mathsf{in}} \\ \mathsf{A}^{\mathsf{in}} & ::= & \mathsf{t}^{\mathsf{in}} \leq \mathsf{t}^{\mathsf{in}} \mid \mathsf{t}^{\mathsf{in}} {=} \mathsf{t}^{\mathsf{in}} \\ \mathsf{t}^{\mathsf{in}} & ::= & m(e) \mid P \mid \mathsf{t}^{\mathsf{in}} + \mathsf{t}^{\mathsf{in}} \mid P \cdot \mathsf{t}^{\mathsf{in}} \mid \mathsf{ite}(\mathsf{F}^{\mathsf{in}}, \mathsf{t}^{\mathsf{in}}, \mathsf{t}^{\mathsf{in}}) \end{array}$$

multiset expressions:

$$M ::= m \mid \emptyset \mid M \cap M \mid M \cup M \mid M \uplus M \mid M \setminus M \mid M \setminus\setminus M \mid \mathsf{set}(M)$$

terminals:

m - multiset variables; e - index variable (fixed)

k - integer variable; C - integer constant; P - non-negative integer constant

Figure 2.3: Quantifier-Free Multiset Constraints with Cardinality Operator

the difference can be expressed using the sum: $x = y - z$ is formulated as $x + z = y$.

Multiset constraints contain some common multiset operations such as disjoint union, intersection, and difference, as well as the set operation that computes the largest set contained in a given multiset. These operations are provided for the sake of illustration; using the constraints $\forall e.\mathsf{F}^{\mathsf{in}}$ it is possible to specify any multiset operation defined pointwise using a QFPA formula. Note also that it is easy to reason about individual elements of sets at the top level by representing them as multisets s such that $|s| = 1$. To express that $s \in M$ we consider s as a singleton: $s \subseteq M \wedge |s| = 1$. If s is such a multiset representing an element and m is a multiset, we can count the number of occurrences of s in m with, for example, the expression $\sum \mathsf{ite}(s(e){=}0, 0, m(e))$. We can also state that a multiset s is a set. This is done by requiring that every element can appear at most once: $\forall e.s(e) \leq 1$.

2.4.1 Reducing Multiset Operations to Sums

We next show that all operations and relations on multisets as a whole can be eliminated from the language of Figure 2.3. To treat operations as relations, we flatten formulas by introducing fresh variables for subterms and using the equality operator. Figure 2.4 summarizes this process.

Definition 2.1 (Sum normal form) *A multiset formula is in a* sum normal form *iff it is of the*

INPUT: multiset formula in the syntax of Figure 2.3
OUTPUT: formula in sum-normal form (Definition 2.1)

1. Flatten expressions that we wish to eliminate:
   $C[e] \rightsquigarrow (x = e \wedge C[x])$
   where e is one of the expressions $\emptyset$, $m_1 \cup m_2$, $m_1 \cup m_2$, $m_1 \uplus m_2$, $m_1 \setminus m_2$, $\mathsf{set}(m_1)$, $|m_1|$, and where the occurrence of e is not already in a top-level conjunct of the form $x = e$ or $e = x$ for some variable x.

2. Reduce multiset relations to pointwise linear arithmetic conditions:
   $$\begin{array}{lcl}
   C[m_0 = \emptyset] & \rightsquigarrow & C[\forall e.\, m_0(e) = 0] \\
   C[m_0 = m_1 \cap m_2] & \rightsquigarrow & C[\forall e.\, m_0(e) = \mathsf{ite}(m_1(e) \leq m_2(e), m_1(e), m_2(e))] \\
   C[m_0 = m_1 \cup m_2] & \rightsquigarrow & C[\forall e.\, m_0(e) = \mathsf{ite}(m_1(e) \leq m_2(e), m_2(e), m_1(e))] \\
   C[m_0 = m_1 \uplus m_2] & \rightsquigarrow & C[\forall e.\, m_0(e) = m_1(e) + m_2(e)] \\
   C[m_0 = m_1 \setminus m_2] & \rightsquigarrow & C[\forall e.\, m_0(e) = \mathsf{ite}(m_1(e) \leq m_2(e), 0, m_1(e) - m_2(e))] \\
   C[m_0 = m_1 \setminus\setminus m_2] & \rightsquigarrow & C[\forall e.\, m_0(e) = \mathsf{ite}(m_2(e) = 0, m_1(e), 0)] \\
   C[m_0 = \mathsf{set}(m_1)] & \rightsquigarrow & C[\forall e.\, m_0(e) = \mathsf{ite}(1 \leq m_1(e), 1, 0)] \\
   C[m_1 \subseteq m_2] & \rightsquigarrow & C[\forall e.\, (m_1(e) \leq m_2(e))] \\
   C[m_1 = m_2] & \rightsquigarrow & C[\forall e.\, (m_1(e) = m_2(e))]
   \end{array}$$

3. Group all top-level universally quantified conjuncts in one formula:
   $F \wedge \forall e.F_1 \wedge \ldots \wedge \forall e.F_q \rightsquigarrow F \wedge \forall e.F_1 \wedge \ldots \wedge F_q$

4. Let the current formula be $F \wedge \forall e.F_u$. The remaining steps perform only on F:

   (a) Express each pointwise constraint using a sum:
   $C[\forall e.F] \rightsquigarrow C[\sum_{e \in \mathbb{E}} \mathsf{ite}(F, 0, 1) = 0]$

   (b) Express each cardinality operator using a sum:
   $C[|m|] \rightsquigarrow C[\sum_{e \in \mathbb{E}} m(e)]$

   (c) Flatten any sums that are not already top-level conjuncts:
   $$C[(u_1, \ldots, u_n) = \sum_F (t_1, \ldots, t_n)] \rightsquigarrow (w_1, \ldots, w_n) = \sum_F (t_1, \ldots, t_n) \wedge C[\bigwedge_{i=1}^{n} u_i = w_i]$$

   (d) Eliminate conditions from sums:
   $C[\sum_F (t_1, \ldots, t_n)] \rightsquigarrow C[\sum_{e \in \mathbb{E}} (\mathsf{ite}(F, t_1, 0), \ldots, \mathsf{ite}(F, t_n, 0))]$

   (e) Group all sums into one:
   $$\begin{array}{ll}
   P \wedge & \bigwedge_{i=1}^{q} (u_1^i, \ldots, u_{n_i}^i) = \sum_{e \in \mathbb{E}} (t_1^i, \ldots, t_{n_i}^i) \rightsquigarrow \\
   P \wedge & (u_1^1, \ldots, u_{n_1}^1, \ldots, u_1^q, \ldots, u_{n_q}^q) = \sum_{e \in \mathbb{E}} (t_1^1, \ldots, t_{n_1}^1, \ldots, t_1^q, \ldots, t_{n_q}^q)
   \end{array}$$

5. Return $P \wedge (u_1, \ldots, u_n) = \sum_{e \in \mathbb{E}} (t_1, \ldots, t_n) \wedge \forall e.F_u$

Figure 2.4: Algorithm for reducing multiset formulas to sum normal form

form

$$P \wedge (u_1, \ldots, u_n) = \sum_{e \in \mathbb{E}} (t_1, \ldots, t_n) \wedge \forall e.F$$

where P is a quantifier-free Presburger arithmetic formula without any multiset variables, and the variables in $t_1, \ldots, t_n$ and F occur only as expressions of the form $m(e)$ for m a multiset variable and e the fixed index variable.

Theorem 2.2 (Reduction to sum normal form) *Algorithm in Figure 2.4 reduces in polynomial time any formula in the language of Figure 2.3 to a formula in sum normal form. The derived formula in sum normal form is at most linear in the size of the original formula.*

2.5 Linear Integer Arithmetic with Stars

In this section, we define an extension of linear integer arithmetic. It will contain an additional atom that checks whether a vector can be represented as a sum of a finite, unbounded number of the solution vectors for a given formula F. We call such logic LIA^* (linear integer arithmetic with the star operator). We first define the star operator.

Definition 2.3 (Star operator) *Let $S \subseteq \mathbb{N}^k$ be a set of vectors of non-negative integers. The star operator is an additive closure operator of set S:*

$$S^* = \{x_1 + \ldots + x_n \mid x_1, \ldots, x_n \in S\}$$

It can be easily seen that for a finite set $S = \{x_1, \ldots, x_n\}$, $S^* = \{\lambda_1 x_1 + \ldots + \lambda_n x_n \mid \forall i.\lambda_i \geq 0\}$. In the operational research literature, S^* is also known under the name the integer conic hull generated by a set S.

Given a formula F and an integer vector u, the expression $u \in \{x \mid F(x)\}^*$ is an atom stating that u can be represented as an unbounded and finite sums of the non-negative solution vectors for F. This atom is an addition to the standard linear integer arithmetic that we consider in LIA^*. Figure 2.5 defines the language of the LIA^* formulas.

2.5.1 From Multisets to LIA* Constraints

We next argue that for every formula in a sum normal form (Definition 2.1) there is an equisatisfiable LIA^* formula (Figure 2.5).

Theorem 2.4 (Multiset elimination) *Consider a sum normal form formula G of the form*

$$P \wedge (u_1, \ldots, u_n) = \sum_{e \in \mathbb{E}} (t_1, \ldots, t_n) \wedge \forall e.F$$

LIA^* formulas: $F_0 \wedge (u_1, \ldots, u_n) \in \{(x_1, \ldots, x_n) \mid F\}^*$ (free variables of F are among x)
LIA formulas:

$$\begin{array}{lll} F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F_1 \\ A & ::= & T_1 \leq T_2 \mid T_1 = T_2 \\ T & ::= & k \mid C \mid T_1 + T_2 \mid C \cdot T_1 \mid \mathsf{ite}(F, T_1, T_2) \end{array}$$

terminals: k - integer variable; C - integer constant

Figure 2.5: Quantifier-free Presburger Arithmetic and an extension with the Star Operator

where free variables of $t_1, \ldots, t_n$ and F are multiset variables $m_1, \ldots, m_q$. Let $k_1, \ldots, k_q$ be fresh integer variables. Then G is equisatisfiable with the formula

$$P \wedge (u_1, \ldots, u_n) \in \{(t'_1, \ldots, t'_n) \mid F' \wedge k_1 \geq 0 \wedge \ldots \wedge k_q \geq 0\}^* \tag{2.7}$$

where $t'_i = t_i[m_1(e) := k_1, \ldots, m_q(e) := k_q]$ (t'_i results from t_i by replacing the multiset variables with the fresh integer variables) and $F' = F[m_1(e) := k_1, \ldots, m_q(e) := k_q]$ (similarly replacing the multiset variables with the freshly introduced integer variables).

Proof. Assume that the LIA^* formula is satisfiable and let $\mathscr{M}$ be its model. In addition, satisfiability also means that $(u_1, \ldots, u_n)$ is a sum of N vectors, each of them satisfying the formula F':

$$(u_1, \ldots, u_n) = \sum_{i=1}^{N} ({t'_1}^i, \ldots, {t'_n}^i) \wedge \forall i. F'(t'^i)$$

We will show that the original multiset formula is satisfiable as well, by constructing a model for it. We denote this model by $\mathscr{M}_M$. First, $\mathscr{M}$ and $\mathscr{M}_M$ overlap on the common integer variables. Next, we need to define the interpretation of the multiset variables occurring in the multiset formula. To do that, we construct a base set $\mathbb{E}$ containing N distinctive elements. Each summand in $\sum_{i=1}^{N} ({t'_1}^i, \ldots, {t'_n}^i)$ stands for the exactly one element of $\mathbb{E}$. Let $e \in \mathbb{E}$ be an element of $\mathbb{E}$ and let $(t'_1, \ldots, t'_n)$ be its corresponding summand. Let M be a multiset occurring in a multiset formula and let q_M be a corresponding integer variable. Then $M(e)$ for this fixed e is the value that q_M has in the corresponding summand. This completes the definition of $\mathscr{M}_M$. Since the set $\mathbb{E}$ contains N elements, it is easy to see that $\mathscr{M}_M$ is a model for

$$P \wedge (u_1, \ldots, u_n) = \sum_{e \in \mathbb{E}} (t_1, \ldots, t_n) \wedge \forall e. F$$

To prove the other direction, that satisfiability of the multiset formula also guarantees the satisfiability of the LIA^* formula, we apply analogous reasoning.

Let us consider an atom $(u_1,\ldots,u_n) \in \{(x_1,\ldots,x_n) \mid F\}^*$. If any of terms t_i in vector x is a complex expression and not a variable, we use the following transformation:

$$(u_1,\ldots,u_{n+m}) \in \{(x_1,\ldots,x_n,t_1,\ldots,t_m) \mid F\}^* \rightsquigarrow$$
$$(u_1,\ldots,u_{n+m}) \in \{(x_1,\ldots,x_n,v_1,\ldots,v_m) \mid F \wedge \bigwedge_{i=1}^{m} v_i = t_i\}^*$$

The same way we treat complex expressions appearing in vector u:

$$P \wedge (u_1,\ldots,u_n,t_1,\ldots,t_m) \in \{(x_1,\ldots,x_{m+n}) \mid F\}^* \rightsquigarrow$$
$$P \wedge \bigwedge_{i=1}^{m} w_i = t_i \wedge (u_1,\ldots,u_n,w_1,\ldots,w_m) \in \{(x_1,\ldots,x_{m+n}) \mid F\}^*$$

From now on we consider a formula $(u_1,\ldots,u_n) \in \{(x_1,\ldots,x_n) \mid F\}^*$, where all x_i and u_j are variables. If there are some free variables in F that do not appear in $\{x_1,\ldots,x_n\}$, we add them to variables x_i and in addition we extend vector $(u_1,\ldots,u_n)$ with fresh integer variables:

$$(u_1,\ldots,u_n) \in \{(x_1,\ldots,x_n) \mid F\}^* \wedge x \in FV(F) \wedge x \notin \{x_1,\ldots,x_n\} \rightsquigarrow$$
$$(u_1,\ldots,u_n,u_f) \in \{(x_1,\ldots,x_n,x) \mid F\}^*$$

To illustrate those transformation, consider the following example:

Example 2.5 *Let $(u_1,u_2) \in \{(3x+2y,x+3y) \mid 5x+2y \leq 7\}^*$ be a LIA^* atom. First we introduce fresh variables v_1 and v_2 and the problem transforms to $(u_1,u_2) \in \{(v_1,v_2) \mid 5x+2y \leq 7 \wedge v_1 = 3x+2y \wedge v_2 = x+3y\}^*$. With F we denote formula $5x+2y \leq 7 \wedge v_1 = 3x+2y \wedge v_2 = x+3y$. The free variables of F are x, y, v_1 and v_2. However, x and y do not appear in the variable vector. We extend the variable vector with x and y and the original problem is reduced to $(u_1,u_2,u_x,u_y) \in \{(v_1,v_2,x,y) \mid F\}^*$*

To summarize, given a multiset formula G belonging to the language defined in Figure 2.4, we translate G to a formula of the form $P \wedge (u_1,\ldots,u_n) \in \{(x_1,\ldots,x_n) \mid F\}^*$. Because the inner terms in the original formula are always non-negative, variables x_i represent non-negative solutions, so it is clear that the resulting formula belongs to LIA^*.

Combining the result of Theorem 2.4 and the above transformations, we obtain the following corollary:

Corollary 2.6 *For a formula in a sum normal form (Definition 2.1) there is a linear time reduction to an equisatisfiable LIA^* formula (Figure 2.5). The derived LIA^* formula is linear in the size of the original formula.*

Because of Corollary 2.6, we reduce reasoning about multisets with cardinality constraints to reasoning about an extension of linear integer arithmetic. The reduction is satisfiability preserving and from now on we focus on the satisfiability question in this new logic.

2.6 Deciding Linear Arithmetic with Sum Constraints

In this section, we describe an algorithm that reduces reasoning about LIA^* formulas to reasoning about linear integer arithmetic. The algorithm and techniques described here establish decidability. This algorithm focuses on the conjunctive fragments making it much simpler than the algorithm given in Sec. 2.7.

To address the satisfiability problem of LIA^* formulas, we first need to find a characterization of the set of non-negative solutions for a linear arithmetic formula. To answer this problem, we use semilinear sets.

Definition 2.7 (Minkowski addition and semilinear sets) *Let $C_1, C_2 \subseteq \mathbb{N}^k$ be sets of vectors of non-negative integers. The Minkowski sum of C_1 and C_2 is a set of vectors in which every element is a result of adding an element of A to an element of B, i.e. the set*

$$C_1 + C_2 = \{x_1 + x_2 \mid x_1 \in C_1 \wedge x_2 \in C_2\}$$

A linear set *is a set of the form $\{x\} + C^*$, where $x \in \mathbb{N}^n$ and $C \subseteq \mathbb{N}^n$ is a finite set of non-negative vectors. The vector $x \in \mathbb{N}^n$ is called the base vectors, while the elements of $C \subseteq \mathbb{N}^n$ are called the step vectors.*

A semilinear set *is a union of a finite number of linear sets.*

2.6.1 Formula Solutions as Semilinear Sets

We first review some relevant results from [Ginsburg and Spanier(1966)].

Theorem 2.8 (Theorem 1.3 in [Ginsburg and Spanier(1966)]) *For a given linear arithmetic formula F let P_F denote the set of all non-negative solutions of F, i.e. $P_F = \{x \mid F(x) \wedge x \in \mathbb{N}^n\}$. For every F, set P_F is a semilinear set and it is effectively computable from F. The converse also holds: for every semilinear set S there exists a formula F such that $S = P_F$.*

Ginsburg and Spanier provided a constructive proof for building a semilinear set from a given formula in [Ginsburg and Spanier(1964)]. However, this proof is of a rather theoretical importance. In a more recent paper by Pottier [Pottier(1991)], the author describes several algorithms for computing semilinear sets and also establishes bounds on the size of the generating sets.

2.6.2 Computing Semilinear Sets and their Bounds

To establish bounds on the number and the size of the base and the step vectors generating a semilinear set, we apply the results of [Pottier(1991)] and [Giles and Pulleyblank(1979)] on the size of the minimal solutions of linear integer arithmetic formulas.

Definition 2.9 *We use the following norms to establish the complexity bounds:*

1. *for an integer vector* $x = (x_1, \ldots, x_n)$, *define* $||x||_1 = \sum_{i=1}^{n} |x_i|$
2. *for an integer vector* $x = (x_1, \ldots, x_n)$, *define* $||x||_\infty = \max\{|x_1|, \ldots, |x_n|\}$
3. *for a matrix* $A = [a_{ij}]$, *define* $||A||_{1,\infty} = \sup_i(\sum_j |a_{ij}|)$

Every linear arithmetic formula can be represented as a finite disjunction of systems of linear inequalities. Since it trivially follows that a union of two semilinear sets is again a semilinear set, it is enough to solve the problem of finding a semilinear set corresponding to a system of linear inequalities. Finding the solution sets for a system of inequalities can be easily reduced to finding the solution set for a system of equalities and vice versa.

Given a system of equations $Ax = 0$, where $A \in \mathbb{Z}^{m,n}$ is an integer matrix with m rows and n columns, the set of all non-negative integer solutions forms a monoid M (a sub-monoid of $\mathbb{N}^n$). On M we define a partial order by $(x_1, \ldots, x_n) \preccurlyeq (y_1, \ldots, y_n) \Leftrightarrow \forall i.x_i \leq y_i$. The monoid M is generated by the non-zero minimal elements for $(M, \preccurlyeq)$. We call this set *the Hilbert basis of* M and denote it by $\mathscr{H}(M)$. The set $\mathscr{H}(M)$ is finite and [Pottier(1991)] describes several algorithms for its construction.

Theorem 2.10 (Theorem 1 in [Pottier(1991)]) *Given a matrix* $A \in \mathbb{Z}^{m,n}$, *let* $r = rank(A)$ *be the rank of A. Then the following bound on the size of the elements of* $\mathscr{H}(M)$ *holds:*

$$\forall x \in \mathscr{H}(M).||x||_1 \leq (1 + ||A||_{1,\infty})^r$$

Theorem 2.11 (Corollary 1 in [Pottier(1991)], applied to non-negative integers) *Consider a system of inequations* $Ax \leq b$ *where* $A \in \mathbb{Z}^{m,n}$ *and* $b \in \mathbb{Z}^m$. *Then there exist two finite sets* $C_1, C_2 \subseteq \mathbb{N}^n$ *such that*

1. *for all* $x \in \mathbb{N}^n$, $Ax \leq b$ *iff* $x \in C_1 + C_2^*$, *and*
2. $\forall h \in C_1 \cup C_2, ||h||_1 \leq (2 + ||A||_{1,\infty} + ||b||_\infty)^m$.

If we consider the system of equations $Ax = b$, *the set of all non-negative solutions can be again represented as a Minkowsky sum of two set* C_1 *and* C_2 *with the same properties, but the bound is weakened to* $(1 + ||A||_{1,\infty} + ||b||_\infty)^{rank(A)+1}$.

Proof. With $y \geq 0$ we denote a vector $y = (y_1, \ldots, y_m)$ such that $\forall i.y_i \geq 0$. We use vector y to express $Ax \leq b \Leftrightarrow \exists y \geq 0.Ax + y - b = 0$. To formulate this equation as the problem of finding the Hilbert basis, we construct a new matrix $A' = [A \mid I_m \mid -b]$. Here I_m represents the identity matrix of the dimension m. Matrix A' has m rows and $n + m + 1$ columns. We construct a variable vector $t = (x, y, z)$ of the dimension $n + m + 1$. To access the first n coordinates we use projection $\pi_x(t) = x$. To access the last coordinate we use projection $\pi_z(t) = z$. To find the set of all non-negative solutions of $Ax \leq b$ we use the following equivalence:

$$Ax \leq b \wedge x \in \mathbb{N}^n \Leftrightarrow A't = 0 \wedge t \in \mathbb{N}^{n+m+1} \wedge \pi_x(t) = x \wedge \pi_z(t) = 1$$

Let $\mathscr{H}_{A'}$ be the Hilbert basis of the monoid of non-negative integer solutions of the system $A't = 0$. We define C_1 and C_2 as follows:

$$C_1 = \{x \mid \exists t.t \in \mathscr{H}_{A'} \wedge \pi_x(t) = x \wedge \pi_z(t) = 1\}$$
$$C_2 = \{x \mid \exists t.t \in \mathscr{H}_{A'} \wedge \pi_x(t) = x \wedge \pi_z(t) = 0\}$$

We next show that every non-negative solution of $Ax \leq b$ can be written as a sum of a vector from C_1 and a finite number of vectors from C_2:

$$\begin{aligned}
&Ax \leq b \wedge x \in \mathbb{N}^n \Leftrightarrow A't = 0 \wedge t \in \mathbb{N}^{n+m+1} \wedge \pi_x(t) = x \wedge \pi_z(t) = 1 \\
&(t \neq 0 \text{ and can be written as a linear combination of the vectors from } \mathscr{H}_{A'}) \\
&\Leftrightarrow t = (x^0, y^0, 1) + \sum_{k=1}^{l} (x^k, y^k, 0) \wedge \forall k.(x^k, y^k, z) \in \mathscr{H}_{A'} \\
&\Leftrightarrow x = c_1^0 + \sum_{k=1}^{l} c_2^k \wedge c_1^0 \in C_1 \wedge \forall k.c_2^k \in C_2 \\
&\Leftrightarrow x \in C_1 + C_2^*
\end{aligned}$$

To prove the upper bounds, first we observe that $\text{rank}(A') = m$, because of I_m. We also derive an additional bound on the $||A'||_{1,\infty}$: $||A'||_{1,\infty} \leq ||A||_{1,\infty} + 1 + ||b||_\infty$. Combining those bounds together with Theorem 2.10 results in the fact that the $||t||_1$ norm of every $t \in \mathscr{H}_{A'}$ is bounded by $(2 + ||A||_{1,\infty} + ||b||_\infty)^m$. The last observation is that for every vector $h \in C_1 \cup C_2$ holds $||h||_1 \leq ||t||_1$ which results in

$$\forall h \in C_1 \cup C_2.||h||_1 \leq (2 + ||A||_{1,\infty} + ||b||_\infty)^m$$

In the case when we consider the system of equations $Ax = b$, we apply analogous reasoning and define sets C_1 and C_2, as well as the bounds on the size of the vectors in $C_1 \cup C_2$.

2.6.3 LIA Formulas Representing LIA* Formulas

Every linear arithmetic formula F can be converted into a disjunction of systems of equations and inequations. The number of such systems is singly exponential in the number of atomic formulas in F. Once F is converted into a disjunction of systems of the form $Ax = b$ or $Ax \leq b$, we invoke Theorem 2.11 and construct sets C_1 and C_2. If $C_1, C_2 \subseteq \mathbb{N}^n$ are finite sets, then $C_1 + C_2^*$ is a particular kind of a semilinear set. Moreover, the values occurring in A and b in the resulting systems are polynomially bounded by the coefficients and constants in the original LIA formula.

Consequently, for a formula F let $B = \max_i (2 + ||A_i||_{1,\infty} + ||b_i||_\infty)^m$ where $A_i x \leq b_i$ are systems, which are the results of decomposition of F. If s is the size of the original formula F, then B is at most singly exponential in s. We denote this bound by $2^{p(s)}$, where p is some polynomial that follows from details of the algorithm for generating all the systems of equations and inequations whose disjunction is equivalent to F. We thus obtain the following theorem.

Theorem 2.12 *Let F be a quantifier-free linear integer arithmetic formula of size s. Then there exist a number n and finite sets A_i, B_i for $1 \leq i \leq n$ such that the set of satisfying assignments for F is given as*

$$\{x \mid F(x)\} \;=\; \bigcup_{i=1}^{n} (A_i + B_i^*)$$

In addition, there is a polynomial p such that $||h||_1 \leq 2^{p(s)}$ for each $h \in \bigcup_{i=1}^{d} (A_i \cup B_i)$.

If $A = \{a_1, \ldots, a_q\}$ and $B = \{b_1, \ldots, b_r\}$ for $a_i, b_j \in \mathbb{N}^n$, then the condition $u \in A + B^*$ is given by the formula $\bigvee_{i=1}^{q} (u = a_i + \sum_{j=1}^{r} \lambda_j b_j)$ where $\lambda_1, \ldots, \lambda_r$ are existentially quantified variables ranging over $\mathbb{N}$. This view leads to the following formulation of Theorem 2.13.

Theorem 2.13 (Semilinear normal form for linear arithmetic) *Let F be a quantifier-free linear integer arithmetic formula of size s. Then there exist a number n, numbers $q_1, \ldots, q_n$ and vectors a_i and b_{ij}, $1 \leq j \leq q_i$, $1 \leq i \leq n$, with $||a_i||_1, ||b_{ij}||_1 \leq 2^{p(s)}$ such that*

$$F(x) \;\Leftrightarrow\; \exists \lambda_{11}, \ldots, \lambda_{nq_n}. \bigvee_{i=1}^{n} (x = a_i + \sum_{j=1}^{q_i} \lambda_{ij} b_{ij}) \tag{2.8}$$

The semilinear sets provide a characterization of the set of all solutions of a formula F. To answer our starting problem of expressing $(u_1, \ldots, u_n) \in \{(t_1, \ldots, t_n) \mid F\}^*$ as a QFPA formula, we first represent the set of solutions of F as a semilinear set. We next applying the $*$ operator on a semilinear set, which results in a linear arithmetic formula. Similar result was also obtained in [Lugiez and Zilio(2002), Section 3.2].

Theorem 2.14 (Elimination of the star operator) *Given a formula F, let a semilinear set representing all non-negative solutions of F be given with the vectors a_i and b_{ij}:*

$$F(x) \Leftrightarrow x \in \bigcup_{i=1}^{n} (\{a_i\} + \{b_{ij}\}^*)$$

The expression $u \in \{t \mid F\}^*$, where u and t are integer vectors, is equisatisfiable to

$$\exists \mu_i, \lambda_{ij}.\ u = \sum_{i=1}^{d} (\mu_i a_i + \sum_{j=1}^{q_i} \lambda_{ij} b_{ij}) \ \wedge \ \bigwedge_{i=1}^{d} (\mu_i = 0 \implies \sum_{j=1}^{q_i} \lambda_{ij} = 0) \tag{2.9}$$

The existentially quantified variables μ_i, λ_{ij} become free variables in the satisfiability problem. The last conjunct in the formula (2.9) states that if a base vector did not appear in the sum, then also its corresponding step vectors cannot contribute to the sum.

To summarize, the starting problem was the satisfiability question for the multiset formulas with cardinality constraints. We reduced this problem to reasoning about the formulas of the form $P \ \wedge \ (u_1, \ldots, u_n) \in \{(t_1, \ldots, t_n) \mid F'\}^*$. Theorem 2.14 shows how the star operator can be entirely eliminated and the resulting formula is a conjunction of P and formula (2.9). The formula that we obtain in the end is a QFPA formula. The satisfiability problem for quantifier-free linear arithmetic is decidable, it is an NP-complete problem [Papadimitriou(1981)].

Theorem 2.15 *Consider a formula F belonging to the language defined in Figure 2.3. Checking whether F is satisfiable is a decidable problem.*

2.7 Complexity of Linear Arithmetic with Stars

While reducing a LIA* formula $P \wedge u \in \{x \mid F(x)\}^*$ to a linear arithmetic formula, we used semilinear sets. The set of all non-negative integer solutions of F can be described with finitely many generating vectors a_i (base vectors) and b_{ij} (step vectors). However, the number of generating vectors can be exponential [Pottier(1991)], so we avoid explicitly constructing them. We instead apply several relevant results from operations research to construct a polynomially large equisatisfiable formula. In this section we describe a construction of such polynomial-sized formula.

2.7.1 Estimating Coefficient Bounds of Disjunctive Form

The results on which we rely are usually expressed for integer linear programming problems. They mostly expressed properties for the systems of the form $Ax \leq b$ or $Ax = b$. Those properties are expressed in terms of m, n and a, where m is the number of rows in the matrix A, n is the number of columns and a is a maximal absolute value of all coefficient occurring

in A. In this sections we will describe how one can estimate those values without actually converting formula into a disjunction of the systems of the form $Ax = b$.

Let F be a QFPA formula. F can be converted into an equivalent disjunction of integer linear programming problems $\bigvee_{i=1}^{l} A_i x = \vec{b}_i$. Let m_i be a number of rows in A_i and let n_i be a number of columns in A_i and let a_i be a maximal absolute value of all coefficient occurring in A_i and b_i. For a given F, define $m_F = \max_{i=1}^{l} m_i$, $n_F = \max_{i=1}^{l} n_i$ and $a_F = \max_{i=1}^{l} a_i$.

Lemma 2.16 (Values of m_F, n_F and a_F) *Let F be a* QFPA *formula. If a subformula does not occur within any* ite *expression we say that it has the positive polarity if it occurs under an even number of negations and say it has the negative polarity if it occurs under an odd number of negations. If a subformula occurs within an* ite *expression we say that it has no polarity. Let g be the number of atomic formula occurrences of the form $t_1 = t_2$ that have the positive polarity in F, and let h be the number of the remaining atomic formulas. Let v be the number of variables in F and a the maximum of absolute values of integer constants. Then $m_F \leq g + h$, $n_F \leq v + h$, and $a_F \leq a + 1$.*

Proof. We can transform $F[\mathsf{ite}(C, t_1, t_2)]$ into a disjunction of $C \wedge F[t_1]$ and $\neg C \wedge F[t_2]$. Repeating this transformation we eliminate all ite expressions and obtain disjuncts whose size is polynomial in the size of F. Let D be one of the disjuncts after such ite elimination. The polarity of all g atomic formulas $t_1 = t_2$ that occur positively in F remains positive in each D. Each of the remaining h atomic formulas becomes of the form $t_1 \leq t_2$, $t_1 = t_2$ or disjunction $t_1 \leq t_2 \vee t_1' \leq t_2'$. In disjunctive normal form of D, each of the h atomic formulas $t_1 \leq t_2$ may require addition of at most one fresh variable to be converted into equality $t_1 + x = t_2$. The resulting number of variables is therefore bounded by $v + h$ whereas the total number of atomic formulas is bounded by $g + h$. When transforming $t_1 < t_2$ into $t_1 + 1 \leq t_2$ there is a possibility that the constants part of t_1 or t_2 will be increased by one, so $a_F \leq a + 1$.

Example 2.17 *As an illustration consider a formula F: $z = \mathsf{ite}(x \leq y, x, y) \wedge z + 2 \neq y$. The number of atomic formulas of the form $t_1 = t_2$ with the positive polarity is one, $z = \mathsf{ite}(x \leq y, x, y)$, which means that $g = 1$. There are two other remaining atomic formulas: $x \leq y$ and $z + 2 \neq y$, i.e. $h = 2$. Finally, $v = 3$ and $a = 2$. From those values we can easily compute values m_F, n_F and a_F: $m_F \leq 3$, $n_F \leq 5$, and $a_F \leq 3$. In this particular example, those bounds are also tight. One of the disjuncts will contain the following formula $z = x \wedge x \leq y \wedge z + 2 < y$. When translated into equalities, the formula becomes $z = x \wedge x + l_1 \leq y \wedge z + 3 + l_2 = y$. Written in the matrix form, it becomes:*

$$\begin{bmatrix} 1 & 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 \\ - & -1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ l_1 \\ l_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -3 \end{bmatrix}$$

2.7.2 Size of the Solution Set Generators

This section combines the results of the previous section (Section 2.7.1) with the bounds on the size of the generators of a semilinear set. It expresses Theorem 2.13 in the terms of Lemma 2.16.

Lemma 2.18 *For every* QFPA *formula F, there exist q base vectors a_i, $1 \le i \le q$, and for each i the corresponding q_i step vectors b_{ij} for $1 \le j \le q_i$ such that*

$$F(x) \;\Leftrightarrow\; \exists \lambda_{ij}. \bigvee_{i=1}^{q} (x = a_i + \sum_{j=1}^{q_i} \lambda_{ij} b_{ij}) \tag{2.10}$$

The norm $||\cdot||_1$ of all vectors a_i and b_{ij} is bounded by $(1+(n_F+1)a_F)^{m_F+1}$ where n_F, m_F, a_F are defined as in Lemma 2.16.

Proof. The existence of the generating vectors was already proved in Theorem 2.13. The only remaining thing to argue is a bound on the size of the generating vectors. Let F be a formula and $\bigvee_{i=1}^{d} A_i x = b$ be its decomposition into a disjunction of the system of equations. In Theorem 2.11 there is a bound for the case of a system of equations $Ax = b$: every generating vector is bounded by $(1 + ||A||_{1,\infty} + ||b||_\infty)^{\mathrm{rank}(A)+1}$. A matrix A will have a maximal $||A||_{1,\infty}$ if there is a row containing the value a or $-a$. That implies that for every A_i: $||A_i||_{1,\infty} \le n_F a_F$. Similarly, we prove that for every b_i: $||b||_\infty \le a_F$. Finally, the rank of A_i is bounded with both n_f and m_F. We choose M_F to keep the consistence with other formulas.

2.7.3 Selecting Polynomially Many Generators

We have already established the bounds on the size of the generating vectors. However, we did not establish the bound on their number. This section finds that bound. We consider the formula

$$x = \sum_{i=1}^{q} (\mu_i a_i + \sum_{j=1}^{q_i} \lambda_{ij} b_{ij}) \;\wedge\; \bigwedge_{i=1}^{q} (\mu_i = 0 \rightarrow \sum_{j=1}^{q_i} \lambda_{ij} = 0) \tag{2.11}$$

Our goal is to show that if x is a linear combination of the generators, then it is also a linear combination of a polynomial subset of the generators that form a smaller semilinear set. We prove this fact using a theorem about sparse solutions of integer linear programming problems [Eisenbrand and Shmonin(2006)].

Given a set of vectors X and a vector $b \in X^*$, the following theorem determines the bound on the number of vectors sufficient for representing b as a linear combination of vectors from X.

Theorem 2.19 (Theorem 1 (ii) in [Eisenbrand and Shmonin(2006)]) *Let $X \subseteq \mathbb{Z}^d$ be a finite set of integer vectors and let $b \in X^*$. Then there exists a subset $\tilde{X}$ such that $b \in \tilde{X}^*$ and $|\tilde{X}| \le$*

$2d\log(4dM)$, *where* $M = \max_{x\in X} ||x||_\infty$.

Proof. The proof can be found in Appendix A.

Theorem 2.19 has been applied in [Kuncak and Rinard(2007)] to show that the satisfiability of constraints on sets with cardinality operators is in NP. In the case of multisets and LIA* we need to generalize this idea because of dependencies between the base vectors and the corresponding step vectors.

Theorem 2.20 *Let* F *be* QFPA *formula and* a_i, b_{ij}, x, q, q_i *be values and vectors from* (2.9). *Then there exist sets* $I_0, I_1 \subseteq \{1,\ldots,q\}$ *and* $J \subseteq \cup_{i=1}^{q}\{(i,1),\ldots,(i,q_i)\}$ *such that*

$$x = \sum_{i\in I_0}(a_i + \sum_{(i,j)\in J}\lambda'_{ij}b_{ij}) + \sum_{i\in I_1}\mu'_i a_i \tag{2.12}$$

and $|I_0| \le |J| \le B$, *and* $|I_1| \le B$, *where* $B = 2n_F(\log 4n_F + (m_F+1)\log(1+(n_F+1)a_F))$.

Proof. By assumption, $x = \sum_{i=1}^{q}(\mu_i a_i + \sum_{j=1}^{q_i}\lambda_{ij}b_{ij})$ and $\bigwedge_{i=1}^{q}(\mu_i = 0 \rightarrow \sum_{j=1}^{q_i}\lambda_{ij} = 0)$. All zero indices can be removed and from now on we assume that all μ_i and λ_{ij} are strictly positive. We define vectors a and b as $a = \sum_i \mu_i a_i$ and $b = \sum_{ij}\lambda_{ij}b_{ij}$, so $x = a + b$. Because of $b = \sum_{ij}\lambda_{ij}b_{ij}$, vector b can be seen as an element of the integer cone generated by vectors b_{ij}: $b \in \{b_{ij}\}^*$. We apply Theorem 2.19 and conclude that there exists a set J of indices (i,j) and coefficients λ'_{ij} such that $b = \sum_{(i,j)\in J}\lambda'_{ij}b_{ij}$ and $|J| \le 2n_F\log(4n_F M)$ where M is the bound on the size generators. We denote this number with B: $B = 2n_F\log(4n_F M)$. To satisfy the dependencies between step vectors b_{ij} and a base vector a_i, let $I_0 = \{i \mid \exists j.(i,j) \in J\}$. Note that $|I_0| \le |J|$. Let $a_b = \sum_{i\in I_0} a_i$. The vector $a_b + b$ is generated by vectors whose indices are I_0 and J. However, the vector $a_b + b$ still does not cover the whole vector x and we define the vector $a_r = a - a_b$. The vector a_r can be again seen as an element of an integer cone: $a_r = \sum_{i\in I_0}(\mu_i - 1)a_i + \sum_{i\in\{1,\ldots,q\}\setminus I_0}\mu_i a_i$. Applying once again Theorem 2.19 we conclude that there exists $I_1 \subseteq \{1,\ldots,q\}$ with $|I_1| \le B$ such that $a_r = \sum_{i\in I_1}\mu'_i a_i$.

We still need to compute the size of the value B. Let H be a set containing all generating vectors a_i and b_{ij}. To compute the value M from Theorem 2.19, we note that since $||v||_1 \le ||v||_\infty$ for all vectors v, then $M \le \max_{h\in H}||h||_1$ Using the bound $(1+(n_F+1)a_F)^{m_F+1}$ from Lemma 2.18, we obtain that $B = 2n_F\log(4n_F(1+(n_F+1)a_F)^{m_F+1}) = 2n_F(\log 4n_F + (m_F+1)\log(1+(n_F+1)a_F))$.

Theorem 2.20 shows that there are only polynomially many generators of a semilinear set needed to represent a solution of a linear arithmetic formula. The number of generators is polynomial in the size of the given formula.

2.7.4 Grouping Generators into Solutions

Having showed that if $u \in \{x \mid F(x)\}^*$, then u is a particular linear combination of polynomially many generating vectors a_i and b_{ij} and using the fact that those generating vectors are

also polynomially bounded, this suggests an idea of guessing polynomially many bounded vectors, checking whether they are generators, and then checking whether u is their linear combination. However, it is not clear how to check if a guessed vector is one of a_i's or b_{ij}'s without calculating them. In this section we show that we can avoid the problem of checking whether a vector is a generator and reduce the problem to checking whether a vector is a solution of F. We show that it is enough to guess polynomially many vectors x such that $F(x)$ holds and check whether $u = \sum_{i=1}^{k} \lambda_i x_i$.

Lemma 2.21 *Let F be a* QFPA *formula and $u \in \{x \mid F(x)\}^*$. Then there exist k vectors $x_1, \ldots, x_k$ for $k \leq 4n_F(\log 4n_F + (m_F+1)\log(1+(n_F+1)a_F))$ such that for some non-negative integers λ_i*

$$u = \sum_{i=1}^{k} \lambda_i x_i \ \wedge \ \bigwedge_{i=1}^{k} F(x_i)$$

Proof. First observe that in Theorem 2.20 vectors $a_i + \sum_{(i,j)\in J} v'_{ij} b_{ij}$ are solutions of F and that their number is bounded by B. Similarly, a_i are also solutions of F and their number is bounded by B as well. The total number of solutions is bounded by $2B$ where B is from Theorem 2.20.

2.7.5 Multiplication by Bounded Bit Vectors

We can outline a new algorithm for checking satisfiability of a LIA^* formula $F_0 \wedge u \in \{v \mid F(v)\}^*$. First, using Lemma 2.16 we calculate the values of m_F, n_F and a_F. Using those values and Lemma 2.21 we estimate an upper bound $k = 4n_F(\log 4n_F + (m_F+1)\log(1+(n_F+1)a_F))$ on the number of solution vectors x_i. We construct an equisatisfiable formula

$$F_0 \wedge u = \lambda_1 x_1 + \ldots + \lambda_k x_k \wedge \bigwedge_{i=1}^{k} F(x_i) \tag{2.13}$$

Formula (2.13) is polynomial in the size of $F_0 \wedge u \in \{v \mid F(v)\}^*$, but it is not a QFPA formula because it contains multiplication of variables in $\lambda_i \cdot x_i$. We address this problem by showing that the values of λ_i in the smallest solutions have a polynomial number of bits, which allows us to express multiplication using bitwise expansion.

To express terms $\lambda_i x_i$ from Lemma 2.21 as a QFPA term, we show that the smallest solution u, if exists, is bounded [Papadimitriou(1981)].

Theorem 2.22 (Theorem, p.767 in [Papadimitriou(1981)]) *Let A be an $m \times n$ integer matrix and b an m-vector, both with entries from $[-a..a]$. Then the system $Ax = b$ has a solution in $\mathbb{N}^n$ if and only if it has a solution in $[0..M]^n$ where $M = n(ma)^{2m+1}$.*

The Theorem 2.22 states that it is enough to find a solution x such that $||x||_\infty \leq M$. Here is a simple algorithm to find a non-negative solution of $Ax = b$: check all vectors consisting of

non-negative integers such that their $||\cdot||_\infty \leq M$ until a solution is found. If no solution was found in that range, then $Ax = b$ has no solution in non-negative integers.

Applying the Theorem 2.22 and using the fact that the whole formula can be represented as a QFPA formula, we know that there is a bound on the solution vector u. Let r_B be a bound on the vector u of formula $F_0 \wedge u \in \{v \mid F(v)\}^*$: $||u||_\infty \leq r_B$. Because every λ_i in formula (2.13) must be a non-negative integer, $\lambda_i \leq ||u||_\infty \leq r_B$, so each λ_i is also bounded by r_B. This means that every λ_i can be represented as a bit-vector of size r for $r = \lceil \log r_B \rceil$. Let $\lambda_i = \overline{\lambda_{ir} \ldots \lambda_{i1}\lambda_{i0}} = \sum_{j=0}^{r} \lambda_{ij} 2^j$. Then

$$\lambda_i x_i = (\sum_{j=0}^{r} \lambda_{ij} 2^j) x_i = \sum_{j=0}^{r} 2^j (\lambda_{ij} x_i) = \sum_{j=0}^{r} 2^j \mathsf{ite}(\lambda_{ij}, x_i, 0) = \\ \mathsf{ite}(\lambda_{i0}, x_i, 0) + 2(\mathsf{ite}(\lambda_{i1}, x_i, 0) + 2(\mathsf{ite}(\lambda_{i2}, x_i, 0) + \ldots)) \quad (2.14)$$

Still it remains to show how to establish and compute the value r_B.

2.7.6 Estimating the Solution Size Bounds

Theorem 2.23 *Let F_0 be a* QFPA *formula. Let $u = (u_1, \ldots, u_d)$ denote a d-dimensional vector of variables ranging over non-negative integers. Let F be a* QFPA *formula which does not share any variable with F_0 and u. If a formula $F_0 \wedge u \in \{x \mid F(x)\}^*$ is satisfiable, then there exists a non-negative solution vector u_S for variables u such that $||u_S||_\infty \leq r_B = n(ma)^{2m+1}$ where n, m and a are defined by*

1. $m := n_F + m_{F_0}$
2. $n := n_{F_0} + n_F(1 + 6(\log 4 n_F + (m_F + 1)\log(1 + (n_F + 1)a_F)))$
3. $a := \max\{a_{F_0},\ (1 + (n_F + 1)a_F)^{m_F+1}\}$

The values $m_{F_0}, n_{F_0}, a_{F_0}, m_F, n_F$ and a_F are computed as in the Lemma 2.16.

Proof. We establish a bound on the size of the solution vector by applying two facts. First, u_S is a solution vector for $u \in \{\vec{v} \mid F(\vec{v})\}^*$. As shown in Theorem 2.14, u_S is a linear combination of the generators of a semilinear set and u_S can be expressed as

$$u_S = \sum_{i=1}^{q} (\mu_i a_i + \sum_{j=1}^{q_i} \lambda_{ij} b_{ij})$$

Assuming that all the generator vectors a_i and b_{ij} are known (they can be computed), we need to solve the equation

$$u = \sum_{i=1}^{q} (\mu_i a_i + \sum_{j=1}^{q_i} \lambda_{ij} b_{ij})$$

for the vector u and variables μ_i and λ_{ij}. If we represent the above condition in the $Ax = b$ form, the matrix A consists of the generators of semilinear set and the negative identity matrix $-\mathbb{1}_n$, while the vector x consists of the parameters μ_i and λ_{ij} followed by the vector u.

$$\begin{bmatrix} \underbrace{\begin{matrix} | & | & | \\ | & | & | \\ | & | & | \end{matrix}}_{\text{generators}} & & \\ & \begin{matrix} -1 & & \\ & \ddots & \\ & & -1 \end{matrix} & \end{bmatrix} \begin{bmatrix} \mu_i \\ \lambda_{ij} \\ u_1 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

We denote this system with $A_g x_g = 0$. The matrix A_g can have at most n_F rows and at most $n_F + n_G$ columns, where n_G is the number of generators. The number of rows is bounded by n_F (and not by d) because the length of the vector u can increase (for an example, when converting inequalities into equalities). Theorem 2.20 established the bound on the number of generators. Let $B = 2n_F(\log 4n_F + (m_F + 1)\log(1 + (n_F + 1)a_F))$ be the bound introduced in the theorem. By reconstructing the proof, we count the needed generating vectors: there are B step vectors b_{ij} and B base vectors a_i associated with those step vectors. However, there may still appear B base vectors. Altogether, there are at most $3B$ generating vectors, i.e. $n_G \leq 6n_F(\log 4n_F + (m_F + 1)\log(1 + (n_F + 1)a_F))$. In addition, this leads to the fact that A_g has at most $n_F(1 + 6(\log 4n_F + (m_F + 1)\log(1 + (n_F + 1)a_F)))$ columns.

The second fact that we will use is the observation that u_S is a component of the solution vector of F_0. This implies that there is a matrix A_0 with dimensions at most m_{F_0} and n_{F_0} and a vector b_0 such that $A_0 w = b_0$. In the vector w we assume that first comes the vector u followed by the remaining variables that we denote with the vector v: $w = (u, v)$.

We combine those two facts into one by constructing a new system that contains both matrices A_g and A_0. We denote this new system with $A_f x = b_f$ and it has the following form:

$$\left[\begin{array}{cc|c} A_g & & 0 \\ \hline 0 & & A_0 \end{array}\right] \begin{bmatrix} \mu_i \\ \lambda_{ij} \\ u \\ v \end{bmatrix} = \begin{bmatrix} 0 \\ b_0 \end{bmatrix}$$

Matrix A_f has at most $n_{F_0} + n_F(1 + 6(\log 4n_F + (m_F + 1)\log(1 + (n_F + 1)a_F)))$ columns and at most $n_F + m_{F_0}$ rows.

To establish an upper bound on the maximum of absolute values in A_f and b_f, we apply Lemma 2.18 in which there is an upper bound on the $||\cdot||_1$ for all the generating vectors. Since for every vector z, $||z||_\infty \leq ||z||_1$, we conclude that the bound given in Lemma 2.18 is also an upper bound for the $\max\{\,|a_{ij}| \mid a_{ij} \in A_g\}$. The maximal absolute value appearing in A_0 and b_0 is bounded by a_{F_0} by the definition. Therefore, the maximal absolute value in the matrix

A_f and vector b_f is the bigger one of those two values.

We obtain the final result by applying Theorem 2.22 to $A_f x = b_f$. Since the vector u is embedded into the vector x, it holds $||u||_\infty \leq ||x||_\infty$ and this gives us the required upper bound.

2.7.7 An NP-Algorithm for LIA* Satisfiability

In this section, we summarize the results of all previous sections and present an algorithm of the optimal complexity for checking satisfiability of LIA* formulas. Let $F_0 \wedge u \in \{x \mid F(x)\}^*$ be a LIA* formula. The algorithm constructs an equisatisfiable QFPA formula through several steps:

1. Apply Lemma 2.16 and compute values $m_{F_0}, n_{F_0}, a_{F_0}, m_F, n_F$ and a_F. To compute those values, it is enough to analyze the formulas F and F_0 without converting them to a disjunctive form. The required values are computed in a linear time of the size of the input formula and they are small (smaller than the size of the input formula).

2. Apply Theorem 2.23 and using $m_{F_0}, n_{F_0}, a_{F_0}, m_F, n_F$ and a_F compute the values n, m and a. Compute the value $r = \lceil \log n + (2m+1)\log(ma) \rceil$. Note that a might be of the exponential value, but it does not matter, because to compute the value r, the log function is applied to a. This implies the final value r is polynomial in the size of the input formula.

3. With t_i we denote the term $t_i = \sum_{j=0}^{r} 2^j \mathsf{ite}(\lambda_{ij}, x_i, 0) = \mathsf{ite}(\lambda_{i0}, x_i, 0) + 2(\mathsf{ite}(\lambda_{i1}, x_i, 0) + 2(\mathsf{ite}(\lambda_{i2}, x_i, 0) + \ldots))$. This is a term from the formula (2.14). The values λ_{ij} are boolean variables and the value x_i is an integer vector (also a variable). The size of the term t_i is polynomial in the size of the input formula, since the term t_i consists of r summands.

4. Apply Lemma 2.21 and compute an upper bound for the number of t_i terms, i.e. compute the value k: $k = 4n_F(\log 4n_F + (m_F+1)\log(1+(n_F+1)a_F))$.

5. Construct the formula F_1:
$$F_0 \wedge u = \sum_{i=1}^{k} t_i \ \wedge \bigwedge_{i=1}^{k} F(x_i)$$
The term t_i contains the vector x_i and variables λ_{ij}. The size of this newly constructed formula is polynomial in the size of the original input formula.

6. Use a solver for linear integer arithmetic and check satisfiability of F_1

Putting everything together, the algorithm constructs an equisatisfiable QFPA formula, which is polynomial in the size of the the original formula. Since checking satisfiability of QFPA formulas is an NP-complete problem, the satisfiability question for LIA* formulas is also an NP-complete problem.

Theorem 2.24 *Checking satisfiability of* LIA* *formulas is an NP-complete problem.*

2.8 Complexity of Multiset Constraints

In this section, we return to the original problem of checking satisfiability of formulas belonging to the language of multiset constraints, defined in Figure 2.3. It was shown in [Kuncak and Rinard(2007)] that satisfiability checking for the set constraints with cardinality operator is an NP-complete problem. This problem is subsumed by constraints defined in Figure 2.3, which means that checking multiset constraints is at least NP-hard. Our goal is to show that checking satisfiability of formulas defined in Figure 2.3 is also an NP-complete problem.

Since the language supports arbitrary propositional operators, the satisfiability problem for this language is clearly NP-hard. The non-trivial part of NP-completeness is therefore establishing the membership in NP. The standard way of proving that a problem belongs to the NP class is by showing that a candidate solution for the problem can be verified in polynomial time. Let F_m be a multiset formula. We can reduce it to an equisatisfiable LIA^* formula in a polynomial time. We denote this formula with F_l, $F_l \in \mathsf{LIA}^*$. Theorem 2.24 proves that checking satisfiability of F_l is an NP-complete problem, which means that its candidate solution can be verified in polynomial time, which in addition also means that verifying the solution for F_M can be done in polynomial time.

This way we proved that checking satisfiability of multiset formulas with the cardinality constraints belongs to the same class of problems as checking satisfiability of set formulas with the cardinality constraint.

Theorem 2.25 *Checking satisfiability of formulas belonging to the language of multiset constraints, defined in Figure 2.3 is an NP-complete problem.*

2.9 Undecidability of Quantified Constraints

Until now, we have discussed quantifier-free multiset formulas. We next show that adding quantifiers to the language of Figure 2.3 results in undecidable constraints.

We already pointed out that the language defined in Figure 2.3 can be seen as a generalization of quantifier-free Boolean algebra with Presburger arithmetic (QFBAPA) defined in [Kuncak and Rinard(2007)]. QFBAPA is a language for reasoning about sets with cardinality constraints. Given that QFBAPA admits quantifier elimination [Feferman and Vaught(1959), Kuncak et al.(2006)Kuncak, Nguyen, and Rinard], it is an interesting question whether multiset quantifiers can be eliminated from constraints. The decision procedure that we described before demonstrated that the multiset formulas without the cardinality operator can be viewed as a product of Presburger arithmetic structures. Therefore, Feferman-Vaught theorem [Feferman and Vaught(1959)] (a summary can be found in [Kuncak and Rinard(2003)] in Section 3.3) gives a way to decide the first-order theory of multiset operations extended with the ability to state cardinality of sets of the form $|\{e \mid F(e)\}|$. This corresponds to multiset theory with counting distinct elements of multisets, which is denoted by $FO^{\#D}_{\mathcal{M}}$ in [Lugiez(2005)].

However, the language $FO^{\#D}_{\mathcal{M}}$ is strictly less expressive than a quantified extension of the language in Figure 2.3. The language in Figure 2.3 contains the summation expressions $\sum_{F(e)} t(e)$ and that corresponds to the language $FO^{\#}_{\mathcal{M}}$ defined in [Lugiez(2005)]. The decidability of $FO^{\#}_{\mathcal{M}}$ was left open in [Lugiez(2005)]. We next prove that this language is undecidable.

The undecidability follows by reduction from Hilbert's 10th problem [Matiyasevich(1970)], because quantified multiset constraints can define addition and multiplication. To define addition, we use disjoint union $\uplus$:

$$x + y = z \iff \exists M_1.\exists M_2.|M_1| = x \wedge |M_2| = y \wedge |M_1 \uplus M_2| = z$$

To define $x \cdot y = z$, we introduce a new multiset P that contains x distinct elements, each of which occurs y times. The following formula encodes this property:

$$\begin{aligned} x \cdot y = z \iff \exists P.\ & z = |P| \ \wedge\ x = |\mathsf{set}(P)| \ \wedge \\ & (\forall M.\ |M| = z \wedge |\mathsf{set}(M)| = 1 \wedge \mathsf{set}(M) \subseteq P \implies |M \cap P| = y) \end{aligned}$$

Because we can define addition and multiplication using the quantified multiset constraints, we can express in this logic also the satisfiability question of Diophantine equations (i.e. Hilbert's tenth problem). In [Matiyasevich(1970)], Matiyasevich proved that there cannot exist an algorithm to check whether a Diophantine equation has any solutions in integers. Applying this result, we conclude that satisfiability of multiset constraints with quantifiers and cardinality is undecidable. Similarly, we obtain undecidable constraints if in the quantified expressions $\forall e.F$ we admit the use of outer integer variables as parameters. This justifies the current "stratified" syntax that distinguishes inner and outer integer variables.

A natural question is whether the presence of the built-in set operator is needed for undecidability of quantified constraints. Nevertheless, the set operator is itself definable using quantifiers. Here is an example how to encode the set operator: $S = \mathsf{set}(M)$ iff S is the smallest multiset that behaves the same as M with respect to a simple set membership. Behaving the same with respect to a simple set membership is given by

$$\mathsf{memSame}(M_1, M_2) \iff (\forall E.|E| = 1 \implies (E \subseteq M_1 \iff E \subseteq M_2))$$

Using the $\mathsf{memSame}(M_1, M_2)$ predicate we encode the set operator as:

$$S = \mathsf{set}(M) \iff (\mathsf{memSame}(S, M) \wedge (\forall S_1.\ \mathsf{memSame}(S_1, M) \implies S \subseteq S_1))$$

Moreover, note that, as in any lattice, $\cap$ and $\subseteq$ are inter-expressible using quantifiers. Therefore, adding quantifiers to a multiset language that contains $\subseteq$ and cardinality constructs already gives undecidable constraints. This answers negatively the question on decidability of $FO^{\#D}_{\mathcal{M}}$ posed in [Lugiez(2005), Section 3.4].

3 Implementation: Automated Reasoner for Sets and Multisets

In this chapter, we provide an overview of the MUNCH reasoner for sets and multisets. MUNCH takes as the input a formula in a logic that supports expressions about sets, multisets, and integers. Constraints over collections and integers are connected using the cardinality operator, as defined in Chapter 2. MUNCH is the first fully automated reasoner for this logic. MUNCH reduces input formulas to equisatisfiable linear integer arithmetic formulas, and then uses an SMT solver Z3 [de Moura and Bjørner(2008b)] to check the satisfiability of the derived formula.

3.1 Motivation

Interactive theorem provers such as Isabelle [Nipkow et al.(2005)Nipkow, Wenzel, Paulson, and Voelker], Why [Filliâtre and Marché(2007)] or KIV [Balser et al.(2000)Balser, Reif, Schellhorn, Stenzel, and Thums] specify theories of multisets with cardinality constraints. They prove a number of theorems about multisets to enable their use in interactive verification. However, all those tools require a certain level of interaction. Our tool is the first automated theorem prover for multisets with cardinality constraints, which can check satisfiability of formulas belonging to a logic defined in Figure 2.3 entirely automatically.

We have evaluated our implementation on the verification conditions for the correctness of mutable data structure implementations. To prove that a formula F is valid, we check unsatisfiability of the negation of F. If $\neg F$ is satisfiable, that means there is a model in which the original formula evaluates to false. In that case MUNCH generates a model, which can be used to construct a counterexample trace of the checked program.

3.2 MUNCH Implementation

In Chapter 2, we have showed that checking satisfiability of the formulas defined in Figure 2.3 is an NP-complete problem (Theorem 2.25). Our first implementation was based on the

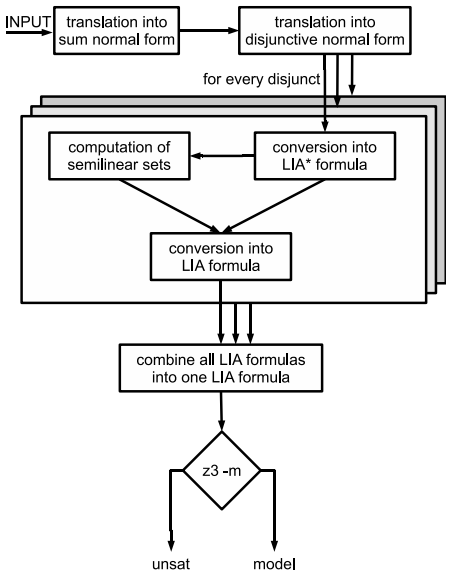

Figure 3.1: Phases in checking formula satisfiability. MUNCH translates the input formula through several intermediate forms, preserving satisfiability in each step.

algorithm in Section 2.7.7, the algorithm used to establish that the decision problem is in NP. But we found that the running times were impractical due to large constants. MUNCH therefore currently uses the conceptually simpler algorithm that reduces a multiset formula to a LIA* formula. The algorithm then, through the computation of semilinear sets, further reduces the LIA* formula to a QFPA formula. Despite its worst-case complexity can be at least NEXPTIME, we have found that the algorithm that uses the semilinear set characterization, when combined with additional simplifications, results in a tool that exhibits acceptable performance. Our implementation often avoids the worst-case complexity of the most critical task, the computation of semilinear sets, by leveraging the special structure of formulas that we need to process (see Section 3.2.2).

3.2.1 System Overview

The MUNCH reasoner is implemented in the Scala [Typesafe(2011)] programming language and currently uses the SMT solver Z3 [de Moura and Bjørner(2008b)] to solve the generated integer linear arithmetic constraints.

Figure 3.1 provides a high-level overview of the reasoner.

Given an input formula F, MUNCH converts it into the sum normal form and then translates it into a LIA* formula. However, having a LIA* formula $P \wedge u \in \{x \mid F(x)\}^*$, we should compute the semilinear set representation of F. The problem with this approach is that computing semilinear sets is expensive. The best know algorithms still run in the exponential time and are fairly complex [Pottier(1991)].

For complexity reasons, we are delaying the computation of semilinear sets. Still, the exponential running time is unavoidable in this approach. Therefore, instead of developing an algorithm which computes semilinear sets for an arbitrary Presburger arithmetic formula, we split a formula into simpler parts for which we can easier compute semilinear sets. Namely, we convert formula F into a disjunctive normal form:

$$F(x) \equiv A_1(x) = b_1 \vee \ldots \vee A_m(x) = b_m$$

This way checking whether $u \in \{x \mid F(x)\}^*$ becomes $= k_1 + \ldots + k_m \wedge \bigwedge_{j=1}^{m} k_j \in \{x \mid A_j(x)\}^*$. The next task is to eliminate the $*$ operator for the formula $k_j \in \{x \mid A_j(x) = b_j\}^*$, where A_j is a conjunction of linear arithmetic atoms. A_j can also be rewritten as a conjunction of equalities by introducing fresh non-negative variables.

3.2.2 Efficient Computation of Semilinear Sets

The MUNCH reasoner is not complete for the full logic described in Figure 2.3. Although the decision procedure is complete, we are avoiding computation of semilinear sets using the Hilbert basis. An implementation, which would preserve completeness and therefore include the computation of semilinear sets for any formula, would significantly increase the running times of our tool. Instead, we noticed that we are mostly using MUNCH to check validity of verification condition generated when proving the correctness of container data structures. The formula that are we generate have very simple structure: they are either about the addition (originating from the disjoint union) or the min and max operator (result of the union and the intersection operators). The min and max operator reduce to the assignment. Another typical formula that often occurs is expressing that a collection is a set. This is represented with $\forall e : s(e) = 1 \vee s(e) = 0$ which at the end again becomes a simple assignment.

In most of the cases, computing a semilinear set is actually computing a linear set which can be done effectively, for example, using the Omega-test [Pugh(1992)]. Since A_j is a conjunction of equations, we use simple rewriting rules. The problem of inequalities expressing that a term is non-negative in most cases is resolved by implicitly using them as non-negative coefficients. As an illustration, consider formula $(u_1, u_2, u_3) \in \{(x, y, z) \mid x < y \wedge z = x + y\}^*$. To compute a semilinear set representing all the non-negative solutions of $x < y \wedge z = x + y$, we first rewrite $x < y$ as $x + l + 1 = y$, where l is a non-negative variable. We further rewrite (x, y, z) as:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ x+y \end{pmatrix} = \begin{pmatrix} x \\ x+l+1 \\ x+x+l+1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} + x \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} + l \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

The semilinear set describing the non-negative solutions of $x < y \wedge z = x + y$ is a linear set $(0, 1, 1) + \{(1, 1, 2), (0, 1, 1)\}^*$. This approach of using equalities and rewriting is highly efficient and works in most of the cases. We also support a simple version of the Omega test.

However, as mentioned earlier, our implementation is not complete for the full logic described in Figure 2.3. There are cases when one cannot avoid the computation of a semilinear set. As an example, it does not work for a formula $u \in \{(x, y, z) \mid 5x + 7y < 6z\}^*$. As a rule, a formula F, where none of the variables have coefficient 1, cannot be rewritten in the above way. Notice also that our tool is always complete for sets, so it can also be used as a complete reasoner for sets with cardinality constraints (with a doubly exponential worst-case bound on running time).

In our experimental results, while processing formulas derived in verification, we did not encounter such a problem. Notice also that our tool is always complete for sets, so it can also be used as a complete reasoner for sets with cardinality constraints (with a doubly exponential worst-case bound on running time).

Assuming that we managed to compute the semilinear sets representing the non-negative solution of $Ax = b$, we construct a Presburger arithmetic formula. As we know, this way the initial multiset constraints problem reduces to satisfiability of quantifier-free Presburger arithmetic. To check satisfiability of such a formula, we invoke the SMT solver Z3 [de Moura and Bjørner(2008a)] with the option "-m". This option ensures that Z3 returns a model in case that the input formula is satisfiable. Since all our transformations are satisfiability preserving, we either return `unsat` or reconstruct a model for the initial multiset formula from the model returned by Z3.

3.3 Examples and Benchmarks

First we illustrate how the MUNCH reasoner works on a simple example, and then we show some benchmarks that we did.

Consider a simple multiset formula $|x \uplus y| = |x| + |y|$. Its validity is proved by showing that $|x \uplus y| \neq |x| + |y|$ is unsatisfiable. We chose such a simple formula so that we can easily present and analyze the tool's output. Figure 3.2 displays the output of MUNCH on this formula. The intermediate formulas in the output correspond to the result of the individual reduction step described in Chapter 2.

Using MUNCH in software verification. We evaluated MUNCH on the verification conditions that we encountered, while proving properties about the container data structures. Those verification conditions are expressible as constraints on sets and multisets. The running times are given in Table 3.3.

The precise description of those verification conditions are given in Figure 3.4.

The main problem we are facing for a more comprehensive evaluation of the MUNCH reasoner is the lack of similar tools and benchmarks. Most benchmarks we were using are originally derived for reasoning about sets. Sometimes those formulas contain conditions that we do not

```
Formula f3:
 NOT (|y PLUS x|  = |y|  + |x|)
Normalized formula f3:
 NOT (k0 = k1 + k2) AND FOR ALL e IN E. (m0(e) = y(e) + x(e)) AND
    (k0, k1, k2) = SUM {e in E,  TRUE } (m0(e), y(e), x(e))
Translated formula f3:
 NOT (k0 = k1 + k2) AND (k0, k1, k2) IN {(m0, y, x) | m0 = y + x }*
No more disjunctions:
  NOT (k0 = k1 + k2) AND k0 = u0  AND k1 = u1 AND k2 = u2 AND
   (u0, u1, u2) IN {(m0, y, x) | m0 = y + x }*
Semilinear set computation :
 ( m0, y, x )  | m0 = y + x,
semilinear set describing it is:
  List(0, 0, 0), List( List(1, 1, 0), List(1, 0, 1))
No more stars:
 NOT (k0 = k1 + k2) AND k0 = u0  AND k1 = u1 AND k2 = u2 AND
 u2 = 0 + 1*nu1 + 0 AND u1 = 0 + 0 + 1*nu0 AND u0 = 0 + 1*nu1 + 1*nu0
 AND ( NOT (mu0 = 0)  OR (nu1 = 0 AND nu0 = 0) )
---------------------
This formula is unsat
---------------------
```

Figure 3.2: Example run of MUNCH on a multiset formula.

Property	#set vars	#multiset vars	time (s)
Efficient emptiness check using sets	2	0	0.40
Efficient emptiness check using multisets	0	2	0.40
Size invariant after insertion (sets)	2	1	0.46
Size invariant after insertion (msets)	0	2	0.40
Size invariant after deleting (msets)	0	2	0.35
Allocation and insertion of 3 (sets)	5	0	3.23
Allocation and insertion of 3 (msets)	0	5	0.40
Allocation and insertion of 4 (sets)	6	0	8.35
Allocation and insertion of 4 (msets)	6	0	0.40

Figure 3.3: Running times for checking verification conditions that arise in proving correctness of container data structures.

VC#	verification condition	property being checked
1	$x \notin \texttt{content} \wedge \texttt{size} = \texttt{card content} \longrightarrow (\texttt{size} = 0 \leftrightarrow \texttt{content} = \emptyset)$	using invariant on size to prove correctness of an efficient emptiness check
2	$x \notin \texttt{content} \wedge \texttt{size} = \texttt{card content} \longrightarrow \texttt{size} + 1 = \texttt{card}(\{x\} \cup \texttt{content})$	maintaining correct size when inserting fresh element
3	$\texttt{size} = \texttt{card content} \wedge \texttt{size1} = \texttt{card}(\{x\} \cup \texttt{content}) \longrightarrow \texttt{size1} \leq \texttt{size} + 1$	maintaining size after inserting any element
4	$\texttt{content} \subseteq \texttt{alloc} \wedge x_1 \notin \texttt{alloc} \wedge x_2 \notin \texttt{alloc} \cup \{x_1\} \wedge x_3 \notin \texttt{alloc} \cup \{x_1\} \cup \{x_2\} \longrightarrow \texttt{card}(\texttt{content} \cup \{x_1\} \cup \{x_2\} \cup \{x_3\}) = \texttt{cardcontent} + 3$	allocating and inserting three objects into a container data structure using sets
5	$\texttt{content} \subseteq \texttt{alloc} \wedge \texttt{card}(\texttt{content} \cup \{x_1\} \cup \{x_2\} \cup \{x_3\}) = \texttt{cardcontent} + 3$	allocating and inserting three objects into a container data structure using multisets

Figure 3.4: Description of the verification conditions proved using MUNCH

need to consider when reasoning about multisets. This can especially be seen in Figure 3.3, when checking allocation and insertion of three elements into a data structure. Proving validity of a multiset formula requires 0.4 seconds. Checking the same property for a data structure implementing a set requires 3.23 seconds. It is due to the fact that the formula is more complicated since we need to also make sure that we are inserting all different elements. Also, to express that we are working with the sets adds the additional disjunctive formula $(\forall e.s(e) = 1 \vee s(e) = 0)$. This contributes to the exponential blow-up. One can see how, in the case of sets, the running times drastically increase.

We could also not compare the MUNCH tool with interactive theorem provers since our tool is completely automated and does not require any interaction.

We plan to continue with the further development of MUNCH. In addition to make it complete, we also plan to incorporate it into software verification systems. This will also enable us to obtain further sets of benchmarks.

4 Decision Procedures for Fractional Collections and Collection Images

In this chapter, we present two extensions of the logic about multisets with cardinality constraints, introduced in Chapter 2. The first extension is a generalization towards a logic that involves collections such as sets, multisets, and fuzzy sets. Element membership in these collections is given by characteristic functions from a finite universe (of unknown size) to a user-defined subset of rational numbers. The logic supports standard operators such as union, intersection, difference, or any operation defined pointwise using mixed linear integer-rational arithmetic. Moreover, it supports the notion of cardinality of the collection, defined as the sum of occurrences of all elements. We describe a decision procedure for the satisfiability problem in this new logic. The decision procedure reduces a formula to a formula in an extension of mixed linear integer-rational arithmetic, with a "star" operator.

The other extension is a logic of uninterpreted functions over sets. A function takes a set as an input and returns a multiset. We add this new construct to the language introduced in Chapter 2. This logic was motivated by examples from verification of data structures. Having a linked data structure, the content of the data structure can be seen as a result of application the function c to the set of nodes, where c is a function that takes a node in a linked data structure and returns the content store in it. We describe a decision procedure for this new logic and show that the satisfiability question is an NEXPTIME-complete problem.

4.1 Motivation for Fractional Collections

A collection of elements can be defined through their characteristic function $f : E \to R$. Inspired by applications in software verification [Kuncak and Rinard(2007)], we assume that the domain E is a finite set but of unknown size. The range R depends on the kind of the collection: for sets, $R = \{0,1\}$; for multisets, $R = \{0,1,2,\ldots,\}$; for fuzzy sets, R is the interval $[0,1]$ of rational numbers, denoted $\mathbb{Q}_{[0,1]}$. With this representation, operations and relations on collections such as union, difference, and subset are all expressed using operations of linear arithmetic. For example, the condition $A \cup B = C$ becomes $\forall e{\in}E.\ \max(A(e),B(e)) = C(e)$, a definition that applies independently of whether A, B are sets, multisets, or fuzzy sets. A distinguishing

feature of our constraints, compared to many other approaches for reasoning about functions $E \to R$, e.g. [Bradley and Manna(2007), Chapter 11], is the presence of the cardinality operator, defined by $|A| = \sum_{e \in \mathbb{E}} A(e)$. The resulting language freely combines the use of linear arithmetic at two levels: the level of individual elements, as in the subformula $\max(A(e), B(e)) = C(e)$, and the level of sizes of collections, as in the formula $|A \cup B| + |A \cap B| = |A| + |B|$. The language subsumes constraints such as quantifier-free Boolean Algebra with Presburger Arithmetic [Kuncak and Rinard(2007)] and therefore contains both set algebra and integer linear arithmetic. It also subsumes decidable constraints on multisets with cardinality bounds from Chapter 2, since in this new language we can express the condition $(\forall e.int(A(e)) \wedge A(e) \geq 0)$, i.e. that the number of occurrences $A(e)$ for each element e is a non-negative integer number. Moreover, these new constraints can express the condition $\forall e.(0 \leq A(e) \leq 1)$, which makes them appropriate for modeling fuzzy sets.

Analogously to the algorithm described in Chapter 2, a decision procedure for the new logic is based on a translation of a formula with collections and cardinality constraints into a conjunction of a mixed linear integer-rational arithmetic (MLIRA) formula and a new form of condition, denoted $\vec{u} \in \{\vec{v} \mid F(\vec{v})\}^*$. Here the star operator has the same semantics as in Chapter 2. Therefore, $\{\vec{v} \mid F(\vec{v})\}^*$ denotes the closure under vector addition of the set of solution vectors $\vec{v}$ of the MLIRA formula F. Formally,

$$\vec{u} \in \{\vec{v} \mid F(\vec{v})\}^* \leftrightarrow \exists K \in \{0,1,2,\ldots\}.\, \exists \vec{v}_1, \ldots, \vec{v}_K.\, \vec{u} = \sum_{i=1}^{K} \vec{v}_i \wedge \bigwedge_{i=1}^{K} F(\vec{v}_i)$$

In contrast to Chapter 2, the formula F in this paper is not restricted to integers, but can be arbitrary MLIRA formula. Consequently, we are faced with the problem of solving an extension of satisfiability of MLIRA formulas with the conditions $\vec{u} \in \{\vec{v} \mid F(\vec{v})\}^*$ where F is an arbitrary MLIRA formula. To solve this problem, we describe a finite and effectively computable representation of the solution set $S = \{\vec{v} \mid F(\vec{v})\}$. We use this representation to express the condition $\vec{u} \in S^*$ as a new MLIRA formula. This gives a "star elimination" algorithm. As one consequence, we obtain a unified decision procedure for sets, multisets, and fuzzy sets in the presence of the cardinality operator.

4.2 Examples

Figure 4.1 shows small example formulas over sets, multisets, and fuzzy sets that are expressible in our logic. The examples for sets and multisets are based on verification conditions from software verification [Kuncak and Rinard(2007)]. The remaining examples illustrate basic differences in valid formulas over multisets and fuzzy sets.

We illustrate our technique on one of the examples shown in Figure 4.1: we show that formula $\forall e.U(e) = 1 \to |A \cap B| + |A \cup B| \leq |A| + |U|$ is valid where U, A, and B are fuzzy sets. To prove formula validity, we prove unsatisfiability of its negation, conjoined with the constraints

Examples of constraints on sets. For each set variable s we assume the constraint $\forall e.(s(e) = 0 \vee s(e) = 1)$.

formula	informal description
$x \notin \texttt{content} \wedge \texttt{size} = \texttt{card content} \longrightarrow$ $(\texttt{size} = 0 \leftrightarrow \texttt{content} = \emptyset)$	using invariant on size to prove correctness of an efficient emptiness check
$x \notin \texttt{content} \wedge \texttt{size} = \texttt{card content} \longrightarrow$ $\texttt{size} + 1 = \texttt{card}(\{x\} \cup \texttt{content})$	maintaining correct size when inserting fresh element into set
$\texttt{size} = \texttt{card content} \wedge$ $\texttt{size1} = \texttt{card}(\{x\} \cup \texttt{content}) \longrightarrow$ $\texttt{size1} \leq \texttt{size} + 1$	maintaining size after inserting an element into set
$\texttt{content} \subseteq \texttt{alloc} \wedge$ $x_1 \notin \texttt{alloc} \wedge$ $x_2 \notin \texttt{alloc} \cup \{x_1\} \wedge$ $x_3 \notin \texttt{alloc} \cup \{x_1\} \cup \{x_2\} \longrightarrow$ $\texttt{card}(\texttt{content} \cup \{x_1\} \cup \{x_2\} \cup \{x_3\}) =$ $\texttt{card content} + 3$	allocating and inserting three objects into a container data structure
$\texttt{content} \subseteq \texttt{alloc0} \wedge x_1 \notin \texttt{alloc0} \wedge$ $\texttt{alloc0} \cup \{x_1\} \subseteq \texttt{alloc1} \wedge x_2 \notin \texttt{alloc1} \wedge$ $\texttt{alloc1} \cup \{x_2\} \subseteq \texttt{alloc2} \wedge x_3 \notin \texttt{alloc2} \longrightarrow$ $\texttt{card}(\texttt{content} \cup \{x_1\} \cup \{x_2\} \cup \{x_3\}) =$ $\texttt{card content} + 3$	allocating and inserting at least three objects into a container data structure
$x \in C \wedge C_1 = (C \setminus \{x\}) \wedge$ $\texttt{card}(\texttt{alloc1} \setminus \texttt{alloc0}) \leq 1 \wedge$ $\texttt{card}(\texttt{alloc2} \setminus \texttt{alloc1}) \leq \texttt{card}\, C_1 \longrightarrow$ $\texttt{card}(\texttt{alloc2} \setminus \texttt{alloc0}) \leq \texttt{card}\, C$	bound on the number of allocated objects in a recursive function that incorporates container C into another container

Examples of constraints on multisets. For each multiset variable m we assume the constraint $\forall e.int(m(e)) \wedge A(e) \geq 0$.

$\texttt{size} = \texttt{card content} \wedge$ $\texttt{size1} = \texttt{card}(\{x\} \uplus \texttt{content}) \longrightarrow$ $\texttt{size1} = \texttt{size} + 1$	maintaining size after inserting an element into multiset

Examples of constraints on fuzzy sets. For each fuzzy set variable f we assume the constraint $\forall e.0 \leq f(e) \leq 1$.

$2\|A\| \neq 2\|B\| + 1$	example formula valid over multisets but invalid over fuzzy sets
$(\forall e.U(e) = 1) \rightarrow \|A \cap B\| + \|A \cup B\| \leq \|A\| + \|U\|$	example formula valid over fuzzy sets but invalid over multisets
$(\forall e.C(e) = \lambda A(e) + (1 - \lambda)B(e)) \rightarrow$ $A \cap B \subseteq C \subseteq A \cup B$	basic property of convex combination of fuzzy sets [Zadeh(1965)], for any fixed constant $\lambda \in [0,1]$

Figure 4.1: Example constraints in our class.

ensuring that the collections are fuzzy sets:

$$\forall e.U(e) = 1 \ \wedge \ |A| + |U| < |A \cap B| + |A \cup B| \ \wedge$$
$$\forall e.0 \leq A(e) \leq 1 \ \wedge \ \forall e.0 \leq B(e) \leq 1 \ \wedge \ \forall e.0 \leq U(e) \leq 1$$

As previously, we eliminate the collections and reduce the formula to a formula in the extended mixed linear integer-rational arithmetic. The reduction follows the algorithm we have seen in Chapter 2. We first reduce the formula to the normal form. We flatten the formula by introducing fresh variables n_i for each cardinality operator. The formula reduces to:

$$n_1 + n_2 < n_3 + n_4 \ \wedge \ n_1 = |U| \ \wedge \ n_2 = |A| \ \wedge \ n_3 = |A \cap B| \ \wedge \ n_4 = |A \cup B| \ \wedge$$
$$\forall e.U(e) = 1 \ \wedge \ \forall e.0 \leq A(e) \leq 1 \ \wedge \ \forall e.0 \leq B(e) \leq 1 \ \wedge \ \forall e.0 \leq U(e) \leq 1$$

We next apply the definition of the cardinality operator, $|C| = \sum_{e \in \mathbb{E}} C(e)$:

$$n_1 + n_2 < n_3 + n_4 \ \wedge \ n_1 = \textstyle\sum_{e \in \mathbb{E}} U(e) \ \wedge \ n_2 = \sum_{e \in \mathbb{E}} A(e) \ \wedge$$
$$n_3 = \textstyle\sum_{e \in \mathbb{E}} (A \cap B)(e) \ \wedge \ n_4 = \sum_{e \in \mathbb{E}} (A \cup B)(e) \ \wedge$$
$$\forall e.U(e) = 1 \ \wedge \ \forall e.0 \leq A(e) \leq 1 \ \wedge \ \forall e.0 \leq B(e) \leq 1 \ \wedge \ \forall e.0 \leq U(e) \leq 1$$

Operators $\cup$ and $\cap$ are defined pointwise using ite operator:

$(C_1 \cup C_2)(e) = \max\{C_1(e), C_2(e)\} = \mathsf{ite}(C_1(e) \leq C_2(e), C_2(e), C_1(e))$

$(C_1 \cap C_2)(e) = \min\{C_1(e), C_2(e)\} = \mathsf{ite}(C_1(e) \leq C_2(e), C_1(e), C_2(e))$, where $\mathsf{ite}(A, B, C)$ is the standard *if-then-else* operator, denoting B when A is true and C otherwise. Using these definitions, the example formula becomes:

$$n_1 + n_2 < n_3 + n_4 \ \wedge \ n_1 = \sum_{e \in \mathbb{E}} U(e) \ \wedge \ n_2 = \sum_{e \in \mathbb{E}} A(e) \ \wedge$$
$$n_3 = \sum_{e \in \mathbb{E}} \mathsf{ite}(A(e) \leq B(e), A(e), B(e)) \ \wedge \ n_4 = \sum_{e \in \mathbb{E}} \mathsf{ite}(A(e) \leq B(e), B(e), A(e)) \ \wedge$$
$$\forall e.U(e) = 1 \ \wedge \ \forall e.0 \leq A(e) \leq 1 \ \wedge \ \forall e.0 \leq B(e) \leq 1 \ \wedge \ \forall e.0 \leq U(e) \leq 1$$

Using vectors of integers, we then group all the sums into one, and also group all universally quantified constraints:

$$n_1 + n_2 < n_3 + n_4 \wedge \big(n_1, n_2, n_3, n_4\big) =$$
$$\sum_{e \in \mathbb{E}} \big(U(e), \ A(e), \ \mathsf{ite}(A(e) \leq B(e), A(e), B(e)), \ \mathsf{ite}(A(e) \leq B(e), B(e), A(e))\big)$$
$$\wedge \ \forall e. \ \big(U(e) = 1 \ \wedge \ 0 \leq A(e) \leq 1 \ \wedge \ 0 \leq B(e) \leq 1 \ \wedge \ 0 \leq U(e) \leq 1\big)$$

As stated in Theorem 4.2 (a counterpart of Theorem 2.4), the last formula is equisatisfiable with

$$n_1 + n_2 < n_3 + n_4 \wedge (n_1, n_2, n_3, n_4) \in$$
$$\{\big(u, a, \mathsf{ite}(a \leq b, a, b), \mathsf{ite}(a \leq b, b, a)\big) \mid u = 1 \ \wedge \ 0 \leq a \leq 1 \ \wedge \ 0 \leq b \leq 1\}^*$$

In this chapter we will explore general techniques for solving such satisfiability problems that contain a MLIRA formula and a star operator applied to another MLIRA formula. We next illustrate some of the ideas of the general technique, taking several shortcuts to keep the exposition brief.

Because the value of the variable u is determined ($u = 1$), we can simplify the last formula to:

$$n_1 + n_2 < n_3 + n_4 \ \wedge \ (n_1, n_2, n_3, n_4) \in S^*$$

where $S = \{\big(1, a, \mathsf{ite}(a \leq b, a, b), \mathsf{ite}(a \leq b, b, a)\big) \mid 0 \leq a \leq 1 \wedge 0 \leq b \leq 1\}$. By case analysis on $a \leq b$, we conclude $S = S_1 \cup S_2$ for

$$S_1 = \{(1, a, a, b) \mid 0 \leq a \leq 1 \wedge 0 \leq b \leq 1 \wedge a \leq b\}$$
$$S_2 = \{(1, a, b, a) \mid 0 \leq a \leq 1 \wedge 0 \leq b \leq 1 \wedge b < a\}$$

This eliminates the ite expressions and we have:

$$n_1 + n_2 < n_3 + n_4 \wedge (n_1, n_2, n_3, n_4) \in (S_1 \cup S_2)^*$$

By definition of star operator, the last condition is equivalent to

$$n_1 + n_2 < n_3 + n_4 \ \wedge \ (n_1, n_2, n_3, n_4) = (n_1^1, n_2^1, n_3^1, n_4^1) + (n_1^2, n_2^2, n_3^2, n_4^2) \ \wedge \\ (n_1^1, n_2^1, n_3^1, n_4^1) \in S_1^* \ \wedge \ (n_1^2, n_2^2, n_3^2, n_4^2) \in S_2^*$$

Let us characterize the condition $(n_1^1, n_2^1, n_3^1, n_4^1) \in S_1^*$. Let K_1 denote the number of vectors in S_1 whose sum is $(n_1^1, n_2^1, n_3^1, n_4^1)$. By definition of the star operator, there are $a_1^1, \ldots, a_{K_1}^1$ and $b_1^1, \ldots, b_{K_1}^1$ such that $0 \leq a_i^1 \leq b_i^1 \leq 1$ and

$$(n_1^1, n_2^1, n_3^1, n_4^1) = \sum_{i=1}^{K_1} (1, a_i^1, a_i^1, b_i^1)$$

We obtain that $n_1^1 = K_1$, $n_2^1 = n_3^1 = \sum_{i=1}^{K_1} a_i^1 =: A_1$, $n_4 = \sum_{i=1}^{K_1} b_i^1 =: B_1$. The other case for S_2 is analogous and we derive $(n_1^2, n_2^2, n_3^2, n_4^2) = (K_2, A_2, B_2, A_2)$.

This way the star operator is "hidden" in the variables A_1 and B_1 and the example formula reduces to:

$$n_1 + n_2 < n_3 + n_4 \wedge (n_1, n_2, n_3, n_4) = (K_1, A_1, A_1, B_1) + (K_2, A_2, B_2, A_2)$$

After eliminating n_i variables, the formula becomes $K_1 + K_2 < B_1 + B_2$. Apply the definitions of B_i and stating the bounding properties of b_i^j, we obtain the following formula:

$$K_1 + K_2 < \sum_{i=1}^{K_1} b_i^1 + \sum_{i=1}^{K_2} b_i^2 \ \wedge \ \bigwedge_{i=1}^{K_1} b_i^1 \leq 1 \wedge \bigwedge_{i=1}^{K_2} b_i^2 \leq 1$$

In this case, it is easy to see that the resulting formula is contradictory, since $K_1 + K_2$ is an integer. The unsatisfiable formula proves that the initial formula is valid over fuzzy sets. In general, such formulas are equivalent to existentially quantified MLIRA formulas, despite the fact that their initial formulation involves sums with parameters such as K_1 and K_2. This is possible thanks to the special structure of the sets of solutions of MLIRA formulas.

Having seen the way to prove validity, we illustrate how to produce counterexamples by showing that the original formula is invalid over multisets. Restricting the range of each collection to integers and using the same reduction, we derive formula

$$n_1 + n_2 < n_3 + n_4 \wedge$$
$$(n_1, n_2, n_3, n_4) \in \{\big(1, a, \mathsf{ite}(a \leq b, a, b), \mathsf{ite}(a \leq b, b, a)\big) \mid a, b \in \mathbb{N}\}^*$$

Applying again a similar case analysis, we deduce $K_1 + K_2 < \sum_{i=1}^{K_1} b_i^1 + \sum_{i=1}^{K_2} b_i^2$ where all b_i^j's are non-negative integers. This formula is satisfiable, for example, with a satisfying variable assignment $K_1 = 1, b_1^1 = 2$ and $K_2 = 0$. Applying the proof of Theorem 4.2, we construct a multiset counterexample. Because $K_2 = 0$, no vector from S_2 contributes to sum, and we consider only S_1. Variable K_1 denotes the number of elements of a domain set E, so we consider the domain set $E = \{e_1\}$. Multisets A, B and U are defined by $A(e_1) = 1$, $B(e_1) = 2$, and $U(e_1) = 1$. It can easily be verified that this is a counterexample for validity of the formula over multisets.

4.3 From Collections to Stars

This section describes the translation from constraints on collections to constraints that use the star operator. We first present the syntax of our constraints and clarify the semantics of selected constructs (the semantics of the remaining constructs can be derived from their translation into simpler ones).

We model each collection c as a function whose domain is a finite set E of unknown size and whose range is the set of rational numbers. When the constraints imply that the range of c is $\{0, 1\}$, then c models sets, when the range of c are non-negative integers, then c denotes standard multisets (bags), in which an element can occur multiple times. We call the number of occurrences of an element e, denoted $c(e)$, the *multiplicity* of an element. When the range of c is restricted to be in interval $[0, 1]$, then c describes a fuzzy set [Zadeh(1965)].

In addition to standard operations on collections (such as plus, union, intersection, difference), we also allow the cardinality operator, defined as $|c| = \sum_{e \in \mathbb{E}} c(e)$. This is the desired definition for sets and multisets, and we believe it is a natural notion for fuzzy sets over a finite universe E. Figure 4.2 shows a context-free grammar of our formulas involving collections.

Semantics of some less commonly known operators is defined as follows. The $\mathsf{set}(C)$ operator takes as an argument collection C and returns the set of all elements for which $C(e)$ is positive.

top-level formulas:

$$\begin{array}{rcl} F & ::= & A \mid F \wedge F \mid \neg F \\ A & ::= & C{=}C \mid C \subseteq C \mid \forall e.\mathsf{F}^{\mathsf{in}} \mid \mathsf{A}^{\mathsf{out}} \end{array}$$

outer linear arithmetic formulas:

$$\begin{array}{rcl} \mathsf{F}^{\mathsf{out}} & ::= & \mathsf{A}^{\mathsf{out}} \mid \mathsf{F}^{\mathsf{out}} \wedge \mathsf{F}^{\mathsf{out}} \mid \neg \mathsf{F}^{\mathsf{out}} \\ \mathsf{A}^{\mathsf{out}} & ::= & \mathsf{t}^{\mathsf{out}} \leq \mathsf{t}^{\mathsf{out}} \mid \mathsf{t}^{\mathsf{out}} {=} \mathsf{t}^{\mathsf{out}} \mid (\mathsf{t}^{\mathsf{out}}, \ldots, \mathsf{t}^{\mathsf{out}}) {=} \sum_{\mathsf{F}^{\mathsf{in}}} (\mathsf{t}^{\mathsf{in}}, \ldots, \mathsf{t}^{\mathsf{in}}) \\ \mathsf{t}^{\mathsf{out}} & ::= & k \mid \mid C \mid \mid K \mid \mathsf{t}^{\mathsf{out}} + \mathsf{t}^{\mathsf{out}} \mid K \cdot \mathsf{t}^{\mathsf{out}} \mid \lfloor \mathsf{t}^{\mathsf{out}} \rfloor \mid \mathsf{ite}(\mathsf{F}^{\mathsf{out}}, \mathsf{t}^{\mathsf{out}}, \mathsf{t}^{\mathsf{out}}) \end{array}$$

inner linear arithmetic formulas:

$$\begin{array}{rcl} \mathsf{F}^{\mathsf{in}} & ::= & \mathsf{A}^{\mathsf{in}} \mid \mathsf{F}^{\mathsf{in}} \wedge \mathsf{F}^{\mathsf{in}} \mid \neg \mathsf{F}^{\mathsf{in}} \\ \mathsf{A}^{\mathsf{in}} & ::= & \mathsf{t}^{\mathsf{in}} \leq \mathsf{t}^{\mathsf{in}} \mid \mathsf{t}^{\mathsf{in}} {=} \mathsf{t}^{\mathsf{in}} \\ \mathsf{t}^{\mathsf{in}} & ::= & c(e) \mid K \mid \mathsf{t}^{\mathsf{in}} + \mathsf{t}^{\mathsf{in}} \mid K \cdot \mathsf{t}^{\mathsf{in}} \mid \lfloor \mathsf{t}^{\mathsf{in}} \rfloor \mid \mathsf{ite}(\mathsf{F}^{\mathsf{in}}, \mathsf{t}^{\mathsf{in}}, \mathsf{t}^{\mathsf{in}}) \end{array}$$

expressions about collections:

$$C ::= c \mid \emptyset \mid C \cap C \mid C \cup C \mid C \uplus C \mid C \setminus C \mid C \setminus\setminus C \mid \mathsf{set}(C)$$

terminals:

c - collection variable; e - index variable (fixed)

k - rational variable; K - rational constant

Figure 4.2: Quantifier-Free Formulas about Collection with Cardinality Operator

To constrain a variable s to denote a set, use formula $\forall e.s(e) = 0 \vee s(e) = 1$. To constraint a variable m to denote a multiset, use formula $(\forall e.int(m(e)) \wedge m(e) \geq 0)$. Here $int(x)$ is a shorthand for $\lfloor x \rfloor = x$ where $\lfloor x \rfloor$ is the largest integer smaller than or equal to x.

The novelty of constraints in Figure 4.2 compared to the language in Figure 2.3 is the presence of the floor operator $\lfloor x \rfloor$ and not only integer but also rational constants. All variables in our current language are interpreted over rationals, but any of them can be restricted to be integer using the constraint $int(x)$. In addition, there are no restrictions that the inner terms should be only non-negative. This way the language in Figure 4.2 is a generalization of the language in Figure 2.3.

To reduce reasoning about collections to reasoning in linear arithmetic with stars, we follow the idea from Chapter 2 and convert a formula to the sum normal form.

Definition 4.1 *A formula is in sum normal form iff it is of the form*

$$P \wedge (u_1, \ldots, u_n) = \sum_{e \in \mathbb{E}} (t_1, \ldots, t_n) \wedge \forall e.F$$

where P is a quantifier-free linear arithmetic formula with no collection variables, and where variables in $t_1, \ldots, t_n$ and F occur only as expressions of the form $c(e)$ for a collection variable c and e the fixed index variable. Formula F can also contain terms of the form $\lfloor t \rfloor$.

To transform a formula in the logic of Figure 4.2 into its sum normal form, we use the algorithm

given in Figure 2.4. Analogously, the translation results with the corresponding theorem:

Theorem 4.2 *A formula* $(u_1,\ldots,u_n) = \sum_{e\in\mathbb{E}}(t_1,\ldots,t_n) \wedge \forall e.F$ *is equisatisfiable with the formula* $(u_1,\ldots,u_n) \in \{(t'_1,\ldots,t'_n) \mid x_i \in \mathbb{Q} \wedge F'\}^*$ *where* t'_j *and* F' *are* t_j *and* F *respectively in which each* $c_i(e)$ *is replaced by a fresh variable* x_i.

Proof. Identical to the proof of Theorem 2.4.

Thanks to Theorem 4.2, in the rest of the paper we investigate the satisfiability problem for such formulas, whose syntax is given in Figure 4.3. These formulas are sufficient to check satisfiability for formulas in Figure 4.2. We extend the semantics of the atom $\vec{u} \in \{\vec{x} \mid F\}^* - \vec{u}$ is a finite sum of the solution vectors of formula F. They do not need to be non-negative.

MLIRA* formulas: $F_0 \wedge (u_1,\ldots,u_n) \in \{(x_1,\ldots,x_n) \mid F\}^*$ (free variables of F are among x)
MLIRA formulas:

$$
\begin{array}{lcl}
F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F_1 \\
A & ::= & T_1 \leq T_2 \mid T_1 = T_2 \\
T & ::= & k \mid C \mid T_1 + T_2 \mid C \cdot T_1 \mid \lfloor T \rfloor \mid \mathsf{ite}(F, T_1, T_2)
\end{array}
$$

terminals: k - rational variable; C - rational constant

Figure 4.3: Syntax of Mixed Integer-Rational Linear Arithmetic with Star

4.4 Separating Mixed Constraints

As justified in previous sections, we consider the satisfiability problem for $G(\vec{r},\vec{w}) \wedge \vec{w} \in \{\vec{x} \mid F(\vec{x})\}^*$ where F and G are quantifier-free, mixed linear integer-rational arithmetic (MLIRA) formulas.

Our goal is to give an algorithm for constructing another MLIRA formula F' such that $\vec{w} \in \{\vec{x} \mid F(\vec{x})\}^*$ is equivalent to $\exists\vec{w}'.F'(\vec{w}',\vec{w})$. This will reduce the satisfiability problem to the satisfiability of $G(\vec{r},\vec{w}) \wedge F'(\vec{w}',\vec{w})$.

As a first step towards this goal, this section shows how to represent the set $\{\vec{x} \mid F(\vec{x})\}$ using solutions of pure integer constraints and solutions of pure rational constraints. We proceed in several steps.

Step 1. Eliminate the floor functions from F using integer and real variables, applying from left to right the equivalence

$$C(\lfloor t \rfloor) \leftrightarrow \exists y_Q \in \mathbb{Q}.\exists y_Z \in \mathbb{Z}.\ t = y_Q \wedge y_Z \leq y_Q < y_Z + 1 \wedge C(y_Z)$$

The result is an equivalent formula without the floor operators, where some of the variables are restricted to be integer.

Step 2. Transform F into linear programming problems, as follows. First, eliminate if-then-else expressions by introducing fresh variables and using disjunction (as in Chapter 2). Then transform formula to negation normal form. Eliminate $t_1 = t_2$ by transforming it into $t_1 \leq t_2 \wedge t_2 \leq t_1$. Eliminate $t_1 \neq t_2$ by transforming it into $t_1 < t_2 \vee t_2 < t_1$. Following [Dutertre and de Moura(2006b), Section 3.3], replace each $t_1 < t_2$ with $t_1 + \delta \leq t_2$ where δ is a special variable (the same for all strict inequalities), for which we require $0 < \delta \leq 1$. We obtain for some d matrices A_i for $1 \leq i \leq d$ such that

$$F(\vec{x}) \leftrightarrow \exists \vec{y}^Z \in \mathbb{Z}^{d_Z}.\exists \vec{y}^Q \in \mathbb{Q}^{d_Q}.\exists \delta \in \mathbb{Q}_{(0,1]}.\ \bigvee_{i=1}^{d} A_i \cdot (\vec{x}, \vec{y}^Z, \vec{y}^Q, \delta) \leq \vec{b}_i$$

where $A_i \cdot (\vec{x}, \vec{y}^Z, \vec{y}^Q, \delta)$ denotes multiplication of matrix A_i by the vector $(\vec{x}, \vec{y}^Z, \vec{y}^Q, \delta)$ obtained by stacking vectors $\vec{x}$, $\vec{y}^Z$, $\vec{y}^Q$ and the value δ.

Step 3. Represent the rational variables $\vec{x}$, $\vec{y}^Q$ as a sum of its integer part and its rational part from $\mathbb{Q}_{[0,1)}$, obtaining

$$F(\vec{x}) \leftrightarrow \Big(\exists (\vec{x}^Z, \vec{y}^Z) \in \mathbb{Z}^{d'_Z}.\exists (\vec{x}^R, \vec{y}^R) \in \mathbb{Q}_{[0,1)}^{d'_Q}.\exists \delta \in \mathbb{Q}_{(0,1]}. \\ \vec{x} = \vec{x}^Z + \vec{x}^R \wedge \bigvee_{i=1}^{d} A'_i \cdot (\vec{x}^Z, \vec{y}^Z, \vec{x}^R, \vec{y}^R, \delta) \leq \vec{b_i}'\Big)$$

Note that $\vec{w} \in \{\vec{x} \mid \exists \vec{y}.H(\vec{x}, \vec{y})\}^*$ is equivalent to

$$\exists \vec{w}'.(\vec{w}, \vec{w}') \in \{(\vec{x}, \vec{y}) \mid H(\vec{x}, \vec{y})\}^*$$

In other words, we can push existential quantifiers to the top-level of the formula. Therefore, the original problem (after renaming) becomes

$$G(\vec{r}, \vec{w}) \wedge \exists \vec{z}.\ (\vec{u}^Z, \vec{u}^Q, \Delta) \in \\ \{(\vec{x}^Z, \vec{x}^R, \delta) \mid \bigvee_{i=1}^{d} A_i \cdot (\vec{x}^Z, \vec{x}^R, \delta) \leq \vec{b}_i,\ \vec{x}^Z \in \mathbb{Z}^{d_Z}, \vec{x}^R \in \mathbb{Q}_{[0,1)}^{d_R}, \delta \in \mathbb{Q}_{(0,1]}\}^*$$

where the vector $\vec{z}$ contains a subset of variables $\vec{u}^Z, \vec{u}^Q, \Delta$.

Step 4. Separate integer and rational parts, as follows. Consider one of the disjuncts $A \cdot (\vec{x}^Z, \vec{y}^R, \delta) \leq \vec{b}$. For $A = [A_Z\ A_R\ c]$ this linear condition can be written as $A_Z\vec{x}^Z + A_R\vec{x}^R + \vec{c}\delta \leq \vec{b}$, that is

$$A_R\vec{x}^R + \vec{c}\delta \leq \vec{b} - A_Z\vec{x}^Z \tag{4.1}$$

Because the right-hand side is integer, for $\vec{a}$ denoting $\lceil A_R\vec{x}^R + \vec{c}\delta \rceil$ (left-hand side rounded up), the equation becomes $A_R\vec{x}^R + \vec{c}\delta \leq \vec{a} \leq \vec{b} - A_Z\vec{x}^Z$. Because $\vec{x}^R \in \mathbb{Q}_{[0,1]}^{d_Q}, \delta \in \mathbb{Q}_{(0,1]}$, vector $\vec{a}$

is bounded by the norm M_1 of the matrix $[A_R\ \vec{c}]$. Formula (4.1) is therefore equivalent to the finite disjunction

$$\bigvee_{\vec{a}\in\mathbb{Z}^d, ||\vec{a}||\leq M_1} A_Z\vec{x}^Z \leq \vec{b}-\vec{a} \wedge A_R\vec{x}^R + \vec{c}\delta \leq \vec{a} \tag{4.2}$$

Note that each disjunct is a conjunction of a purely integer constraint and a purely rational constraint.

Step 5. Propagate star through disjunction, using the property

$$\vec{w} \in \{\vec{x} \mid \bigvee_{i=1}^{n} H_i(\vec{x})\}^* \leftrightarrow \exists \vec{w}_1,\ldots,\vec{w}_n.\ \vec{w} = \sum_{i=1}^{n} \vec{w}_i \ \wedge \ \bigwedge_{i=1}^{n} \vec{w}_i \in \{\vec{x} \mid H_i(\vec{x})\}^*$$

The final result is an equivalent conjunction of a MLIRA formula and an existentially quantified conjunction of formulas of the form

$$(\vec{u}^Z, \vec{u}^Q, \Delta) \in \{(\vec{x}^Z, \vec{x}^R, \delta) \mid \begin{array}{l} A_Z\vec{x}^Z \leq \vec{b}_Z, A_R\cdot(\vec{x}^R,\delta) \leq \vec{b}_R, \\ \vec{x}^Z \in \mathbb{Z}^{d_Z}, \vec{x}^R \in \mathbb{Q}^{d_R}_{[0,1)}, \delta \in \mathbb{Q}_{(0,1]}\}^* \end{array} \tag{4.3}$$

4.4.1 Example

In Section 4.2 we briefly outlined how to checked the validity of formulas about fractional collections. The reasoning used in that example was fairly simple and tailor-made. In this section, we will show how can any given formula be reduced to a conjunction of MLIRA* formulas. To do that, we apply the algorithm which was just introduced in the previous subsection. We will apply it blindly and without using further simplifications or any additional reasoning on the same formula as in the Section 4.2.

We first need to reduce the formula that reasons about fuzzy sets to the formula of the form $G(\vec{r},\vec{w}) \wedge \vec{w} \in \{\vec{x} \mid F(\vec{x})\}^*$. The translation of the original formula to the required form is described in details in Figure 2.4 and after the translation we obtain the formula:

$$\begin{array}{l} n_1 + n_2 < n_3 + n_4 \wedge (n_1,n_2,n_3,n_4) \in \\ \qquad \left\{\left(u, a, \mathsf{ite}(a\leq b, a, b), \mathsf{ite}(a\leq b, b, a)\right) \mid u = 1 \ \wedge\ 0 \leq a \leq 1 \ \wedge\ 0 \leq b \leq 1\right\}^* \end{array}$$

which reduces to

$$\begin{array}{l} n_1 + n_2 < n_3 + n_4 \wedge (n_1,n_2,n_3,n_4) \in \{(u,a,t_1,t_2) \mid \\ \qquad u = 1 \ \wedge\ 0 \leq a \leq 1 \ \wedge\ 0 \leq b \leq 1 \wedge t_1 = \mathsf{ite}(a\leq b, a, b) \wedge t_2 = \mathsf{ite}(a\leq b, b, a)\}^* \end{array}$$

This formula has the desired form so we can apply the algorithm from the previous section that will separate it into reasoning about integers and small rationals. First we need to transform

the formula $F(u,a,t_1,t_2) \equiv u = 1 \wedge 0 \leq a \leq 1 \wedge 0 \leq b \leq 1 \wedge t_1 = \mathsf{ite}(a \leq b, a, b) \wedge t_2 = \mathsf{ite}(a \leq b, b, a)$. As the floor function does not occur in F, Step 1 is not applied. In Step 2 we transform F into a linear programming problem. After eliminating if-then-else expression, F becomes $F(u,a,t_1,t_2) \equiv ((u = 1 \wedge 0 \leq a \leq 1 \wedge 0 \leq t_2 \leq 1 \wedge a \leq t_2 \wedge t_1 = a) \vee (u = 1 \wedge 0 \leq a \leq 1 \wedge 0 \leq t_1 \leq 1 \wedge t_1 < a \wedge t_2 = a))$. Next we transform equalities and strict inequalities into equalities, and finally we obtain the following formula:

$$
\begin{aligned}
&F(u,a,t_1,t_2) \equiv \\
&((u \leq 1 \wedge 1 \leq u \wedge 0 \leq a \leq 1 \wedge 0 \leq t_2 \leq 1 \wedge a \leq t_2 \wedge t_1 \leq a \wedge a \leq t_1) \\
&\qquad\qquad \vee (u \leq 1 \wedge 1 \leq u \wedge 0 \leq a \leq 1 \wedge 0 \leq t_1 \leq 1 \wedge t_1 + \delta \leq a \wedge a \leq t_2 \wedge t_2 \leq a))
\end{aligned}
$$

This way we obtain the following representation of F:

$$F(u,a,t_1,t_2) \equiv \exists \delta \in \mathbb{Q}_{(0,1]}.\ \bigvee_{i=1}^{2} A_i \cdot (u,a,t_1,t_2,\delta) \leq \vec{b}_i$$

where $A_i = \begin{bmatrix} A_i^Z & A_i^Q & A_i^\delta \end{bmatrix}$ and those matrices has the following values:

$$
A_1^Z = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},\ A_1^Q = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix},\ A_1^\delta = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},\ \vec{b}_1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},
$$

$$
A_2^Z = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},\ A_2^Q = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \\ -1 & 1 & 0 \\ 1 & 0 & -1 \\ -1 & 0 & 1 \end{bmatrix},\ A_2^\delta = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},\ \vec{b}_2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.
$$

A variable u is an integer variable, while a, t_1 and t_2 are rational variables. In Step 3 we

represent each rational variable as a sum of its integer and rational part.

$$F(u,a,t_1,t_2) \equiv \exists \vec{v}^Z \in \mathbb{Z}^3.\exists \vec{v}^R \in \mathbb{Q}^3_{[0,1)}.\exists \delta \in \mathbb{Q}_{(0,1]}.$$
$$(a,t_1,t_2) = \vec{v}_Z + \vec{v}_R \ \wedge \bigvee_{i=1}^{2} A'_i \cdot (u, \vec{v}_Z, \vec{v}_Q, \delta) \leq \vec{b}'_i)$$

We think about the vector $\vec{v}_Z$ as an integer part of a, t_1 and t_2 variables: $\vec{v}_Z = (a^Z, t_1^Z, t_2^Z)$, while $\vec{v}_R$ represents their rational part. Matrices A'_1 and A'_2 are constructed using previously derived matrices: $A'_i = \begin{bmatrix} A_i^Z & A_i^Q & A_i^Q & A_i^\delta \end{bmatrix}$, while $\vec{b}'_i = \vec{b}_i$.

Finally, we push the existential quantifiers that appear in formula F to the top level formula and we obtain the following:

$$n_1 + n_2 < n_3 + n_4 \wedge \exists (n_2^Z, n_3^Z, n_4^Z) \in \mathbb{Z}^3.\exists \vec{(} n_2^Q, n_3^Q, n_4^Q), \in \mathbb{Q}^3_{\geq 0}.\exists \Delta \in \mathbb{Q}_{>0}.$$
$$(n_1, n_2^Z, n_3^Z, n_4^Z, n_2^Q, n_3^Q, n_4^Q, \Delta) \in \{\big(u, \vec{v}_Z, \vec{v}_Q, \delta)\big) \mid \bigvee_{i=1}^{2} A'_i \cdot (u, \vec{v}_Z, \vec{v}_Q, \delta) \leq \vec{b}'_i\}^*$$

In order to make it more readable, we introduce the following shorthands: $\vec{u}^Z = (n_1, n_2^Z, n_3^Z, n_4^Z)$ and $\vec{u}^Q = \vec{(} n_2^Q, n_3^Q, n_4^Q)$. Similarly, $\vec{x}^Z = (u, \vec{v}_Z)$ and $\vec{x}^R = \vec{v}_Q$. At the end, we also rename the variables for matrices A'_i and vectors $\vec{b}'_i$ and finally we obtain a formula that closely corresponds to the formula described in the algorithm:

$$n_1 + n_2 < n_3 + n_4 \wedge \exists (n_2^Z, n_3^Z, n_4^Z) \in \mathbb{Z}^3.\exists \vec{(} n_2^Q, n_3^Q, n_4^Q), \in \mathbb{Q}^3_{\geq 0}.\exists \Delta \in \mathbb{Q}_{>0}.$$
$$(\vec{u}^Z, \vec{u}^Q, \Delta) \in \{\big(\vec{x}^Z, \vec{x}^R, \delta)\big) \mid \bigvee_{i=1}^{2} A_i \cdot (\vec{x}^Z, \vec{x}^R, \delta) \leq \vec{b}_i\}^*$$

Step 4 separates integer and rational parts. We demonstrate the technique on only one disjunct (the second one as it contains the non-zero δ-part). All other disjuncts can be transformed similarly. In the matrix A we identify three parts: A_Z, A_R and A_δ that corresponds to the parts which are multiplying integer, rational and δ variables. Applying this on A_2, we obtain, $A_2 = \begin{bmatrix} A_Z & A_R & A_\delta \end{bmatrix}$, where

$$A_Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \end{bmatrix}, A_R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \\ -1 & 1 & 0 \\ 1 & 0 & -1 \\ -1 & 0 & 1 \end{bmatrix}, A_\delta = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Next, we estimate the values that the vector $A_R\vec{x}^R + A_\delta\delta$ can reach. Our goal is to construct an integer vector $\vec{a}$ such that $A_R\vec{x}^R + A_\delta\delta \le \vec{a} \le \vec{b} - A_Z\vec{x}^Z$. The set of values that $\vec{a}$ can have, we denote with $\mathscr{A}$.

$$A_R\vec{x}^R + A_\delta\delta \in \begin{bmatrix} 0 \\ 0 \\ (-1,0] \\ [0,1) \\ (-1,0] \\ [0,1) \\ (-1,1] \\ (-1,1) \\ (-1,1) \end{bmatrix} \Longrightarrow \mathscr{A} = \{ \begin{bmatrix} 0 \\ 0 \\ 0 \\ x_1 \\ 0 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \mid x_i \in \{0,1\}\}$$

Once we have constructed $\mathscr{A}$, the original problem $A_Z\vec{x}^Z + A_R\vec{x}^R + A_\delta\delta \le \vec{b}$ becomes equivalent to the finite disjunction

$$\bigvee_{\vec{a}\in\mathscr{A}} A_Z\vec{x}^Z \le \vec{b} - \vec{a} \wedge A_R\vec{x}^R + A_\delta\delta \le \vec{a}$$

and integer and rational part are fully separated. This way the original formula becomes:

$$n_1 + n_2 < n_3 + n_4 \wedge \exists(n_2^Z, n_3^Z, n_4^Z) \in \mathbb{Z}^3.\exists\vec{(}n_2^Q, n_3^Q, n_4^Q), \in \mathbb{Q}^3_{\ge 0}.\exists\Delta \in \mathbb{Q}_{>0}.$$
$$(\vec{u}^Z, \vec{u}^Q, \Delta) \in \{\big(\vec{x}^Z, \vec{x}^R, \delta\big) \mid \bigvee_{i=1}^{2} \bigvee_{\vec{a}\in\mathscr{A}_i} (A_{Z\,i}\vec{x}^Z \le \vec{b}_i - \vec{a} \wedge A_{R\,i}\vec{x}^R + \vec{A}_{\delta\,i}\delta \le \vec{a})\}^*$$

The last step is to propagate the star operator though disjunctions. The number of disjuncts is finite and thus $(\vec{u}^Z, \vec{u}^Q, \Delta)$ can be written as a finite summation. Each its summand is an elements of a set described by a MLIRA* formula. For the simplicity of notation, let us assume that all disjunctions are numbered with natural numbers $1, 2, \ldots, 64$ and corresponding matrices A and vectors b are indexed with those numbers.

$$n_1 + n_2 < n_3 + n_4 \wedge \exists (n_2^Z, n_3^Z, n_4^Z) \in \mathbb{Z}^3 . \exists \vec{(n_2^Q, n_3^Q, n_4^Q)}, \in \mathbb{Q}^3_{\geq 0} . \exists \Delta \in \mathbb{Q}_{>0} .$$
$$(\vec{u}^Z, \vec{u}^Q, \Delta) = \sum_{i=1}^{64} (\vec{u^Z}_i, \vec{u^Q}_i, \Delta_i) \wedge$$
$$\bigwedge_{i=1}^{64} (\vec{u^Z}_i, \vec{u^Q}_i, \Delta_i) \in \{(\vec{x}^Z, \vec{x}^R, \delta)) \mid A_{Z\,i} \vec{x}^Z \leq \vec{b_{Z\,i}} \wedge A_{R\,i} \vec{x}^R + \vec{A_{\delta\,i}} \delta \leq \vec{b_{R\,i}}\}^*$$

In the rest of the chapter we show how to describe vector $(\vec{u}^Z, \vec{u}^Q, \Delta)$ without the star operator.

4.5 Eliminating the Star Operator from Formulas

The previous section sets the stage for the following star-elimination theorem, Theorem 4.4, which is the core result for the decidability problem of the logic defined in Figure 4.3. To prove Theorem 4.4 we need Theorem 4.3, which can be seen as a generalization of the fact that non-negative solutions of integer linear arithmetic formulas are semilinear sets.

Theorem 4.3 (Corollary 1 in [Pottier(1991)]) *Consider a system of inequations $A\vec{x} \leq \vec{b}$ where $A \in \mathbb{Z}^{m,n}$ and $b \in \mathbb{Z}^m$. Let C be a set of all solutions of $A\vec{x} \leq \vec{b}$ Then there exist two finite sets $C_1, C_2 \subseteq \mathbb{Z}^n$ such that*

$$\vec{x} \in C \Leftrightarrow \vec{x} = \vec{x}_0 + \vec{x}_1 + \ldots + \vec{x}_k, \text{ with } \vec{x}_0 \in C_1 \text{ and } \vec{x}_1, \ldots, \vec{x}_k \in C_2$$

Theorem 4.4 *Let F be a quantifier-free* MLIRA *formula. Then there exist effectively computable integer vectors $\vec{a}_i$ and $\vec{b}_{ij}$ and effectively computable rational vectors $\vec{c}_1, \ldots, \vec{c}_n$ with coordinates in $\mathbb{Q}_{[0,1]}$ such that formula* (4.3) *is equivalent to a formula of the form*

$$(\vec{u}^Z, \vec{u}^Q, \Delta) = \vec{0} \vee \exists K \in \mathbb{N} . \exists \mu_1, \ldots, \mu_q, \nu_{11}, \ldots, \nu_{qq_q} \in \mathbb{N} . \exists \beta_1, \ldots, \beta_n \in \mathbb{Q} .$$
$$\vec{u}^Z = \sum_{i=1}^{q} (\mu_i \vec{a}_i + \sum_{j=1}^{q_i} \nu_{ij} \vec{b}_{ij}) \wedge \bigwedge_{i=1}^{q} (\mu_i = 0 \rightarrow \sum_{j=1}^{q_i} \nu_{ij} = 0) \wedge \sum_{i=1}^{q} \mu_i = K$$
$$\wedge K \geq 1 \wedge \Delta > 0 \wedge (\vec{u}^Q, \Delta) = \sum_{i=1}^{n} \beta_i \vec{c}_i \wedge \bigwedge_{i=1}^{n} \beta_i \geq 0 \wedge \sum_{i=1}^{n} \beta_i = K \quad (4.4)$$

Proof. For a set of vectors S and an integer variable K, we define $KS = \{v_1 + \ldots + v_K \mid v_1, \ldots, v_K \in S\}$. Formula (4.3) is satisfiable iff there exists non-negative integer $K \in \mathbb{N}$ such that both

$$\vec{u}^Z \in K\{\vec{x}^Z \mid A_Z \vec{x}^Z \leq \vec{b}^Z\} \quad (4.5)$$

and

$$(\vec{u}^Q, \Delta) \in K\{(\vec{x}^R, \delta) \mid A_R \cdot (\vec{x}^R, \delta) \leq \vec{b}^R, \vec{x}^R \in \mathbb{Q}^{d_R}_{[0,1)}, \delta \in \mathbb{Q}_{(0,1]}\} \quad (4.6)$$

hold. We show how to describe (4.5) and (4.6) as existentially quantified MLIRA formulas that share the variable K.

If $K = 0$ then $(\vec{u}^Z, \vec{u}^Q, \Delta)$ must be a zero vector. This is a trivial case as both formulas are satisfiable. In the rest of the proof we consider a non-trivial case when $(\vec{u}^Z, \vec{u}^Q, \Delta)$ is a non-zero vector. Then K must be $K \geq 1$ and $\Delta > 0$.

Following Theorem 2.14 and Theorem 4.3, $\vec{u} \in S^*$ can be expressed as a Presburger arithmetic formula. In particular, formula (4.5) is equivalent to

$$\begin{aligned}\exists \mu_1, \ldots, \mu_q, \nu_{11}, \ldots, \nu_{qq_q} \in \mathbb{N}.\ \ & \vec{u}^Z = \textstyle\sum_{i=1}^{q} (\mu_i \vec{a}_i + \sum_{j=1}^{q_i} \nu_{ij} \vec{b}_{ij}) \wedge \\ & \bigwedge_{i=1}^{q} (\mu_i = 0 \rightarrow \textstyle\sum_{j=1}^{q_i} \nu_{ij} = 0) \ \wedge\ (\sum_{i=1}^{q} \mu_i = K)\end{aligned} \tag{4.7}$$

where vectors $\vec{a}_i$'s and $\vec{b}_{ij}$ can be computed effectively from A_Z and b^Z.

We next characterize condition (4.6). Renaming variables and incorporating the boundedness of $\vec{x}, \delta$ into the linear inequations, we can write such condition in the form

$$(\vec{u}^Q, \Delta) \in K\{(\vec{x}, \delta) \mid A \cdot (\vec{x}, \delta) \leq \vec{b}, \delta > 0\} \tag{4.8}$$

Here A is a new matrix such that $A \cdot (\vec{x}, \delta) \leq b$ subsumes the conditions $\vec{0} \leq \vec{x} \leq \vec{1}$ and $0 \leq \delta \leq 1$. The fact that $\vec{x} \in \mathbb{Q}_{[0,1)}$ was converted to $\vec{x} \in \mathbb{Q}_{[0,1]}$ using δ as before. From the theory of linear programming [Schrijver(1998)] it follows that the set $\{(\vec{x}, \delta) \mid A \cdot (\vec{x}, \delta) \leq \vec{b}\}$ is a polyhedron, and because the solution set is bounded, it is in fact a polytope. Therefore, there exist finitely many vertices $\vec{c}_1, \ldots, \vec{c}_n \in \mathbb{Q}^d_{[0,1]}$ for some d such that $A \cdot (\vec{x}, \delta) \leq \vec{b}$ is equivalent to

$$\exists \lambda_1, \ldots, \lambda_n \in \mathbb{Q}_{[0,1]}.\ \sum_{i=1}^{n} \lambda_i = 1 \ \wedge\ (\vec{x}, \delta) = \sum_{i=1}^{n} \lambda_i \vec{c}_i$$

Consequently, (4.8) is equivalent to

$$\begin{aligned}&\exists \vec{u}_1, \ldots, \vec{u}_K.\ \exists \delta_1, \ldots, \delta_K.\ (\vec{u}^Q, \Delta) = \sum_{j=1}^{K} (\vec{u}_j, \delta_j) \ \wedge\ \exists \lambda_{11}, \ldots, \lambda_{Kn}. \\ &\quad \bigwedge_{j=1}^{K} \Big(\bigwedge_{i=1}^{n} \lambda_{ij} \geq 0 \ \wedge\ \sum_{i=1}^{n} \lambda_{ij} = 1 \ \wedge\ (\vec{u}_j, \delta_j) = \sum_{i=1}^{n} \lambda_{ij} \vec{c}_i \wedge \delta_j > 0 \Big)\end{aligned} \tag{4.9}$$

It remains to show that the above condition is equivalent to

$$\exists \beta_1, \ldots, \beta_n.\ \ (\vec{u}^Q, \Delta) = \sum_{i=1}^{n} \beta_i \vec{c}_i \ \wedge\ \bigwedge_{i=1}^{n} \beta_i \geq 0 \ \wedge\ \sum_{i=1}^{n} \beta_i = K \tag{4.10}$$

Consider a solution of (4.9). Letting $\beta_i = \sum_{j=1}^{K} \lambda_{ij}$ we obtain a solution of (4.10). Conversely, consider a solution of (4.10). Letting $\alpha_{ij} = \beta_i / K$, $\vec{u}_j = \vec{u}/K$, $\delta_j = \Delta / K$ we obtain a solution of (4.9). This shows the equivalence of (4.9) and (4.10).

Conjoining formulas (4.10) and (4.7) we complete the proof of Theorem 4.4. ■

4.5.1 Satisfiability Checking for Collection Formulas

Because star elimination (as well as the preparatory steps in Section 4.4) introduce only existential quantifiers, and the satisfiability of MLIRA formulas is decidable (see e.g. [Dutertre and de Moura(2006b), Dutertre and de Moura(2006a)]), we obtain the decidability of the initial formula $G(\vec{r}, \vec{w}) \wedge \vec{w} \in \{\vec{x} \mid F(\vec{x})\}^*$. Thanks to transformation to sum normal form and Theorem 4.2, we obtain the decidability of formulas involving sets, multisets and fuzzy sets.

Techniques for deciding satisfiability of MLIRA formulas are part of implementations of modern satisfiability modulo theory theorem provers [Dutertre and de Moura(2006b), Dutertre and de Moura(2006a), Berezin et al.(2003)Berezin, Ganesh, and Dill] and typically use SAT solving techniques along with techniques from mixed integer-linear programming.

4.5.2 Satisfiability Checking for Generalized Multisets Formulas

Consider the logic defined in Figure 2.3 but without the requirement that inner terms need to be non-negative. We also apply the same semantics for the $*$ operator as in this Section: $\vec{u} \in \{\vec{x} \mid F\}^*$ indicates that the vector u is a finite sum of the solution vectors of F. We show that the satisfiability of this new logic remains an NP-complete problem.

To establish the NP-completeness, we follow the steps in the original proof of NP-completeness, as presented in Section 2.7, with few small modifications. First, instead of semilinear sets, we use the characterization given in Theorem 4.3: vector u is a sum of integer vectors, not necessarily non-negative ones. The bound on their size is exactly the same as the bound given in Theorem 2.11 (cf. Corollary 1 in [Pottier(1991)]). Next, there are only polynomially many vectors that will participate in the sum: Theorem 2.19 is defined for all integers, not just natural numbers. The number of generating vectors depends on their maximal size and the dimension of the problem. Using those facts we can construct formula (2.13).

The last step is to establish a bound on the size of λ_i. Although Theorem 2.22 talks only about non-negative integers, we can still use exactly the same bound as defined in Theorem 2.23. By analyzing the proof of Theorem 2.23, we can see that the only major difference is that the matrix A_g in the case of semilinear sets contains non-negative integer generators, while in the new proof it can also contain negative values. Nevertheless, we can still apply Theorem 2.22 because it establishes a bound on the size of non-negative solutions of an integer system of equalities. From this bound we can derive a bound on λ_i.

By having a bound on λ_i we are able to construct a linear integer arithmetic formula equisatisfiable with a given formula. The new formula is polynomial in the size of the original formula. Since the satisfiability checking of linear integer arithmetic formulas is an NP-complete problem, checking satisfiability of the extended multiset formulas is also an NP-complete problem.

4.6 Decision Procedures for Collection Images

Logics that involve collections (sets, multisets), and cardinality constraints are useful for reasoning about unbounded data structures and concurrent processes. To make such logics more useful in verification, we extend them in this section with the ability to compute function images. We establish decidability and complexity bounds for the extended logics.

4.6.1 Motivating Examples for Collection Images

We start by listing several examples from verification of data structures that have motivated us to consider extending the logic of sets with cardinalities (BAPA) with functions.

$$\begin{array}{l} \mathsf{nodes} \subseteq \mathsf{alloc} \wedge \mathsf{card}\ \mathsf{tmp} = 1 \wedge \mathsf{tmp} \cap \mathsf{alloc} = \emptyset \wedge \mathsf{data}[\mathsf{tmp}] = \mathsf{e} \wedge \\ \mathsf{content} = \mathsf{data}[\mathsf{nodes}] \wedge \mathsf{nodes1} = \mathsf{nodes} \cup \mathsf{tmp} \wedge \mathsf{content1} = \mathsf{data}[\mathsf{nodes1}] \rightarrow \\ \qquad \mathsf{card}\ \ \mathsf{content1} \leq \mathsf{card}\ \mathsf{content} + 1 \end{array}$$

Figure 4.4: Verification condition for verifying that by inserting an element into a list, the size of the list does not decrease. The variables occurring in the formula have the following types: $\mathsf{nodes}, \mathsf{alloc}, \mathsf{tmp}, \mathsf{e}, \mathsf{content}, \mathsf{content1} :: \mathsf{Set}\langle \mathsf{E} \rangle$, $\mathsf{data} :: \mathsf{E} \rightarrow \mathsf{E}$.

$$\begin{array}{l} \mathsf{nodes} \subseteq \mathsf{alloc} \wedge \mathsf{card}\ \mathsf{tmp} = 1 \wedge \mathsf{tmp} \cap \mathsf{alloc} = \emptyset \wedge \mathsf{data}[\mathsf{tmp}] = \mathsf{e} \wedge \\ \mathsf{content} = \mathsf{data}[\mathsf{nodes}] \wedge \mathsf{nodes1} = \mathsf{nodes} \cup \mathsf{tmp} \wedge \mathsf{content1} = \mathsf{data}[\mathsf{nodes1}] \rightarrow \\ \qquad \mathsf{card}\ \ \mathsf{content1} = \mathsf{card}\ \mathsf{content} + 1 \end{array}$$

Figure 4.5: Verification condition for verifying that by inserting an element into a list, the size of a list increases by one. The variables occurring in the formula have the following types: $\mathsf{nodes}, \mathsf{alloc}, \mathsf{tmp} :: \mathsf{Set}\langle \mathsf{E} \rangle$, $\mathsf{content}, \mathsf{content1}, \mathsf{e} :: \mathsf{Multiset}\langle \mathsf{E} \rangle$, $\mathsf{data} :: \mathsf{E} \rightarrow \mathsf{E}$.

We start with a dynamically allocated data structure (such as a list or a tree) that manipulates a set of linked nodes denoted by the variable nodes. The useful content in the data structure is stored in the data fields of the elements of nodes. The nodes set can be either explicitly manipulated through a library data type or built-in type [Dewar(1979)], or it can be verified to correspond to a set of reachable objects using techniques such as [Wies et al.(2006)Wies, Kuncak, Lam, Podelski, and Rinard]. The content of the list, stored in the content specification variable, is then an image of nodes under the function data. We consider two cases of specification in our example: 1) content is a *set*, that is, multiple occurrences of elements are ignored and 2) content is a *multiset*, preserving the counts of occurrences of each element in the data structure.

The verification condition generated for the case when the image is a set is given in Figure 4.4. A more precise abstraction is obtained if content is viewed as a multiset. Figure 4.5 shows the verification condition for this case. In the next section, we describe a decision procedure that can reason about such logic, where functions can also return a multiset, not only set. The approach also rewrites sets as a disjoint union of Venn regions. It then constrains the cardinality of the multiset obtained through the image to be equal to the cardinality of the original set. This final formula is a formula in the NP-complete logic for reasoning about

multisets and cardinality constraints defined in Chapter 2.

4.6.2 Logic of Multiset Images of Functions

In this section, we extend the logic of multisets with cardinalities to also include a function image operator that maps a set into a multiset.

We define the function image of a set A to be a multiset $f[A] : \mathbb{E} \to \mathbb{N}$:

$$(f[A])(e) = |\{x \mid x \in A \wedge f(x) = e\}|$$

Figure 4.6 shows the logic that embeds the logic of multisets (defined in Figure 2.3), and extends it with the multiset image operator. The logic distinguishes the sorts of sets and multisets, but also includes a casting function $\mathsf{mset}(B)$ which treats a set as a multiset, and an abstraction function $\mathsf{set}(M)$ which extracts the set of distinct elements that occur in a multiset.

$$\begin{array}{rcl}
F & ::= & A \mid F \vee F \mid \neg F \\
A & ::= & B \subseteq B \mid M \subseteq M \mid T \leq T \mid K \ \mathsf{dvd}\ T \\
B & ::= & x \mid \emptyset \mid \mathscr{U} \mid B \cup B \mid B \cap B \mid B^c \mid \mathsf{set}(M) \\
M & ::= & m \mid \emptyset_M \mid M \cap M \mid M \cup M \mid M \uplus M \mid M \setminus M \mid M \setminus\setminus M \mid \mathsf{mset}(B) \mid f[B] \\
T & ::= & k \mid K \mid \mathsf{MAXC} \mid T_1 + T_2 \mid K \cdot T \mid |B| \mid |M| \\
K & ::= & \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots
\end{array}$$

Figure 4.6: Logic of multisets, cardinality operator, and multiset images of sets

Given a formula F in the language described in Figure 4.6, a decision procedure for F works as follows:

1. Apply the algorithm in Figure 4.7 to translate F into an equisatisfiable multiset formula F' in the syntax of the multiset logic defined in Figure 2.3. In this step we eliminate function symbols. The new formula F' has size singly exponential in the size of F.
2. Invoke on the formula F' the decision procedure described in Chapter 2. The decision procedure runs in NP time.

The correctness of the reduction is stated in the following theorem.

Theorem 4.5 *Given a formula F as an input to the algorithm described in Figure 4.7, let the formula F' be its output. Then formulas F and F' are equisatisfiable and their satisfying assignments have the same projections on the set and multiset variables occurring in F.*

Proof. Let α be a model for F. From α we construct a model for F' by interpreting M_i as $\alpha(f[s_i])$.

INPUT: formula in the syntax of Figure 4.6
OUTPUT: multiset formula in the syntax of Figure 2.3

1. Flatten expressions containing the operator set:
   $C[\ldots \mathsf{set}(M) \ldots] \rightsquigarrow (B_F = \mathsf{set}(M) \wedge C[\ldots B_F \ldots])$
   where the occurrence of $\mathsf{set}(M)$ is not already in a top-level conjunct of the form $B = \mathsf{set}(M)$ for some set variable B, and B_F is a fresh unused set variable

2. Let S be the set of variables occurring in the formula
   Define the set $S_N = \{s_1, \ldots, s_Q\}$ of Venn regions over elements of S

3. Rewrite each set expression as a disjoint union of the Venn regions from S_N

4. Eliminate function symbols:
   $C[\ldots f[s_{i_1} \cup \ldots \cup s_{i_k}] \ldots] \rightsquigarrow C[\ldots (M_{i_1} \uplus \ldots \uplus M_{i_k}) \ldots]$
   where each M_{i_j} is a fresh multiset variable denotes $f[s_{i_j}]$

5. Add the conjuncts which states a necessary condition for $M_{i_j} = f[s_{i_j}]$
   $F \rightsquigarrow F \wedge \bigwedge_{i=1}^{Q} |s_i| = |M_i|$

6. Add the conjuncts which state that s_{i_j} are disjoint sets
   $F \rightsquigarrow F \wedge \forall e. \bigwedge_{i=1}^{Q} (s_i(e) = 0 \vee s_i(e) = 1) \wedge \bigwedge_{i \neq j} (s_i \cap s_j = \emptyset)$

Figure 4.7: Algorithm for eliminating function symbols

Conversely, let α' be a model for F', and we need to define α, a model for F. We only need to interprete functions and for each function symbol f we interprete it on each disjoint set s_i independently. Because $|\alpha'(s_i)| = |\alpha'(M_i)|$, we can enumerate both s_i and M_i into sequences $a_1, \ldots, a_K$ and $b_1, \ldots, b_K$ of same length, i.e. $s_i = \{a_1, \ldots, a_K\}$ and $M_i = \{b_1, \ldots, b_K\}$. This enumeration defines an interpretation of function assigning a_j to b_j for $1 \leq j \leq K$ such that $f_\alpha[\alpha(s_i)] = \alpha(M_i)$.

Finally, we prove that our logic is NEXPTIME-complete. To do that we use the following facts from [Givan et al.(2002)Givan, McAllester, Witty, and Kozen].

Definition 4.6 (Lewis clause) *Let a be a constant and f be a unary function symbol. A* Lewis clause *(over a and f) is a first-order clause C that has one of the following forms:*

- *C is an atom $P(a)$ for a monadic predicate symbol P*
- *C is a clause involving a single variable x where every literal is an application of a monadic predicate to either x or $f(x)$*
- *C is a clause involving exactly two variable x and y where every literal is an application of a monadic predicate to either x or y*

Theorem 4.7 (p. 21 in [Givan et al.(2002)Givan, McAllester, Witty, and Kozen]) *Checking satisfiability for a set of Lewis clauses is an NEXPTIME-complete problem.*

The proof of Theorem 4.7 relies on the result [Lewis(1980)] that acceptance of nondeterministic exponential-time bounded Turing machines can be reduced to satisfiability of formulas of the form $\exists z.F_1 \wedge \forall y \exists x.F_2 \wedge \forall y_1 \forall y_2.F_3$ where F_1, F_2, and F_3 have no quantifiers and are monadic (have only unary predicates). This class is in fact a fragment of quantifier BAPA, where quantification occurs only over singleton elements. Note that this class of formulas has a finite model property, which is preserved by Skolemization. Therefore, the theorem continues to hold if we consider finite satisfiability, which is the version that we need. We adapt the proof of Theorem 4.7 for establishing NEXPTIME-hardness of our constraints.

Theorem 4.8 *Checking satisfiability of formulas belonging to the logic defined in Figure 4.6 is an NEXPTIME-complete problem.*

Proof. Let F be a formula from Figure 4.6. The algorithm in Figure 4.7 reduces F to an equisatisfiable formula defined in the syntax of the multiset logic from Figure 2.3. This reduction produces a formula of a singly exponential size by introducing set variables for Venn regions over set variables in the original formula for each function. The resulting formula belongs to logic defined in Figure 2.3, which we have proved to be NP-complete (Theorem 2.25). From those two facts, we conclude that checking satisfiability of formulas belonging to the logic defined in Figure 4.6 is in NEXPTIME.

To prove NEXPTIME-hardness, we construct a reduction from checking satisfiability of a set of Lewis clauses to checking satisfiability of a set of formulas belonging to the logic defined in Figure 4.6. Given a set of Lewis clauses, we identify monadic predicate symbols with set variables, using the same symbols for both. We encode the set of Lewis clause into our logic as follows:

- let $P_1(a),\ldots,P_n(a)$ be all clauses in the set of the form $P(a)$. We encode them with the formula $P_1 \cap \ldots \cap P_n \neq \emptyset$.
- for every clause of the form

  $$\forall x.P_1(x) \vee P_2(x) \vee \ldots \vee P_m(x) \vee Q_1(f(x)) \vee Q_2(f(x)) \vee \ldots \vee Q_n(f(x))$$

  we generate a constraint

  $$f\left(P_1^c \cap P_2^c \cap \ldots \cap P_m^c\right) \subseteq Q_1 \cup Q_2 \cup \ldots \cup Q_n$$

  To illustrate why we generate such a constraint, here is a sequence of equivalences that hold in the set theory. For readability reasons we restrict ourselves to one P predicate

and Q predicate:

$$\begin{aligned}
& f(P^c) \subseteq Q \Leftrightarrow \forall e.e \in f(P^c) \Rightarrow e \in Q \\
\Leftrightarrow & \forall e.(\exists e_1.e_1 \in P^c \wedge e = f(e_1)) \Rightarrow e \in Q \\
\Leftrightarrow & \forall e.\forall e_1.e_1 \in P \vee e \neq f(e_1) \vee e \in Q \\
\Leftrightarrow & \neg\exists e.\exists e_1.e_1 \in P^c \wedge e = f(e_1) \wedge e \in Q^c \\
\Leftrightarrow & \neg\exists e_1.e_1 \in P^c \wedge f(e_1) \in Q^c \\
\Leftrightarrow & \forall x.x \in P \vee f(x) \in Q
\end{aligned}$$

- a clause of the form

$$\forall x \forall y.\ P_1(x) \vee P_2(x) \vee \ldots \vee P_m(x) \vee Q_1(y) \vee Q_2(y) \vee \ldots \vee Q_n(y)$$

  is translated into a formula

$$(P_1 \cup P_2 \cup \ldots \cup P_m = \mathscr{U}) \ \vee \ (Q_1 \cup Q_2 \cup \ldots \cup Q_n = \mathscr{U})$$

If we denote the given set of Lewis clauses with L and the translated set of the formulas with $T[L]$, it can be easily shown that L and $T[L]$ are equisatisfiable. Let α be a model of L. We extend α to be a model for $T[L]$ as follows. To construct a model for the set variables in $T[L]$ we check non-emptiness of every Venn region, i.e. we check whether the corresponding clause describing a Venn region evaluate to true in α. Once we have an interpretation for sets, we construct an interpretation for the function symbols similarly as in the proof of Theorem 4.5.

The proof of the other direction is analogous. Using the results of Theorem 4.7, we conclude that the satisfiability problem for formulas belonging to the logic defined in Figure 4.6 is an NEXPTIME-complete problem.

5 Decision Procedures for Automating Termination Proofs

In this chapter we introduce a new logic, called POSSUM , for expressing ordering constraints on finite multisets. The main motivation for POSSUM is proving program termination. There was no decision procedure that would enable automated reasoning in this logic until recently [Piskac and Wies(2011)]. In this chapter we describe the decision procedure for POSSUM and prove its correctness. The logic is parametrized by the theory of the base set, which can be an arbitrary theory equipped with a preorder (not necessarily well-founded). We show that, if the base theory is decidable, then so is its extension to a multiset ordering. Moreover, if the base theory is decidable in NP, then the satisfiability problem for its POSSUM extension is also in NP. Our decision procedure reduces a given input formula to an equisatisfiable formula in the base theory. The decision procedure can be implemented using off-the-shelf SMT solvers. We therefore believe that it can be a useful component of future automated termination provers.

5.1 Motivation

The standard technique for proving program termination is to construct a *ranking function* [Turing(1949), Floyd(1967)]. A ranking function maps the states of the program into some well-founded domain, i.e., a set equipped with a well-founded ordering. The mapping is such that, with each transition taken by the program, the value of the ranking function decreases in the ordering. The canonical well-founded ordering for constructing ranking functions is the strict order on the natural numbers. However, constructing *global* ranking functions for this ordering (i.e., functions that decrease with every transition of the program) requires a lot of ingenuity.

Despite the general result of undecidability of the halting problem, recent advances in program analysis have brought forth tools that can automatically prove termination of real-world programs [Berdine et al.(2006)Berdine, Cook, Distefano, and O'Hearn, Cook et al.(2006)Cook, Podelski, and Rybalchenko]. The success of these tools is due to the development of new proof techniques for termination [Lee et al.(2001)Lee, Jones, and Ben-Amram, Podelski and Rybalchenko(2004b)]. These techniques avoid the construction of a global termination argu-

ment and, instead, decompose the program into simpler ones. Each of these simpler programs is then proved terminating independently, by constructing a simpler ranking function. The automation of these proof techniques relies on decision procedures for reasoning about constraints on well-founded domains. The existing tools use known decidable logics such as linear arithmetic to express these constraints [Podelski and Rybalchenko(2004a), Colón and Sipma(2001)], which effectively restricts the range of ranking functions that can be constructed automatically. We believe that by providing decision procedures for more sophisticated well-founded domains, one can significantly increase the class of programs that are amenable to automated termination proofs.

Among the most powerful well-founded domains for proving program termination are multiset orderings [Dershowitz and Manna(1979)]. In this chapter, we present a decision procedure for automated reasoning about such orderings. A (strict) ordering $\prec$ on the base set S can be lifted to an ordering $\prec_{\mathsf{m}}$ on (finite) multisets over S as follows. For two multisets X and Y, $X \prec_{\mathsf{m}} Y$ holds iff X and Y are different, and for every element $x \in S$ which occurs more times in X than in Y, there exists an element $y \in S$ which occurs more times in Y than in X and $x \prec y$. For instance, $\{1,1,1,2,2\} \prec_{\mathsf{m}} \{1,3\}$ since $1 < 3$ and $2 < 3$. Multiset orderings are interesting because they inherit important properties of the ordering on the base set. In particular, the multiset ordering $\prec_{\mathsf{m}}$ is well-founded iff the ordering $\prec$ on the base set is well-founded [Dershowitz and Manna(1979)]. Multiset orderings have been traditionally used for manual termination proofs in program verification [Dershowitz and Manna(1979), Deng and Sangiorgi(2006)], term rewriting systems [Dershowitz(1979), Baader and Nipkow(1998)], and theorem proving [Martín-Mateos et al.(2005)Martín-Mateos, Ruiz-Reina, Alonso, and Hidalgo, Bachmair and Ganzinger(2001b)]. The question whether reasoning about multiset orderings can be effectively automated was open.

5.2 Examples

We motivate the usefulness of our decision procedure for proving termination through two examples.

Example: counting leaves in a tree. Our first example is a program taken from [Dershowitz and Manna(1979)] and shown in Figure 5.1. The termination behavior of this program is representative for many programs that traverse algebraic data types.

The program CountLeaves counts the number of leaves in a binary tree. For this purpose, it maintains a stack S that contains all subtrees of the input tree *root* that still need to be traversed. In each iteration, the first element y is removed from S. If y is a leaf then the count is increased. Otherwise, the subtrees of y are pushed on the stack. Then the computation continues with the updated stack.

In order to prove termination of program CountLeaves, we need to find a well-founded ordering on the states of the program that decreases with every iteration of the loop. This

```
prog CountLeaves(root : Tree) : int =
    var S : Stack[Tree] = root
    var c : int = 0
    do
        y := head(S)
        if leaf(y) then
            S := tail(S)
            c := c + 1
        else S := left(y) · right(y) · tail(S)
    until S = ϵ
    return c
```

Figure 5.1: Program CountLeaves: counting the leaves in a binary tree

```
prog AbsCountLeaves(root : Tree) =
    var X_S : multiset[Tree] = {root}
    do
        y := choose(X_S)
        if leaf(y) then X_S := X_S \ {y}
        else X_S := (X_S \ {y}) ⊎ {left(y)} ⊎ {right(y)}
    until X_S = ∅
```

Figure 5.2: Multiset abstraction of program CountLeaves

well-founded ordering needs to capture the fact that in each loop iteration either some tree is removed from the stack, or some tree on the stack is replaced by finitely many smaller trees. This can be naturally expressed in terms of a multiset ordering. We therefore abstract the program CountLeaves by a program over multisets. The result of this abstraction is shown in Figure 5.2. The program AbsCountLeaves is obtained from program CountLeaves by mapping the stack S to a multiset X_S, i.e. in program AbsCountLeaves we abstract from the order of the elements in S. In program AbsCountLeaves, the stack operations are replaced by operations on multisets. For instance, the operation $head(S)$ is abstracted by the operation $choose(X_S)$ that non-deterministically chooses an element from the multiset X_S. The computation of such multiset abstractions of programs could be automated by combining techniques developed in [Suter et al.(2010)Suter, Dotta, and Kuncak] and [Podelski and Rybalchenko(2007b), Cook et al.(2005)Cook, Podelski, and Rybalchenko]. In this chapter, we focus on automating the termination proofs for the resulting multiset program.

We prove termination of program AbsCountLeaves by proving that for every iteration of the loop, the variable X_S decreases in the ordering $\prec_{\mathsf{m}}$. The ordering $\prec_{\mathsf{m}}$ is the multiset extension of the subtree ordering $\prec$ on the trees stored in the multiset. The subtree ordering is well-founded; consequently, so is its multiset extension. Termination of program AbsCountLeaves is therefore implied by the validity of the *termination condition* given in Figure 5.3. The decision procedure presented in this chapter decides the validity of such termination conditions

$$X_S \neq \emptyset \wedge X_S(y) > 0 \wedge$$
$$(X'_S = X_S \setminus \{y\} \vee X'_S = (X_S \setminus \{y\}) \uplus \{\mathit{left}(y)\} \uplus \{\mathit{right}(y)\}) \rightarrow X'_S <_{\mathsf{m}} X_S$$

Figure 5.3: Termination condition for program AbsCountLeaves

$$\begin{aligned} \neg\neg F &\rightsquigarrow F \\ \neg(F \wedge G) &\rightsquigarrow \neg F \vee \neg G \\ \neg(F \vee G) &\rightsquigarrow \neg F \wedge \neg G \end{aligned}$$

Figure 5.4: Rewrite system for computing negation normal form

(respectively, unsatisfiability of their negation). In Section 5.3 we show how the decision procedure works on a formula similar to the one shown in Figure 5.3.

Example: computing negation normal form. Our second example is a rewrite system that computes the negation normal form of a propositional formula. It consists of the three rewrite rules shown in Figure 5.4. The three rules are applied non-deterministically to any matching subformula.

In order to prove termination of this rewrite system, Dershowitz [Dershowitz(1979)] suggested the following mapping from a propositional formula F to a multiset of natural numbers X_F. Let $[G]$ denote the number of operators other than $\neg$ that occur in G, then define

$$X_F = \{[G] \mid \neg G \text{ is a subformula of } F\}$$

We can then prove that, for each rewrite rule applied to a formula F, X_F decreases in the multiset extension $<_{\mathsf{m}}$ of the ordering $<$ on natural numbers. This amounts to checking validity of the following two implications:

$$X_F = X'_F \uplus \{x, x\} \rightarrow X'_F <_{\mathsf{m}} X_F$$
$$X_F = Y \uplus \{x + y + 1\} \wedge x > 0 \wedge X'_F = Y \uplus \{x, y\} \rightarrow X'_F <_{\mathsf{m}} X_F$$

Again, these checks can be automated using our decision procedure.

5.3 Decision Procedure through an Example

We now explain our decision procedure through an example. The decision procedure is parameterized by the theory of the base elements comprising the multisets. For instance, in the first example given in Section 5.2, the base theory is the theory of trees with the subtree ordering. This theory is decidable in NP [Venkataraman(1987)]. In general, the base theory can be any decidable theory equipped with a preorder. Our decision procedure reduces the formula with ordering constraints over multisets to a formula containing ordering constraints

on the base elements. Satisfiability of the reduced formula is then checked using the decision procedure of the base theory.

To demonstrate how our decision procedure works, we apply it to the following formula, which is a slightly generalized version of the negated termination condition given in Figure 5.3:

$$Y \subseteq X \wedge X' = (X \setminus Y) \uplus Z \wedge Z \prec_{\mathsf{m}} Y \wedge \neg(X' \prec_{\mathsf{m}} X) \tag{5.1}$$

This formula is unsatisfiable in the theory of preordered multisets (where the base theory is the theory of all preordered sets). The reduction of the formula works as follows. First, we purify and flatten the input formula by introducing fresh variables for multisets and base elements to separate the multiset constraints from constraints in the base theory. In our example, there are no base theory constraints. Purification and flattening of formula (5.1) results in the formula:

$$Y \subseteq X \wedge X' = X_1 \uplus Z \wedge X_1 = X \setminus Y \wedge Z \prec_{\mathsf{m}} Y \wedge \neg(X' \prec_{\mathsf{m}} X)$$

The next step is to replace all multiset atoms by their point-wise definitions on the base elements. This gives the following formula:

$$\begin{array}{l}
(\forall x.\, Y(x) \leq X(x)) \wedge \\
(\forall x.\, X'(x) = X_1(x) + Z(x)) \wedge \\
(\forall x.\, X_1(x) = \max\{X(x) - Y(x), 0\}) \wedge \\
(\exists y.Z(y) \neq Y(y)) \wedge (\forall z.\, Y(z) < Z(z) \rightarrow \exists y.\, Z(y) < Y(y) \wedge z \prec y) \wedge \\
(\ \ (\forall x.X'(x) = X(x)) \vee \exists x'.\, X(x') < X'(x') \wedge \forall x.\, X'(x) < X(x) \rightarrow \neg(x' \prec x)))
\end{array}$$

Next, we skolemize all existentially quantified variables. In our example this introduces two Skolem constants e_1, e_2 and one Skolem function w. Skolemization and Skolem constants are defined in Section 5.4. They are used to eliminate the existential quantifiers from a first-order formula and produce an equisatisfiable formula. The resulting formula is:

$$\begin{array}{l}
(\forall x.\, Y(x) \leq X(x)) \wedge \\
(\forall x.\, X'(x) = X_1(x) + Z(x)) \wedge \\
(\forall x.\, X_1(x) = \max\{X(x) - Y(x), 0\}) \wedge \\
Z(e_1) \neq Y(e_1) \wedge (\forall y'.\, Y(y') < Z(y') \rightarrow Z(w(y')) < Y(w(y')) \wedge y' \prec w(y'))) \wedge \\
(\ \ (\forall x.X'(x) = X(x)) \vee \ \ X(e_2) < X'(e_2) \wedge \forall x.\, X'(x) < X(x) \rightarrow \neg(e_2 \prec x))
\end{array}$$

The idea is now to replace each remaining universal quantifier with a finite conjunction by instantiating each quantifier with finitely many ground terms generated from the constants appearing in the formula and the introduced Skolem functions. The problem is that finite instantiation is in general incomplete because the Skolem functions coming from the ordering constraints generate an infinite Herbrand universe. The Herbrand universe, defined in Section 5.4, is a model for a formula that exists for every satisfiable first-order formula. Before instantiation we therefore first conjoin the skolemized formula with additional axioms that

further constrain the Skolem functions. In our example, we add the two axioms:

$$\forall x\, y.\, Y(x) < Z(x) \wedge w(x) \prec y \rightarrow Y(y) \leq Z(y), \quad \forall x.\, Z(x) = Y(x) \rightarrow w(x) = x$$

We will show in Section 5.6 that this step is sound and ensures that instantiation of the strengthened formula with the terms $e_1, e_2, w(e_1)$ and $w(e_2)$ is sufficient for proving unsatisfiability of the original constraint. The instantiated formula is a quantifier-free formula over symbols of the base theory (such as the preorder $\prec$), the theory of linear arithmetic, and the theory of free function symbols (the multisets and the Skolem functions). The satisfiability of such formulas can be decided using a Nelson-Oppen combination of the decision procedures for the corresponding component theories. In our example, the instantiated formula implies the following disjunction:

$$\begin{aligned}
&Z(e_1) \neq Y(e_1) \wedge X'(e_1) = X(e_1) - Y(e_1) + Z(e_1) \wedge X'(e_1) = X(e_1) \vee \\
&X'(e_2) = X(e_2) - Y(e_2) + Z(e_2) \wedge X(e_2) < X'(e_2) \wedge Y(e_2) \geq Z(e_2) \vee \\
&\quad X'(w(e_2)) = X(w(e_2)) - Y(w(e_2)) + Z(w(e_2)) \wedge \\
&\qquad Z(w(e_2)) < Y(w(e_2)) \wedge X'(w(e_2)) \geq X(w(e_2)) \vee \\
&e_2 \prec w(e_2) \wedge \neg(e_2 \prec w(e_2))
\end{aligned}$$

Observe that each of the disjuncts is unsatisfiable and, hence, so is the original formula (5.1).

5.4 Basic Definitions

Before we describe the logic and decision procedure for multiset orderings, we recall some definitions and facts that are widely used in theorem proving.

Sorted logic. A *signature* Σ is a tuple (S, Ω), where S is a countable set of sorts and Ω is a countable set of function symbols f. Every $f \in \Omega$ is associated with an arity $n \geq 0$ and a sort $s_1 \times \cdots \times s_n \rightarrow s_0$ with $s_i \in S$ for all $i \leq n$. Function symbols of arity 0 are called *constant symbols*. For the description of our problem we will consider three sorts: $S = \{\mathsf{int}, \mathsf{bool}, \mathsf{elem}\}$. We treat predicates of sort $s_1 \times \cdots \times s_n$ as function symbols of sort $s_1 \times \ldots \times s_n \rightarrow \mathsf{bool}$. We say that a signature Σ_1 extends a signature Σ_2 if Σ_1 contains at least the sorts and function symbols of Σ_2. Let V be a countably infinite set of sorted variables, disjoint from Ω. Terms are built as usual from the function symbols in Ω and variables taken from V. We denote by $t : s$ that term t has sort s. A term t is *ground*, if no variable appears in t. We denote by $\mathsf{Terms}(\Sigma)$ the set of all ground Σ-terms. An *atom* is either constructed from the equality symbol $t_1 = t_2$ applied to terms t_1 and t_2 of the same sort, or by applying a predicate symbol to terms of the respective sorts. Formulas are built from atoms as usual, using boolean connectives and quantifiers. A formula F is called *closed* or a *sentence* if no variable appears free in F.

Structures. Given a signature $\Sigma = (S, \Omega)$, a Σ-*structure* α is a function that maps each sort $s \in S$ to a non-empty set $\alpha(s)$ and each function symbol $f \in \Omega$ of sort $s_1 \times \cdots \times s_n \rightarrow s_0$ to a function $\alpha(f) : \alpha(s_1) \times \cdots \times \alpha(s_n) \rightarrow \alpha(s_0)$. Set $\alpha(s)$ is also called α-domain of the sort s. We assume that

all structures interpret the sort bool by the set of Booleans {true, false}, and the sort int by the set of all integers $\mathbb{Z}$. The sort elem will serve as our base set for defining multisets. We speak of $\alpha(\mathsf{elem})$ simply as the *domain* of α and often identify the two.

For a Σ-structure α and a *variable assignment* $\beta : V \to \alpha(S)$, the evaluation of a term (respectively a formula) in α, β is defined as usual:

$$\begin{aligned} \alpha, \beta(x) &= \beta(x), \quad \text{for } x \in V \\ \alpha, \beta(f(t_1, \ldots, t_n)) &= \alpha(f)(\alpha, \beta(t_1), \ldots, \alpha, \beta(t_n)) \end{aligned}$$

We use the standard interpretations for the equality symbol and propositional connectives. A quantified variable of sort s ranges over all elements of $\alpha(s)$. For ground terms t, we skip the variable assignment and simply write $\alpha(t)$ for its evaluation in α. The notions of satisfiability, validity, and entailment of formulas are also defined as usual. We write $\alpha, \beta \models F$ if α satisfies F under β. Similarly, we write $\alpha \models F$ if α satisfies F for all variable assignments β. In this case, we also call α a *model* of F.

Herbrand model and Herbrand's theorem. A Herbrand structure is a Σ-structure, where every sort is interpreted by the set of its ground terms. A function symbol $f \in \Omega$ of sort $s_1 \times \cdots \times s_n \to s_0$ is interprated in a natural way: given $t_1 : s_1, \ldots, t_n : s_n$, $\alpha(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n) : s_0$. Herbrand's theorem states that if a set of closed formula is satisfiable, then it also has a Herbrand model. In first-order theorem proving, it is enough to consider only a Herbrand structure when constructing a model.

Theories. A Σ-*theory* $\mathcal{T}$ for a signature Σ is simply a set of Σ-structures. Sometimes we identify a theory by a set of Σ-sentences $\mathcal{K}$, meaning the set of all Σ-models of $\mathcal{K}$. We then call $\mathcal{K}$ the *axioms* of the theory. The satisfiability problem for a Σ-theory $\mathcal{T}$ and a set of Σ-formulas $\mathcal{F}$ is to decide whether a given $F \in \mathcal{F}$ is satisfiable in some structure of $\mathcal{T}$. If the set of formulas $\mathcal{F}$ is clear from the context, we simply speak of the satisfiability problem of the theory $\mathcal{T}$. A Σ_2-theory $\mathcal{T}_2$ is an *extension* of a Σ_1-theory $\mathcal{T}_1$ if Σ_2 is an extension of Σ_1 and for every $\alpha \in \mathcal{T}_2$, the restriction $\alpha|_{\Sigma_1}$ of α to the sorts and symbols of Σ_1 is a structure in $\mathcal{T}_1$. A Σ-theory $\mathcal{T}$ is called *stably infinite* with respect to a set of formulas $\mathcal{F}$, if for every formula $F \in \mathcal{F}$ which is satisfiable in $\mathcal{T}$, there exists a model α of F in $\mathcal{T}$, such that the domain of α has infinite cardinality.

Skolemization. Skolemization is a method for removing existential quantifiers from a first order formula. It is among the first steps performed by first-order theorem provers so that the resulting formula contains only universally quantified variables. The intuition behind skolemization is to replace every $\exists y$ by a concrete choice function computing y from all the arguments y depends on. Computation of the Skolem form of a formula is described with the

following translation step:

$$\forall x_1,\ldots,x_n \exists y F \quad \Rightarrow_S \quad \forall x_1,\ldots,x_n F[f(x_1,\ldots,x_n)/y]$$

Here f is a new function symbol (called a Skolem function or a Skolem constant). This reduction has to be applied to the outermost existential quantifier and repeated as along as they are the existential quantifiers in the formula. This transformation is satisfiability preserving. As an illustration, formula $\exists x.\forall y.\forall z.\exists u.p(x,y) \vee q(z,u)$ is skolemized to formula $\forall y.\forall z.p(s_0,y) \vee q(z,s_1(y,z))$. We introduced Skolem constant s_0 and Skolem function s_1.

5.5 POSSUM : Multiset Constraints over Preordered Sets

In this section, we formally define the constraints whose satisfiability we study in this chapter. We first explain the definition of preordered sets and afterwards we describe multiset constraints over preordered sets. We consider the following signature $\Sigma = (\{\mathsf{bool},\mathsf{int},\mathsf{elem}\},\Omega)$, where Ω contains the symbols for the boolean connectives and arithmetic operators. In addition, it also contains the symbol $\preceq$ with the sort $\mathsf{elem}\times\mathsf{elem}\rightarrow\mathsf{bool}$.

5.5.1 Finite Multisets over Preordered Sets

We assume that Σ_{elem} is a signature containing at least the binary predicate symbol $\preceq$ over sort elem. Let $\mathscr{F}_{\mathsf{elem}}$ be the set of all quantifier-free ground formulas over signature Σ_{elem}. We will use the formula $t_1 \prec t_2$ as syntactic shorthand for the formula $t_1 \neq t_2 \wedge t_1 \preceq t_2$. A binary relation R defined on a set E, such that R is reflexive and transitive is called a *preorder* and set (E,R) is called a preordered set. A theory of preordered sets $\mathscr{T}_{\mathsf{elem}}$ is a Σ_{elem}-theory such that for all structures $\alpha \in \mathscr{T}_{\mathsf{elem}}$, $(\alpha(\mathsf{elem}),\alpha(\preceq))$ is a preordered set, i.e., every structure $\alpha \in \mathscr{T}_{\mathsf{elem}}$ satisfies the following two axioms:

$$\forall x:\mathsf{elem}.\ x \preceq x \quad \text{(refl)} \qquad\qquad \forall x,y,z:\mathsf{elem}.\ x \preceq y \wedge y \preceq z \rightarrow x \preceq z \quad \text{(trans)}$$

For the rest of this chapter, we fix such a theory $\mathscr{T}_{\mathsf{elem}}$. We require that the satisfiability problem for $\mathscr{F}_{\mathsf{elem}}$ and $\mathscr{T}_{\mathsf{elem}}$ is decidable. We further require that $\mathscr{T}_{\mathsf{elem}}$ is stably-infinite with respect to the formulas $\mathscr{F}_{\mathsf{elem}}$. We call $\mathscr{T}_{\mathsf{elem}}$ the *base theory*.

Let Ω_{la} be the function and constant symbols of linear integer arithmetic

$$\Omega_{\mathsf{la}} = \{+,-,\max,\min,\ldots,-2,-1,0,1,2,\ldots,-2\cdot,-1\cdot,0\cdot,1\cdot,2\cdot\}$$

with their appropriate sorts (the function symbol $C\cdot$ denotes multiplication with integer constant C). We assume that these symbols are disjoint from the symbols in Σ_{elem}. We represent multisets as function symbols of sort $\mathsf{elem}\rightarrow\mathsf{int}$. Let $\mathscr{M}$ be a countably infinite set of function symbols of this sort, disjoint from the symbols in Σ_{elem} and Ω_{la}. Further, let Σ_{mset}

be the signature Σ_{elem} extended with the symbols $\mathscr{M}$ and Ω_{la}. We then define the theory $\mathscr{T}_{\mathsf{mset}}$ of finite preordered multisets over $\mathscr{T}_{\mathsf{elem}}$ as follows. The theory $\mathscr{T}_{\mathsf{mset}}$ is the set of all structures α such that α is an extension of a structure in $\mathscr{T}_{\mathsf{elem}}$ to a Σ_{mset}-structure and α satisfies the following conditions:

- α gives the standard interpretation to the arithmetic symbols, and
- α interprets each $X \in \mathscr{M}$ as a finite multiset, i.e.,
  1. for all $e \in \alpha(\mathsf{elem})$, $\alpha(X)(e) \geq 0$
  2. there are only finitely many $e \in \alpha(\mathsf{elem})$ such that $\alpha(X)(e) > 0$

5.5.2 Syntax and Semantics of POSSUM Formulas

Syntax. Figure 5.5 defines the POSSUM formulas. A POSSUM formula is an arbitrary propositional combination of atomic formulas. The atomic formulas are relations between multiset expressions, relations between arithmetic expressions, atoms over the base signature Σ_{elem}, and restricted quantified formulas $\mathsf{F}^{\forall}$. An example of a base signature atom is the formula $e_1 \preceq e_2$, where e_1 and e_2 are two constants of sort elem. The formulas $\mathsf{F}^{\forall}$ express universal quantification over variables of sort elem. The formulas below the quantifiers can express arithmetic relations between multiplicities $X(x)$ of the quantified variables or ordering constraints between these variables. Using these quantified formulas we can express that some constant e is maximal in a multiset X: $\forall x.\, X(x) > 0 \rightarrow x \preceq e$. The important restriction for the formulas below the universal quantifiers is that the quantified variables x are not allowed to appear below function symbols of the base signature Σ_{elem}. This is enforced by allowing only ground Σ_{elem}-terms t below the quantifiers. Note also that there are no POSSUM formulas with $\mathsf{F}^{\forall}$ atoms that have an alternating quantifier prefix. We call a subset $\mathscr{F}$ of POSSUM formulas *quantifier-bounded* if the number of quantified variables appearing in $\mathsf{F}^{\forall}$ subformulas of formulas in $\mathscr{F}$ is bounded.

Semantics. POSSUM formulas are interpreted in the structures of the theory $\mathscr{T}_{\mathsf{mset}}$. The semantics of POSSUM formulas extends the semantics of first-order formulas defined in Section 5.4. Note that with the exception of atomic formulas that express relations on multisets, all atomic formulas are first-order formulas. Thus, we only need to define the semantics of formulas of the form $M_1 = M_2$, $M_1 \subseteq M_2$, and $M_1 \preceq_{\mathsf{m}} M_2$. Let α be a structure in $\mathscr{T}_{\mathsf{mset}}$. First, we extend the interpretation $\alpha(X)$ of multisets $X \in \mathscr{M}$ in α to multiset expressions. The interpretation is defined point-wise for all $e \in \alpha$ and recursively on the structure of the expression:

$$\begin{aligned} \alpha(\emptyset)(e) &= 0 \\ \alpha(\{t^K\})(e) &= \text{if } \alpha(t) = e \text{ then } \alpha(K) \text{ else } 0 \\ \alpha(M_1 \cup M_2)(e) &= \max\{\alpha(M_1)(e), \alpha(M_2)(e)\} \end{aligned}$$

top-level formulas:

$$\begin{array}{rcl} F & ::= & A \mid F \wedge F \mid \neg F \\ A & ::= & M = M \mid M \subseteq M \mid K = K \mid K \leq K \mid M \preceq_{\mathsf{m}} M \mid A_{\mathsf{elem}} \mid \mathsf{F}^{\forall} \\ M & ::= & X \mid \emptyset \mid \{t^K\} \mid M \cap M \mid M \cup M \mid M \uplus M \mid M \setminus M \mid \mathsf{set}(M) \\ K & ::= & k \mid C \mid K + K \mid C \cdot K \end{array}$$

restricted quantified formulas:

$$\begin{array}{rcl} \mathsf{F}^{\forall} & ::= & \forall x : \mathsf{elem}.\mathsf{F}^{\forall} \mid \forall x : \mathsf{elem}.\mathsf{F}^{\mathsf{in}} \\ \mathsf{F}^{\mathsf{in}} & ::= & \mathsf{A}^{\mathsf{in}} \mid \mathsf{F}^{\mathsf{in}} \wedge \mathsf{F}^{\mathsf{in}} \mid \neg \mathsf{F}^{\mathsf{in}} \\ \mathsf{A}^{\mathsf{in}} & ::= & \mathsf{t}^{\mathsf{in}} \leq \mathsf{t}^{\mathsf{in}} \mid \mathsf{t}^{\mathsf{in}} = \mathsf{t}^{\mathsf{in}} \mid \mathsf{E}^{\mathsf{in}} \preceq \mathsf{E}^{\mathsf{in}} \mid \mathsf{E}^{\mathsf{in}} = \mathsf{E}^{\mathsf{in}} \\ \mathsf{t}^{\mathsf{in}} & ::= & X(\mathsf{E}^{\mathsf{in}}) \mid C \mid \mathsf{t}^{\mathsf{in}} + \mathsf{t}^{\mathsf{in}} \mid C \cdot \mathsf{t}^{\mathsf{in}} \\ \mathsf{E}^{\mathsf{in}} & ::= & x \mid t \end{array}$$

terminals:

A_{elem} - ground Σ_{elem}-atom ; X - multiset ; k - integer variable; C - integer constant
t - ground Σ_{elem}-term of sort elem; x - variable of sort elem

Figure 5.5: Syntax for Multiset Constraints over Preordered Sets (POSSUM)

$$\begin{aligned} \alpha(M_1 \cap M_2)(e) &= \min\{\alpha(M_1)(e), \alpha(M_2)(e)\} \\ \alpha(M_1 \uplus M_2)(e) &= \alpha(M_1)(e) + \alpha(M_2)(e) \\ \alpha(M_1 \setminus M_2)(e) &= \max\{\alpha(M_1)(e) - \alpha(M_2)(e), 0\} \\ \alpha(\mathsf{set}(M))(e) &= \min\{\alpha(M)(e), 1\} \end{aligned}$$

For defining the interpretations of the predicate symbols $=$, $\subseteq$, and $\preceq_{\mathsf{m}}$ on multisets, we define corresponding relations $=_{\mathsf{m}}$, $\subseteq_{\mathsf{m}}$, and $\preceq_{\mathsf{m}}$ at the meta-level. Let m_1, m_2 be functions $\alpha(\mathsf{elem}) \to \mathbb{N}$. The relations $=_{\mathsf{m}}$ and $\subseteq_{\mathsf{m}}$ are defined point-wise as expected:

$$\begin{array}{rcl} m_1 =_{\mathsf{m}} m_2 & \Leftrightarrow & \forall e \in \alpha(\mathsf{elem}).m_1(e) = m_2(e) \\ m_1 \subseteq_{\mathsf{m}} m_2 & \Leftrightarrow & \forall e \in \alpha(\mathsf{elem}).m_1(e) \leq m_2(e) \end{array}$$

For defining the multiset ordering we identify $\prec$ with the irreflexive reduct of the relation $\alpha(\preceq)$. The relation $\preceq_{\mathsf{m}}$ is then defined as follows:

$$\begin{array}{rcl} m_1 \preceq_{\mathsf{m}} m_2 & \Leftrightarrow & \forall e_1 \in \alpha.\ m_1(e_1) > m_2(e_1) \Rightarrow \\ & & \qquad \exists e_2 \in \alpha.m_2(e_2) > m_1(e_2) \wedge e_1 \prec e_2 \end{array} \tag{5.2}$$

Note that this is not the standard definition of the multiset ordering that was originally used in [Dershowitz and Manna(1979)]. However, in order to reduce the number of multiset variables we use the simpler definition (5.2). For finite multisets, definition (5.2) is equivalent to the standard one (for proof see [Baader and Nipkow(1998), Lemma 2.5.6, p.24]).

5.6 Decidability of POSSUM

We now describe the decision procedure for POSSUM . The idea of the decision procedure is to reduce satisfiability of a POSSUM formula to satisfiability of a formula in a particular first-order theory, namely, the disjoint combination of the base theory $\mathcal{T}_{\mathsf{elem}}$, the theory of linear integer arithmetic, and the theory of uninterpreted function symbols.

Reduction to a first-order theory. In the following, we show how to decide conjunctions of POSSUM literals. The extension of the decision procedure to arbitrary Boolean combinations of literals is straightforward. Thus, let F be a fixed POSSUM conjunction. The first step of our decision procedure is to rewrite F into a quantified first-order formula by expanding all multiset constraints to their point-wise definitions.

For two multiset variables X and Y we denote by $L_{X,Y}$ the multiset $X \setminus Y$ and by $U_{X,Y}$ the multiset $Y \setminus X$. Similarly, for a given element x we use $L_{X,Y}(x)$ as a shorthand for the expression $X(x) - Y(x)$ and $U_{X,Y}(x)$ for $Y(x) - X(x)$. The algorithm for rewriting F is then as follows:

1. Purify and flatten all multiset constraints in F:
   $C[M] \rightsquigarrow X_f = M \wedge C[X_f]$
   where $X_f \in \mathscr{M}$ is a fresh multiset and M is

   (a) either of the form $M_1 \cup M_2$, $M_1 \cap M_2$, $M_1 \uplus M_2$, $M_1 \setminus M_2$, and at least one M_i is not a multiset $X \in \mathscr{M}$

   (b) or of the form $\emptyset$, $\{t^k\}$, $\mathsf{set}(M_1)$, and they are not in a conjunct of the form $X = M$ or $M = X$ for some multiset $X \in \mathscr{M}$.

2. Replace all multiset atoms by their point-wise definitions

$$
\begin{array}{lcl}
C[X = \emptyset] & \rightsquigarrow & C[\forall x.\, X(x) = 0] \\
C[X = \{e^k\}] & \rightsquigarrow & C[X(e) = k \wedge \forall x.\, x \neq e \rightarrow X(e) = 0] \\
C[X = Y \cup Z] & \rightsquigarrow & C[\forall x.\, X(x) = \max\{Y(x), Z(x)\}] \\
C[X = Y \cap Z] & \rightsquigarrow & C[\forall x.\, X(x) = \min\{Y(x), Z(x)\}] \\
C[X = Y \uplus Z] & \rightsquigarrow & C[\forall x.\, X(x) = Y(x) + Z(x)] \\
C[X = Y \setminus Z] & \rightsquigarrow & C[\forall x.\, X(x) = \max\{Y(x) - Z(x), 0\}] \\
C[X = Y] & \rightsquigarrow & C[\forall x.\, X(x) = Y(x)] \\
C[X \preceq_{\mathsf{m}} Y] & \rightsquigarrow & C[\forall x.\, L_{X,Y}(x) > 0 \rightarrow \exists y.\, U_{X,Y}(y) > 0 \wedge x \prec y]
\end{array}
$$

3. Compute negation normal form, i.e., push all negations down to the atoms

4. Skolemize all existentially quantified variables

5. For every multiset X occurring in the formula, add the formula $\forall x.\, X(x) \geq 0$ as an additional conjunct

After rewriting, the resulting formula is of the form $\mathscr{K} \wedge G$ where G is a ground formula and $\mathscr{K}$ is a conjunction of universally quantified formulas. Clearly, each step of the rewriting transforms the input formula into an equisatisfiable formula.

Lemma 5.1 *The formulas F and $\mathscr{K} \wedge G$ are equisatisfiable in the theory of preordered multisets.*

Quantifier instantiation.

If the original formula F does not contain any atom of the form $X \preceq_{\mathsf{m}} Y$ (or if there is some but it appears under the negation), then checking satisfiability of $\mathscr{K} \wedge G$ is relatively simple: it is enough to construct the Herbrand universe, which is in this particular case finite. We would instantiate all universally quantified variables with the elements of the Herbrand domain. This way the whole formula becomes ground and checking its satisfiability can be done using any SMT solver. However, $X \preceq_{\mathsf{m}} Y$ introduces the Skolem function of arity 1. In that case the Herbrand domain becomes infinite and the previous solution cannot be applied.

We will now show that there exists a finite and computable set of ground terms $T_{\mathscr{K},G}$ of sort elem such that $\mathscr{K} \wedge G$ is equisatisfiable to the formula $\mathscr{K}[T_{\mathscr{K},G}] \wedge G$, where $\mathscr{K}[T_{\mathscr{K},G}]$ is a ground formula obtained by instantiating all quantified variables appearing in $\mathscr{K}$ with the terms in $T_{\mathscr{K},G}$.

Throughout the rest of this section, we denote by E the set of all ground terms of sort elem appearing in $\mathscr{K} \wedge G$. The set E contains the ground terms appearing in the initial formula F and Skolem constants that have been introduced for top-level existentially quantified variables in Step 4 of the rewrite algorithm. Zarba showed in [Zarba(2002a)] that for formulas F without ordering constraints on multisets and formulas $\mathsf{F}^{\forall}$, the theory $\mathscr{K}$ is (what is now known as) a *stably local theory extension* [Sofronie-Stokkermans(2005)]. This means that if F does not contain ordering constraints then F is equisatisfiable to the formula $\mathscr{K}[E] \wedge G$. The reason for locality of $\mathscr{K}$ in this case is simply that instantiation of the quantifiers in $\mathscr{K}$ with terms of sort elem will not create new terms of the same sort. Unfortunately, in the presence of ordering constraints this is no longer true, i.e., instantiation of $\mathscr{K}$ with the terms in E alone is not sufficient.

For an illustration of this behavior, reconsider the defining formula (5.2) for the ordering constraint $X \preceq_{\mathsf{m}} Y$. This formula contains $\forall\exists$ quantification over variables of sort elem. Skolemization of this formula thus gives

$$\forall x.\, L_{X,Y}(x) > 0 \rightarrow U_{X,Y}(w_{X,Y}(x)) > 0 \wedge x \prec w_{X,Y}(x) \tag{5.3}$$

where $w_{X,Y}$ is a fresh Skolem function. We call these Skolem functions $\prec_{\mathsf{m}}$*-witness functions* and terms constructed from these functions $\prec_{\mathsf{m}}$*-witnesses*. Figure 5.6 represents a $\prec_{\mathsf{m}}$-witnesses of an element e. Instantiation of formula (5.3) with a term $e \in E$ generates a new $\prec_{\mathsf{m}}$-witness $w_{X,Y}(e)$ of sort elem, which is not already contained in E. For completeness, we

have to instantiate $\mathscr{K}$ recursively with these $\prec_{\mathsf{m}}$-witnesses.

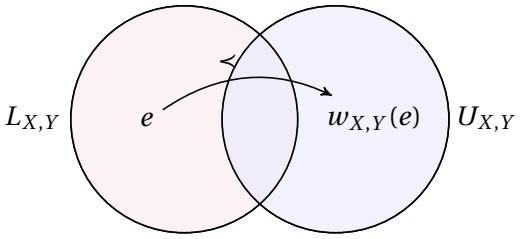

Figure 5.6: An element and its $\prec_{\mathsf{m}}$-witnesses

We now show that we can put additional constraints on the $\prec_{\mathsf{m}}$-witness functions such that we only need to consider finitely many $\prec_{\mathsf{m}}$-witnesses for the instantiation of $\mathscr{K}$. These additional constraints are as follows. First, we enforce that the $\prec_{\mathsf{m}}$-witness function $w_{X,Y}$ only chooses maximal elements in the multiset $U_{X,Y}$ and, second, we require that each element outside $L_{X,Y}$ is mapped to itself. Figure 5.7 depicts those additional rules. Formally, these constraints are expressed by the following two axioms:

$$\forall x\, y.\, L_{X,Y}(x) > 0 \wedge w_{X,Y}(x) \prec y \rightarrow U_{X,Y}(y) = 0 \tag{5.4}$$

$$\forall x.\, L_{X,Y}(x) = 0 \rightarrow w_{X,Y}(x) = x \tag{5.5}$$

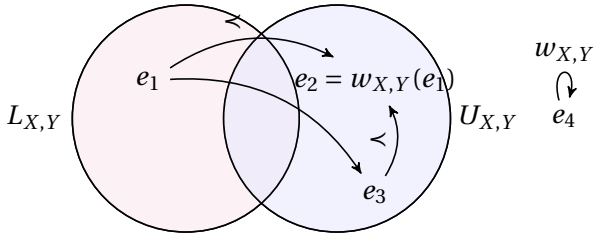

Figure 5.7: An illustration for the rules described by formulas (5.4) and (5.5)

The existence of such constrained witness functions is guaranteed by the fact that we restrict ourselves to finite multisets. In particular, given a $\prec_{\mathsf{m}}$-witness function $w_{X,Y}$ satisfying axiom (5.3), we can define a new witness function that maps every e in $L_{X,Y}$ to the maximal element of some ascending chain starting from $w_{X,Y}(e)$ in $U_{X,Y}$. Finiteness of the multiset $U_{X,Y}$ guarantees the existence of such a maximal element.

For the rest of this section let W be the set of all $\prec_{\mathsf{m}}$-witness functions occurring in $\mathscr{K}$ and let $\mathscr{K}_W$ be the conjunction of axioms (5.4) and (5.5) for all $w_{X,Y} \in W$.

Lemma 5.2 *The formulas $\mathscr{K} \wedge G$ and $\mathscr{K} \wedge \mathscr{K}_W \wedge G$ are equisatisfiable in the theory of preordered multisets.*

Let $T_{W,E}$ be the smallest set of ground terms that satisfies the following two conditions:

1. $E \subseteq T_{W,E}$
2. if $t \in T_{W,E}$ and $w_{X,Y} \in W$ then $w_{X,Y}(t) \in T_{W,E}$

For a term $t \in T_{W,E}$ of the form $t = w_n \dots w_1(e)$ where $e \in S$, we define $t_0 = e$ and denote by t_i, for $1 \leq i \leq n$, the subterm $w_i \dots w_1(e)$ of t. We call $t \in T_{W,E}$ a strict chain in a structure α iff α satisfies $t_i \prec t_{i+1}$ for all i with $0 \leq i < n$. We say that a strict chain $t \in T_{W,E}$ in a structure α is *maximal* if t is not a proper subterm of any other strict chain $t' \in T_{W,E}$ in α. For a structure α and a set of ground terms T, we denote by $\alpha(T)$ the set $\alpha(T) = \{\alpha(t) \mid t \in T\}$.

Now, define $T_{\mathcal{K},G}$ as the set of all terms $t \in T_{W,E}$ such that each function $w_{X,Y}$ occurs at most once in t. Clearly, the set $T_{\mathcal{K},G}$ is finite, since W is finite. We can now show that in models of $\mathcal{K} \wedge \mathcal{K}_W$, the terms $T_{W,E}$ are partitioned into finitely many equivalence classes, each of which is represented by some term in $T_{\mathcal{K},G}$.

Lemma 5.3 *For all models α of $\mathcal{K} \wedge \mathcal{K}_W$, $\alpha(T_{W,E}) = \alpha(T_{\mathcal{K},G})$.*

Proof. Let α be a model of $\mathcal{K} \wedge \mathcal{K}_W$. Note that from strictness of $\prec$, and axioms (5.3) and (5.5) it follows that for all terms t of sort elem and $w_{X,Y} \in W$, either $\alpha \models w_{X,Y}(t) = t$ or $\alpha \models t \prec w_{X,Y}(t)$ holds.

The proof goes by contradiction. Thus, assume there exists $t \in T_{W,E}$ such that $\alpha(t) \notin \alpha(T_{\mathcal{K},G})$. Then remove all function applications w_i from t for which $\alpha \models w_i(t_{i-1}) = t_{i-1}$, obtaining a term $t' \in T_{W,E}$. Then t' is a strict chain and $\alpha \models t = t'$. From this we conclude that $\alpha(t') \notin \alpha(T_{\mathcal{K},G})$ and therefore $t' \notin T_{\mathcal{K},G}$. Hence, there exists i, j with $1 \leq i < j < k$ and multiset variables X, Y such that $w'_i = w'_j = w_{X,Y} \in W$. We then have $\alpha \models t'_{i-1} \prec w_{X,Y}(t'_{i-1})$. Based on strictness of $\prec$, axiom (5.5) and axiom $\forall x.L_{X,Y}(x) \geq 0$ we conclude $\alpha \models L_{X,Y}(t'_{i-1}) > 0$. Similarly, we conclude $\alpha \models L_{X,Y}(t'_{j-1}) > 0$. By transitivity of $\prec$ and construction of t', we further have that α satisfies $w_{X,Y}(t'_{i-1}) \prec w_{X,Y}(t'_{j-1})$. From axiom (5.4) we then conclude $\alpha \models U_{X,Y}(w_{X,Y}(t'_{j-1})) = 0$. However, axiom (5.3) implies $\alpha \models U_{X,Y}(w_{X,Y}(t'_{j-1})) > 0$, which gives us a contradiction.

From Lemma 5.3 it follows that we only need to instantiate the axioms $\mathcal{K} \wedge \mathcal{K}_W$ with the terms in $T_{\mathcal{K},G}$.

Lemma 5.4 *The formulas $\mathcal{K} \wedge \mathcal{K}_W \wedge G$ and $\mathcal{K}[T_{\mathcal{K},G}] \wedge \mathcal{K}_W[T_{\mathcal{K},G}] \wedge G$ are equisatisfiable in the theory of preordered multisets.*

The formula $\mathcal{K}[T_{\mathcal{K},G}] \wedge \mathcal{K}_W[T_{\mathcal{K},G}] \wedge G$ can now be purified obtaining an equisatisfiable formula $G_{\mathsf{elem}} \wedge G_{\mathsf{la}} \wedge G_{\mathsf{euf}}$ such that the three conjuncts G_{elem}, G_{la}, and G_{euf} only share constant symbols and:

- G_{elem} is a constraint over symbols in the theory $\mathcal{T}_{\mathsf{elem}}$

- G_{la} is a linear integer arithmetic constraint, and
- G_{euf} is a constraint built from uninterpreted function symbols and equality

We can thus check satisfiability of F by checking satisfiability of $G_{\mathsf{elem}} \wedge G_{\mathsf{la}} \wedge G_{\mathsf{euf}}$ in the disjoint combination of the theory $\mathcal{T}_{\mathsf{elem}}$, the theory of linear integer arithmetic, and the theory of uninterpreted function symbols with equality. By our assumptions on the theory $\mathcal{T}_{\mathsf{elem}}$, this combined theory can be decided using standard Nelson-Oppen combination techniques [Nelson and Oppen(1979)].

Theorem 5.5 *The satisfiability problem for* POSSUM *formulas is decidable.*

5.7 Complexity of POSSUM

We will now establish that the satisfiability problem for the quantifier-bounded fragments of POSSUM is in NP, provided the base theory $\mathcal{T}_{\mathsf{elem}}$ is also decidable in NP. Since POSSUM formulas subsume propositional logic this bound is tight.

We have seen in the previous section that we can reduce a POSSUM conjunction F to a ground formula $\mathcal{K}[T_{\mathcal{K},G}] \wedge \mathcal{K}_W[T_{\mathcal{K},G}] \wedge G$ whose satisfiability can be decided using the decision procedure of the base theory. However, the size of the resulting formula can be exponential in the size of the input formula F because the size of the set $T_{\mathcal{K},G}$ used for the instantiation is exponential in the number of $\prec_{\mathsf{m}}$-witness functions W. The following theorem implies that this exponential blowup can be avoided.

Theorem 5.6 *If the formula $\mathcal{K} \wedge \mathcal{K}_W \wedge G$ is satisfiable then it has a model α such that $|\alpha(T_{\mathcal{K},G})| \in \mathcal{O}(|W|^2 \cdot |E|)$, where W is the set of all $\prec_{\mathsf{m}}$-witness functions occurring in $\mathcal{K}$ and E is the set of all ground terms of sort* elem *appearing in $\mathcal{K} \wedge G$.*

Proof.

Assume $\mathcal{K} \wedge \mathcal{K}_W \wedge G$ is satisfiable and let α_0 be one of its models. Further, let $n = |W|$ and $m = |E|$. From α_0 we construct a model α with $|\alpha(T_{\mathcal{K},G})| \in \mathcal{O}(n^2 m)$ by collapsing redundant strict chains in α_0.

For this purpose, we choose a set T of strict chains in α_0 such that for every term $e \in E$ and witness function $w_{X,Y} \in W$, there is at most one chain $t \in T$ that starts in $L_{X,Y}$ with e, i.e., t contains $w_{X,Y}(e)$ as a subterm. Figure 5.8 shows an example of two redundant chains. Since we consider an arbitrary preorder, not necessarily total, the figure shows that it is possible for two elements e_1 and e_2, such that $e_1 \in [e_2]$, to find the witnesses $w_{X,Y}(e_1)$ and $w_{X,Y}(e_2)$ that are incomparable. The witness of e_1 is clearly also a witness for e_2 and we do not need any

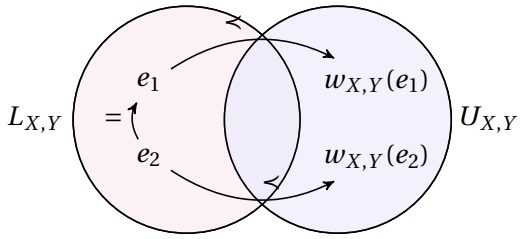

Figure 5.8: An example of two redundant chains in α_0

of the chains that contain $w_{X,Y}(e_2)$. Therefore we demand for the set T to contain only *one* chain passing through $[e_2]$ and containing a term $w_{X,Y}(t')$, for some t' such that $\alpha_0(t') \in [e_2]$.

Formally, let $E_=$ be the quotient of E with respect to the interpretation of the equality predicate $=$ in α_0 and denote by $[e] \in E_=$ the equivalence class of $e \in E$. Let T be a maximal subset T of $T_{K,G}$ such that

1. each $t \in T$ is a maximal strict chain in α_0
2. for each $w \in W$, if there is some $t \in T$ which contains w and starts in $e \in E$, then there is no other $t' \in T$ which contains $w(e')$ as a subterm, for any $e' \in [e]$

Clearly such a set T exists. A simple algorithm to construct T runs as follows. First, create a set T_0 that contains all the maximal strict chain in α_0. Then, for every $w \in W$ take a strict chain $t \in T_0$ such that w is contained in t. Let e be the starting element of t. Delete all t' from T_0 such that t' starts with $w(e')$, where $e' \in [e]$. Set T_0 is finite so this algorithm terminates and at the end we obtain set T. Let T^* be the set of all subterms $t_0, \ldots, t_k$ of the chains $t \in T$, where k is the length of chain t.

We now construct α from α_0 by collapsing all strict chains in α_0 to the chains in T. First, we let α agree with α_0 on the interpretation of all sorts and all symbols that are not witness functions. For each witness function $w \in W$ and $v \in \alpha(\mathsf{elem})$, we then define

$$\alpha(w)(v) = \begin{cases} \alpha_0(t) & \text{if } v = \alpha_0(e) \text{ for some } e \in E,\ \alpha_0(w(e)) \neq \alpha_0(e), \text{ and there is some term} \\ & t \in T^* \text{ with } t = w(t') \text{ for some } t' \text{ containing } e' \in [e], \\ \alpha_0(w)(v) & \text{otherwise} \end{cases}$$

Note that from the definition of T and α it follows that for all $t \in T^*$, $\alpha(t) = \alpha_0(t)$. Thus all terms in T^* are still strict chains in α.

We first prove that α is still a model of $\mathscr{K} \wedge \mathscr{K}_W \wedge G$. Since α_0 is a model of $\mathscr{K} \wedge \mathscr{K}_W \wedge G$ and α agrees with α_0 on all symbols that are not witness functions, we immediately conclude that α is also a model of G and all axioms of $\mathscr{K}$ that do not mention the witness functions. The fact that α still satisfies the remaining axioms (5.3)-(5.5) for all $w \in W$ also easily follows from the

definition of α.

As an illustration, we will show that α satisfies axiom (5.3). Let β be a valuation and let $\alpha, \beta \models L_{X,Y}(x) > 0$. Since α and α_0 agree on all the symbols but the witness function, it clearly means $\alpha, \beta \models U_{X,Y}(w_{X,Y}(x))$. We need to show that $\alpha, \beta \models x \prec w_{X,Y}(x)$. Let $v = \beta(x)$. If $\alpha(w_{X,Y})(v) = \alpha_0(w_{X,Y})(v)$ then clearly $\alpha, \beta \models x \prec w_{X,Y}(x)$, so let us assume that $\alpha(w_{X,Y})(v) \neq \alpha_0(w_{X,Y}\, v)(v)$. Then, by definition, $v = \alpha_0(e)$ for some $e \in E$ and $\alpha(w_{X,Y})(v) = \alpha_0(t)$ for some term $t \in T^*$ such that $t = w_{X,Y}(t')$ and t' contains $e' \in [e]$. The fact that t is a strict chain in α_0 also means that t is a strict chain in α. By the transitivity of $\prec$ we conclude $\alpha \models e' \prec t$. Since $e' \in [e]$ we further have $\alpha \models e = e'$ and hence $\alpha \models e \prec w_{X,Y}(e)$. Element $e \in E$ is a ground term and therefore $\alpha(e) = \alpha_0(e) = v$. Since $v = \beta(x)$, we have proved that $\alpha, \beta \models x \prec w_{X,Y}(x)$. The proofs for the other two axioms are similar.

We next observe that $\alpha(T^*) = \alpha(T_{K,G})$. Since $T^* \subseteq T_{\mathcal{K},G}$, it is obvious that $\alpha(T^*) \subseteq \alpha(T_{K,G})$. To show that $\alpha(T_{K,G}) \subseteq \alpha(T^*)$, let $t \in T_{K,G}$ be a term. If $t = e$, where e is a ground term, clearly $e \in T^*$. If $t = w(t')$ for some t', either there is only one chain containing w going through $[\alpha_0(t')]$ or there are several of them. In first case $t \in T^*$. In the second case t does not need to be contained in T^*, but by construction of α we have $\alpha(t) \in \alpha(T^*)$.

For proving that $|\alpha(T_{\mathcal{K},G})| \in \mathcal{O}(n^2 m)$ it is therefore enough to count the number of elements in T^*. For this purpose, fix $e \in E$ and let k be the maximal length of the chains t in T that start from some $e' \in [e]$. From strictness of the chains and Lemma 5.3 it follows that $k \leq n$. Let t be a chain of the maximal length. There are k witness functions occurring in t and thus by the second condition of the definition of T, none of those k witness functions can start a new chain. Using the second condition of the definition of T again, we conclude that there can be at most $n-k$ additional maximal chains, different from t, that will start from some $e' \in [e]$. That means that, if k is the maximal length of the chains starting from some $e' \in [e]$, then there are at most $n-k+1$ maximal chains in T that start from some $e' \in [e]$. Each of these chains has at most length k by assumption and thus at most $k+1$ subterms. Using this observation we derive that T^* contains at most $(n-k+1)(k+1)$ terms with some $e' \in [e]$ as a subterm. From $\max_{1 \leq k \leq n}\{(n-k+1)(k+1)\} \in \mathcal{O}(n^2)$ we then conclude $|T^*| \in \mathcal{O}(n^2 m)$.

Theorem 5.6 implies that we can guess a polynomial subset T of the terms $T_{K,G}$ and then use this subset to instantiate the axioms in $\mathcal{K} \wedge \mathcal{K}_W$. The size of the resulting formula $\mathcal{K}[T] \wedge \mathcal{K}_W[T] \wedge G$ is then polynomial in the size of the input formula, provided we bound the number of quantified variables in $\mathsf{F}^\forall$ subformulas of the input.

Theorem 5.7 *If the base theory $\mathcal{T}_{\mathsf{elem}}$ is decidable in NP then for the quantifier-bounded fragments of its* POSSUM *extension, the satisfiability problem is NP-complete.*

Practical Considerations. Our decision procedure is amenable to practical implementations using off-the-shelf SMT-solvers. In particular, using techniques developed for local theory extensions [Jacobs(2009)], we can postpone the exponential decomposition phase of guessing

the terms used for instantiation, by generating these terms lazily from models produced by the SMT solver. Also note that in practical applications such as checking validity of constraints generated from termination proofs, all multiset ordering constraints $X \prec_{\mathsf{m}} Y$ will typically have negative polarity. Since only positive occurrences of such constraints generate $\prec_{\mathsf{m}}$-witness functions, the set of terms $T_{K,G}$ will, in most practical cases, already be polynomial in the size of the input constraint.

5.8 Further Related Work

The logic POSSUM extends the logic of multisets with integers, which was shown to be NP-complete by Zarba [Zarba(2002a)]. This extension is non-trivial. In particular, Zarba only considers a disjoint combination of a base theory with the theory of multisets and does not support ordering constraints on multisets. Such constraints generate axioms with $\forall\exists$ quantification, which require a more intricate argument to establish completeness of local instantiation. The logic of multisets with cardinality constraints [Piskac and Kuncak(2008a)] also subsumes Zarba's logic and was shown to be NP-complete [Piskac and Kuncak(2008c)]. It is incomparable to our logic because it also does not support ordering constraints. On the other hand, POSSUM can only express very restricted cardinality constraints. In [Kuncak et al.(2010c)Kuncak, Piskac, and Suter] the theory of sets with cardinality constraints over totally ordered base sets was shown to be decidable in NP. This result can be generalized to multisets. Decidability of multisets over partially ordered base sets and with general cardinality constraints is open.

Local theory extensions [Sofronie-Stokkermans(2005)] formalize the general category of theories for which local quantifier instantiation techniques are complete. Some local theory extension of orders have been studied in [Sofronie-Stokkermans and Ihlemann(2007)]. Our extension of preorders to multiset orderings is an instance of the so called Ψ-local theory extensions, which have been introduced in [Ihlemann et al.(2008)Ihlemann, Jacobs, and Sofronie-Stokkermans].

Simplification orderings are a common tool to prove termination of term rewrite systems [Dershowitz(1979), Baader and Nipkow(1998)]. Among the most widely used simplification orderings are recursive path orderings [Dershowitz(1979)] (which have originally been defined in terms of multiset orderings), lexicographic path orderings [Baader and Nipkow(1998)], and Knuth-Bendix orderings [Dick et al.(1990)Dick, Kalmus, and Martin]. Constraint solving has been shown to be decidable in NP for each of these orderings [Korovin and Voronkov(2001), Narendran et al.(1998)Narendran, Rusinowitch, and Verma, Zhang et al.(2005)Zhang, Sipma, and Manna, Nieuwenhuis(1993)]. Unlike simplification orderings, we do not require that the underlying order is total. Thus, one can use our decision procedure to prove termination even in cases where there are no natural total orderings, such as Example 1 in Section 5.2.

6 Combining Theories with Shared Set Operations

Motivated by applications in software verification, we explore automated reasoning about the non-disjoint combination of theories of infinitely many finite structures, where the theories share set variables and set operations. We prove a combination theorem and apply it to show the decidability of the satisfiability problem for a class of formulas obtained by applying propositional connectives to formulas belonging to: 1) Boolean Algebra with Presburger Arithmetic (with quantifiers over sets and integers), 2) weak monadic second-order logic over trees (with monadic second-order quantifiers), 3) two-variable logic with counting quantifiers (ranging over elements), 4) the Bernays-Schönfinkel-Ramsey class of first-order logic with equality (with $\exists^*\forall^*$ quantifier prefix), and 5) the quantifier-free logic of multisets with cardinality constraints.

6.1 Motivation

Constraint solvers based on satisfiability modulo theories (SMT) [de Moura and Bjørner(2008a), Barrett and Tinelli(2007), Ge et al.(2007)Ge, Barrett, and Tinelli] are a key enabling technique in software and hardware verification systems [Ball et al.(2002)Ball, Podelski, and Rajamani, Barnett et al.(2004a)Barnett, DeLine, Fähndrich, Leino, and Schulte]. The range of problems amenable to such approaches depends on the expressive power of the logics supported by the SMT solvers. Current SMT solvers implement the combination of quantifier-free stably infinite theories with disjoint signatures, in essence following the approach pioneered by Nelson and Oppen [Nelson and Oppen(1979)]. Such solvers serve as decision procedures for quantifier-free formulas, typically containing uninterpreted function symbols, linear arithmetic, and bit vectors. The limited expressiveness of SMT prover logics translates into a limited class of properties that automated verification tools can handle.

To support a broader set of applications, this chapter considers decision procedures for the combination of possibly quantified formulas in non-disjoint theories. The idea of combining rich theories within an expressive language has been explored in interactive provers [Owre et al.(1992)Owre, Rushby, and Shankar, Boyer and Moore(1988), Basin and Friedrich(2000),

McLaughlin et al.(2006)McLaughlin, Barrett, and Ge]. Such integration efforts are very useful, but do not result in complete decision procedures for the combined logics. The study of completeness for non-disjoint combination is relatively recent [Zarba(2002b), Tinelli and Ringeissen(2003)] and provides foundations for the general problem. Under certain conditions, such as local finiteness, decidability results have been obtained even for non-disjoint theories [Ghilardi(2005)]. We consider a case of combination of non-disjoint theories sharing operations on sets of uninterpreted elements, a case that was not considered before. The theories that we consider have the property that the tuples of cardinalities of Venn regions over shared set variables in the models of a formula are a semi-linear set (i.e., expressible in Presburger arithmetic).

Reasoning about combinations of decidable logics. The idea of deciding a combination of logics is to check the satisfiability of a conjunction of formulas $A \wedge B$ by using one decision procedure, D_A, for A, and another decision procedure, D_B, for B. To obtain a complete decision procedure, D_A and D_B must communicate to ensure that a model found by D_A and a model found by D_B can be merged into a model for $A \wedge B$.

Reduction-based decision procedure. We follow a reduction approach to decision procedures. The first decision procedure, D_A, computes a projection, S_A, of A onto shared set variables, which are free in both A and B. This projection is semantically equivalent to existentially quantifying over predicates and variables that are free in A but not in B; it is the strongest consequence of A expressible only using the shared set variables. D_B similarly computes the projection S_B of B. This reduces the satisfiability of $A \wedge B$ to satisfiability of the formula $S_A \wedge S_B$, which contains only set variables.

A logic for shared constraints on sets. A key parameter of our combination approach is the logic of sets used to express the projections S_A and S_B. A suitable logic depends on the logics of formulas A and B. Inspired by verification of linked data structures, we consider as the logics for A, B the following: weak monadic second-order logic of two successors WS2S [Thatcher and Wright(1968)], two-variable logic with counting C^2 [Grädel et al.(1997)Grädel, Otto, and Rosen, Pacholski et al.(2000)Pacholski, Szwast, and Tendera, Pratt-Hartmann(2005)], the Bernays-Schönfinkel-Ramsey class of first-order logic [Börger et al.(1997)Börger, Grädel, and Gurevich], BAPA [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard], and quantifier-free logics of multisets [Piskac and Kuncak(2008c), Piskac and Kuncak(2008a)]. Remarkably, the smallest logic needed to express the projection formulas in these logics has the expressive power of Boolean Algebra with Presburger Arithmetic (BAPA), described in [Kuncak and Rinard(2007)] and in Fig. 6.4. We show that the decision procedures for these four logics can be naturally extended to a reduction to BAPA that captures precisely the constraints on set variables. The existence of these reductions, along with quantifier elimination [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard] and NP membership of the quantifier-free fragment [Kuncak and Rinard(2007)], make BAPA an appealing reduction target for expressive logics.

```
class Node {Node left,right ; Object data;}
class Tree {
    private static Node root;
    private static int size ; /*:
    private static specvar nodes :: objset ;
    vardefs "nodes==={x. (root,x) ∈ {(x,y). left x = y ∨ right x = y}*}";
    private static specvar content :: objset ;
    vardefs "content==={x. ∃ n. n ≠ null ∧ n ∈ nodes ∧ data n = x} " */

    private void insertAt (Node p, Object e) /*:
      requires "tree [left ,right ] ∧ nodes ⊆ Object.alloc ∧ size = card content ∧
                e ∉ content ∧ e ≠ null ∧ p ∈ nodes ∧ p ≠ null ∧ left p = null"
      modifies nodes,content, left , right , data, size
      ensures "size = card content" */
    {
    Node tmp = new Node();
    tmp.data = e;
    p.left = tmp;
    size = size + 1;
    }
}
```

Figure 6.1: Fragment of insertion into a tree

An earlier version of some of these results is available in [Kuncak and Wies(2009)].

6.2 Example: Verifying a Code Fragment

Our example shows a verification condition formula generated when verifying an unbounded linked data structure. The formula belongs to our new decidable class obtained by combining several decidable logics.

Specification and verification in Jahob. Fig. 6.1 shows a fragment of Java code for insertion into a binary search tree, factored out into a separate `insertAt` method. The search tree has fields (`left`, `right`) that form a tree, and field `data`, which is not necessarily an injective function (an element may be stored multiple times in the tree). The `insertAt` method is meant to be invoked when the insertion procedure has found a node `p` that has no left child. It inserts the given object `e` into a fresh node `tmp` that becomes the new left child of `p`. In addition to Java statements, the example in Fig. 6.1 contains preconditions and postconditions, written in the notation of the Jahob verification system [Kuncak(2007),Zee et al.(2008)Zee, Kuncak, and Rinard,Wies(2009)]. The *vardefs* notation introduces two sets: 1) the set of auxiliary objects *nodes*, denoting the `Node` objects stored in the binary tree, and 2) the set *content* denoting the useful content of the tree. To verify such examples in the previously reported approach [Zee et al.(2008)Zee, Kuncak, and Rinard], the user of the system had to manually provide the definitions of auxiliary sets, and to manually introduce certain lemmas describing changes to these sets. Our decidability result means that there is no need to manually introduce these lemmas.

$$
\begin{array}{l}
\mathsf{tree}\,[\mathsf{left}\ ,\mathsf{right}\,]\ \wedge\ \mathsf{left}\ \mathsf{p} = \mathsf{null}\ \wedge\ \mathsf{p} \in \mathsf{nodes}\ \wedge \\
\mathsf{nodes}{=}\{\mathsf{x}.\ (\mathsf{root},\mathsf{x})\ \in\ \{(\mathsf{x},\mathsf{y}).\ \mathsf{left}\ \ \mathsf{x} = \mathsf{y} | \mathsf{right}\ \ \mathsf{x} = \mathsf{y}\}\hat{}*\}\ \wedge \\
\mathsf{content}{=}\{\mathsf{x}.\ \exists\ \mathsf{n}.\ \mathsf{n} \neq \mathsf{null}\ \wedge\ \mathsf{n} \in \mathsf{nodes}\ \wedge\ \mathsf{data}\ \mathsf{n} = \mathsf{x}\}\ \wedge \\
\mathsf{e} \notin \mathsf{content}\ \wedge\ \mathsf{nodes} \subseteq \mathsf{alloc}\ \wedge \\
\mathsf{tmp} \notin \mathsf{alloc}\ \wedge\ \mathsf{left}\ \mathsf{tmp} = \mathsf{null}\ \wedge\ \mathsf{right}\ \mathsf{tmp} = \mathsf{null}\ \wedge \\
\mathsf{data}\ \mathsf{tmp} = \mathsf{null}\ \wedge\ (\forall\ \mathsf{y}.\ \mathsf{data}\ \mathsf{y} \neq \mathsf{tmp})\ \wedge \\
\mathsf{nodes1}{=}\{\mathsf{x}.\ (\mathsf{root},\mathsf{x})\ \in\ \{(\mathsf{x},\mathsf{y}).\ (\mathsf{left}\ \ (\mathsf{p}{:=}\mathsf{tmp}))\ \mathsf{x} = \mathsf{y})\ |\ \mathsf{right}\ \ \mathsf{x} = \mathsf{y}\}\ \wedge \\
\mathsf{content1}{=}\{\mathsf{x}.\ \exists\ \mathsf{n}.\ \mathsf{n} \neq \mathsf{null}\ \wedge\ \mathsf{n} \in \mathsf{nodes1}\ \wedge\ (\mathsf{data}(\mathsf{tmp}{:=}\mathsf{e}))\ \mathsf{n} = \mathsf{x}\}\ \rightarrow \\
\qquad \mathsf{card}\ \ \mathsf{content1}\ =\ \mathsf{card}\ \mathsf{content}\ +\ 1
\end{array}
$$

Figure 6.2: Verification condition for Fig. 6.1

$$
\begin{array}{l}
\mathsf{SHARED\ SETS}\text{: } \mathsf{nodes},\ \mathsf{nodes1},\ \mathsf{content},\ \mathsf{content1},\ \{\mathsf{e}\},\ \{\mathsf{tmp}\} \\
\mathsf{WS2S\ FRAGMENT}\text{:} \\
\ \mathsf{tree}\,[\mathsf{left}\ ,\mathsf{right}\,]\ \wedge\ \mathsf{left}\ \mathsf{p} = \mathsf{null}\ \wedge\ \mathsf{p} \in \mathsf{nodes}\ \wedge\ \mathsf{left}\ \mathsf{tmp} = \mathsf{null}\ \wedge\ \mathsf{right}\ \mathsf{tmp} = \mathsf{null}\ \wedge \\
\ \mathsf{nodes}{=}\{\mathsf{x}.\ (\mathsf{root},\mathsf{x})\ \in\ \{(\mathsf{x},\mathsf{y}).\ \mathsf{left}\ \ \mathsf{x} = \mathsf{y} | \mathsf{right}\ \ \mathsf{x} = \mathsf{y}\}\hat{}*\}\ \wedge \\
\ \mathsf{nodes1}{=}\{\mathsf{x}.\ (\mathsf{root},\mathsf{x})\ \in\ \{(\mathsf{x},\mathsf{y}).\ (\mathsf{left}\ \ (\mathsf{p}{:=}\mathsf{tmp}))\ \mathsf{x} = \mathsf{y})\ |\ \mathsf{right}\ \ \mathsf{x} = \mathsf{y}\} \\
\mathsf{CONSEQUENCE}\text{: } \mathsf{nodes1}{=}\mathsf{nodes} \cup \{\mathsf{tmp}\} \\
\mathsf{C2\ FRAGMENT}\text{:} \\
\ \ \mathsf{data}\ \mathsf{tmp} = \mathsf{null}\ \wedge\ (\forall\ \mathsf{y}.\ \mathsf{data}\ \mathsf{y} \neq \mathsf{tmp})\ \wedge\ \mathsf{tmp} \notin \mathsf{alloc}\ \wedge\ \mathsf{nodes} \subseteq \mathsf{alloc}\ \wedge \\
\ \ \mathsf{content}{=}\{\mathsf{x}.\ \exists\ \mathsf{n}.\ \mathsf{n} \neq \mathsf{null}\ \wedge\ \mathsf{n} \in \mathsf{nodes}\ \wedge\ \mathsf{data}\ \mathsf{n} = \mathsf{x}\}\ \wedge \\
\ \ \mathsf{content1}{=}\{\mathsf{x}.\ \exists\ \mathsf{n}.\ \mathsf{n} \neq \mathsf{null}\ \wedge\ \mathsf{n} \in \mathsf{nodes1}\ \wedge\ (\mathsf{data}(\mathsf{tmp}{:=}\mathsf{e}))\ \mathsf{n} = \mathsf{x}\} \\
\mathsf{CONSEQUENCE}\text{: } \mathsf{nodes1} \neq \mathsf{nodes} \cup \{\mathsf{tmp}\}\ \vee\ \mathsf{content1} = \mathsf{content} \cup \{\mathsf{e}\} \\
\mathsf{BAPA\ FRAGMENT}\text{: } \mathsf{e} \notin \mathsf{content}\ \wedge\ \mathsf{card}\ \mathsf{content1} \neq \mathsf{card}\ \mathsf{content}\ +\ 1 \\
\mathsf{CONSEQUENCE}\text{: } \mathsf{e} \notin \mathsf{content}\ \wedge\ \mathsf{card}\ \mathsf{content1} \neq \mathsf{card}\ \mathsf{content}\ +\ 1
\end{array}
$$

Figure 6.3: Negation of Fig. 6.2, and consequences on shared sets

Decidability of the verification condition. Fig. 6.2 shows the verification condition formula for a method (`insertAt`) that inserts a node into a linked list. The validity of this formula implies that invoking a method in a state satisfying the precondition results in a state that satisfies the postcondition of `insertAt`. The formula contains the transitive closure operator, quantifiers, set comprehensions, and the cardinality operator. Nevertheless, there is a (syntactically defined) decidable class of formulas that contains the verification condition in Fig. 6.2. This decidable class is a set-sharing combination of three decidable logics, and can be decided using the method we present in this chapter.

To understand the method for proving the formula in Fig. 6.2, consider the problem of showing the unsatisfiability of the negation of the formula. Fig. 6.3 shows the conjuncts of the negation, grouped according to three decidable logics to which the conjuncts belong: 1) weak monadic second-order logic of two successors WS2S [Thatcher and Wright(1968)], 2) two-variable logic with counting C^2 [Pratt-Hartmann(2005)], and 3) Boolean Algebra with Presburger Arithmetic (BAPA) [Feferman and Vaught(1959), Kuncak et al.(2006)Kuncak, Nguyen, and Rinard, Kuncak and Rinard(2007)]. For the formula in each of the fragments, Fig. 6.3 also shows a consequence formula that contains only shared sets and statements about their cardinalities. (We represent elements as singleton sets, so we admit formulas sharing elements as well.)

A decision procedure. Note that the conjunction of the consequences of three formula fragments is an unsatisfiable formula. This shows that the original verification condition is valid. In general, our decidability result shows that the decision procedures of logics such as WS2S and C^2 can be naturally extended to compute strongest consequences of formulas involving given shared sets. These consequences are all expressed in BAPA, which is decidable. In summary, the following is a decision procedure for satisfiability of combined formulas: 1) split the formula into fragments (belonging to WS2S, C^2, or BAPA); 2) for each fragment compute its strongest BAPA consequence; 3) check the satisfiability of the conjunction of consequences.

Higer-order logic. We present our problem in a fragment of classical higher-order logic [Andrews(2002), Chapter 5] with a particular set of types, which we call sorts. We assume that formulas are well-formed according to sorts of variables and logical symbols. Each variable and each logical symbol have an associated sort. The primitive sorts we consider are 1) bool, interpreted as the two-element set {true, false} of booleans; 2) int, interpreted as the set of integers $\mathbb{Z}$; and 3) obj, interpreted as a non-empty set of elements. The only sort constructor is the binary function space constructor '$\rightarrow$'. We represent a function mapping elements of sorts $s_1, \ldots, s_n$ into an element of sort s_0 as a term of sort $s_1 \times \ldots \times s_n \rightarrow s_0$ where $s_1 \times s_2 \times \ldots \times s_n \rightarrow s_0$ is a shorthand for $s_1 \rightarrow (s_2 \rightarrow \ldots (s_n \rightarrow s_0))$. When $s_1, \ldots, s_n$ are all the same sort s, we abbreviate $s_1 \times \ldots \times s_n \rightarrow s_0$ as $s^n \rightarrow s_0$. We represent a relation between elements of sorts $s_1, \ldots, s_n$ as a function $s_1 \times \ldots \times s_n \rightarrow$ bool. We use set as an abbreviation for the sort obj $\rightarrow$ bool. We call variables of sort set set variables. The equality symbol applies only to terms of the same sort. We assume to have a distinct equality symbol for each sort of interest, but we use the same notation to denote all of them. Propositional operations connect terms of sort bool. We write $\forall x{:}s.F$ to denote a universally quantified formula where the quantified variable has sort s (analogously for $\exists x{:}s.F$ and $\exists x{:}s^K.F$ for counting quantifiers of Section 6.4.3). We denote by $\mathsf{FV}(F)$ the set of all free variables that occur free in F. We write $\mathsf{FV}_s(F)$ for the free variables of sort s. Note that the variables can be higher-order (we will see, however, that the shared variables are of sort set). A theory is simply a set of formulas, possibly with free variables.

Structures. A structure α specifies a finite set, which is also the meaning of obj, and we denote it $\alpha(\mathsf{obj})$. We focus on the case of finite $\alpha(\mathsf{obj})$ primarily for simplicity; we believe the extension to the case where domains are either finite or countable is possible and can be done using results from [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard, Section 8.1], [Pratt-Hartmann(2005), Section 5], [Thatcher and Wright(1968)]. When α is understood we use $[\![X]\!]$ to denote $\alpha(X)$, where X denotes a sort, a term, a formula, or a set of formulas. If S is a set of formulas then $\alpha(S) = \mathsf{true}$ means $\alpha(F) = \mathsf{true}$ for each $F \in X$. In every structure we let $[\![\mathsf{bool}]\!] = \{\mathsf{false}, \mathsf{true}\}$. Instead of $\alpha(F) = \mathsf{true}$ we often write simply $\alpha(F)$. We interpret terms of the sort $s_1 \times \ldots \times s_n \rightarrow s_0$ as total functions $[\![s_1]\!] \times \ldots \times [\![s_n]\!] \rightarrow [\![s_0]\!]$. For a set A, we identify a function $f : A \rightarrow \{\mathsf{false}, \mathsf{true}\}$ with the subset $\{x \in A \mid f(x) = \mathsf{true}\}$. We thus interpret variables of the sort $\mathsf{obj}^n \rightarrow \mathsf{bool}$ as subsets of $[\![\mathsf{obj}]\!]^n$. If s is a sort then $\alpha(s)$ depends only on $\alpha(\mathsf{obj})$ and we denote it also by $[\![s]\!]$. We interpret propositional operations $\wedge, \vee, \neg$ as usual in classical logic.

$$
\begin{array}{rcl}
F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \forall x{:}s.F \mid \exists x{:}s.F \\
s & ::= & \mathsf{int} \mid \mathsf{obj} \mid \mathsf{set} \\
A & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid T_1 = T_2 \mid T_1 < T_2 \mid K \,\mathsf{dvd}\, T \\
B & ::= & x \mid \emptyset \mid \mathsf{Univ} \mid \{x\} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c \\
T & ::= & x \mid K \mid \mathsf{CardUniv} \mid T_1 + T_2 \mid K \cdot T \mid \mathsf{card}\, B \\
K & ::= & \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
\end{array}
$$

Figure 6.4: Boolean Algebra with Presburger Arithmetic (BAPA)

A quantified variable of sort s ranges over all elements of $[\![s]\!]$. (Thus, as in standard model of HOL [Andrews(2002), Section 54], quantification over variables of sort $s_1 \to s_2$ is quantification over all total functions $[\![s_1]\!] \to [\![s_2]\!]$.)

6.2.1 Boolean Algebra with Presburger Arithmetic

It will be convenient to enrich the language of our formulas with operations on integers, sets, and cardinality operations. These operations could be given by a theory or defined in HOL, but we choose to simply treat them as built-in logical symbols, whose meaning must be respected by all structures α we consider. Fig. 6.4 shows the syntax of Boolean Algebra with Presburger Arithmetic (BAPA) [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard, Feferman and Vaught(1959)]. The following are the sorts of symbols appearing in BAPA formulas: $\subseteq : \mathsf{set}^2 \to \mathsf{bool}$, $< : \mathsf{int}^2 \to \mathsf{bool}$, $\mathsf{dvd}_K : \mathsf{int} \to \mathsf{bool}$ for each integer constant K (with $\mathsf{dvd}_K(t)$ denoted by $\mathsf{dvd}\, K\, t$), $\emptyset, \mathsf{Univ} : \mathsf{set}$, $\mathsf{singleton} : \mathsf{obj} \to \mathsf{set}$ (with $\mathsf{singleton}(x)$ denoted as $\{x\}$), $\cap, \cup : \mathsf{set}^2 \to \mathsf{set}$, $\mathsf{complement} : \mathsf{set} \to \mathsf{set}$ (with $\mathsf{complement}(A)$ denoted by A^c), $K : \mathsf{int}$ for each integer constant K, $\mathsf{CardUniv} : \mathsf{int}$, $+ : \mathsf{int}^2 \to \mathsf{int}$, $\mathsf{mul}_K : \mathsf{int} \to \mathsf{int}$ for each integer constant K (with $\mathsf{mul}_K(t)$ denoted by $K \cdot t$), and $\mathsf{card} : \mathsf{set} \to \mathsf{int}$.

We sketch the meaning of the less common among the symbols in Fig. 6.4. Univ denotes the universal set, that is, $[\![\mathsf{Univ}]\!] = [\![\mathsf{obj}]\!]$. $\mathsf{card}\, A$ denotes the cardinality of the set A. CardUniv is interpreted as $\mathsf{card}\,\mathsf{Univ}$. The formula $\mathsf{dvd}\, K\, t$ denotes that the integer constant K divides the integer t. We note that the condition $x \in A$ can be written in this language as $\{x\} \subseteq A$. Note that BAPA properly extends the first-order theory of Boolean Algebras over finite structures, which in turn subsumes the first-order logic with unary predicates and no function symbols, because e.g. $\exists x{:}\mathsf{obj}.F(x)$ can be written as $\exists X{:}\mathsf{set}.\mathsf{card}\, X{=}1 \wedge F'(X)$ where in F' e.g. $P(x)$ is replaced by $X \subseteq P$.

BAPA-definable relations between sets. We recall the definitions from Chapter 2. A semilinear set is a finite union of linear sets. A linear set is a set of the form $\{\vec{a} + k_1\vec{b}_1 + \ldots + k_n\vec{b}_n \mid k_1, \ldots, k_n \in \mathbb{N}\}$ where $\vec{a}, \vec{b}_1, \ldots, \vec{b}_n \in \mathbb{N}^M$. We represent a linear set by its generating vectors

$\vec{a}, \vec{b}_1, \ldots, \vec{b}_n$, and a semilinear set by the finite set of representations of its linear sets. It was shown in [Ginsburg and Spanier(1966)] that a set of integer vectors $S \subseteq \mathbb{N}^M$ is a set of non-negative solutions of a Presburger arithmetic formula P i.e. $S = \{(v_1, \ldots, v_n).P\}$ iff S is a semilinear set. We then have the following characterization of relationships between sets expressible in BAPA, which follows from [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard].

Lemma 6.1 (BAPA-expressible means Venn-cardinality-semilinear) *Given a finite set U and a relation $\rho \subseteq (2^U)^p$ the following are equivalent:*

1. *there exists a BAPA formula F whose free variables are $A_1, \ldots, A_p$, and have the sort* set, *such that $\rho = \{(s_1, \ldots, s_p) \mid \{A_1 \mapsto s_1, \ldots, A_p \mapsto s_p\}(F)\}$;*
2. *the following subset of $\mathbb{Z}^M$ for $M = 2^p$ is semilinear:*
   $\{(|s_1^c \cap s_2^c \cap \ldots \cap s_p^c|, |s_1 \cap s_2^c \cap \ldots \cap s_p^c|, \ldots, |s_1 \cap s_2 \cap \ldots \cap s_p|) \mid (s_1, \ldots, s_p) \in \rho\}$.

Structures of interest in this chapter. In the rest of this chapter we consider structures that interpret the BAPA symbols as defined above. Because the meaning of BAPA-specific symbols is fixed, a structure α that interprets a set of formulas is determined by a finite set $\alpha(\mathsf{obj})$ as well as the values $\alpha(x)$ for each variable x free in the set of formulas. Let $\{\mathsf{obj} \mapsto u, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ denote the structure α with domain u that interprets each variable x_i as v_i.

6.3 Combination by Reduction to BAPA

The Satisfiability Problem. We are interested in an algorithm to determine whether there exists a structure $\alpha \in \mathscr{M}$ in which the following formula is true

$$B(F_1, \ldots, F_n) \tag{6.1}$$

where

1. $F_1, \ldots, F_n$ are formulas with $\mathsf{FV}(F_i) \subseteq \{A_1, \ldots, A_p, x_1, \ldots, x_q\}$.
2. $V_S = \{A_1, \ldots, A_p\}$ are variables of sort set, whereas $x_1, \ldots, x_q$ are the remaining variables.[1]
3. Each formula F_i belongs to a given class of formulas, $\mathscr{F}_i$. For each $\mathscr{F}_i$, we assume that there is a corresponding theory $\mathscr{T}_i \subseteq \mathscr{F}_i$.
4. $B(F_1, \ldots, F_n)$ denotes a formula built from $F_1, \ldots, F_n$ using the propositional operations $\wedge, \vee$. The absence of negation is usually not a loss of generality because most $\mathscr{F}_i$ are closed under negation so B is the negation-normal form of a quantifier-free combination.

[1] For notational simplicity we do not consider variables of sort obj because they can be represented as singleton sets, of sort set.

5. As the set of structures $\mathscr{M}$, we consider all structures α of interest (with finite $[\![\mathsf{obj}]\!]$, interpreting BAPA symbols in the standard way) for which $\alpha(\cup_{i=1}^{n}\mathscr{T}_i)$.

6. (Set Sharing Condition) If $i \neq j$, then $\mathsf{FV}(\{F_i\} \cup \mathscr{T}_i) \cap \mathsf{FV}(\{F_j\} \cup \mathscr{T}_j) \subseteq V_S$.

Note that, as a special case, if we embed a class of first-order formulas into our framework, we obtain a framework that supports sharing unary predicates, but not e.g. binary predicates.

Combination Theorem. The formula B in (6.1) is satisfiable iff one of the disjuncts in its disjunctive normal form is satisfiable. Consider a disjunct $F_1 \wedge \ldots \wedge F_m$ for $m \leq n$. By definition of the satisfiability problem (6.1), $F_1 \wedge \ldots \wedge F_m$ is satisfiable iff there exists a structure α such that for each $1 \leq i \leq m$, for each $G \in \{F_i\} \cup \mathscr{T}_i$, we have $\alpha(G) = \mathsf{true}$. Let each variable x_i have some sort s_i (such as $\mathsf{obj}^2 \to \mathsf{bool}$). Then the satisfiability of $F_1 \wedge \ldots \wedge F_m$ is equivalent to the following condition:

$$\begin{array}{l} \exists \text{ finite set } u.\ \exists a_1,\ldots,a_p \subseteq u.\ \exists v_1 \in [\![s_1]\!]^u.\ldots.\ \exists v_q \in [\![s_q]\!]^u.\ \bigwedge_{i=1}^{m} \\ \{\mathsf{obj} \to u, A_1 \mapsto a_1,\ldots,A_p \mapsto a_p, x_1 \mapsto v_1,\ldots,x_q \mapsto v_q\}(\{F_i\} \cup \mathscr{T}_i) \end{array} \tag{6.2}$$

By the set sharing condition, each of the variables $x_1,\ldots,x_q$ appears only in one conjunct and can be moved inwards from the top level to this conjunct. Using x_{ij} to denote the j-th variable in the i-th conjunct we obtain the condition

$$\exists \text{ finite set } u.\ \exists a_1,\ldots,a_p \subseteq u.\ \bigwedge_{i=1}^{m} C_i(u,a_1,\ldots,a_p) \tag{6.3}$$

where $C_i(u,a_1,\ldots,a_p)$ is

$$\begin{array}{l} \exists v_{i1}.\ldots\exists v_{iw_i}. \\ \{\mathsf{obj} \to u, A_1 \mapsto a_1,\ldots,A_p \mapsto a_p, x_{i1} \mapsto v_{i1},\ldots,x_{iw_i} \mapsto v_{iw_i}\}(\{F_i\} \cup \mathscr{T}_i) \end{array}$$

The idea of our combination method is to simplify each condition $C_i(u,a_1,\ldots,a_p)$ into the truth value of a BAPA formula. If this is possible, we say that there exists a BAPA reduction.

Definition 6.2 (BAPA Reduction) *If $\mathscr{F}_i$ is a set of formulas and $\mathscr{T}_i \subseteq \mathscr{F}_i$ a theory, we call a function $\rho : \mathscr{F}_i \to \mathscr{F}_{\mathsf{BAPA}}$ a BAPA reduction for $(\mathscr{F}_i, \mathscr{T}_i)$ iff for every formula $F_i \in \mathscr{F}_i$ and for all finite u and $a_1,\ldots,a_p \subseteq u$, the condition*

$$\begin{array}{l} \exists v_{i1}.\ldots\exists v_{iw_i}. \\ \{\mathsf{obj} \to u, A_1 \mapsto a_1,\ldots,A_p \mapsto a_p, x_{i1} \mapsto v_{i1},\ldots,x_{iw_i} \mapsto v_{iw_i}\}(\{F_i\} \cup \mathscr{T}_i) \end{array}$$

is equivalent to the condition $\{\mathsf{obj} \to u, A_1 \mapsto a_1,\ldots,A_p \mapsto a_p\}(\rho(F_i))$.

A computable BAPA reduction is a BAPA reduction which is computable as a function on formula syntax trees.

$$
\begin{array}{rcl}
F & ::= & P \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \forall x{:}s.F \mid \exists x{:}s.F \\
s & ::= & \mathsf{obj} \mid \mathsf{set} \\
P & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid r(x,y) \\
r & ::= & \mathsf{succ}_L \mid \mathsf{succ}_R \\
B & ::= & x \mid \epsilon \mid \emptyset \mid \mathsf{Univ} \mid \{x\} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c
\end{array}
$$

Figure 6.5: Monadic Second-Order Logic of Finite Trees (FT)

Theorem 6.3 *Suppose that for every* $1 \le i \le n$ *for* $(\mathscr{F}_i, \mathscr{T}_i)$ *there exists a computable BAPA reduction* ρ_i*. Then the problem* (6.1) *in Section 6.3 is decidable.*

Specifically, to check satisfiability of $B(F_1,\ldots,F_n)$, compute $B(\rho_1(F_1),\ldots,\rho_n(F_n))$ and then check its satisfiability using a BAPA decision procedure [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard, Kuncak and Rinard(2007)].

6.4 BAPA Reductions

6.4.1 Monadic Second-Order Logic of Finite Trees

Figure 6.5 shows the syntax of (our presentation of) monadic second-order logic of finite trees (FT), a variant of weak monadic second-order logic of two successors (WS2S) [Thatcher and Wright(1968), Klarlund and Møller(2001)]. The following are the sorts of variables specific to FT formulas: $\mathsf{succ}_L, \mathsf{succ}_R : \mathsf{obj}^2 \to \mathsf{bool}$.

We interpret the sort obj over finite, prefix-closed sets of binary strings. More precisely, we use $\{1,2\}$ as the binary alphabet, and we let $[\![\mathsf{obj}]\!] \subset \{1,2\}^*$ such that

$$\forall w \in \{1,2\}^*.\ (w1 \in [\![\mathsf{obj}]\!] \vee w2 \in [\![\mathsf{obj}]\!]) \to w \in [\![\mathsf{obj}]\!]$$

Note that the $*$ operator here refers to the Kleene star. In each model, $[\![\mathsf{set}]\!]$ is the set of all subsets of $[\![\mathsf{obj}]\!]$. We let $[\![\epsilon]\!]$ be the empty string which we also denote by ϵ. We define

$$[\![\mathsf{succ}_L]\!] = \{(w, w1) \mid w1 \in [\![\mathsf{obj}]\!]\} \quad \text{and} \quad [\![\mathsf{succ}_R]\!] = \{(w, w2) \mid w2 \in [\![\mathsf{obj}]\!]\}$$

The remaining constants and operations on sets are interpreted as in BAPA.

Let $\mathscr{F}_{\mathsf{FT}}$ be the set of all formulas in Figure 6.5. Let $\mathscr{M}_{\mathsf{FT}}$ be the set of all (finite) structures described above. We define $\mathscr{T}_{\mathsf{FT}}$ as the set of all formulas $F \in \mathscr{F}_{\mathsf{FT}}$ such that F is true in all structures from $\mathscr{M}_{\mathsf{FT}}$.

The models of the theory $\mathscr{T}_{\mathsf{FT}}$ correspond up to isomorphism with the interpretations in $\mathscr{M}_{\mathsf{FT}}$.

Lemma 6.4 *If α is a structure such that $\alpha(\mathscr{T}_{\mathsf{FT}})$, then α is isomorphic to some structure in $\mathscr{M}_{\mathsf{FT}}$.*

Note that any FT formula $F(x)$ with a free variable x of sort obj can be transformed into the equisatisfiable formula $\exists x : \mathsf{obj}.\, y = \{x\} \wedge F(x)$ where y is a fresh variable of sort set. For conciseness of presentation, in the rest of this section we only consider FT formulas F with $\mathsf{FV}_{\mathsf{obj}}(F) = \emptyset$.

Finite tree automata. In the following, we recall the connection between FT formulas and finite tree automata. Let Σ be a finite ranked alphabet. We call symbols of rank 0 constant symbols and a symbol of rank $k > 0$ a k-ary function symbol. We denote by $\mathsf{Terms}(\Sigma)$ the set of all terms over Σ. We associate a position $p \in \{1, \ldots, r_{\max}\}^*$ with each subterm in a term t where $r_{\max}$ is the maximal rank of all symbols in Σ. We denote by $t[p]$ the topmost symbol of the subterm at position p. For instance, consider the term $t = f(g(a,b,c),a)$ then we have $t[\epsilon] = f$ and $t[13] = c$.

A finite (deterministic bottom-up) tree automaton A for alphabet Σ is a tuple (Q, Q_f, ι) where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, and ι is a function that associates with each constant symbol $c \in \Sigma$ a state $\iota(c) \in Q$ and with each k-ary function symbol $f \in \Sigma$ a function $\iota(f) : Q^k \to Q$. We homomorphically extend ι from symbols in Σ to Σ-terms. We say that A accepts a term $t \in \mathsf{Terms}(\Sigma)$ if $\iota(t) \in Q_f$. The language $\mathscr{L}(A)$ accepted by A is the set of all Σ-terms accepted by A.

Let F be an FT formula and let $\mathsf{SV}(F)$ be the set $\mathsf{SV}(F) = \mathsf{FV}(F) \cup \{\mathsf{Univ}\}$. We denote by Σ_F the alphabet consisting of the constant symbol $\bot$ and all binary function symbols f_ν where ν is a function $\nu : \mathsf{SV}(F) \to \{0,1\}$. We inductively associate a Σ_F-term $t_{\alpha,w}$ with every structure $\alpha \in \mathscr{M}_{\mathsf{FT}}$ and string $w \in \{1,2\}^*$ as follows:

$$t_{\alpha,w} = \begin{cases} f_{\nu_{\alpha,w}}(t_{\alpha,w1}, t_{\alpha,w2}) & \text{if } w \in \alpha(\mathsf{obj}) \\ \bot & \text{otherwise} \end{cases}$$

such that for all $x \in \mathsf{SV}(F)$, $\nu_{\alpha,w}(x) = 1$ iff $w \in \alpha(x)$. The language $\mathscr{L}(F) \subseteq \mathsf{Terms}(\Sigma_F)$ of F is then defined by $\mathscr{L}(F) = \{\, t_{\alpha,\epsilon} \mid \alpha \in \mathscr{M}_{\mathsf{FT}} \wedge \alpha(F) \,\}$.

The following theorem states the connection between the structures satisfying FT formulas and the languages accepted by finite tree automata[2].

Theorem 6.5 (Thatcher and Wright [Thatcher and Wright(1968)]) *For every FT formula F there exists a finite tree automaton A_F over alphabet Σ_F such that $\mathscr{L}(F) = \mathscr{L}(A_F)$ and A_F can be effectively constructed from F.*

[2] The theorem was originally stated for WS2S where the universe of all structures is fixed to the infinite binary tree $\{1,2\}^*$ and where all set variables range over finite subsets of $\{1,2\}^*$. It carries over to finite trees in a straightforward manner.

Parikh image. We recall Parikh's commutative image [Parikh(1966)]. The Parikh image for an alphabet Σ is the function $\mathsf{Parikh} : \Sigma^* \to \Sigma \to \mathbb{N}$ such that for any word $w \in \Sigma^*$ and symbol $\sigma \in \Sigma$, $\mathsf{Parikh}(w)(\sigma)$ is the number of occurrences of σ in w. The Parikh image is extended pointwise from words to sets of words: $\mathsf{Parikh}(W) = \{\mathsf{Parikh}(w) \mid w \in W\}$. In the following, we implicitly identify $\mathsf{Parikh}(W)$ with the set of integer vectors $\{(\chi(\sigma_1), \ldots, \chi(\sigma_n)) \mid \chi \in \mathsf{Parikh}(W)\}$ where we assume some fixed order on the symbols $\sigma_1, \ldots, \sigma_n$ in Σ.

Theorem 6.6 (Parikh [Parikh(1966)]) *Let G be a context-free grammar and $\mathscr{L}(G)$ the language generated from G then the Parikh image of $\mathscr{L}(G)$ is a semilinear set and its finite representation is effectively computable from G.*

We generalize the Parikh image from words to terms as expected: the Parikh image for a ranked alphabet Σ is the function $\mathsf{Parikh} : \mathsf{Terms}(\Sigma) \to \Sigma \to \mathbb{N}$ such that for all $t \in \mathsf{Terms}(\Sigma)$ and $\sigma \in \Sigma$, $\mathsf{Parikh}(t)(\sigma)$ is the number of positions p in t such that $t[p] = \sigma$. Again, we extend this function pointwise from terms to sets of terms.

Lemma 6.7 *Let A be a finite tree automaton over alphabet Σ. Then the Parikh image of $\mathscr{L}(A)$ is a semilinear set and its finite representation is effectively computable from A.*

6.4.2 BAPA Reduction for Monadic Second-Order Logic of Finite Trees

In the following, we prove the existence of a computable BAPA reduction for the theory of monadic second-order logic of finite trees.

Let F be an FT formula and let Σ_F^2 be the set of all binary function symbols in Σ_F, i.e., $\Sigma_F^2 \stackrel{def}{=} \Sigma_F \setminus \{\bot\}$. We associate with each $\sigma_\nu \in \Sigma_F^2$ the Venn region $\mathsf{vr}(\sigma_\nu)$, which is given by a set-algebraic expression over $\mathsf{SV}(F)$: let $\mathsf{SV}(F) = \{x_1, \ldots, x_n\}$ then

$$\mathsf{vr}(\sigma_\nu) \stackrel{def}{=} x_1^{\nu(x_1)} \cap \cdots \cap x_n^{\nu(x_n)} \quad .$$

Hereby x_i^0 denotes x_i^c and x_i^1 denotes x_i. Let $\alpha \in \mathscr{M}_{\mathsf{FT}}$ be a model of F. Then the term $t_{\alpha,\epsilon}$ encodes for each $w \in \alpha(\mathsf{obj})$ the Venn region to which w belongs in α, namely $\mathsf{vr}(t_{\alpha,\epsilon}[w])$. Thus, the Parikh image $\mathsf{Parikh}(t_{\alpha,\epsilon})$ encodes the cardinality of each Venn region over $\mathsf{SV}(F)$ in α.

Lemma 6.8 *Let F be an FT formula then*

$$\mathsf{Parikh}(\mathscr{L}(F))|_{\Sigma_F^2} = \{S(\alpha, \Sigma_F^2) \mid \alpha \in \mathscr{M}_{\mathsf{FT}} \wedge \alpha(F)\},$$

where $S(\alpha, \Sigma_F^2) = \{\sigma \mapsto |\alpha(\mathsf{vr}(\sigma))| \mid \sigma \in \Sigma_F^2\}$.

$$
\begin{array}{lcl}
F & ::= & P \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists^K x{:}\mathsf{obj}.F \\
P & ::= & x_1 = x_2 \mid \{x\} \subseteq A \mid r(x_1, x_2)
\end{array}
$$

Figure 6.6: Two-Variable Logic with Counting (C^2)

According to Theorem 6.5 we can construct a finite tree automaton A_F over Σ_F such that $\mathscr{L}(F) = \mathscr{L}(A_F)$. From Lemma 6.7 follows that $\mathsf{Parikh}(\mathscr{L}(F))$ is a semilinear set whose finite representation in terms of base and step vectors is effectively computable from A_F. From this finite representation, we can construct a Presburger arithmetic formula ϕ_F over free integer variables $\{x_\sigma \mid \sigma \in \Sigma_F\}$ whose set of solutions is the Parikh image of $\mathscr{L}(F)$, i.e.

$$
\mathsf{Parikh}(\mathscr{L}(F)) = \{\{\sigma \mapsto k_\sigma \mid \sigma \in \Sigma_F\} \mid \{x_\sigma \mapsto k_\sigma \mid \sigma \in \Sigma\}(\phi_F)\} \tag{6.4}
$$

Using the above construction of the Presburger arithmetic formula ϕ_F for a given FT formula F, we define the function $\rho_{\mathsf{FT}} : \mathscr{F}_{\mathsf{FT}} \to \mathscr{F}_{\mathsf{BAPA}}$ as follows:

$$
\rho_{\mathsf{FT}}(F) \stackrel{def}{=} \exists \vec{x}_\sigma . \phi_F \wedge \bigwedge_{\sigma \in \Sigma_F^2} \mathsf{card}\,\mathsf{vr}(\sigma) = x_\sigma
$$

where $\vec{x}_\sigma$ are the free integer variables of ϕ_F.

Theorem 6.9 *The function ρ_{FT} is a BAPA reduction for $(\mathscr{F}_{\mathsf{FT}}, \mathscr{T}_{\mathsf{FT}})$.*

6.4.3 Two-Variable Logic with Counting

Figure 6.6 shows the syntax of (our presentation of) two-variable logic with counting (denoted C^2) [Pratt-Hartmann(2004)]. As usual in C^2, we require that every sub-formula of a formula has at most two free variables. In the atomic formula $r(x_1, x_2)$, variables x_1, x_2 are of sort obj and r is a relation variable of sort $\mathsf{obj}^2 \to \mathsf{bool}$. The formula $\{x\} \subseteq A$ replaces $A(x)$ in predicate-logic notation, and has the expected meaning, with the variable x is of sort obj and A of sort set. The interpretation of the counting quantifier $\exists^K x{:}\mathsf{obj}.F$ for a positive constant K is that there exist at least K distinct elements x for which the formula F holds.

Let $\mathscr{F}_{C2}$ be the set of all formulas in Figure 6.6. Let $\mathscr{M}_{C2}$ be the set of structures that interpret formulas in $\mathscr{F}_{C2}$. We define $\mathscr{T}_{C2}$ as the set of all formulas $F \in \mathscr{F}_{C2}$ such that F is true in all structures from $\mathscr{M}_{C2}$. Modulo our minor variation in syntax and terminology (using relation and set variables instead of predicate symbols), $\mathscr{T}_{C2}$ corresponds to the standard set of valid C^2 formulas over finite structures [Pratt-Hartmann(2004)].

6.4.4 BAPA Reduction for Two-Variable Logic with Counting

We next build on the results in [Pratt-Hartmann(2005)] to define a BAPA reduction for C^2. We fix set variables $A_1,\ldots,A_p$ and relation variables $r_1,\ldots,r_q$. Throughout this section, let $\Sigma_A = \{A_1,\ldots,A_p\}$, $\Sigma_R = \{r_1,\ldots,r_q\}$, and $\Sigma_0 = \Sigma_A \cup \Sigma_R$. We call $\Sigma_A, \Sigma_R, \Sigma_0$ signatures because they correspond to the notion of signature in the traditional first-order logic formulation of C^2.

Model theoretic types. Define the model-theoretic notion of n-type $\pi_\Sigma(x_1,\ldots,x_n)$ in the signature Σ as the maximal consistent set of non-equality literals in Σ whose obj-sort variables are included in $\{x_1,\ldots,x_n\}$. [3] Given a structure α such that $\alpha(x_1),\ldots,\alpha(x_n)$ are all distinct, α induces an n-type

$$\mathsf{ityp}^{\alpha,\Sigma}(x_1,\ldots,x_n) = \{L \mid \alpha(L) \wedge \mathsf{FV}(L) \subseteq \{x_1,\ldots,x_n\},\ L \text{ is } \Sigma\text{-literal without '='}\}$$

We also define the set of n-tuples for which a type π holds in a structure α:

$$S^\alpha(\pi(x_1,\ldots,x_n)) = \{(e_1,\ldots,e_n) \in \alpha(\mathsf{obj})^n \mid \alpha(x_1 := e_1,\ldots,x_n := e_n)(\pi)\}$$

If $\Sigma \subseteq \Sigma'$ and π' is an n-type in signature Σ', by $\pi'|_\Sigma$ we denote the subset of π containing precisely those literals from π whose sets and relations belong to Σ. The family of sets $\{S^\alpha(\pi') \mid \pi'|_\Sigma = \pi\}$ is a partition of $S^\alpha(\pi')$. We will be particularly interested in 1-types. We identify a 1-type $\pi(x)$ in the signature Σ_A with the corresponding Venn region

$$\bigcap\{A_i \mid (\{x\} \subseteq A_i) \in \pi(x)\} \cap \bigcap\{A_i^c \mid (\neg(\{x\} \subseteq A_i)) \in \pi(x)\}.$$

If $\pi_1,\ldots,\pi_m$ is the sequence of all 1-types in the signature Σ and α is a structure, let $I^\alpha(\Sigma) = (|S^\alpha(\pi_1)|,\ldots,|S^\alpha(\pi_m)|)$. If $\mathscr{M}$ is a set of structures let $I^{\mathscr{M}}(\Sigma) = \{I^\alpha(\Sigma) \mid \alpha \in \mathscr{M}\}$.

Observation 6.10 *If π is a 1-type in Σ and π' a 1-type in Σ' for $\Sigma \subseteq \Sigma'$, then*

$$|I^\alpha(\pi)| = \sum_{\pi'|_\Sigma = \pi} |I^\alpha(\pi')|$$

Making structures differentiated, chromatic, sparse preserves 1-types. Let ϕ be a C^2 formula with signature Σ_0 of relation symbols. By Scott normal form transformation [Pratt-Hartmann(2005), Lemma 1], it is possible to introduce fresh set variables and compute another C^2 formula ϕ^* in an extended signature $\Sigma^* \supseteq \Sigma_0$, and compute a constant C_ϕ such that, for all sets u with $|u| \geq C_\phi$: 1) if α_0 is a Σ_0 interpretation with domain u such that $\alpha_0(\phi)$, then there exists its Σ^* extension $\alpha^* \supseteq \alpha_0$ such that $\alpha^*(\phi^*)$, and 2) if α^* is a Σ^* interpretation with domain u such that $\alpha^*(\phi^*)$, then for its restriction $\alpha_0 = \alpha^*|_\Sigma$ we have $\alpha_0(\phi)$. By introducing

[3] For example, if Σ has one relation variable r, and two set variables A_1, A_2, then each 2-type with free variables x, y contains, for each of the atomic formulas with variables x, y (i.e. $\{x\} \subseteq A_1$, $\{y\} \subseteq A_1$, $\{x\} \subseteq A_2$, $\{y\} \subseteq A_2$, $r(x,x)$, $r(y,y)$, $r(x,y)$, $r(y,x)$), either the formula or its negation.

further fresh set- and relation- symbols, [Pratt-Hartmann(2005), lemmas 2 and 3] shows that we can extend the signature from Σ^* to Σ such that each model α^* in Σ^* extends to a model α in Σ, where α satisfies some further conditions of interest: α is chromatic and differentiated. [Pratt-Hartmann(2005), Lemma 10] then shows that it is possible to transform a model of a formula into a so-called X-sparse model for an appropriately computed integer constant X. What is important for us is the following.

Observation 6.11 *The transformations that start from α_0 with $\alpha_0(\phi)$, and that produce a chromatic, differentiated, X-sparse structure α with $\alpha(\phi)$, have the property that, for structures of size C_ϕ or more,*

1. *the domain remains the same: $\alpha_0(\mathsf{obj}) = \alpha(\mathsf{obj})$,*
2. *the induced 1-types in the signature Σ_0 remain the same: for each 1-type π in signature Σ_0, $S^{\alpha_0}(\pi) = S^{\alpha}(\pi)$.*

Star types. [Pratt-Hartmann(2005), Definition 9] introduces a star-type $(\pi, \vec{v})$ (denoted by letter σ) as a description of a local neighborhood of a domain element, containing its induced 1-type π as well as an integer vector $\vec{v} \subseteq \mathbb{Z}^N$ that counts 2-types in which the element participates, where N is a function of the signature Σ. A star type thus gives a more precise description of the properties of a domain element than a 1-type. Without repeating the definition of star type [Pratt-Hartmann(2005), Definition 9], we note that we can similarly define the set $S^\alpha((\pi, \vec{v}))$ of elements that realize a given star type $(\pi, \vec{v})$. Moreover, for a given 1-type π, the family of the non-empty among the sets $S^\alpha((\pi, \vec{v}))$ partitions the set $S^\alpha(\pi)$.

Frames. The notion of Y-bounded chromatic frame [Pratt-Hartmann(2005), Definition 11] can be thought of as a representation of a disjunct in a normal form for the formula ϕ^*. It summarizes the properties of elements in the structure and specifies (among others), the list of possible star types $\sigma_1, \ldots, \sigma_N$ whose integer vectors $\vec{v}$ are bounded by Y. For a given ϕ^*, it is possible to effectively compute the set of C_ϕ-bounded frames $\mathscr{F}$ such that $\mathscr{F} \models \phi^*$ holds. The '$\models$' in $\mathscr{F} \models \phi^*$ is a certain syntactic relation defined in [Pratt-Hartmann(2005), Definition 13].

For each frame $\mathscr{F}$ with star-types $\sigma_1, \ldots, \sigma_N$, [Pratt-Hartmann(2005), Definition 14] introduces an effectively computable Presburger arithmetic formula $P_\mathscr{F}$ with N free variables. We write $P_\mathscr{F}(w_1, \ldots, w_N)$ if $P_\mathscr{F}$ is true when these variables take the values $w_1, \ldots, w_N$. The following statement is similar to the main [Pratt-Hartmann(2005), Theorem 1], and can be directly recovered from its proof and the proofs of the underlying [Pratt-Hartmann(2005), lemmas 12,13,14].

Theorem 6.12 *Given a formula ϕ^*, and the corresponding integer constant C_ϕ, there exists a computable constant X such that if $N \leq X$, if $\sigma_1, \ldots, \sigma_N$ is a sequence of star types in Σ whose integer vectors are bounded by C_ϕ, and $w_1, \ldots, w_N$ are integers, then the following are equivalent:*

1. *There exists a chromatic differentiated structure α such that $\alpha(\phi^*)$, $w_i = |S^\alpha(\sigma_i)|$ for $1 \le i \le N$, and $\alpha(\mathsf{obj}) = \bigcup_{i=1}^N S^\alpha(\sigma_i)$.*
2. *There exists a chromatic frame $\mathscr{F}$ with star types $\sigma_1, \ldots, \sigma_N$, such that $\mathscr{F} \models \phi^*$ and $P_{\mathscr{F}}(w_1, \ldots, w_N)$.*

We are now ready to describe our BAPA reduction. Fix $V_1, \ldots, V_M$ to be the list of all 1-types in signature Σ_A; let $s_1, \ldots, s_M$ be variables corresponding to their counts. By the transformation of models into chromatic, differentiated, X-sparse ones, the observations 6.11, 6.10, and Theorem 6.12, we obtain

Corollary 6.13 *If $\mathscr{M} = \{\alpha \mid \alpha(\phi^*)\}$, then there is a computable constant X such that $I^{\mathscr{M}}(\Sigma_A) = \{(s_1, \ldots, s_M) \mid F_{\phi^*}(s_1, \ldots, s_M)\}$ where $F_{\phi^*}(s_1, \ldots, s_M)$ is the following Presburger arithmetic formula*

$$\bigvee_{N, \sigma_1, \ldots, \sigma_N, \mathscr{F}} \exists w_1, \ldots, w_N.\ P_{\mathscr{F}}(w_1, \ldots, w_N) \wedge \bigwedge_{j=1}^M s_j = \sum \{w_i \mid V_j = (\pi_i|_{\Sigma_A})\}$$

where N ranges over $\{0, 1, \ldots, X\}$, $\sigma_1, \ldots, \sigma_N$ range over sequences of C_ϕ-bounded star types, and where $\mathscr{F}$ ranges over the C_ϕ-bounded frames with star types $\sigma_1, \ldots, \sigma_N$ such that $\mathscr{F} \models \phi^*$.

By adjusting for the small structures to take into account Scott normal form transformation, we further obtain

Corollary 6.14 *If $\mathscr{M} = \{\alpha \mid \alpha(\phi)\}$, then*

$$I^{\mathscr{M}}(\Sigma_A) = \{(s_1, \ldots, s_M) \mid G_\phi(s_1, \ldots, s_M)\}$$

where $G_\phi(s_1, \ldots, s_M)$ is the Presburger arithmetic formula

$$\sum_{i=1}^M s_i \ge C_\phi \wedge F_{\phi^*}(s_1, \ldots, s_M) \quad \bigvee$$
$$\bigvee \{ \bigwedge_{i=1}^M s_i = d_i \mid \exists \alpha.\ |\alpha(\mathsf{obj})| < C_\phi \wedge (d_1, \ldots, d_M) \in I^\alpha(\Sigma_A)\}$$

Theorem 6.15 *The following is a BAPA reduction for C^2 over finite models to variables Σ_A: given a two-variable logic formula ϕ, compute the BAPA formula $\exists s_1, \ldots, s_M.\ G_\phi(s_1, \ldots, s_M) \wedge \bigwedge_{i=1}^M \mathsf{card}\, V_i = s_i$.*

6.4.5 Bernays-Schönfinkel-Ramsey Fragment of First-Order Logic

Figure 6.7 shows the syntax of (our presentation of) the Bernays-Schönfinkel-Ramsey fragment of first-order logic with equality [Börger et al.(1997)Börger, Grädel, and Gurevich], often called

$$
\begin{array}{lcl}
F & ::= & \exists z_1{:}\mathsf{obj}. \ldots \exists z_n{:}\mathsf{obj}.\, \forall y_1{:}\mathsf{obj}. \ldots \forall y_m{:}\mathsf{obj}.\, B \\
B & ::= & P \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B \\
P & ::= & x_1 = x_2 \mid \{x\} \subseteq A \mid r(x_1, \ldots, x_k)
\end{array}
$$

Figure 6.7: Bernays-Schönfinkel-Ramsey Fragment of First-Order Logic

effectively propositional logic (EPR). The interpretation of atomic formulas is analogous as for C^2 in previous section. Quantification is restricted to variables of sort obj and must obey the usual restriction of $\exists^* \forall^*$-prenex form that characterizes the Bernays-Schönfinkel-Ramsey class.

6.4.6 BAPA Reduction for Bernays-Schönfinkel-Ramsey Fragment

Our BAPA reduction for the Bernays-Schönfinkel-Ramsey fragment (EPR) is in fact a reduction from EPR formulas to unary EPR formulas, in which all free variables have the sort set. To convert a unary EPR formula into BAPA, treat first-order variables as singleton sets and apply quantifier elimination for BAPA [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard].

Theorem 6.16 (BAPA Reduction for EPR) *Let ϕ be a quantifier-free formula whose free variables are: 1) $A_1, \ldots, A_p$, of sort* set*, 2) $r_1, \ldots, r_q$, each r_i of sorts* $\mathsf{obj}^{K(i)} \to \mathsf{bool}$ *for some $K(i) \geq 2$, 3) $z_1, \ldots, z_n, y_1, \ldots, y_m$, of sort* obj*. Then*

$$\exists r_1, \ldots r_q.\, \exists z_1, \ldots, z_n.\, \forall y_1, \ldots, y_m.\, \phi$$

is equivalent to an effectively computable BAPA formula.

The proof of Theorem 6.16 builds on and generalizes, for finite models, the results on the spectra of EPR formulas [Fontaine(2007), Fontaine(2009), Ramsey(1930)]. We here provide some intuition. The key insight [Ramsey(1930)] is that, when a domain of a model of an EPR formula has sufficiently many elements, then the model contains an induced submodel S of m nodes such that for every $0 \leq k < m$ elements $e_1, \ldots, e_k$ outside S the m-type induced by $e_1, \ldots, e_k$ and any $m - k$ elements in S is the same. Then an element of S can be replicated to create a model with more elements, without changing the set of all m-types in the model and thus without changing the truth value of the formula. Moreover, every sufficiently large model of the EPR formula that has a submodel S with more than m such symmetric elements can be shrunk to a model by whose expansion it can be generated. This allows us to enumerate a finite (even if very large) number of characteristic models whose expansion generates all models. The expansion of a characteristic model increases by one the number of elements of some existing 1-type, so the cardinalities of Venn regions of models are a semilinear set

whose base vectors are given by characteristic models and whose step vectors are given by the 1-types being replicated.

6.4.7 Quantifier-free Mutlisets with Cardinality Constraints

Figure 2.3 in Chapter 2 defines the syntax of quantifier-free multiset constraints with cardinality operators. We first give their interpretation in our new settings. Multiset expressions have sort $\mathsf{obj} \to \mathsf{int}$, which we abbreviate by mset in the following. Formulas are built from set expressions over set variables of sort set, multiset expressions over multiset variables of sort mset and inner and outer linear arithmetic formulas. Formally, we have distinct variables for operations on sets and multisets, e.g., we have a variable $\cup_s : \mathsf{set}^2 \to \mathsf{set}$ and a variable $\cup_m : \mathsf{mset}^2 \to \mathsf{mset}$, but we use the same symbol $\cup$ for both of them.

We restrict ourself to structures α that interpret multiset variables as functions from $[\![\mathsf{obj}]\!]$ to the nonnegative integers. Set and arithmetic operations are interpreted as in BAPA. Multiset operations are interpreted as expected, in particular, '$\uplus$' denotes additive union, '\' denotes multiset difference, and '\\' denotes set difference. The variables $\mathsf{multisetof}$ and set are interpreted as functions that convert between multisets and sets, e.g., $[\![\mathsf{set}]\!]$ maps a multiset $M : [\![\mathsf{obj}]\!] \to \mathbb{N}$ to the set $\{e \mid M(e) > 0\}$. The variable ite is interpreted as the conditional choice function, i.e., $[\![\mathsf{ite}(F, t_1, t_2)]\!]$ denotes $[\![t_1]\!]$ if $[\![F]\!] = \mathsf{true}$ and $[\![t_2]\!]$ otherwise. Finally, the atom $(u_1, \ldots, u_n) = \sum(t_1, \ldots, t_n)$ denotes true iff for all $i \in [1, n]$

$$\alpha(u_i) = \sum_{o \in [\![obj]\!]} \alpha[e := o](t_i)$$

Let $\mathscr{F}_{\mathsf{MS}}$ be the set of all formulas defined in Figure 2.3 and let $\mathscr{M}_{\mathsf{MS}}$ be the set of all structures interpreting formulas in $\mathscr{F}_{\mathsf{MS}}$ as described above. We define the theory of quantifier-free multisets with cardinality constraints $\mathscr{T}_{\mathsf{MS}}$ as the set of all formulas $F \in \mathscr{F}_{\mathsf{MS}}$ such that $\alpha(F)$ is true for all structures α in $\mathscr{M}_{\mathsf{MS}}$.

6.4.8 BAPA Reduction for Quantifier-free Multiset Constraints

The satisfiability of the quantifier-free fragment of multisets with cardinality operators is decidable (Theorem 2.25). There is, in fact, also a BAPA reduction from a quantifier-free multiset formula over multiset and set variables to a BAPA formula ranging only over the set variables.

Let $F \in \mathscr{F}_{\mathsf{MS}}$ be a multiset constraint containing set variables $A_1, \ldots, A_p$ and multiset variables $M_1, \ldots, M_q$. To obtain a BAPA reduction, we apply the decision procedure described in Chapter 2 to the formula F_1

$$F_1 \equiv F \wedge \bigwedge_{i=1}^{w} \mathsf{card}\, V_i = k_i$$

where $k_1, \ldots, k_w$ are fresh integer variables and $V_1, \ldots, V_w$ are the Venn regions over the set variables $A_1, \ldots, A_p$. Before applying the decision procedure we convert F_1 to a formula F_2 that only ranges over multiset variables. This is done by replacing every set operation in F by the corresponding multiset operation, replacing every set variable A_i by a fresh multiset variable M_{A_i}, and conjoining the formula

$$\forall e : \mathsf{obj}. \bigwedge_{i=1}^{p} (M_{A_i}(e) = 0 \vee M_{A_i}(e) = 1) \ .$$

The decision procedure constructs a Presburger arithmetic formula P with $\{k_1, \ldots, k_w\} \subseteq \mathsf{FV}(P)$. From the proofs of Theorem 2.2, Theorem 2.4 and Theorem 2.14 follows that for every structure α in which formula F_2 evaluates to true, there exists a structure α' in which formula P evaluates to true and for every variable k_i, $\alpha(k_i) = \alpha'(k_i)$. The converse holds as well, i.e. for every model of P there is a model of F_2 with the previous property. We conclude that the following holds:

$$\{\alpha|_{\{k_1,\ldots,k_w\}} \mid \alpha(F_2)\} = \{\alpha|_{\{k_1,\ldots,k_w\}} \mid \alpha(P)\}$$

If $x_1, \ldots, x_n$ are the variables in P other than $k_1, \ldots, k_w$ then the result of the BAPA reduction is the formula

$$P_F \stackrel{def}{=} \exists k_1 : \mathsf{int}. \ldots \exists k_w : \mathsf{int}. \left(\bigwedge_{i=1}^{w} \mathsf{card}\, V_i = k_i\right) \wedge (\exists x_1 : \mathsf{int}. \ldots \exists x_n : \mathsf{int}.\ P) \tag{6.5}$$

Theorem 6.17 *The function mapping a formula $F \in \mathscr{F}_{\mathsf{MS}}$ to the BAPA formula P_F is a BAPA reduction for $(\mathscr{F}_{\mathsf{MS}}, \mathscr{T}_{\mathsf{MS}})$.*

The proof of Theorem 6.17 uses Equation 6.5 following a similar argument than in the proof of Theorem 6.9.

6.5 Further Related Work

There are combination results for the disjoint combinations of non-stably infinite theories [Tinelli and Zarba(2005), Krstic et al.(2007)Krstic, Goel, Grundy, and Tinelli, Fontaine(2007), Fontaine(2009)]. These results are based on the observation that such combinations are possible whenever one can decide for each component theory whether a model of a specific cardinality exists. Our combination result takes into account not only the cardinality of the models, i.e. the interpretation of the universal set, but cardinalities of Venn regions over the interpretations of arbitrary shared set variables. It is a natural generalization of the disjoint case restricted to theories that share the theory of finite sets, thus, leading to a non-disjoint combination of non-stably infinite theories.

Ghilardi [Ghilardi(2005)] proposes a model-theoretic condition for decidability of the non-disjoint combination of theories based on quantifier elimination and local finiteness of the

shared theory. Note that BAPA is not locally finite and that, in general, we need the full expressive power of BAPA to compute the projections on the shared set variables. For instance, consider the C^2 formula

$$(\forall x.\exists^{=1} y.r(x,y)) \wedge (\forall x.\exists^{=1} y.r(y,x)) \wedge (\forall y.\ y \in B \leftrightarrow (\exists x.x \in A \wedge r(x,y)))$$

where r is a binary relation variable establishing the bijection between A and B. This constraint expresses $|A| = |B|$ without imposing any additional constraint on A and B. Similar examples can be given for weak monadic second-order logic of finite trees.

Another example, belonging to WS1S logic, further demonstrates why Venn regions are important in constructing a reduction. Consider a formula $((A \wedge \neg B)(B \wedge \neg A))^*(\neg B \wedge \neg A)^*$. Every Venn region denotes a letter of the alphabet Σ of the accepting automaton for the formula: $\Sigma = (A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B)$. The Parikh image of the all models of the formula is the set $\{(0,p,p,q) \mid q,p \geq 0\}$. The elements of this semilinear set are all the solutions of the formula $|A \cap B^c| = |A^c \cap B| \wedge |A \cap B| = 0$, which is exactly the projection of the original formula.

The reduction approach to combination of decision procedures has previously been applied in the simpler scenario of reduction to propositional logic [Lahiri and Seshia(2004)]. Like propositional logic, quantifier-Free BAPA is NP-complete, so it presents an appealing alternative for combination of theories that share sets.

Gabbay and Ohlbach [Gabbay and Ohlbach(1992)] present a procedure, called SCAN, for second-order quantifier elimination. However, [Gabbay and Ohlbach(1992)] gives no characterization of when SCAN terminates. We were therefore unable to use SCAN to derive any BAPA reductions.

The general combination of weak monadic second-order logics with linear cardinality constraints has been proven undecidable by Klaedtke and Rueß [Klaedtke and Rueß(2002), Klaedtke and Rueß(2003)]. They introduce the notion of Parikh automata to identify decidable fragments of this logic which inspired our BAPA reduction of MSOL of finite trees. Our combined logic is incomparable to the decidable fragments identified by Klaedtke and Rueß because it supports non-tree structures as well. However, by applying projection to C^2 and the Bernays-Schönfinkel-Ramsey class, we can combine our logic with [Klaedtke and Rueß(2002), Klaedtke and Rueß(2003)], obtaining an even more expressive decidable logic.

6.6 Conclusions

Many verification techniques rely on decision procedures to achieve a high degree of automation. The class of properties that such techniques are able to verify is therefore limited by the expressive power of the logics supported by the underlying decision procedures. We have presented a combination result for logics that share operations on sets. This result yields an expressive decidable logic that is useful for software verification. We therefore believe that we

made an important step in increasing the class of properties that are amenable to automated verification.

7 Complete Functional Synthesis

In this chapter we discuss complete functional synthesis. Synthesis of program fragments from specifications can make programs easier to write and easier to reason about. To integrate synthesis into programming languages, synthesis algorithms should behave in a predictable way - they should always find a code for a well-defined class of specifications. To guarantee correctness and applicability to software (and not just hardware), these algorithms should also support unbounded data types, such as numbers and data structures.

This chapter describes how to generalize decision procedures into predictable and complete synthesis procedures. Such procedures are guaranteed to find code that satisfies the specification if such code exists. Moreover, we identify conditions under which synthesis will statically decide whether the solution is guaranteed to exist, and whether it is unique. We demonstrate our approach by starting from a quantifier elimination decision procedure for Boolean Algebra of set with Presburger Arithmetic (BAPA) and transforming it into a synthesis procedure.

7.1 Motivation

Synthesis of software from specifications, discussed already in [Manna and Waldinger(1971), Manna and Waldinger(1980), Green(1969)], promises to make programmers more productive. Despite substantial recent progress [Solar-Lezama et al.(2006)Solar-Lezama, Tancau, Bodík, Seshia, and Saraswat, Solar-Lezama et al.(2008)Solar-Lezama, Jones, and Bodík, Vechev et al.(2009)Vechev, Yahav, and Yorsh, Srivastava et al.(2010)Srivastava, Gulwani, and Foster], synthesis is limited to small pieces of code. We expect that this will continue to be the case for some time in the future, for two reasons: 1) synthesis is algorithmically a difficult problem, and 2) synthesis requires detailed specifications, which for large programs become difficult to write.

We therefore expect that practical applications of synthesis lie in its integration into the compilers of general-purpose programming languages. To make this integration feasible, we aim to identify well-defined classes of expressions and synthesis algorithms guaranteed

to succeed for these classes of expressions, just like a compilation attempt succeeds for any well-formed program. Our starting point for such synthesis algorithms are *decision procedures*.

A decision procedure for satisfiability of a class of formulas accepts a formula in its class and checks whether the formula has a solution. On top of this basic functionality, many decision procedure implementations provide the additional feature of generating a satisfying assignment (a model) whenever the given formula is satisfiable. Such a model-generation functionality has many uses, including better error reporting in verification [Moskal(2009)] and test-case generation [Anand et al.(2008)Anand, Godefroid, and Tillmann]. An important insight is that model generation facility of decision procedures could also be used as an advanced computation mechanism. Given a set of values for some of the variables, a constraint solver can at run-time find the values of the remaining variables such that a given constraint holds. Two recent examples of integrating such a mechanism into a programming language are the quotations of the *F#* language [Syme et al.(2007)Syme, Granicz, and Cisternino] and a Scala library [Köksal et al.(2011)Köksal, Kuncak, and Suter], both interfacing to the Z3 satisfiability modulo theories (SMT) solver [de Moura and Bjørner(2008a)]. Such mechanisms promise to bring the algorithmic improvements of SMT solvers to declarative paradigms such as Constraint Logic Programming [Jaffar and Maher(1994)]. However, they involve a possibly unpredictable search at run-time, and require the deployment of the entire decision procedure as a component of the run-time system.

Our goal is to provide the benefits of the declarative approach in a more controlled way: we aim to run a decision procedure at *compile time* and use it to generate code. The generated code then computes the desired values of variables at run-time. Such code is thus specific to the desired constraint, and can be more efficient. It does not require the decision procedure to be present at run-time, and gives the developer static feedback by checking the conditions under which the generated solution will exist and be unique. We use the term *synthesis* for our approach because it starts from an implicit specification, and involves compile-time precomputation. Because it computes a function that satisfies a given input/output relation, we call our synthesis *functional*, in contrast to reactive synthesis approaches [Pnueli and Rosner(1989)] (another term for the general direction of our approach is AE-paradigm or Skolem paradigm). Finally, we call our approach *complete* because it is guaranteed to work for all specification expressions from a well-specified class.

We demonstrate our approach by describing synthesis algorithms for the domains of linear arithmetic and collections of objects. We have implemented these synthesis algorithms and deployed them as a compiler extension of the Scala programming language [Odersky et al.(2008)Odersky, Spoon, and Venners]. We have found that using such constraints we were able to express a number of program fragments in a more natural way, stating the invariants that the program should satisfy as opposed to the computation details of establishing these invariants.

In the area of integer arithmetic, we obtain a language extension that can implicitly define

integer variables to satisfy given constraints. The applications of integer arithmetic synthesis include conversions of quantities expressed in terms of multiple units of measure, coordinate transformations, as well as a substantially more general notion of pattern matching on integers, going well beyond matching on constants or $(n+k)$-patterns of the Haskell programming language [Jones and group of authors(2010)].

In the area of data structures, we describe a synthesis procedure that can compute sets of elements subject to constraints expressed in terms of basic set operations (union, intersection, set difference, subset, equality) as well as linear constraints on sizes of sets. We have found these constraints to be useful for manipulating sets of objects in high-level descriptions of algorithms, from simple operations such as choosing an element from a set or a fresh element, or splitting sets subject to size constraints. Such constructs arise in pseudo code notations, and they provide a useful addition to the transformations previously developed for the SETL programming language [Dewar(1979), Sharir(1982)]. Regarding data structures, in our synthesis procedure we focus on sets, but the approach applies to other constraints for which decision procedures are available [Kuncak et al.(2010d)Kuncak, Piskac, Suter, and Wies], including multisets (Chapters 2 and 4) and algebraic data types [Suter et al.(2010)Suter, Dotta, and Kuncak].

We implemented our approach to program synthesis as a plugin for the Scala compiler. Our implementation can be used as a starting point for the development of further synthesis approaches.

7.2 Example

We first illustrate the use of a synthesis procedure for integer linear arithmetic. Consider the following example to break down a given number of seconds (stored in the variable `totsec`) into hours, minutes, and leftover seconds.

```
val (hrs, mns, scs) = choose((h: Int, m: Int, s: Int) ⇒
 h * 3600 + m * 60 + s == totsec &&
 0 ≤ m && m ≤ 60 &&
 0 ≤ s && s ≤ 60)
```

Our synthesizer succeeds, because the constraint is in integer linear arithmetic. However, the synthesizer emits the following warning:

```
Synthesis predicate has multiple solutions
for variable assignment: totsec = 0
  Solution 1: h = 0, m = 0, s = 0
  Solution 2: h = -1, m = 59, s = 60
```

The reason for this warning is that the bounds on `m` and `s` are not strict. After correcting the

error in the specification, replacing $m \leq 60$ with $m < 60$ and $s \leq 60$ with $s < 60$, the synthesizer emits no warnings and generates code corresponding to the following:

```
val (hrs, mns, scs) = {
  val loc1 = totsec div 3600
  val num2 = totsec + ((−3600)∗ loc1)
  val loc2 = min(num2 div 60, 59)
  val loc3 = totsec + ((−3600)∗ loc1) + (−60 ∗ loc2)
  (loc1, loc2, loc3)
}
```

The absence of warnings guarantees that the solution always exists and that it is unique. The developer directly ensures that the condition $h * 3600 + m * 60 + s == totsec$ will be satisfied, making program understanding easier, by writing the code in this style. Note that, if the developer imposes the constraint

```
val (hrs, mns, scs) = choose((h: Int, m: Int, s: Int) ⇒
 h ∗ 3600 + m ∗ 60 + s == totsec &&
 0 ≤ h < 24 &&
 0 ≤ m && m < 60 &&
 0 ≤ s && s < 60)
```

our system emits the following warning:

```
Synthesis predicate is not satisfiable
for variable assignment: totsec = 86400
```

pointing to the fact that the constraint has no solutions when the `totsec` parameter is too large.

Our approach and implementation also work for parametrized integer arithmetic formulas, which become linear only once the parameters are known. For example, our synthesizer accepts the following specification that decomposes an offset of a linear representation of a three-dimensional array with statically unknown dimensions into indices for each coordinate:

```
val (x1, y1, z1) = choose((x: Int, y: Int, z: Int) ⇒
  offset == x + dimX ∗ y + dimX ∗ dimY ∗ z &&
  0 ≤ x && x < dimX &&
  0 ≤ y && y < dimY &&
  0 ≤ z && z < dimZ)
```

Here dimX, dimY, dimZ are variables whose value is unknown until runtime. Note that the satisfiability of constraints that contain multiplications of variables is in general undecidable.

In such parametrized case, our synthesizer is complete in the sense that it generates code that 1) always terminates, 2) detects at run-time whether a solution exists for current parameter values, and 3) computes one solution whenever a solution exists.

In addition to integer arithmetic, other theories are amenable to synthesis and provide similar benefits. Consider the problem of splitting a set collection in a balanced way. The following code attempts to do that:

```
val (a1,a2) = choose((a1:Set[O],a2:Set[O]) ⇒
  a1 union a2 == s && a1 intersect a2 == empty &&
  a1.size == a2.size)
```

It turns out that for the above code our synthesizer emits a warning indicating that there are cases where the constraint has no solutions. Indeed, there are no solutions when the set `s` is of odd size. If we weaken the specification to

```
val (a1,a2) = choose((a1:Set[O],a2:Set[O]) ⇒
  a1 union a2 == s && a1 intersect a2 == empty &&
  a1.size − a2.size ≤ 1 &&
  a2.size − a1.size ≤ 1)
```

then our synthesizer can prove that the code has a solution for all possible input sets `s`. The synthesizer emits code that, for each input, computes one such solution. The nature of constraints on sets is that if there is one solution, then there are many solutions. Our synthesizer resolves these choices at compile time, which means that the generated code is deterministic.

7.3 From Decision to Synthesis Procedures

We next define precisely the notion of a synthesis procedure and describe a methodology for deriving synthesis procedures from decision procedures.

Preliminaries. Each of our algorithms works with a set of formulas, Formulas, build from terms, whose set we denote with Terms. Formulas denote truth values, whereas terms and variables denote values from the domain (e.g. integers). We denote the set of variables by Vars. $\mathsf{FV}(q)$ denotes the set of free variables in a formula or a term q. If $\vec{x} = (x_1, \ldots, x_n)$ then $\vec{x}_s$ denotes the set of variables $\{x_1, \ldots, x_n\}$. If q is a term or formula, $\vec{x} = (x_1, \ldots, x_n)$ is a vector of variables and $\vec{t} = (t_1, \ldots, t_n)$ is a vector of terms, then $q[\vec{x} := \vec{t}]$ denotes the result of substituting in q the free variables $x_1, \ldots, x_n$ with terms $t_1, \ldots, t_n$, respectively. Given a substitution $\sigma : \mathsf{FV}(F) \to \mathsf{Terms}$, we write $F\sigma$ for the result of substituting each $x \in \mathsf{FV}(F)$ with $\sigma(x)$. Formulas are interpreted over elements of a first-order structure $\mathscr{D}$ with a countable domain D. We assume that for each $e \in D$ there exists a ground term c_e whose interpretation in $\mathscr{D}$ is e; let

$C = \{c_e \mid e \in D\}$. We further assume that if $F \in \mathsf{Formulas}$ then also $F[x := c_e] \in \mathsf{Formulas}$ (the class of formulas is closed under partial grounding with constants).

The choose programming language construct. We integrate into a programming language a construct of the form

$$\vec{r} = \mathsf{choose}(\vec{x} \Rightarrow F) \tag{7.1}$$

Here F is a formula (typically represented as a boolean-valued programming language expressions) and $\vec{x} \Rightarrow F$ denotes an anonymous function from $\vec{x}$ to the value of F (that is, $\lambda\vec{x}.F$). Two kinds of variables can appear within F: output variables $\vec{x}$ and parameters $\vec{a}$. The parameters $\vec{a}$ are program variables that are in scope at the point where choose occurs; their values will be known when the statement is executed. Output variables $\vec{x}$ denote values that need to be computed so that F becomes true, and they will be assigned to $\vec{r}$ as a result of the invocation of choose.

We can translate the above choose construct into the following sequence of commands in a guarded command language [Dijkstra(1976)]:

assert$(\exists\vec{x}.F)$;
havoc $(\vec{r})$;
assume $(F[\vec{x} := \vec{r}])$;

The simplicity of the above translation indicates that it is natural to represent choose within existing verification systems (e.g. [Flanagan et al.(2002)Flanagan, Leino, Lilibridge, Nelson, Saxe, and Stata, Zee et al.(2008)Zee, Kuncak, and Rinard]) The use of choose can help verification because the desired property F is explicitly assumed and can aid in proving the subsequent program assertions.

Model-generating decision procedures. As a starting point for our synthesis algorithms for choose invocations we consider a model-generating decision procedure. Given $F \in \mathsf{Formulas}$ we expect this decision procedure to produce either

a) a substitution $\sigma : \mathsf{FV}(F) \to C$ such that $F\sigma$ is a true, or

b) a special value unsat indicating that the formula is unsatisfiable.

We assume that the decision procedure is deterministic and behaves as a function. We write $Z(F)$=σ or $Z(F)$=unsat to denote the result of applying the decision procedure to F.

Baseline: invoking a decision procedure at run-time. Just like an interpreter can be considered as a baseline implementation for a compiler, deploying a decision procedure at run-time

can be considered as a baseline for our approach. In this scenario, we replace the statement (7.1) with the code

```
F = makeFormulaTree(makeVars(x⃗), makeGroundTerms(a⃗));
r⃗ = (Z(F) match {
  case σ ⇒ (σ(x_1),...,σ(x_n))
  case unsat ⇒ throw new Exception("No solution exists")
})
```

Such dynamic invocation approach is flexible and useful. However, there are important performance and predictability advantages of an alternative *compilation* approach.

Synthesis based on decision procedures. Our goal is therefore to explore a compilation approach where a modified decision procedure is invoked at compile time, converting the formula into a solved form.

Definition 7.1 (Synthesis Procedure) *We denote an invocation of a synthesis procedure by* $[\![\vec{x}, F]\!] = (\mathsf{pre}, \vec{\Psi})$. *A synthesis procedure takes as input a formula* F *and a vector of variables* $\vec{x}$ *and outputs a pair of*

1. *a precondition formula* pre *with* $\mathsf{FV}(\mathsf{pre}) \subseteq \mathsf{FV}(F) \setminus \vec{x}_s$
2. *a tuple of terms* $\vec{\Psi}$ *with* $\mathsf{FV}(\vec{\Psi}) \subseteq \mathsf{FV}(F) \setminus \vec{x}_s$

such that the following two implications are valid:

$$(\exists \vec{x}.F) \rightarrow \mathsf{pre}$$
$$\mathsf{pre} \rightarrow F[\vec{x} := \vec{\Psi}]$$

Observation 7.2 *Because another implication always holds:*

$$F[\vec{x} := \vec{\Psi}] \rightarrow \exists \vec{x}.F$$

the above definition implies that the three formulas are all equivalent: $(\exists \vec{x}.F), \mathsf{pre}, F[\vec{x} := \vec{\Psi}]$. *Consequently, if we can define a function* witn *where for* $\mathsf{witn}(\vec{x}, F) = \vec{\Psi}$ *we have* $\mathsf{FV}(\vec{\Psi}) \subseteq \mathsf{FV}(F) \setminus \vec{x}_s$ *and* $\exists \vec{x}.F$ *implies* $F[\vec{x} := \vec{\Psi}]$, *then we can define a synthesis procedure by*

$$[\![\vec{x}, F]\!] = (F[\vec{x} := \mathsf{witn}(\vec{x}, F)], \mathsf{witn}(\vec{x}, F))$$

The reason we use the translation that computes pre in addition to $\mathsf{witn}(\vec{x}, F)$ is that the synthesizer performs simplifications when generating pre, which can produce a formula faster to evaluate than $F[\vec{x} := \mathsf{witn}(\vec{x}, F)]$.

The synthesizer emits the terms $\vec{\Psi}$ in compiler intermediate representation; the standard compiler then processes them along with the rest of the code. We identify the syntax tree of $\vec{\Psi}$ with its meaning as a function from the parameters $\vec{a}$ to the output variables $\vec{x}$. The overall compile-time processing of the choose statement (7.1) involves the following:

1. emit a non-feasibility warning if the formula $\neg \mathsf{pre}$ is satisfiable, reporting the counterexample for which the synthesis problem has no solutions;
2. emit a non-uniqueness warning if the formula

   $$F \wedge F[\vec{x} := \vec{y}] \wedge \vec{x} \neq \vec{y}$$

   is satisfiable, reporting the values of all free variables as a counterexample showing that there are at least two solutions;
3. as the compiled code, emit the code that behaves as

   $\mathbf{assert}(\mathsf{pre});\ \vec{r} = \vec{\Psi}$

The existence of a model-generating decision procedure implies the existence of a 'trivial' synthesis procedure, which satisfies Definition 7.1 but simply invokes the decision procedure at run-time. (In the realm of conventional programming languages, this would be analogous to 'compiling' the code by shipping its source code bundled with an interpreter.) The usefulness of the notion of synthesis procedure therefore comes from the fact that we can often create compiled code that avoids this trivial solution. Among the potential advantages of the compilation approach are:

- improved run-time efficiency, because part of the reasoning is done at compile-time;
- improved error reporting: the existence and uniqueness of solutions can be checked at compile time;
- simpler deployment: the emitted code can be compiled to any of the targets of the compiler, and requires no additional run-time support.

We decided therefore pursue the compilation approach. As for the processing of more traditional programming language constructs, we do believe that there is space in the future for mixed approaches, such as 'just-in-time synthesis' and 'profiling-guided synthesis'.

Efficiency of synthesis. We introduce the following measures to quantify the behavior of synthesis procedures as a function of the specification expression F:

- time to synthesize the code, as a function of F;

- size of the synthesized code, as a function of F;
- running time of the synthesized code as a function of F and a measure of the run-time values for the parameters $\vec{a}$.

When using F as the argument of the above measures, we often consider not only the size of F as a syntactic object, but also the dimension of the variable vector $\vec{x}$ and the parameter vector $\vec{a}$ of F.

From quantifier elimination to synthesis. The precondition pre can be viewed as a result of applying quantifier elimination (see e.g. [Hodges(1993), Page 67], [Nipkow(2008)]) to remove $\vec{x}$ from F, with the following differences.

1. Synthesis procedures strengthen quantifier elimination procedures by identifying not only pre but also emitting the code $\vec{\Psi}$ that efficiently computes a *witness* for $\vec{x}$.
2. Quantifier elimination is typically applied to arbitrary quantified formulas of first-order logic and aims to successively eliminate all variables. To enable recursive application of variable elimination, pre must be in the same language of formulas as F. This condition is not required in the final step of synthesis procedure, because no further elimination is applied to the final precondition. Therefore, if the final precondition becomes a run-time check, it can contain arbitrary executable code. If the final precondition becomes a compile-time satisfiability check for the totality of the relation, then it suffices for it to be in any decidable logic.
3. Worst-case bounds on quantifier elimination algorithms measure the size of the generated formula and the time needed to generate it, but not the size of $\vec{\Psi}$ or the time to evaluate $\vec{\Psi}$. For some domains, it can be computationally more difficult to compute (or even 'print') the solution than to simply check the existence of a solution.

Despite the differences, we have found that we can naturally extend existing quantifier elimination procedures with explicit computation of witnesses that constitute the program $\vec{\Psi}$.

7.4 Selected Generic Techniques

We next describe some basic observations and techniques for synthesis that are independent of a particular theory.

7.4.1 Synthesis for Multiple Variables

Suppose that we have a function $\mathsf{witn}(x, F)$ that corresponds to constructive quantifier elimination step for one variable and produces a term Ψ such that $F[x := \Psi]$ holds iff $\exists x.F$ holds. We

$$[\![\_ , \_]\!] : \bigcup_n \left(\mathsf{Vars}^n \times \mathsf{Formulas} \to \mathsf{Formulas} \times \mathsf{Terms}^n\right)$$

$$[\![(), F]\!] = (F, ())$$

$$[\![(x_1, \ldots, x_n), F]\!] = (\mathsf{pre}, (\Psi_1, \ldots, \Psi_{n-1}, \Psi'_n)),$$

where

$$\Psi_n = \mathsf{witn}(x_n, F)$$

$$F' = \mathsf{simplify}(F[x_n := \Psi_n])$$

$$(\mathsf{pre}, (\Psi_1, \ldots, \Psi_{n-1})) = [\![(x_1, \ldots, x_{n-1}), F']\!]$$

$$\Psi'_n = \Psi_n[x_1 := \Psi_1, \ldots, x_{n-1} := \Psi_{n-1}]$$

Figure 7.1: Successive Elimination of Variables for Synthesis

can then lift $\mathsf{witn}(x, F)$ to synthesis for any number of variables, using the (non-tail recursive) translation scheme in Figure 7.1. This translation includes the base case in which there are no variables to eliminate, so F becomes the precondition, and the recursive case that applies the witn function.

In implementation, we can use local variable definitions instead of substitutions. Given (7.1), we generate as $\vec{\Psi}$ a Scala code block

$$\left\{\begin{array}{l} \mathsf{val}\ x_1 \quad = \Psi_1 \\ \ldots \\ \mathsf{val}\ x_{n-1} = \Psi_{n-1} \\ \mathsf{val}\ x_n \quad = \Psi_n \\ \vec{x} \end{array}\right\}$$

where the variables in Ψ_n directly refer to variables computed in $\Psi_1, \ldots, \Psi_{n-1}$ and where $\mathsf{FV}(\Psi_i) \subseteq \mathsf{FV}(F) \setminus \{x_i, \ldots, x_n\}$. A consequence of this recursive translation pattern is that the synthesized code computes values in reverse order compared to the steps of a quantifier elimination procedure. This observation can be helpful in understanding the output of our synthesis procedures.

7.4.2 One-Point Rule Synthesis

If $x \notin \mathsf{FV}(t)$ we can define

$$\mathsf{witn}(x,\ x = t \wedge F) = t$$

If the formula does not have the form $x = t \wedge F$, we can often rewrite it into this form using theory-specific transformations.

7.4.3 Output-Independent Preconditions

Whenever $\mathsf{FV}(F_1) \cap \vec{x}_s = \emptyset$, we can apply the following synthesis rule:

$$[\![\vec{x}, F_1 \wedge F_2]\!] = (\mathsf{pre} \wedge F_1, \vec{\Psi}),$$
where
$$(\mathsf{pre}, \vec{\Psi}) = [\![\vec{x}, F_2]\!]$$

which moves a 'constant' conjunct of the specification into the precondition. We assume that this rule is applied whenever possible and do not explicitly mention it in the rest of the chapter.

7.4.4 Propositional Connectives in First-Order Theories

Consider a quantifier-free formula in some first-order theory. Consider the tasks of checking formula satisfiability or applying elimination of a variable. For both tasks, we can first rewrite the formula into disjunctive normal form and then process each disjunct independently. This allows us to focus on handling conjunctions of literals as opposed to arbitrary propositional combination.

We next show that we can similarly use disjunctive normal form in synthesis. Consider a formula $D_1 \vee \ldots \vee D_n$ in disjunctive normal form. We can apply synthesis to each D_i yielding a precondition pre_i and the solved form $\vec{\Psi}_i$. We can then synthesize code with conditionals that select the first $\vec{\Psi}_i$ that applies:

$$[\![\vec{x}, D_1 \vee \ldots \vee D_n]\!] = \left(\bigvee_{i=1}^{n} \mathsf{pre}_i, \left\{ \begin{array}{ll} \mathsf{if}\ (\mathsf{pre}_1) & \vec{\Psi}_1 \\ \mathsf{else\ if}\ (\mathsf{pre}_2) & \vec{\Psi}_2 \\ \ldots & \\ \mathsf{else\ if}\ (\mathsf{pre}_n) & \vec{\Psi}_n \\ \mathsf{else} & \\ \quad \mathsf{throw\ new}\ \text{Exception}(\text{“No solution”}) & \end{array} \right\} \right),$$
where
$$(\mathsf{pre}_1, \vec{\Psi}_1) = [\![\vec{x}, D_1]\!]$$
$$\ldots$$
$$(\mathsf{pre}_n, \vec{\Psi}_n) = [\![\vec{x}, D_n]\!]$$

Although the disjunctive normal form can be exponentially larger than the original formula, the transformation to disjunctive normal form is used in practice [Pugh(1992)] and has advantages in terms of the quality of synthesized code generated for individual disjuncts. What further justifies this approach is that we expect a small number of disjuncts in our specifications, and may need different synthesized values for variables in different disjuncts.

Other methods can have better worst-case quantifier elimination complexity [Cooper(1972),

Ferrante and Rackoff(1979), Weispfenning(1997), Nipkow(2008)] than disjunctive normal form approaches. We discuss these alternative approaches in the sequel as well, but it is the above disjunctive normal form approach that we currently use in our implementation.

7.4.5 Synthesis for Propositional Logic

We are focused on synthesis for formulas over unbounded domains. Nonetheless, to illustrate the potential asymptotic gain of precomputation in synthesis, we illustrate synthesis for the case when F is a propositional formula (see e.g. [Kukula and Shiple(2000)] for a more sophisticated approach to this problem). Suppose that $\vec{x}$ are output variables and $\vec{a}$ are the remaining propositional variables (parameters) in F.

To synthesize a function from $\vec{a}$ to $\vec{x}$, build an ordered binary decision diagram (OBDD) [Bryant(1986)] for F, treating both $\vec{a}$ and $\vec{x}$ as variables for OBDD construction, and using a variable ordering that puts all parameters $\vec{a}$ before all output variables $\vec{x}$. Then split the OBDD graph at the point where all the decisions on $\vec{a}$ have been made. That is, consider the set of nodes that terminate on some paths on which all decisions on $\vec{a}$ have been made and no decisions on $\vec{x}$ have been made. For each of these OBDD nodes, we precompute whether this node reaches the true sink node. As the result of synthesis, we emit the code that consists of nested if-then-else tests encoding the decisions on $\vec{a}$, followed by the code that assigns values to $\vec{x}$ that will lead to the true sink node.

Consider the code generated using the method above. Note that, although the size of the code is bounded by a single exponential, the code executes in time close to linear in the total number of variables $\vec{a}$ and $\vec{x}$. This is in contrast to NP-hardness of finding a satisfying assignment for a propositional formula F, which would occur in the baseline approach of invoking a SAT solver at run-time. In summary, for propositional logic synthesis (and, more generally, for NP-hard constraints over bounded domains) we can precompute solutions and generate code that computes the desired values in deterministic polynomial time in the size of inputs and outputs.

In the next several sections, we describe synthesis procedures for several useful decidable logics over *infinite* domains (numbers and data structures) and discuss the efficiency improvements due to synthesis.

7.5 Synthesis for Linear Rational Arithmetic

We next consider synthesis for quantifier-free formulas of linear arithmetic over rationals. In this theory, variables range over rational numbers, terms are linear expressions $c_0 + c_1 x_1 + \ldots + c_n x_n$, and the relations in the language are $<$ and $=$. Synthesis for this theory can be used to synthesize exact fractional arithmetic computations (or floating-point computations if we are willing to ignore the rounding errors). It also serves as an introduction to the more complex

problem of integer arithmetic synthesis that we describe in the following sections.

Given a quantifier-free formula, we can efficiently transform it to negation-normal form. Furthermore, we observe that $\neg(t_1 < t_2)$ is equivalent to $(t_2 < t_1) \vee (t_1 = t_2)$ and that $\neg(t_1 = t_2)$ is equivalent to $(t_1 < t_2) \vee (t_2 < t_1)$. Therefore, there is no need to consider negations in the formula. We can also normalize the equalities to the form $t = 0$ and the inequalities to the form $0 < t$.

7.5.1 Solving Conjunctions of Literals

Given the observations in Section 7.4.4, we consider conjunctions of literals. The method follows Fourier-Motzkin elimination [Schrijver(1998)]. Consider the elimination of a variable x.

Equalities. If x occurs in an equality constraint $t = 0$, then solve the constraint for x and rewrite it as $x = t'$, where t' does not contain x. Then simply apply the one-point rule synthesis (Section 7.4.2). This step amounts to Gaussian elimination. We follow this step whenever possible, so we first eliminate those variables that occur in some equalities and only then proceed to inequalities.

Inequalities. Next, suppose that x occurs only in strict inequalities $0 < t$. Depending on the sign of x in t, we can rewrite these inequalities into $a_p < x$ or $x < b_q$ for some terms a_p, b_q. Consider the more general case when there is both at least one lower bound a_p and at least one upper bound b_q. We can then define:

$$\mathsf{witn}(x, F) = (\max_p\{a_p\} + \min_q\{b_q\})/2$$

As one would expect from quantifier elimination, the pre corresponding to this case results from F by replacing the conjunction of all inequalities containing x with the conjunction

$$\bigwedge_{p,q} a_p < b_q$$

In case there are no lower bounds a_p, we define $\mathsf{witn}(x, F) = \min_q\{b_q\} - 1$; if there are no upper bounds b_q, we define $\mathsf{witn}(x, F) = \max_p\{a_p\} + 1$.

Complexity of synthesis for conjunctions. We next examine the size of the generated code for linear rational arithmetic. The elimination of input variables using equalities is a polynomial-time transformation. Suppose that after this elimination we are left with N inequalities and V remaining input variables. The above inequality elimination step for one variable replaces N inequalities with $(N/2)^2$ inequalities in the worst case. After eliminating all output variables, an upper bound on the formula increase is $(N/2)^{2^V}$. Therefore, the generated formula can

be in the worst case doubly exponential in the number of output variables V. However, for a fixed V, the generated code size is a (possibly high-degree) polynomial of the size of the input formula. Also, if there are 4 or fewer inequalities in the original formula, the final size is polynomial, regardless of V. Finally, note that the synthesis time and the execution time of synthesized code are polynomial in the size of the generated formula.

7.5.2 Disjunctions for Linear Rational Arithmetic

We next consider linear arithmetic constraints with disjunctions, which are constraints for which the satisfiability is NP-complete. One way to lift synthesis for rational arithmetic from conjunctions of literals to arbitrary propositional combinations is to apply the disjunctive normal form method of Section 7.4.4. We then obtain a complexity that is one exponential higher in formula size than the complexity of synthesis for conjunctions.

In the rest of this section, we consider an alternative to disjunctive normal form. This alternative synthesizes code that can execute exponentially faster (even though it is not smaller) compared to the disjunctive normal form approach of Section 7.4.4.

The starting point of this method are quantifier elimination techniques that avoid disjunctive normal form transformation, e.g. [Ferrante and Rackoff(1979)], [Nipkow(2008)], [Bradley and Manna(2007), Section 7.3]. To remove a variable from negation normal form, this method finds relevant lower bounds a_p and upper bounds b_q in the formula, then computes the values $m_{pq} = (a_p + b_q)/2$ and replaces a variable x_i with the values from the set $\{m_{pq}\}_{p,q}$ extended with "sufficiently small" and "sufficiently large" values [Nipkow(2008)]. This quantifier elimination method gives us a way to compute pre.

We next present how to extend this quantifier elimination method to synthesis, namely to the computation of $\mathsf{witn}(x, F)$. Consider a substitution in quantifier elimination step that replaces variable x_i with the term m. We then extend this step to also attach to each literal a special substitution syntactic form $(x_i \mapsto m)$. When using this process to eliminate one variable, the size of the formula can increase quadratically. After eliminating all output variables, we obtain a formula pre with additional annotations; the size of this formula is bounded by $n^{2^{O(V)}}$ where n is the original formula size. (Again, although it is doubly exponential in V, it is not exponential in n.)

We can therefore build a decision tree that evaluates the values of all $n^{2^{O(V)}}$ literals in pre. On each complete path of this tree, we can, at synthesis time, determine whether the truth values of literals imply that pre is true. Indeed, such computation reduces to evaluating the truth value of a propositional formula in a given assignment to all variables. In the cases when the literals imply that pre holds, we use the attached substitution $(x_i \mapsto m)$ in true literals to recover the synthesized values of variables x_i. Such decision tree has the depth $n^{2^{O(V)}}$, because it tests the values of all literals in the result of quantifier elimination. For a constant number of variables V, this tree represents a synthesized program whose running time is polynomial in

n. Thus, we have shown that using basic methods of quantifier elimination (without relying on detailed geometric facts about the theory of linear rational arithmetic) we can synthesize for each specification formula a polynomial-time function that maps the parameters to the desired values of output variables.

7.6 Synthesis for Linear Integer Arithmetic

We next describe our main algorithm, which performs synthesis for quantifier-free formulas of Presburger arithmetic (integer linear arithmetic). In this theory variables range over integers. Terms are linear expressions of the form $c_0 + c_1 x_1 + \ldots + c_n x_n$, $n \geq 0$, c_i is an integer constant and x_i is an integer variable. Atoms are built using the relations $\geq$, $=$ and $|$. The atom $c|t$ is interpreted as true iff the integer constant c divides term t. We use $a < b$ as a shorthand for $a \leq b \wedge \neg(a = b)$. We describe a synthesis algorithm that works for conjunction of literals.

Pre-processing. We first apply the following pre-processing steps to eliminate negations and divisibility constraints. We remove negations by transforming a formula into its negation-normal form and translating negative literals into equivalent positive ones: $\neg(t_1 \geq t_2)$ is equivalent to $t_2 \geq t_1 + 1$ and $\neg(t_1 = t_2)$ is equivalent to $(t_1 \geq t_2 + 1) \vee (t_2 \geq t_1 + 1)$. We also normalize equalities into the form $t = 0$ and inequalities into the form $t \geq 0$.

We transform divisibility constraints of a form $c|t$ into equalities by adding a fresh variable q. The value obtained for the fresh variable q is ignored in the final synthesized program:

$$\begin{array}{l} [\![\vec{x}, (c|t) \wedge F]\!] = (\mathsf{pre}, \vec{\Psi}), \\ \quad \text{where} \qquad (\mathsf{pre}, (\vec{\Psi}, \Psi_{n+1})) = [\![(\vec{x}, q),\ t = c\,q \wedge F]\!] \end{array}$$

The negation of divisibility $\neg(c|t)$ can be handled in a similar way by introducing two fresh variables q and r:

$$\begin{array}{l} [\![\vec{x}, \neg(c|t) \wedge F]\!] = (\mathsf{pre}, \vec{\Psi}), \\ \quad \text{where} \\ \qquad F' \ \equiv\ t + r = c\,q \wedge 1 \leq r \leq c - 1 \wedge F \\ \qquad (\mathsf{pre}, (\vec{\Psi}, \Psi_{n+1}, \Psi_{n+2})) \ =\ [\![(\vec{x}, q, r), F']\!] \end{array}$$

In the rest of this section we assume the input formula F to have no negation or divisibility constraints (these constructs can, however, appear in the generated code and precondition).

7.6.1 Solving Equality Constraints for Synthesis

Because equality constraints are suitable for deterministic elimination of output variables, our procedure groups all equalities from a conjunction and solves them first, one by one. Let E be one such equation, so the entire formula is of the form $E \wedge F$. Let $\vec{y}$ be the output variables

$[\![\_\ ,\ \_]\!] : \bigcup_n \big(\mathsf{Vars}^n \times \mathsf{Formulas} \to \mathsf{Formulas} \times \mathsf{Terms}^n\big)$

$[\![(\vec{y},\vec{x}), E \wedge F]\!] = (\mathsf{pre}_Y \wedge \mathsf{pre}, (\vec{\Psi}_{Y0}, \vec{\Psi}_X)),$
where
$(\mathsf{pre}_Y, \vec{\Psi}_Y, \vec{\lambda}) = \mathsf{eqSyn}(\vec{y}, E)$
$F' = \mathsf{simplify}(F[\vec{y} := \vec{\Psi}_Y])$
$(\mathsf{pre}, (\vec{\Psi}_\lambda, \vec{\Psi}_X)) = [\![(\vec{\lambda},\vec{x}), F']\!]$
$\vec{\Psi}_{Y0} = \vec{\Psi}_Y[\vec{\lambda} := \vec{\Psi}_\lambda]$

$\mathsf{eqSyn}: \bigcup_n \mathsf{Vars}^n \times \mathsf{Formulas} \to \mathsf{Formulas} \times \mathsf{Terms}^n \times \mathsf{Vars}^{n-1}$

$\mathsf{eqSyn}(y_1, \Sigma_{i=1}^m \beta_i b_i + \gamma_1 y_1 = 0) \ = \ (\gamma_1 | (\Sigma_{i=1}^m \beta_i b_i),\ -(\Sigma_{i=1}^m \beta_i b_i)/\gamma_1,\ ())$

$\mathsf{eqSyn}(y_1, \ldots, y_n, \Sigma_{i=1}^m \beta_i b_i + \Sigma_{j=1}^n \gamma_j y_j = 0) \ =$
$\mathsf{eqSyn}(y_1, \ldots, y_n,\ t/d + \Sigma_{j=1}^n (\gamma_j/d) y_j = 0),$
where
$t = \Sigma_{i=1}^m \beta_i b_i$
$d = \gcd(\beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_n)$
$d > 1$

$\mathsf{eqSyn}(y_1, \ldots, y_n, \Sigma_{i=1}^m \beta_i b_i + \Sigma_{j=1}^n \gamma_j y_j = 0) \ = (\mathsf{pre}, \vec{\Psi}, \vec{\lambda}),$
where
$(\vec{s}_1, \ldots, \vec{s}_{n-1}) = \mathsf{linearSet}(\gamma_1, \ldots, \gamma_n)$
$(w_1, \ldots, w_n) = \mathsf{particularSol}(\Sigma_{i=1}^m \beta_i b_i, \gamma_1, \ldots, \gamma_n)$
$\mathsf{pre} \equiv \gcd(\gamma_1, \ldots, \gamma_n) | (\Sigma_{i=1}^m \beta_i b_i)$
$\lambda_1, \ldots, \lambda_{n-1}$ – fresh variable names
$\vec{\Psi} = (w_1, \ldots, w_n) + \lambda_1 \vec{s}_1 + \ldots + \lambda_{n-1} \vec{s}_{n-1}$

Figure 7.2: Algorithm for Synthesis Based on Integer Equations

that appear in E.

Given an output variable y_1 and E of the form $cy_1 + t = 0$ for $c \neq 0$, a simple way to solve it would be to impose the precondition $c|t$, use the witness $y_1 = -t/c$ in synthesized code, and substitute $-t/c$ instead of y_1 in the remaining formula. However, to keep the equations within linear integer arithmetic, this would require multiplying the remaining equations and disequations in F by c, potentially increasing the sizes of coefficients substantially.

We instead perform synthesis based on one of the improved algorithms for solving integer equations. This algorithm avoids the multiplication of the remaining constraints by simultaneously replacing all n output variables $\vec{y}$ in E with $n-1$ fresh output variables $\vec{\lambda}$. Using this algorithm we obtain the synthesis procedure in Figure 7.2. An invocation of $\mathsf{eqSyn}(\vec{y}, F)$ is similar to $[\![\vec{y}, F]\!]$ but returns a triple $(\mathsf{pre}, \vec{\Psi}, \vec{\lambda})$, which in addition to the precondition pre and

the witness term tuple $\vec{\Psi}$ also has the fresh variables $\vec{\lambda}$.

The eqSyn Synthesis Algorithm

Consider the application of eqSyn in Figure 7.2 to the equation $\Sigma_{i=1}^{m}\beta_i b_i + \Sigma_{j=1}^{n}\gamma_j y_j = 0$. If there is only one output variable, y_1, we directly eliminate it from the equation. Assume therefore $n > 1$. Let $d = \gcd(\beta_1,\ldots,\beta_m,\gamma_1,\ldots,\gamma_n)$. If $d > 1$ we can divide all coefficients by d, so assume $d = 1$.

Our goal is to derive an alternative definition of the set $K = \{\vec{y} \mid \Sigma_{i=1}^{m}\beta_i b_i + \Sigma_{j=1}^{n}\gamma_j y_j = 0\}$ which will allow a simple and effective computation of elements in K. Note that the set K describes the set of all solutions of a Presburger arithmetic formula.

Recall that a semilinear set (cf. Chapter 2) is a set of all non-negative solutions of a Presburger arithmetic formula. However, beside the fact that we search for all, non-negative and negative solutions, we cannot apply this result because the values of parameter variables are not known until run-time. Instead, we proceed in the following steps, as described in Figure 7.2:

1. obtain a linear set representation of the set

   $$S_H = \{\vec{y} \mid \sum_{j=1}^{n}\gamma_j y_j = 0\}$$

   of solutions for the homogeneous part using the function linearSet (defined in Theorem 7.3) to compute $\vec{s}_1,\ldots,\vec{s}_{n-1}$ such that

   $$S_H = \{\vec{y} \mid \exists\lambda_1,\ldots,\lambda_{n-1} \in \mathbb{Z}.\ \vec{y} = \sum_{i=1}^{n-1}\lambda_i\vec{s}_i\}$$

2. find one particular solution, that is, use the function particularSol (defined in Figure 7.3) to find a vector of terms $\vec{w}$ (containing the parameters b_i) such that $t + \sum_{j=1}^{n}\gamma_j w_j = 0$ for all values of parameters b_i.

3. return as the solution $\vec{w} + \sum_{i=1}^{n-1}\lambda_i\vec{s}_i$

To see that the algorithm is correct, fix the values of parameters and let $\vec{\gamma} = (\gamma_1,\ldots,\gamma_n)$. From linearity we have $t + \vec{\gamma}\cdot(\vec{w} + \sum_j \lambda_j\vec{s}_j) = t - t + 0 = 0$, which means that each $\vec{w} + \sum_j \lambda_j\vec{s}_j$ is a solution. Conversely, if $\vec{y}$ is a solution of the equation then $\vec{\gamma}(\vec{y} - \vec{w}) = 0$, so $\vec{y} - \vec{w} \in S_H$, which means $\vec{y} - \vec{w} = \sum_{i=1}^{n}\lambda_i\vec{s}_i$ for some λ_i. Therefore, the set of all solutions of $t + \sum_{j=1}^{n}\gamma_j w_j = 0$ is the set $\{\vec{w} + \sum_{i=1}^{n-1}\lambda_i\vec{s}_i \mid \lambda_i \in \mathbb{Z}\}$. It remains to define linearSet to find $\vec{s}_i$ and particularSol to find $\vec{w}$.

Computing a Linear Set for a Homogeneous Equation

This section describes our version of the algorithm $\mathsf{linearSet}(\gamma_1,\ldots,\gamma_n)$ that computes the set of solutions of an equation $\Sigma_{i=1}^{n}\gamma_i y_i = 0$. A related algorithm is a component of the Omega test [Pugh(1992)].

Theorem 7.3 *Let $\gamma_1,\ldots,\gamma_n \in \mathbb{Z}$ be integer coefficients. The set of all solutions of $\Sigma_{i=1}^{n}\gamma_i y_i = 0$ is described with:*

$$\mathsf{linearSet}(\gamma_1,\ldots,\gamma_n) = (\vec{s}_1,\ldots,\vec{s}_{n-1})$$

where $\vec{s}_j = (K_{1j},\ldots,K_{nj})$ and the integers K_{ij} are computed as follows:

- *if $i < j$, $K_{ij} = 0$ (the matrix K is lower triangular)*
- $K_{jj} = \dfrac{\gcd((\gamma_k)_{k\geq j+1})}{\gcd((\gamma_k)_{k\geq j})}$
- *for each index j, $1 \leq j \leq n-1$, we compute K_{ij} as follows. Consider the equation*

  $$\gamma_j K_{jj} + \sum_{i=j+1}^{n} \gamma_i u_{ij} = 0$$

  and find any solution. That is, compute

  $$(K_{(j+1)j},\ldots,K_{nj}) = \mathsf{particularSol}(-\gamma_j K_{jj}, \gamma_{j+1},\ldots,\gamma_n)$$

  where particularSol *is given in Figure 7.3.*

Proof. Let $S_H = \{\vec{y} \mid \Sigma_{i=1}^{n}\gamma_i y_i = 0\}$ and let

$$S_L = \{\lambda_1\vec{s}_1 + \ldots + \lambda_n\vec{s}_n \mid \lambda_1,\ldots,\lambda_n \in \mathbb{Z}\} = \left\{ \lambda_1 \begin{pmatrix} K_{11} \\ \vdots \\ K_{n1} \end{pmatrix} + \ldots + \lambda_{n-1} \begin{pmatrix} K_{1(n-1)} \\ \vdots \\ K_{n(n-1)} \end{pmatrix} \middle| \lambda_i \in \mathbb{Z} \right\}$$

We claim $S_H = S_L$.

First we show that each vector $\vec{s}_j$ belongs to S_H. Indeed, by definition of K_{ij} we have $\gamma_j K_{jj} + \sum_{i=j+1}^{n}\gamma_i K_{ij} = 0$. This means precisely that $\vec{s}_j \in S_H$, by definition of $\vec{s}_j$ and S_H. Next, observe that S_H is closed under linear combinations. Because S_L is the set of linear combinations of vectors $\vec{s}_j$, we have $S_L \subseteq S_H$.

To prove that the converse also holds, let $\vec{y} \in S_H$. We will show that the triangular system of equations $\sum_{i=1}^{n-1}\lambda_i\vec{s}_i = \vec{y}$ has some solution $\lambda_1,\ldots,\lambda_{n-1}$. We start by showing that we can find λ_1. Let $G_1 = \gcd((\gamma_k)_{k\geq 1})$. From $\vec{y} \in S_H$ we have $\Sigma_{i=1}^{n}\gamma_i y_i = 0$, that is, $G_1(\Sigma_{i=1}^{n}\beta_i y_i) = 0$

for $\beta_i = \gamma_i / G_1$. This implies $\beta_1 y_1 + \Sigma_{i=2}^{n} \beta_i y_i = 0$ and $\gcd((\beta_k)_{k\geq 1}) = 1$. Let $G_2 = \gcd((\beta_k)_{k\geq 2})$. From $\beta_1 y_1 + \Sigma_{i=2}^{n} \beta_i y_i = 0$ we then obtain $\beta_1 y_1 + G_2(\Sigma_{i=2}^{n} \beta_i^{'} y_i) = 0$ for $\beta_i' = \beta_i / G_2$. Therefore $y_1 = -G_2(\Sigma_{i=2}^{n} \beta_i^{'} y_i)/\beta_1$. Because $\gcd(\beta_1, G_2) = 1$ we have $\beta_1 | \Sigma_{i=2}^{n} \beta_i^{'} y_i$ so we can define the integer $\lambda_1 = -\Sigma_{i=2}^{n} \beta_i^{'} y_i / \beta_1$ and we have $y_1 = \lambda_1 G_2$. Moreover, note that

$$G_2 = \gcd((\beta_k)_{k\geq 2}) = \gcd((\gamma_k)_{k\geq 2})/G_1 = K_{11}$$

Therefore, $y_1 = \lambda_1 K_{11}$, which ensures that the first equation is satisfied.

Consider now a new vector $\vec{z} = \vec{y} - \lambda_1 \vec{s}_1$. Because $\vec{y} \in S_H$ and and $\vec{s}_1 \in S_H$ also $\vec{z} \in S_H$. Moreover, note that the first component of $\vec{z}$ is 0. We repeat the described procedure on $\vec{z}$ and $\vec{s}_2$. This way we derive the value for an integer α_2 and a new vector that has 0 as the first two components.

We continue with the described procedure until we obtain a vector $\vec{u} \in S_H$ that has all components set to 0 except for the last two. From $\vec{u} \in S_H$ we have $\gamma_{n-1} u_{n-1} + \gamma_n u_n = 0$. Letting $\beta_{n-1} = \gamma_{n-1} / \gcd(\gamma_{n-1}, \gamma_n)$ and $\beta_n = \gamma_n / \gcd(\gamma_{n-1}, \gamma_n)$ we conclude that $\beta_{n-1} u_{n-1} + \beta_n u_n = 0$, so u_{n-1}/β_n is an integer and we let $\lambda_{n-1} = u_{n-1}/\beta_n$. By definitions of β_i it follows $\lambda_{n-1} = u_{n-1} \cdot \gcd(\gamma_{n-1}, \gamma_n)/\gamma_n$. Next, observe the special form of the vector $\vec{s}_{n-1}$: $\vec{s}_{n-1}$ has the form $(0, \ldots, 0, \gamma_n / \gcd(\gamma_{n-1}, \gamma_n), -\gamma_{n-1} / \gcd(\gamma_{n-1}, \gamma_n))$. It is then easy to verify that $\vec{u} = \lambda_{n-1} \vec{s}_{n-1}$.

This procedure shows that every element of S_H can be represented as a linear combination of vectors $\vec{s}_j$, which shows $S_H \subseteq S_L$ and concludes the proof.

Finding a Particular Solution of an Equation

We finally describe the particularSol function to find a solution (as a vector of terms) for an equation $t + \Sigma_{i=1}^{n} \gamma_i u_i = 0$. We use the Extended Euclidean algorithm (for a detailed description see for example, [Cormen et al.(2001)Cormen, Leiserson, Rivest, and Stein, Figure 31.1]). Given the integers a_1 and a_2, the Extended Euclidean algorithm finds their greatest common divisor d and two integers w_1 and w_2 such that $a_1 w_1 + a_2 w_2 = d$. Our algorithm generalizes the Extended Euclidean Algorithm to arbitrary number of variables and uses it to find a solution of an equation with parameters. We chose the algorithm presented here because of its simplicity. Other algorithms for finding a solution of an equation $t + \Sigma_{i=1}^{n} \gamma_i u_i = 0$ can be found in [Banerjee(1988), Ford and Havas(1996)]. They also run in polynomial time. [Banerjee(1988)] additionally allows bounded inequality constraints, whereas [Ford and Havas(1996)] guarantees that the returned numbers are no larger than the largest of the input coefficients divided by 2.

The equation $t + \Sigma_{i=1}^{n} \gamma_i u_i = 0$ has a solution iff $\gcd((\gamma_k)_{k\geq 1}) | t$, and the result of particularSol is guaranteed to be correct under this condition. Our synthesis procedure ensures that when the results of this algorithm are used, the condition $\gcd((\gamma_k)_{k\geq 1}) | t$ is satisfied.

We start with the base case where there are only two variables, $t + \gamma_1 u_1 + \gamma_2 u_2 = 0$. By the Extended Euclidean Algorithm let v_1 and v_2 be integers such that $\gamma_1 v_1 + \gamma_2 v_2 = \gcd(\gamma_1, \gamma_2)$.

With d we denote $d = \gcd(\gamma_1, \gamma_2)$ and let $r = t/d$. Then one solution is the pair of terms $(-v_1 r, -v_2 r)$:

$$\begin{aligned}&\mathsf{particularSol2}(t, \gamma_1, \gamma_2) = (-v_1 r, -v_2 r),\\ &\quad \text{where}\\ &\qquad (d, v_1, v_2) = \mathsf{ExtendedEuclid}(\gamma_1, \gamma_2)\\ &\qquad r = t/d\end{aligned}$$

If there are more than two variables, we observe that $\Sigma_{i=2}^{n} \gamma_i u_i$ is a multiple of $\gcd((\gamma_k)_{k\geq 2})$. We introduce the new variable u' and find a solution of the equation $t + \gamma_1 u_1 + \gcd((\gamma_k)_{k\geq 2}) \cdot u' = 0$ as described above. This way we obtain terms (w_1, w') for (u_1, w'). To derive values of $u_2, \ldots, u_n$ we solve the equation $\Sigma_{i=2}^{n} \gamma_i u_i = \gcd((\gamma_k)_{k\geq 2}) \cdot w'$. Given that the initial equation was assumed to have a solution, the new equation can also be showed to have a solution. Moreover, it has one variable less, so we can solve it recursively:

$$\begin{aligned}&\mathsf{particularSol}(t, \gamma_1, \ldots, \gamma_n) = (w_1, \ldots, w_n),\\ &\quad \text{where}\\ &\qquad (w_1, w') = \mathsf{particularSol2}(t, \gamma_1, \gcd((\gamma_k)_{k\geq 2}))\\ &\qquad (w_2, \ldots, w_n) = \mathsf{particularSol}(-\gcd((\gamma_k)_{k\geq 2})\, w', \gamma_2, \ldots, \gamma_n)\end{aligned}$$

Figure 7.3: Algorithm for Computing one Solution of the Equation

Example. We demonstrate the process of eliminating equations on an example. Consider the following synthesis problem

$$[\![(x, y, z), 2a - b + 3x + 4y + 8z = 0 \wedge 5x + 4z \leq 2y - b]\!]$$

To eliminate an equation from the formula and to reduce a number of output variables, we first invoke $\mathsf{eqSyn}((x, y, z), 2a - b + 3x + 4y + 8z = 0)$, which works in two phases. In the first phase, we compute the linear set describing a set of solutions of the homogeneous equality $3x + 4y + 8z = 0$. Applying Theorem 7.3, the resulting set S_L is:

$$S_L = \left\{ \lambda_1 \begin{pmatrix} 4 \\ -3 \\ 0 \end{pmatrix} + \lambda_2 \begin{pmatrix} 0 \\ 2 \\ -1 \end{pmatrix} \middle| \lambda_1, \lambda_2 \in \mathbb{Z} \right\}$$

The second phase computes a witness vector $\vec{w}$ and a precondition formula. Applying the procedure described in Section 7.6.1 results in the vector $\vec{w} = (2a - b, b - 2a, 0)$ and the formula $1|2a - b$. Finally, we compute the output of eqSyn applied to $2a - b + 3x + 4y + 8z = 0$: it is a triple consisting of

1. a precondition $1|2a - b$

2. a list of terms denoting witnesses for (x, y, z):

$$\Psi_1 = 2a - b + 4\lambda_1$$
$$\Psi_2 = b - 2a - 3\lambda_1 + 2\lambda_2$$
$$\Psi_3 = -\lambda_2$$

3. a list of fresh variables (λ_1, λ_2).

We then replace each occurrence of x, y and z by the corresponding terms in the rest of the formula. This results in a new formula $7a - 3b + 13\lambda_1 \leq 4\lambda_2$, that has the same input variables, but the output variables are now λ_1 and λ_2. To find a solution for the initial problem, we let

$$(\mathsf{pre}_X, (\Phi_1, \Phi_2)) = [\![(\lambda_1, \lambda_2), 7a - 3b + 13\lambda_1 \leq 4\lambda_2]\!]$$

Since $1|2a - b$ is a valid formula, we do not add it to the final precondition. Therefore, the final result has the form

$$(\mathsf{pre}_X, (2a - b + 4\Phi_1, b - 2a - 3\Phi_1 + 2\Phi_2, -\Phi_2))$$

7.6.2 Solving Inequality Constraints for Synthesis

In the following, we assume that all equalities are already processed and that a formula is a conjunction of inequalities. Dealing with inequalities in the integer case is similar to the case of rational arithmetic: we process variables one by one and proceed further with the resulting formula.

Let x be an output variable that we are processing. Every conjunct can be rewritten in one of the two following forms:

$$\begin{array}{lll} \text{[Lower Bound]} & A_i \leq & \alpha_i x \\ \text{[Upper Bound]} & & \beta_j x \leq B_j \end{array}$$

As for rational arithmetic, x should be a value which is greater than all lower bounds and smaller than all upper bounds. However, this time we also need to enforce that x must be an integer. Let $a = \max_i \lceil A_i/\alpha_i \rceil$ and $b = \min_j \lfloor B_j/\beta_j \rfloor$. If b is defined (i.e. at least one upper bound exists), we use b as the witness for x, otherwise we use a.

The corresponding formula with which we proceed is a conjunction stating that each lower bound is smaller than every upper bound:

$$\bigwedge_{i,j} \lceil A_i/\alpha_i \rceil \leq \lfloor B_j/\beta_j \rfloor \tag{7.2}$$

Because of the division, floor, and ceiling operators, the above formula is not in integer linear arithmetic. However, in the absence of output variables, it can be evaluated using standard

programming language constructs. On the other hand, if the terms A_i and B_j contain output variables, we convert the formula into an equivalent linear integer arithmetic formula as follows.

With lcm we denote the least common multiple. Let $L = \text{lcm}_{i,j}(\alpha_i, \beta_j)$. We introduce new integer linear arithmetic terms $A'_i = \frac{L}{\alpha_i} A_i$ and $B'_j = \frac{L}{\beta_j} B_j$. Using these terms we derive an equivalent integer linear arithmetic formula:

$$\lceil A_i/\alpha_i \rceil \leq \lfloor B_j/\beta_j \rfloor \Leftrightarrow \lceil A'_i/L \rceil \leq \left\lfloor B'_j/L \right\rfloor \Leftrightarrow \frac{A'_i}{L} \leq \frac{B'_j - B'_j \mod L}{L}$$
$$\Leftrightarrow B'_j \mod L \leq B'_j - A'_i \Leftrightarrow B'_j = L \cdot l_j + k_j \wedge k_j \leq B'_j - A'_i$$

Formula (7.2) is then equivalent to

$$\bigwedge_j (B'_j = L \cdot l_j + k_j \wedge \bigwedge_i (k_j \leq B'_j - A'_i))$$

Although this formula belongs to linear integer arithmetic, we still cannot simply apply the synthesizer on that formula. Let $\{1, \ldots, J\}$ be a range of j indices. The newly derived formula contains J equalities and $2 \cdot J$ new variables. The process of eliminating equalities as described in Section 7.6.1 will at the end result in a new formula which contains J new output variables and this way we cannot assure termination. Therefore, this is not a suitable approach.

However, we observe that the value of k_j is always bounded: $k_j \in \{0, \ldots, L-1\}$. Thus, if the value of k_j were known, we would have a formula with only J new variables and J additional equations. The equation elimination procedure described before would then result in a formula that has one variable less than the original starting formula, and that would guarantee termination of the approach.

Since the value of each k_j variable is always bounded, there are finitely many $(J \cdot L)$ possible instantiations of k_j variables. Therefore, we need to check for each instantiation of all k_j variables whether it leads to a solution. As soon as a solution is found, the generated code stops and proceeds with the obtained values of output variables. If no solution is found, we raise an exception, because the original formula has no integer solution. This leads to a translation schema that contains $J \cdot L$ conditional expression. In our implementation we generate this code as a loop with constant bounds.

We finish the description of the synthesizer with an example that illustrates the above algorithm.

Example. Consider the formula $2y - b \leq 3x + a \wedge 2x - a \leq 4y + b$ where x and y are output variables and a and b are input variables. If the resulting formula $\lceil 2y - b - a/3 \rceil \leq \lfloor 4y + a + b/2 \rfloor$ has a solution, then the synthesizer emits the value of x to be $\lfloor 4y + a + b/2 \rfloor$. This newly derived formula has only one output variable y, but it is not an integer linear arithmetic formula.

It is converted to an equivalent integer linear arithmetic formula $(4y+a+b)\cdot 3 = 6l+k \wedge k \leq 8y+5a+5b$, which has three variables: y, k and l. The value of k is bounded: $0 \leq k \leq 5$, so we treat it as a parameter. We start with elimination of the equality: it results in the precondition $6|3a+3b-k$, the list of terms $l = (3a+3b-k)/6+2\alpha, y = \alpha$ and a new variable: α. Using this, the inequality becomes $k-5a-5b \leq 8\alpha$. Because α is the only output variable, we can compute it as $\lceil (k-5a-5b)/8 \rceil$. The synthesizer finally outputs the following code, which computes values of the initial output variables x and y:

```
val kFound = false
for k = 0 to 5 do {
  val v1 = 3 * a + 3 * b - k
  if (v1 mod 6 == 0){
    val alpha = ((k - 5 * a - 5 * b)/8).ceiling
    val l = (v1 / 6) + 2 * alpha
    val y = alpha
    val kFound = true
    break } }
if (kFound)
  val x = ((4 * y + a + b)/2).floor
else
  throw new Exception("No solution exists")
```

The precondition formula is $\exists k.\ 0 \leq k \leq 5 \wedge\ 6|3a+3b-k$, which our synthesizer emits as a loop that checks $6|3a+3b-k$ for $k \in \{0,\ldots,5\}$ and throws an exception if the precondition is false.

7.6.3 Disjunctions in Presburger Arithmetic

We can again lift synthesis for conjunctions to synthesis for arbitrary propositional combinations by applying the method of Section 7.4.4. We also obtain a complexity that is one exponential higher than the complexity of synthesis from the previous section. Approaches that avoid disjunctive normal form can be used in this case as well [Nipkow(2008), Ferrante and Rackoff(1979), Weispfenning(1997)].

7.6.4 Optimizations used in the Implementation

In this section, we describe some optimizations and heuristics that we use in our implementation. Using some of them, we obtained a speedup of several orders of magnitude.

Merging inequalities. Whenever two inequalities $t_1 \leq t_2$ and $t_2 \leq t_1$ appear in a conjunction, we substitute them with an equality $t_1 = t_2$. This makes the process of variable elimination

more efficient.

Heuristic for choosing the right equality for elimination. When there are several equalities in a formula, we choose to eliminate an equality for which the least common multiple of all the coefficients is the smallest. We observed that this reduces the number of integers to iterate over.

Some optimizations on modulo operations. When processing inequalities, as described in Section 7.6.2, as soon as we introduce the modulo operator, we face a potentially longer processing time. This is because finding the suitable value of the remainder in equation $B'_j \bmod L \leq B'_j - A'_i$ requires invoking a loop. While searching for a witness, we might need to test all possible L values. Therefore, we try not to introduce the modulo operator in the first place. This is possible in several cases. One of them is when either $\alpha_i = 1$ or $b_j = 1$. In that case, if for example $\alpha_i = 1$, an equivalent integer arithmetic formula is easily derived:

$$\lceil A_i/\alpha_i \rceil \leq \lfloor B_j/\beta_j \rfloor \Leftrightarrow A_i \leq \lfloor B_j/\beta_j \rfloor \Leftrightarrow \beta_j A_i \leq B_j$$

Another example where we do not introduce the modulo operator is when $A'_i - B'_j$ evaluates to a number N such that $N > L$. In that case, it is clear that $B'_j \bmod L \leq B'_j - A'_i$ is a valid formula and thus the returned formula is true.

Finally, we describe an optimization that leads to a reduction in the number of loop executions. This is possible when there exists an integer N such that $B'_j = N \cdot T_j$ and $L = N \cdot L_1$. (Unless $L = \beta_j$, this is almost always the case.) In the case where N exists, then k_j also has to be a multiple of N. Putting this together, an equivalent formula of $B'_j \bmod L \leq B'_j - A'_i$ is the formula $T_j \bmod L_1 = k_j \wedge N \cdot k_j \leq B'_j - A'_i$. This reduces the number of loop iterations by at least a factor of N.

7.7 Synthesis Algorithm for Parametrized Presburger Arithmetic

In addition to handling the case when the specification formula is an integer linear arithmetic formula of both parameters and output variables, we have generalized our synthesizer to the case when the coefficients of the output variables are not only integers, but can be any arithmetic expression over the input variables. This extension allows us to write e.g. the offset decomposition program from Section 7.2 with statically unknown dimensions dimX, dimY, dimZ. As a slightly simpler example, consider the following invocation:

```
val (valueX, valueY) = choose((x: Int, y: Int) ⇒
  (offset == x + dim * y && 0 ≤ x && x < dim ))
```

Here offset and dim are input variables, whereas x and y are output variables. Note that dim$*y$

is not a linear term. However, at run-time we know the exact value of dim, so the term will become linear. Our synthesizer can handle such cases as well through a generalization of the algorithm in Section 7.6.

Given the problem above, we first eliminate the equality $\mathsf{offset} = x + \mathsf{dim} * y$ and we obtain the new problem consisting of two inequalities: $\mathsf{dim} * t \leq \mathsf{offset} \wedge \mathsf{offset} - \mathsf{dim} + 1 \leq \mathsf{dim} * t$. The variable t is a freshly introduced integer variable and it is also the only output variable. At this point, the synthesizer needs to divide a term by the variable dim. In general it thus needs to generate code that distinguishes the cases when dim is positive, negative, or zero. In this particular example, due to the constraint $0 \leq x < \mathsf{dim}$, only one case applies. The synthesizer returns the following precondition:

$$\mathsf{pre} \equiv \lceil(\mathsf{offset} - \mathsf{dim} + 1)/\mathsf{dim}\rceil \leq \lfloor \mathsf{offset}/\mathsf{dim}\rfloor$$

It can easily be verified that this is a valid formula for all positive values of dim. The synthesizer also returns the code that computes the values for x and y:

```
val t = (offset /dim).floor
val valueY = t
val valueX = offset - dim * t
```

Our general algorithm for handling parametrized Presburger arithmetic follows the algorithm described in Section 7.6. The main difference is that instead of manipulating known integer coefficients, it manipulates arbitrary arithmetic expressions as coefficients. It therefore needs to postpone to run-time certain decisions that involve coefficients. The key observation that makes this algorithm possible is that many compile-time decisions depend not on the particular values of the coefficients, but only on their sign (positive, negative, or zero). In the presence of a coefficient that depends on a parameter, the synthesizer therefore generates code with multiple branches that cover the different cases of the sign.

As an illustration, consider using synthesis to compute, when it exists, the positive integer ratio x between two integers a and b:

```
val a: Int = ...
val b: Int = ...
val x = choose((x: Int) ⇒ a * x == b && x ≥ 0)
```

In this example, the synthesizer needs to distinguish between the cases where a, which is used as a coefficient, is zero, negative and positive: when a is zero, it computes as a precondition

$$\mathsf{pre}_0 \equiv b = 0$$

when a is negative, the precondition is

$$\mathsf{pre}_\ominus \equiv -b \geq 0 \wedge a|b$$

and similarly, when a is positive

$$\text{pre}_{\oplus} \equiv b \geq 0 \wedge a|b$$

In fact, when the positive and negative cases differ only by a sign, our synthesized factors this out by using the expression $\frac{a}{|a|}$ for the sign of a (note that since the case where a is zero is treated before, there is no risk of a division by zero). The generated code for computing x is:

```
if (a == 0 && b == 0){
  0
} else if (−(a/Math.abs(a)) * b ≥ 0 && b % a == 0) {
  b / a
} else {
  throw new Exception("No solution exists")
}
```

(Note that when both a and b are zero, any value for x is valid, 0 is just the option picked by the synthesizer.)

The coefficients of the invocation of the Extended Euclidean algorithm generally also become known only at run-time, so the generated code invokes this algorithm as a library function. The situation is analogous for the gcd function. The following example illustrates this situation:

```
choose((x: Int) ⇒ 6*x + a*y = b
```

On this example, our synthesizer produces the following code:

```
if (b % gcd(6,a) == 0){
  val t1 = gcd(6,a)
  val t2 = −b / t1
  val (t3, t4) = coeffs(1, 6/t1, a/t1)
  (t2 * t3, t2 * t4)
} else {
  throw new Exception("No solution exists")
}
```

In this code, `gcd` computes the greatest common divisor, and `(a,b) = coeffs(1,c,d)` computes `a` and `b` such that `a*c + b*d + 1 == 0` holds. Note that there are no tests on the signs of a and b, because the precondition and the code are the same in all cases (we define `gcd(x,0)` to be `x`).

Finally, note that the running time of the programs in this case is not uniform with respect to the values of all parameters. In particular, the upper bounds of the generated for loops in Section 7.6.2 can now be a function of parameters. Nevertheless, for each value of the

$$
\begin{array}{lcl}
F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \\
A & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid T_1 = T_2 \mid T_1 < T_2 \mid (K|T) \\
B & ::= & x \mid \emptyset \mid \mathscr{U} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c \\
T & ::= & k \mid K \mid T_1 + T_2 \mid K \cdot T \mid \ \mid B \mid \\
K & ::= & \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
\end{array}
$$

Figure 7.4: A Logic of Sets and Size Constraints (BAPA)

parameter, the generated code terminates.

7.8 Synthesis for Sets with Size Constraints

In this section, we recall the definiton a logic of sets with cardinality constraints (introduced in Chapter 6) and describe a synthesis procedure for it. The logic we consider is BAPA (Boolean Algebra with Presburger Arithmetic). It supports the standard operators union, intersection, complement, subset, and equality. In addition, it supports the size operator on sets, as well as integer linear arithmetic constraints over these sizes. Its syntax is shown in Figure 7.4. Decision procedures for BAPA were considered in a number of scenarios [Feferman and Vaught(1959), Zarba(2004), Zarba(2005), Kuncak et al.(2006)Kuncak, Nguyen, and Rinard, Kuncak and Rinard(2007)]. As in the previous sections, we consider the problem (7.1)

$$\vec{r} = \mathsf{choose}(\vec{x} \Rightarrow F(\vec{x}, \vec{a}))$$

where the components of vectors $\vec{a}, \vec{x}, \vec{r}$ are either set or integer variables and F is a BAPA formula.

Figure 7.5 describes our BAPA synthesis procedure that returns a precondition predicate $\mathsf{pre}(\vec{a})$ and a solved form $\vec{\Psi}$. The procedure is based on the quantifier elimination algorithm presented in [Kuncak et al.(2006)Kuncak, Nguyen, and Rinard], which reduces a BAPA formula to an equisatisfiable integer linear arithmetic formula. The algorithm eliminates set variables in two phases. In the first phase all set expressions are rewritten as unions of disjoint Venn regions. The second phase introduces a fresh integer variable for the cardinality of each Venn region. It thus reduces the entire formula to an integer linear arithmetic formula F_1. The input variables in F_1 are the integer input variables from the original formula, as well as fresh integer variables denoting cardinalities of Venn regions of the input set variables. Note that all values of those input variables are known from the program. The output variables are the original integer output variables and freshly introduced integer variables denoting cardinalities of Venn regions that are contained in the output set variables.

We can therefore build a synthesizer for BAPA on top of the synthesizer for integer linear

INPUT: a formula $F(\vec{X}, \vec{k})$ in the logic defined in Figure 7.4 with input variables $X_1, \ldots, X_n, k_1, \ldots, k_m$ and output variables $Y_1, \ldots, Y_s, l_1, \ldots, l_t$, where X_i and Y_j are set variables, k_i and l_j are integer variables

OUTPUT: code that computes values for the output variables from the input variables

1. Apply the first steps towards a Presburger arithmetic formula:
   (a) Replace each atom $S_1 = S_2$ with $S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$
   (b) Replace each atom $S_1 \subseteq S_2$ with $|S_1 \cap S_2^c| = 0$
2. Introduce the Venn regions of sets X_i's and Y_j's: let u be a binary word of the length $n+m$. The set variable R_u represents a Venn region where each '1' stands for a set and '0' stands for a complement. To illustrate, if $n = 2$, $m = 1$ and $u = 001$, then $R_{001} = X_1^c \cap X_2^c \cap Y_1$. Rewrite each set expression as a disjoint union of corresponding Venn regions.
3. Create a Presburger arithmetic formula: an integer variable h_u denotes the cardinality of the Venn region R_u. Use the fact that $|S_1 \cup S_2| = |S_1| + |S_2|$ iff S_1 and S_2 are disjoint to rewrite the whole formula as the Presburger arithmetic formula. We denote the resulting formula by $F_1(\vec{h_u}, \vec{l})$.
4. Create a Presburger arithmetic formula that corresponds to quantifier elimination: let v be a binary word of length n. A set variable P_v denotes a Venn region of input set variables, which means that $|P_v|$ is a known value. Create a formula that expresses each $|P_v|$ as a sum of corresponding h_u's. Define the formula $F_2(\vec{h_u}, |\vec{P_v}|)$ as the conjunction of all those formulas.
5. Create code that computes values of output vectors. First invoke the linear arithmetic synthesizer described in Section 7.6 to generate the code corresponding to:
   val $(\vec{h_{un}}, \vec{l_n}) = \mathsf{choose}((\vec{h_u}, \vec{l}) \Rightarrow F_1(\vec{h_u}, \vec{l}) \wedge F_2(\vec{h_u}, |\vec{P_v}|))$

   Invoking the synthesizer returns code that computes expressions for the integer output variables $\vec{l_n}$ and for the variables $\vec{h_{un}}$. For each set output variable Y_i, do the following: let S_i be a set containing already known or defined set variables, let T_j be a Venn region of $S_i \cup Y_i$ that is contained in Y_i. Each T_j region is contained in the bigger Venn region U_j which is a Venn region of sets in Y_i. For each T_j do: take all R_u that belong to T_j and let d_j be the sum of all corresponding h_{un}. Based on the value of d_j, output the following code:

   - if $T_j \subseteq \cap_{S \in S_i} S^c$ and $d_j > 0$, output the assignment $K_j = \texttt{fresh}(d_j)$
   - if $d_j = 0$, output the assignment $K_j = \emptyset$
   - if $d_j = |U_j|$, output the assignment $K_j = U_j$
   - otherwise output the assignment $K_j = \texttt{take}(d_j, U_j)$

   Finally, construct Y_i as a union of all K_j sets: $Y_i = \cup_j K_j$

Figure 7.5: Algorithm for synthesizing a function Ψ such that $F[\vec{x} := \Psi(\vec{a})]$ holds, where F has the syntax of Figure 7.4

arithmetic described in Section 7.6. The integer arithmetic synthesizer outputs the precondition predicate pre and emits the code for computing values of the new output variables. The generated code can use the returned integer values to reconstruct a model for the original formula. Notice that the precondition predicate pre will be a Presburger arithmetic formula with the terms built using the original integer input variables and the cardinalities of Venn regions of the original input set variables. As an example, if i is an integer input variable and a and b are set input variables then the precondition predicate might be the following formula $\mathsf{pre}(i, a, b) = |a \cap b| < i \wedge |a| \leq |b|$.

In the last step of the BAPA synthesis algorithm, when outputting code, we use functions `fresh` and `take`. The function `take` takes as arguments an integer k and a set S, and returns a subset of S of size k. The function `fresh`(k) is invoked when k fresh elements need to be generated. These functions are used only in the code that computes output values of set variables (the linear integer arithmetic synthesizer already produces the code to compute the values of integer output variables). The set-valued output variables are computed one by one. Given an output set variable Y_i, the code that effectively computes the value of Y_i is emitted in several steps. With S_i we denote a set containing set variables occurring in the original formula whose values are already known. Initially, S_i contains only the input set variables. Our goal is to describe the construction of Y_i in terms of sets that are already in S_i. We start by computing the Venn regions for Y_i and all the sets in S_i in order to define Y_i as a union of those Venn regions. Therefore we are interested only in those Venn regions that are subset of Y_i. Let T_j be one such a Venn region. It can be represented as $T_j = Y_i \cap U_j$ where U_j has a form $U_j = \cap_{S \in S_i} S^{(c)}$ and $S^{(c)}$ denotes either S or S^c. On the other hand, T_j can also be represented as a disjoint union of the original R_u Venn regions. Those R_u are Venn regions that were constructed in the beginning of the algorithm for all input and output set variables. As the linear integer arithmetic synthesizer outputs the code that computes the values h_u, where $h_u = |R_u|$, we can effectively compute the size of each T_j. If $T_j = R_{u_1} \cup \ldots \cup R_{u_k}$, then the size of T_j is $|T_j| = d_j = \sum_{l=1}^{k} h_{u_l}$. Note that d_j is easily computed from the linear integer arithmetic synthesizer and based on the value of d_j we define a set K_j as $K_j = \texttt{take}(d_j, U_j)$. Finally, we emit the code that defines Y_i as a finite union of K_j's: $Y_i = \cup_j K_j$.

Based of the values of d_j, we can introduce further simplifications. If $d_j = 0$, none of elements of U_j contributes to Y_i and thus $K_j = \emptyset$. On the other hand, if $d_j = |U_j|$, applying a simple rule $S = \texttt{take}(|S|, S)$ results in $K_j = U_j$. A special case is when $U_j = \cap_{S \in S_i} S^c$. If in this case it also holds that $d_j > 0$, we need to take d_j elements that are not contained in any of the already known sets, i.e. we need to generate fresh d_j elements. For this purpose we invoke the command `fresh`.

Partitioning a Set. We illustrate the BAPA synthesis algorithm through an example. Consider the following invocation of the `choose` function that generalizes the example in Section 7.2.

```
val (setA, setB) = choose((a: Set[O], b: Set[O]) ⇒
```

```
  (−maxDiff ≤ a.size − b.size  && a.size − b.size  ≤ maxDiff
      && a union b == bigSet && a intersect b == empty
))
```

This example combines integer and set variables. Given a set `bigSet`, the goal is to divide it into two partition. The previously defined integer variable `maxDiff` specifies the maximum amount by which the sizes of the two partitions may differ. We apply the algorithm from Figure 7.5 step-by-step to illustrate how it works. After completing Step 3, we obtain the formula

$$F_1(\vec{h}_u) \equiv h_{100} = h_{110} = h_{010} = h_{001} = h_{111} = 0 \\ \wedge \text{-maxDiff} \leq h_{101} - h_{011} \wedge h_{101} - h_{011} \leq \text{maxDiff}$$

We simplify the formula obtained in Step 4 by applying the constraints from Step 3 and obtain the formula

$$F_2(\vec{h}_u) \equiv |\text{bigSet}| = h_{101} + h_{011} \wedge |\text{bigSet}^c| = h_{000}$$

Next we call the linear arithmetic synthesizer on the formula $F_1(\vec{h}_u) \wedge F_2(\vec{h}_u)$. The only two variables whose values we need to find are h_{101} and h_{011}. The synthesizer first eliminates the equation $|\text{bigSet}| = h_{101} + h_{011}$: a fresh new integer variable k is introduced such that $h_{101} = k$ and $h_{011} = |\text{bigSet}| - k$. This way there is only one output variable: k. Variable k has to be a solution of the following two inequalities: $|\text{bigSet}| - \text{maxDiff} \leq 2k \wedge 2k \leq |\text{bigSet}| + \text{maxDiff}$. This results in the precondition

$$\text{pre} \equiv \left\lceil \frac{|\text{bigSet}| - \text{maxDiff}}{2} \right\rceil \leq \left\lfloor \frac{|\text{bigSet}| + \text{maxDiff}}{2} \right\rfloor$$

Note that pre is defined entirely in terms of the input variables and can be easily checked at run-time. The synthesizer outputs the following code, which computes values for the output variables:

```
val k = ((bigSet.size  + maxDiff)/2).floor
val h101 = k
val h011 = bigSet.size  − k
val setA = take(h101, bigSet)
val setB = take(h011, bigSet  −− setA)
```

In the code above, '-' denotes the set difference operator. The synthesized code first computes the size k of one of the partitions, as approximately one half of the size of `bigSet`. It then selects k elements from `bigSet` to form `setA`, and selects `bigSet.size` $- k$ of the remaining elements for `setB`.

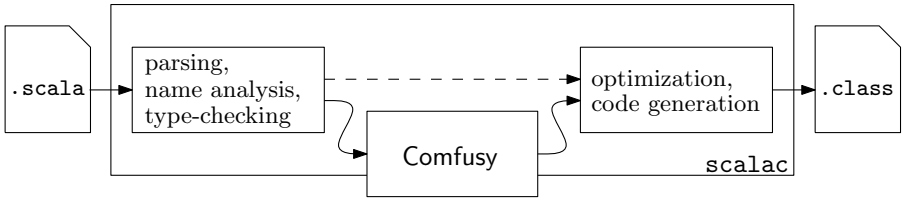

Figure 7.6: Interaction of Comfusy with `scalac`, the Scala compiler. Comfusy takes as an input the abstract syntax tree of a Scala program and rewrites calls to `choose` to syntax trees representing the synthesized function.

7.9 Implementation and Experience

Comfusy **tool.** We have implemented our synthesis procedures as a Scala compiler extension, which we call Comfusy. We chose Scala because it supports higher-order functions that make the concept of a `choose` function natural, and extensible pattern matching in the form of extractors [Emir et al.(2007)Emir, Odersky, and Williams]. Moreover, the compiler supports plugins that work as additional compilation phases, so our extension is seamlessly integrated into compilation process (see Figure 7.6). We used an off-the-shelf decision procedure [de Moura and Bjørner(2008a)] to handle the compile-time checks (we could, in principle, also use our synthesis procedure for compile-time checks because synthesis subsumes satisfiability checking).

Our plugin supports the synthesis of integer values through the `choose` function constrained by linear arithmetic predicates (including predicates in parametrized linear arithmetic), as well as the synthesis of set values constrained by predicates of the logic described in Section 7.8. Additionally, it can synthesize code for pattern-matching expressions on integers such as the ones presented in Section 7.2.

Compilation times. Figure 7.7 shows the compile times for a set of benchmarks, with and without our plugin. Without the plugin, the code is of no use (the `choose` function, when not rewritten, just throws an exception), but the difference between the timings indicates how much time is spent generating the synthesized code. We also measure how much time is used for the compile-time checks for satisfiability and uniqueness. The examples *SecondsToTime*, *FastExponentiation*, *SplitBalanced* and *Coordinates* were presented in Section 7.2. *ScaleWeights* computes solutions to a puzzle, *PrimeHeuristic* contains a long pattern-matching expression where every pattern is checked for reachability, and *SetConstraints* is a variant of *SplitBalanced*. There is no measurement for *Coordinates* with compile-time checks, because the formulas to check are in an undecidable fragment, as the original formula is in parametrized linear arithmetic. We also measured the times with all benchmarks placed in a single file, as an attempt to balance out the time taken by the Scala compiler to start up.

Our numbers show that the additional time required for the code synthesis is minimal. More-

	scalac	w/ plugin	w/ checks
SecondsToTime	3.05	3.2	3.25
FastExponentiation	3.1	3.15	3.25
ScaleWeights	3.1	3.4	3.5
PrimeHeuristic	3.1	3.1	3.1
SetConstraints	3.3	3.5	3.5
SplitBalanced	3.3	3.9	4.0
Coordinates	3.2	4.2	−−
All	5.75	6.35	6.75

Figure 7.7: Measurement of compile times: without applying synthesis (`scalac`), with synthesis but with no call to Z3 (`w/ plugin`) and with both synthesis and compile-time checks activated (`w/ checks`). All times are in seconds.

over, note that the code we tested contained almost exclusively calls to the synthesizer. The increase in compilation time in practice would thus be lower for code that mixes standard Scala with selected `choose` construct invocations.

Execution times of generated code. In our experience, the execution time of the synthesized code is similar to equivalent hand-written code. Our experience so far was restricted to small examples, not because of performance problems, but rather because this is the intended way of using the tool: to synthesize code blocks as opposed to entire procedures or algorithms.

Code size. An older version of Comfusy generated if-then-else statements that correspond to large disjunctions that appear in quantifier elimination algorithms. In certain cases, this led to formulas of large size. We have improved this by generating code that executes about as fast but uses a "for" loop instead of disjunctions. This eliminated the problems with code size, and enabled synthesis for parametric coefficients, discussed above.

7.10 Further Related Work

Early work on synthesis [Manna and Waldinger(1971), Manna and Waldinger(1980)] focused on synthesis using expressive and undecidable logics, such as first-order logic and logic containing the induction principle. Consequently, while it can synthesize interesting programs containing recursion, it cannot provide completeness and termination guarantees as synthesis based on decision procedures.

Recent work on synthesis [Srivastava et al.(2010)Srivastava, Gulwani, and Foster] resolves some of these difficulties by decoupling the problem of inferring program control structure and the problem of synthesizing the computation along the control edges. Furthermore,

the work leverages verification techniques that use both approximation and lattice theoretic search along with decision procedures. As such, it is more ambitious and aims to synthesize entire algorithms. By nature, it cannot be both terminating and complete over the space of all programs that satisfy an input/output specification (thus the approach of specifying program resource bounds). In contrast, we focus on synthesis of program fragments with very specific control structure dictated by the nature of the decidable logical fragment.

Our work further differs from the past ones in 1) using decision procedures to guarantee the computation of synthesized functions whenever a synthesized function exists, 2) bounds on the running times of the synthesis algorithm and the synthesized code size and running time, and 3) deployment of synthesis in well-delimited pieces of code of a general-purpose programming language.

Program sketching has demonstrated the practicality of program synthesis by focusing its use on particular domains [Solar-Lezama et al.(2006)Solar-Lezama, Tancau, Bodík, Seshia, and Saraswat, Solar-Lezama et al.(2007)Solar-Lezama, Arnold, Tancau, Bodík, Saraswat, and Seshia, Solar-Lezama et al.(2008)Solar-Lezama, Jones, and Bodík]. The algorithms employed in sketching are typically focused on appropriately guided search over the syntax tree of the synthesized program. Search techniques have also been applied to automatically derived concurrent garbage collection algorithms [Vechev et al.(2007)Vechev, Yahav, Bacon, and Rinetzky]. In contrast, our synthesis uses the mathematical structure of a decidable theory to explore the space of all functions that satisfy the specification. This enables our approach to achieve completeness without putting any a priori bound on the syntax tree size. Indeed, some of the algorithms we describe can generate fairly large yet efficient programs. We expect that our techniques could be fruitfully integrated into search-based frameworks.

Synthesis of reactive systems generates programs that run forever and interact with the environment. However, known complete algorithms for reactive synthesis work with finite-state systems [Pnueli and Rosner(1989)] or timed systems [Asarin et al.(1995)Asarin, Maler, and Pnueli]. Such techniques have applications to control the behavior of hardware and embedded systems or concurrent programs [Vechev et al.(2009)Vechev, Yahav, and Yorsh]. These techniques usually take specifications in temporal logic [Piterman et al.(2006)Piterman, Pnueli, and Sa'ar] and have resulted in tools that can synthesize useful hardware components [Jobstmann et al.(2007)Jobstmann, Galler, Weiglhofer, and Bloem, Jobstmann and Bloem(2006)]. Our work examines non-reactive programs, but supports infinite data without any approximation, and incorporates the algorithms into a compiler for a general-purpose programming language.

Computing optimal bounds on the size and running time of the synthesized code for Presburger Arithmetic is beyond the scope of the current state of our research. Relevant results in the area of decision procedures are automata-based decision procedures [Boigelot et al.(2005)Boigelot, Jodogne, and Wolper, Klaedtke(2003)], the bounds on quantifier elimination [Weispfenning(1997)] and results on integer programming in fixed dimensions [Eisenbrand and Shmonin(2008)].

Automata-based decision procedures, such as those implemented in the MONA tool [Klarlund and Møller(2001)] could be used to synthesize efficient (even if large) code from expressive specifications. The work on graph types [Klarlund and Schwartzbach(1993)] proposes to synthesize fields given by definitions in monadic second-order logic. Automata have also been applied to the synthesis of efficient code for pattern-matching expressions [Sekar et al.(1995)Sekar, Ramesh, and Ramakrishnan].

Synthesis of constraints for rational arithmetic has been previously applied to automatically construct abstract transfer functions in abstract interpretation of linear constraints over rationals [Monniaux(2009)]. Our results apply this technique to integer linear arithmetic and constraints on sets. More generally, we observe that such synthesis is useful as a general-purpose programming construct.

Our approach is also sharing some of the goals with partial evaluation [Jones et al.(1993)Jones, Gomard, and Sestoft]. However, we do not need to employ general-purpose partial evaluation techniques (which typically provide linear speedup), because we have the knowledge of a particular decision procedure. We use this knowledge to devise a synthesis algorithm that, given formula F, generates the code corresponding to the invocation of this particular decision procedure. This synthesis process checks the uniqueness and the existence of the solutions, emitting appropriate warnings. Moreover, the synthesized code can have reduced complexity compared to invoking the decision procedure at run time, especially when the number of variables to synthesize is bounded.

8 Interactive Synthesis of Code Snippets

In this chapter we describe a synthesis tool called InSynth that applies theorem proving technology to synthesize code fragments. InSynth interactively displays a ranked list of suggested code fragments that are appropriate for the current program point (see Figure 8.1). To determine candidate code fragments, our tool takes into account polymorphic type constraints coming from the library functions, as well as test cases. In our experiments, InSynth was useful for synthesizing code fragments for common programming tasks.

8.1 Motivation

Algorithmic software synthesis from specifications is a difficult problem. Yet software developers perform a form of synthesis on a daily basis, by transforming their intentions into concrete programming language expressions. The goal of our tool, InSynth, is to explore the relationship and synergy between algorithmic synthesis and developers' activities by deploying synthesis for code fragments in interactive settings. To make the problem more tractable, InSynth aims to synthesize small fragments, as opposed to entire algorithms. InSynth builds code fragments containing functions drawn from large and complex libraries. The goal of InSynth to save the developers the effort of searching for appropriate methods and their compositions. InSynth is deployed within an integrated development environment. When invoked, it suggests multiple meaningful expressions at a given program point, using type information and test cases.

InSynth primarily relies on type information to perform its synthesis task. When the developer needs a piece of code that computes a value of a given type, they declare the type of this value, using the usual syntax of the Scala programming language [Odersky et al.(2008)Odersky, Spoon, and Venners]. They then invoke InSynth to find suggested code fragments of this type. It uses Ensime [Cannon(2011)], an incremental Scala compiler integrated into the editor, to gather the available values, fields, and functions. The use of type information is inspired by Prospector [Mandelin et al.(2005)Mandelin, Xu, Bodík, and Kimelman], but InSynth has an important additional dimension: it handles generic (parametric) types [Damas and Milner(1982)]. Generic types are a mainstream mechanism to write safe and reusable

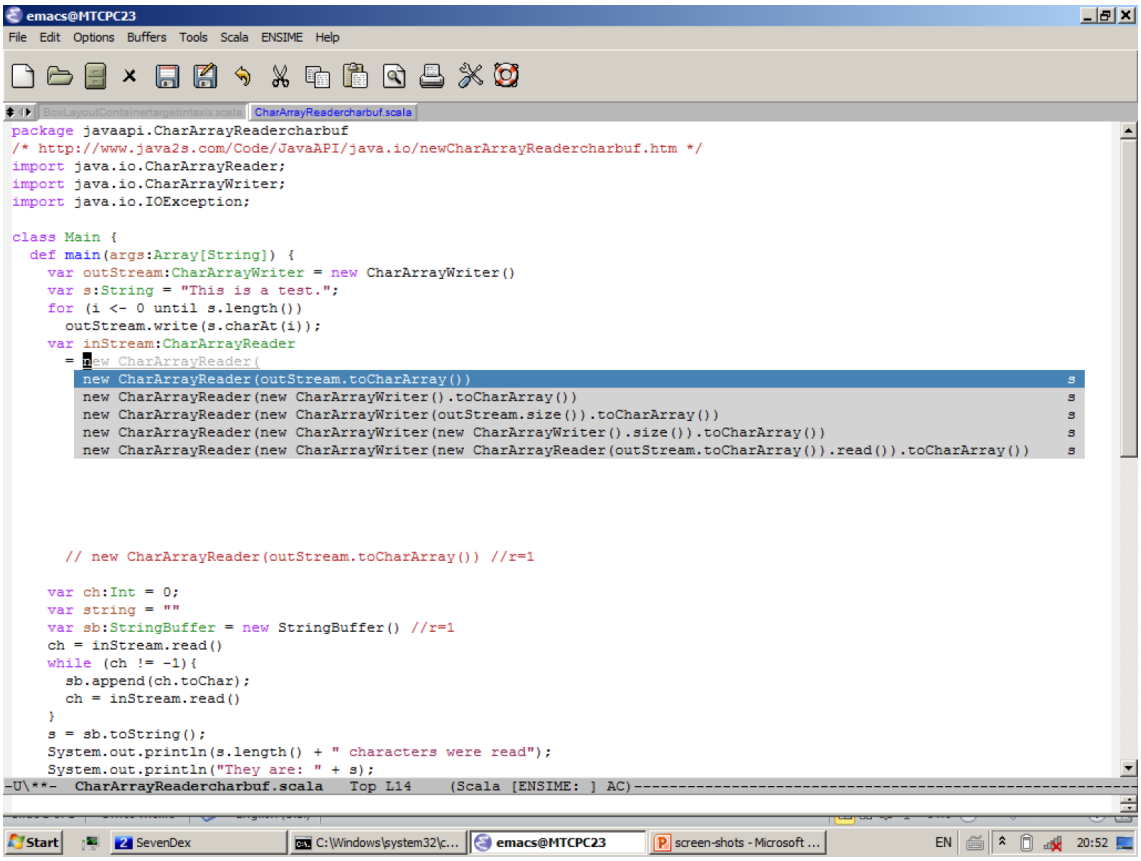

Figure 8.1: InSynth displays suitable code fragments

code in, e.g., Java, ML, and Scala. They are particularly frequent in libraries.

The support for generic types is a fundamental generalization compared to previous tools, which handled only ground types. With generic types, a finite set of declarations will generate an infinite set of possible values, and the synthesis of a value of a given type becomes undecidable. InSynth therefore encodes the synthesis problem in first-order logic. This encoding has the property that a value of the desired type can be built from functions of given types iff there exists a proof for the corresponding theorem in first-order logic. It is therefore related to known connections between proof theory and type theory. In type-theoretic terms, InSynth attempts to check whether there exists a term of a given type in a given polymorphic type environment. This is known as the type inhabitation problem. If such terms exist, the goal of InSynth is to produce a finite subset of them, ranked according to some criterion.

InSynth implements a prover, which finds multiple proofs representing candidate code fragments. Our implementation was inspired by first-order resolution. The use of resolution is related to the traditional deductive program synthesis [Manna and Waldinger(1980)], but our approach attempts to derive code fragments by using type information instead of the code

itself. As a post-processing step, InSynth filters out the candidate code fragments that crash the program, or that violate assertions or postconditions. This functionality incorporates input/output behavior [Jha et al.(2010)Jha, Gulwani, Seshia, and Tiwari], but uses it mostly to improve the precision of the primary mechanism, the type-driven synthesis.

In the software development process an accurate specification is often not available. A synthesis tool should thus be equipped to deal with under-specified problems, and be prepared to generate multiple alternative solutions when asked to do so. Our algorithm fulfills this requirement: it generates multiple solutions and ranks them using a system of weights. The current weight computation takes into account the proximity of values to the point in which the values are used, as well as user-specified hints, if any. A database of code samples is additionally used to derive weights, providing effects similar to some of the previous systems [Sahavechaphan and Claypool(2006), Mandelin et al.(2005)Mandelin, Xu, Bodík, and Kimelman]. Given a weight function, InSynth directs its search using a technique related to ordered resolution [Bachmair and Ganzinger(2001a)].

8.2 Examples

Consider the problem of retrieving data stored in a file. Suppose that we have the following definitions:

```
def fopen(name:String):File = { ... }
def fread(f:File, p:Int):Data = { ... }
var currentPos : Int = 0
var fname : String = ""
def getData():Data = ■
```

The developer is about to define, at the position marked by ■, the body of the function getData that computes a value of type Data. When the developer invokes InSynth, the result is a list of valid expressions (snippets) for the given program point, composed from the values in the scope. Assuming that among the definitions we have functions fopen and fread, with the types shown above, InSynth will return as one of the suggestions fread(fopen(fname),currentPos), which is a simple way to retrieve data from the file given the available operations. In our experience, InSynth often returns snippets in a matter of milliseconds. Such snippets may be difficult to find manually for complex and unknown APIs, so InSynth can also be seen as a sophisticated extension of a search and code completion functionality.

Parametric polymorphism. We next illustrate the support of parametric polymorphism in InSynth. Consider the standard higher-order function map that applies a given function to each element of the list. Assume that the map function is in the scope. Further assume that we wish to define a method that takes as arguments a function from integers to strings and a list of strings, and returns a list of strings.

```
def map[A,B](fun:A ⇒ B, array:Array[A]):Array[B] = { ... }
def createStringArray (name:String):Array[String] = { ... }
def createIntArray (fun:String ⇒ Int, name:String):Array[Int] = ■
```

As seen in Figure 8.2, InSynth returns map[String, Int](fun, createStringArray(name)) as a result, instantiating polymorphic definition of map and composing it with createStringArray. InSynth efficiently handles polymorphic types through resolution and unification.

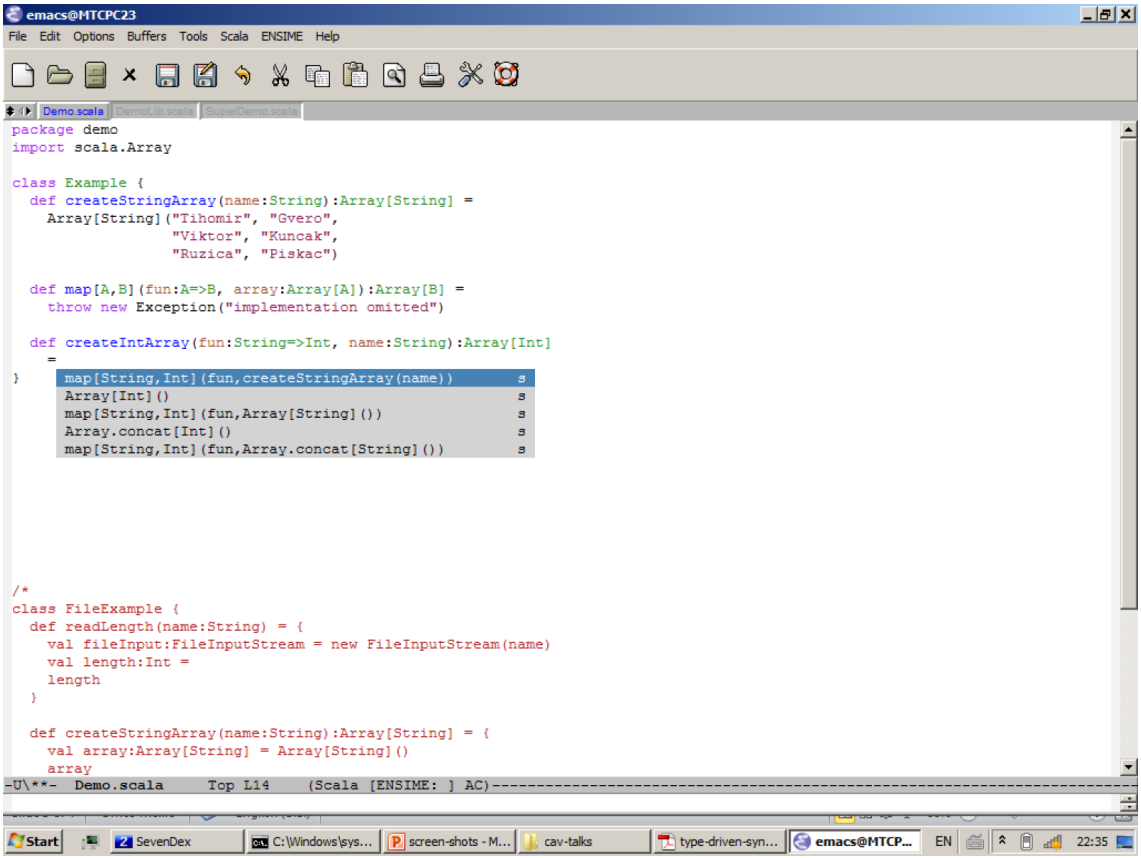

Figure 8.2: Polymorphic behavior of InSynth

Using code behavior. The next example shows how InSynth applies testing to discard those snippets that would make code inconsistent. Define the class FileManager containing methods for opening files either for reading or for writing.

```
class Mode(mode:String)
class File (name:String, val state : Mode)
object FileManager {
  private final val WRITE:Mode = new Mode("write")
  private final val READ:Mode = new Mode("read")
  def openForReading(name:String):File = ■
    ensuring { result => result.state == READ}
}
object Tests { FileManager.openForReading("book.txt") }
```

If it were based only on types, InSynth would return both snippets **new** File(name, WRITE) and **new** File(name, READ). However, InSynth also checks run-time method contracts (pre- and

post-conditions) and verifies whether each of the returned snippets passes the test cases with them. Because of postconditions requiring that the file is open for reading, InSynth discards the snippet **new** File(name, WRITE) and returns only **new** File(name, READ).

Applying user preferences. The last example demonstrates one way in which a developer can influence the ranking of the returned solutions. We consider the following functionality for managing calendar events.

```
private val events: List [Event] = List.empty[Event]
def reserve (user: User, date: Date):Event = { ... }
def getEvent(user: User, date: Date):Event = { ... }
def remove(user:User, date: Date):Event = ■
```

Assume that a user wishes to obtain a code snippet for remove. In general, InSynth ranks the results based on the weight function. We have found that the default computation of the weight is often adequate. Running the above example returns reserve(user, date) and getEvent(user, date), in this order. If this order is not the preferred one, the developer can modify it using elements of text search. To do so, the developer supplies a list of suggested strings indicating the names of some of the methods expected to appear in the code snippet. For example, if the developer invokes InSynth with "getEvent" as a suggestion, the ranking of returned snippets changes, and getEvent(user, date) appears first in the list.

8.3 From Scala to Types

For the main question of finding a code snippet for the given type, the corresponding problem in type theory is the *type inhabitation problem*. In this section we review basic definitions and facts and establish a connection between the type inhabitation problem and the problem of finding code snippets.

Let T be set of types and let E be a set of expressions. A type environment Γ is a finite set $\{e_1 : \tau_1, \ldots, e_n : \tau_n\}$, containing pairs of the form $e_i : \tau_i$, where x_i is an expression and τ_i is a type. The pair $e_i : \tau_i$ is called a type binding.

An expression $\Gamma \vdash e : \tau$ denotes that from an environment Γ we can derive a type binding $e : \tau$ by applying rules of some calculus. The type inhabitation problem for the given calculus is stated as: given a type τ and a type environment Γ, does there exist an expression e such that $\Gamma \vdash e : \tau$.

The first step is to construct the type environment from a Scala program. For every type declaration that appears in the given program, we create a type binding and add it to the context. The bindings is constructed in the following way:

1. Primitive types such as `Int`, `Bool`, `String` are represented with the constants of the same name:

val x : **Int** $\leadsto$ $x : Int$

2. To encode complex types, such as `Map[Int, String]`, lists, sets and similar, we use the same formalism as for the primitive types:
   val x : **List**[**String**] $\leadsto$ $x : List(String)$
   val x : **Map**[**Int**, **String**] $\leadsto$ $x : Map(Int, String)$

3. To capture type constraints on functions and methods, we use the Hindley-Milner type description [Damas and Milner(1982)] and the $\rightarrow$ notation. Function f that returns a value of type R and takes n arguments as an input, with i-th argument being of type T_i, is declared as $f : T_1 \rightarrow \ldots \rightarrow T_n \rightarrow R$:
   def f : (a : **Int**, b: **String**): **Bool** $\leadsto$ $f : Int \rightarrow String \rightarrow Bool$
   Additionally, for the instance methods we add the receiver type. In a standard implementation, to an instance method is passed a hidden reference to the object where it belongs to. We model this by adding an arrow from the receiver type to the method. This will also help us in term reconstruction. Consider the following instance method m:

```
class C {
  def m (a:T1, b:T2): R
}
```

   It is encoded as $m : C \rightarrow T_1 \rightarrow T_2 \rightarrow R$.

4. In Scala it is possible to pass a function as a method's argument. For example,

```
def m(f: String => Int, a: String): Int = f(a) + 2
```

   is a higher order function that takes function f as an argument. Both f and m are encoded using the $\rightarrow$ symbol:
   $m : (String \rightarrow Int) \rightarrow String \rightarrow Int$
   $f : String \rightarrow Int$

5. InSynth supports polymorphic functions as well. This is done using universal quantifiers. Consider as an illustration a generic method that takes a value of any type and creates a list:

```
def elem2list [A] (x: A): List [A] = {List(x)}
```

   The type binding derived from this method is
   $elem2list : \forall \alpha.\alpha \rightarrow List(\alpha)$
   By adding quantifiers we are above the expressivity of the ground types and the propositional logic.

6. Finally, to encode the query, which is to answer whether there is a value of a type τ, we add the following type binding:

$\mathsf{goal} : \tau \to \bot$

Both symbols goal and $\bot$ have the special meaning and they cannot be used for any other encoding. We solve the type inhabitation problem by find an inhabitant of type $\bot$. Once there is an inhabitant of type $\bot$, we should be able easily to derive an inhabitant of type τ.

8.4 Type Inhabitation in the Ground Applicative Calculus

Before going to the general framework that includes polymorphic types, we frist describe reasoning about the ground types.

Definition 8.1 (Ground Types) *Let C be a fixed finite set. For every $c \in C$, with $c/_n$ we denote the arity of the element. The elements of arity 0 are called constants. The set of all ground types T_g is defined by the grammar:*

$$T_g ::= C(T_g, \ldots, T_g) \mid T_g \to T_g$$

To establish a connection between T_g and the Scala types, one could consider the set C as a set containing the Scala primitive types (such as `Int` or `String`) and type constructors (such as $\texttt{List}/_1$, $\texttt{Map}/_2$).

Let S be a set containing function symbols. The set of all ground terms E_g is formed inductively from S as follows: all constants of S are ground terms. If $t_1, \ldots, t_n$ are ground terms and $f/_n \in S$, then $f(t_1, \ldots, t_n)$ is a ground term.

Figure 8.3 lists the rules of a calculus for the ground types. We call this calculus *the ground applicative calculus*. It supports the application of a function to a term, and the function composition. Those two rules have a natural interpretation in a programming language. Through the application we construct a snippet, where a method is applied on its argument, while the composition represents a combination of several methods.

$$\textsc{Axiom}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash f(x) : \tau_2}$$

$$\textsc{Compose}\ \frac{\Gamma \vdash f_1 : \tau_0 \to \tau_1 \qquad \ldots \qquad \Gamma \vdash f_n : \tau_{n-1} \to \tau_n}{\Gamma \vdash f_n \circ \ldots \circ f_1 : \tau_0 \to \tau_n}$$

Figure 8.3: Calculus for the Ground Types

8.4.1 Type Inhabitation in the Ground Applicative Calculus

The problem of a type inhabitation is widely studied for various calculi. However, very often this problem is undecidable. The ground applicative calculus can be seen as a sub-calculus of the simply typed lambda calculus, which additionally contains the lambda abstraction. In the simply typed lambda calculus the type inhabitation problem is decidable, but very hard. By reduction to the canonical quantified Boolean formula (QBF) problem, it was shown in [Statman(1979)] that the problem is PSPACE-complete. In this section we show that if the lambda abstraction is disabled, the type inhabitation problem can be solved much faster.

Theorem 8.2 *The type inhabitation problem in the ground applicative calculus can be solved in polynomial time.*

Proof. Let Γ be a type environment $\Gamma = \{e_1 : \tau_1, \ldots, e_n : \tau_n\}$, with $e_i \in E_g$ and $\tau_i \in T_g$. Let τ_0 be a type for which we ask if there is an expression e_0 such that $\Gamma \vdash e_0 : \tau_0$. We encode the query as the type binding $\mathsf{goal} : \tau_0 \to \bot$, where $\bot$ is a designated symbol, previously unused. The goal of an algorithm is to derive a type binding $e : \bot$. The expression e can only be of the form $\mathsf{goal}(x)$ and the term x has the desired type τ_0.

Let $\mathsf{TParts}(\Gamma)$ denote the set of all types appearing in Γ, together with τ_0. In addition, if the type is not a constant, then $\mathsf{TParts}(\Gamma)$ also contains all its components: if $f(t_1, \ldots, t_n) \in \mathsf{TParts}(\Gamma)$, then also $t_1, \ldots, t_n \in \mathsf{TParts}(\Gamma)$. The cardinality of the set $\mathsf{TParts}(\Gamma)$ is clearly finite and polynomial in the size of Γ.

We consider a sequence of type bindings that starts with an enumeration of Γ and continues with the application of inference rules until reaching type judgment of the form $e_0 : \tau_0$. We show that there is a term e_0 such that that $\Gamma \vdash e_0 : \tau_0$, then it can be derived in polynomial time. For this purpose we can assume that each step produces a term that is non-redundant, that is, it is subsequently used in the derivation (otherwise we could eliminate it).

We first assume that there is no COMPOSE rule, so we only apply the APP rule. In that case each derived term has a type from $\mathsf{TParts}(\Gamma)$. The set of derived types does not change if we always adopt the following principle: never use in premises elements $t : \tau$ of a sequence if there is a term $t' : \tau$ with the same type appearing earlier in the sequence. If we adopt this policy, the number of newly introduced elements is bounded by $|\mathsf{TParts}(\Gamma)|^2$. Therefore, the process terminates. The resulting sequence also gives a representation of the (possibly infinite) set of terms that have given type. The infinite sets of solutions appear precisely from derivations that use a term of some type to derive a new term of the same type. However, the policy described ensures that such loops are detected and not followed.

We next assume that we can also use the composition rule. This problem does not reduce the to the case of application because viewing $\circ$ as a higher-order function would require assigning it a polymorphic type. Nonetheless, we show that we also obtain a polynomial bound.

First we observe that, if the COMPOSE rule is used to obtain a term of the form $(f_1 \circ \ldots \circ f_n)(x)$ then it was not not necessary for producing a new type: we could have instead directly used APP alone to construct $f_1(\ldots f_n(x)\ldots)$. We next use the fact that the COMPOSE rule already accounts for any number of function symbols and that the composition is associative and producing the same type for different composition orders, to conclude that it is not necessary to use the result of a compose rule multiple times in a sequence. From those observations we conclude that COMPOSE always produces either an argument of APP or the required type τ_0. In the second case, by using the APP rule, we derive an inhabitant of the $\bot$ type, i.e. that COMPOSE again produced an argument of APP.

We can therefore replace COMPOSE rule with the following APPCOMPOSE rule, in a process similar to completion in term rewriting. This results in the following system, which again has the crucial property that its result is always an element of $\mathsf{TParts}(\Gamma)$:

$$\text{APP}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash f(x) : \tau_2}$$

$$\text{APPCOMPOSE}\ \frac{\Gamma \vdash c : (\tau \to \tau_n) \to \sigma \qquad \Gamma \vdash f_1 : \tau \to \tau_1 \qquad \ldots \qquad \Gamma \vdash f_n : \tau_{n-1} \to \tau_n}{\Gamma \vdash c(f_n \circ \ldots \circ f_1) : \sigma}$$

Therefore, application of such rules also finishes in at most $|\mathsf{TParts}(\Gamma)|^2$ steps. This completes the proof that type inhabitation problem where we restrict terms to be obtained from application and function composition is polynomial.

8.5 Quantitative Applicative Ground Inhabitation

There might be many terms belonging to a given type, and the question of finding the best term naturally arises. We address this problem by assigning a weight to every expression. Similar to resolution-based theorem proving, a lower weight indicates the higher relevance of the term.

The ranking of the snippets and the entire algorithm strongly rely on a system of weights. The system considers snippets of a smaller weight as preferable to those of a larger weight. The weights of terms extend to the weights of clauses, as in the multiset ordering of clauses in first-order resolution [Bachmair and Ganzinger(2001a)].

To begin with, we define an ordering on the symbols and assign a weight to each symbol. The user-preferred symbols have the smallest weight (highest preference). They are followed by the local symbols occurring in the current method. The remaining symbols of the corresponding class have a larger weight than the local symbols. Finally, the symbols outside the current class have the largest weight. This includes symbols from the imported libraries and APIs.

Once the ordering and the weights of the symbols are fixed, we compute the weights of types

and expressions. The weight of a type or of an expression is computed as the sum of the weights of all the symbols occurring in the type or in the expression.

8.5.1 Finding the Best Type Inhabitant

In this section we further extend the type inhabitation problem with the additional requirement to find an expression of the minimal weight.

Theorem 8.3 *Let w be a weight function defined on the type and expression symbols. We extend w to type bindings as described above. For a type environment Γ and a type τ_0 in the ground applicative calculus it is possible to find in polynomial time an expression e of the type τ_0, such that the weight of e is smaller than the weight of all other expressions of the type τ_0.*

Proof. The proof extends the proof of Theorem 8.2. It builds a sequence of type bindings that can be derived from Γ. To every element τ of $\mathsf{TParts}(\Gamma)$ we assign a pair (n,t) where n is the minimum weight of all terms of type τ, which are currently in the sequence, and t is an expression such that $w(t) = n$. Initially, to all the elements of $\mathsf{TParts}(\Gamma)$ we assign $(\infty, -)$. As before, we construct a sequence of type bindings. With every type binding $e : \tau$ added to the sequence, we recalculate the annotation of τ. If its current minimum weight is strictly greater than $w(e:\tau)$, then the new annotation becomes $(w(e), e)$. In the sequence we also replace every occurrence of the expression e' of the type τ by e. We can do such a replacement safely, since e' does not appear in the derivation of e (otherwise it would not hold $w(e') > w(e)$). We continue with the enumeration of the derived type bindings as in the proof of Theorem 8.2, using the same restrictive principle about the type bindings that can participate in the derivations. Applying the same arguments we prove that it is possible to find a term of the minimum weight in polynomial time.

8.6 Quantitative Inhabitation for Generics

This section presents our algorithm for type inhabitation in the presence of generic (parametric) types as in the Hindley-Milner type system, without nested type quantifiers. We represent type variables implicitly, as in resolution proof systems for first-order logic.

Definition 8.4 (Generic Types) *Let C be a fixed finite set. For every $c \in C$, with $c/_n$ we denote the arity of the element. Let V be a set of type variables. The set of all generic types T is defined by the grammar:*

$$T_b ::= V \mid C(T_b, \ldots, T_b) \mid T_b \to T_b$$
$$T ::= T_b \mid \forall V.T$$

$$\textsc{App}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1' \quad \sigma = \mathsf{mgu}(\tau_1, \tau_1')}{\Gamma \vdash f(x) : \tau_2' \quad \tau_2' = \sigma(\tau_2)}$$

$$\textsc{Compose}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash g : \tau_0 \to \tau_1' \quad \sigma = \mathsf{mgu}(\tau_1, \tau_1')}{\Gamma \vdash (f \circ g) : \tau_0' \to \tau_2' \quad \tau_0' = \sigma(\tau_0) \quad \tau_2' = \sigma(\tau_2)}$$

Figure 8.4: Rules for Generic Types used by Our Algorithm

Those types are also known under the names ML-style types or Hindley-Milner types. Figure 8.4 shows the rules for application, as well as the rule for composition (which we introduce to improve performance).

Before we execute an algorithm for type inhabitation, we add a complete set of combinators belongs to the initial environment, with their polymorphic types. We denote this set by Γ_{Comb}. For example, we can use

$$\{\mathsf{K}{:}\alpha \to \beta \to \alpha, \mathsf{S}{:}(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma\}$$

This make the application rule complete for the purpose of finding a term of a given type, thanks to the translation from lambda calculus to combinatory logic. We therefore omit the lambda abstraction rule. This approach is also used in [Rehof and Urzyczyn(2011)], but for a non-generic type system with intersection types.

Description of the algorithm. Figure 8.5 shows the algorithm that systematically applies rules in Figure 8.4, while avoiding cycles due to repeated types whose terms have non-minimal weights.

The algorithm maintains two sets of bindings (pairs of expressions and their types): Γ, which holds all initial and derived bindings, and q, which is a work list containing the bindings that still need to be processed. Initially, Γ contains program declarations, as well as the combinators and the goal encoded as $(\mathsf{G}{:}\tau_G \to \bot_{fresh})$ where τ_G is the type for which the user wishes to generate expressions. The work list initially contains all these declarations as well. The algorithm accumulates the expressions of the desired type in the set res. The main loop of the algorithm runs until the timeout is reached or the work list q becomes empty.

The body of the main loop of the algorithm selects a minimal (given by $\mathsf{best}(\_)$) binding $(e_1{:}\tau_1)$ from the work list q and attempts to combine it with all other bindings in Γ for which the types τ_1 and τ_2 can be unified to participate in one of the inference rules (we denote this condition using the $\mathsf{cmpt}(\tau_1, \tau_2)$ relation). Note, however, that there is no point to combine $(e_1{:}\tau_1)$ with a $(e_2{:}\tau_2)$ if there is another $(e_2'{:}\tau_2)$, with the same τ_2 but with a strictly smaller $w(e_2')$. Therefore,

INPUT: Γ_0 - environment at program point
INPUT: τ_G - desired type
OUTPUT: res - set of resulting expressions e with $\Gamma_0 \vdash e{:}\tau_G$
Definitions:
$w(e{:}\tau) := w(e) + w(\tau)$
$\mathsf{bestT}(\tau, \Gamma) := \{(e{:}\tau) \in \Gamma \mid (\forall (e'{:}\tau) \in \Gamma.\ w(e) \leq w(e'))\}$
$\mathsf{bestT}(e'{:}\tau', \Gamma) := \mathsf{bestT}(\tau', \Gamma)$
$w(\mathsf{bestT}(b, \Gamma)) = w(b), \text{if } \exists b \in \mathsf{bestT}(b, \Gamma),\ +\infty \text{ otherwise}$
$\mathsf{best}(\mathsf{q}) := \{b \in \mathsf{q} \mid \forall b' \in \Gamma.\ w(b) \leq w(b')\}$
$\mathsf{cmpt}(\tau_1, \tau_2) :=$ an mgu in App or Compose of Figure 8.4 exists

Code:
$\Gamma = \Gamma_0 \cup \Gamma_{\mathsf{Comb}} \cup \{(\mathsf{G}{:}\tau_G \to \bot_{fresh})\}$
$\mathsf{q} = \Gamma$
$\mathsf{res} = \emptyset$
while $\neg\mathsf{timeout} \wedge \mathsf{q} \neq \emptyset$ **do**
  let $(e_1{:}\tau_1) \in \mathsf{best}(\mathsf{q})$
  $\mathsf{q} = \mathsf{q} \setminus \{(e_1{:}\tau_1)\}$
  for all $(e_2{:}\tau_2) \in \{(e_2{:}\tau_2) \in \mathsf{bestT}(\tau_2, \Gamma) \mid \mathsf{cmpt}(\tau_1, \tau_2)\}$ **do**
    $\mathsf{derived} = \mathsf{App}(e_1{:}\tau_1, e_2{:}\tau_2) \cup \mathsf{Comp}(e_1{:}\tau_1, e_2{:}\tau_2)$
    $\mathsf{res} = \{e' \mid (e : \bot_{fresh}) \in \mathsf{derived}, e[\mathsf{G} := \mathsf{I}] \overset{\mathsf{I}(t)\to t}{\rightsquigarrow^*} e'\}$
    $\mathsf{q} = \mathsf{q} \cup \{b \in \mathsf{derived} \mid w(b) < w(\mathsf{bestT}(b, \Gamma))\}$
    $\Gamma = \Gamma \cup \mathsf{derived}$
  end for
end while

Figure 8.5: The Search Algorithm for Quantiative Inhabitation for Generic Types

the algorithm restricts the choice of $(e_2{:}\tau_2)$ to those where $w(e_2)$ is minimal for a given τ_2. We formalize this using the function $\mathsf{bestT}(\tau_2, \Gamma)$ that finds a set of such bindings with minimal e_2. We also extend the function to accept a candidate e_2' (which is ignored in looking up the minimal e_2). Moreover, we define $w(\mathsf{bestT}(\tau_2, \Gamma))$ to denote the value of this minimum (if it exists).

The sets $\mathsf{App}(e_1{:}\tau_1, e_2{:}\tau_2)$ and $\mathsf{Comp}(e_1{:}\tau_1, e_2{:}\tau_2)$ are results of applying the rules from Figure 8.4. If no rule can be applied the result is the empty set. We use derived to denote the set of results of applying the inference rules to selected bindings. These results may need to be processed further and therefore the algorithm may need to insert them into q. However, it avoids inserting them if the derived binding has a type that already exists in Γ and the newly derived expression does not have a strictly smaller weight. This reduces the amount of search that the algorithm needs to perform.

Because of the declaration $(\mathsf{G}{:}\tau_G \to \bot_{fresh})$, the algorithm detects expressions of type τ_G using the expressions e of fresh type $\bot_{fresh}$. To obtain the expression of the desired type, we replace in e every occurrence of G with the identity combinator I. This is justified because $\bot_{fresh}$ is a

fresh constant, so replacing it with τ_G in a derivation of $\Gamma \cup \{(\mathsf{G}{:}\tau_G \to \bot_{fresh})\}(e{:}\bot_{fresh})$ yields a derivation of $\Gamma \cup \{(\mathsf{G}{:}\tau_G \to \tau_G)\} \vdash (e{:}\tau_G)$, in which we can use I instead of G. The algorithm also simplifies the accumulated expressions by reducing I where possible. In the presence of higher-order functions I may still remain in the expressions, which is not a problem because it is deducible from any complete set of combinators.

Finally, under the assumption that a linear weight function is given, and the weight of each expression symbol is strictly positive, it is straightforward to see that the algorithm finds the derivations for all types that can be obtained using the rules from Figure 8.4. Indeed, the weight of an expression strictly increases during the derivation, so an algorithm, if it runs long enough, reaches arbitrarily long value as the minimum of the work list. This shows that the algorithm is complete.

8.7 Subtyping using Coercions

A powerful method to model subtyping is to use coercion functions [Luo(2008),Reynolds(1980), Breazu-Tannen et al.(1991)Breazu-Tannen, Coquand, Gunter, and Scedrov]. This approach raises non-trivial issues when we perform type checking or type inference, but becomes simple and natural if the types are given but we search for the terms.

Simple conversions. In the absence of variant constructors and type bounds, we can model the subtyping relation $A <: B$ by the existence of a coercion expression $c{:}A \to B$. For example, if a class $A[\vec{T}]$ with type parameters $\vec{T}$ extends or mixes-in another class $B[\vec{\tau}(\vec{T})]$, we introduce into the environment a conversion function $c{:}A[\vec{T}] \to B[\vec{\tau}(\vec{T})]$. Note that the composition of coercion functions immediately accounts for the transitivity of the subtyping relation.

We demonstrate the use of coercions on the following example:

Example. Consider the following code:

```
class ArrayList[T] extends AbstractList[T] with List[T]
  with RandomAccess with Cloneable with Serializable { ... }
abstract class AbstractList[E] extends AbstractCollection[E]
  with List[E] {
  ...
  def iterator(): Iterator[E] = {...}
}
```

Because ArrayList [T] extends AbstractList [T], and AbstractList [E] extends AbstractCollection [E], we generate two coercion functions:

c1 : $\forall\alpha$. ArrayList[α] $\to$ AbstractList[α]
c2 : $\forall\beta$. AbstractList[β] $\to$ AbstractCollection[β]

There is a declared member of AbstractList [E], which is encoded as a type binding:

iterator : $\forall\gamma$. AbstractList[γ] → Iterator[γ]

Let us assume that there is local variable declaration in main method of the example that yields the binding

al : ArrayList [String]

Finally, let us assume that the goal in the example is to find an expression of type Iterator [String]. This results in a type binding

τ_G : Iterator[String] → $\bot_{fresh}$

Using rules in Figure 8.4, the algorithm in Figure 8.5 unifies the type variables and ground type of String and derives in Γ the type binding

τ_G(iterator(c1(al))) : $\bot_{fresh}$

This produces I(iterator (c1(al))) in the res variable of the algorithm in Figure 8.5. We further simplify this expression to to iterator (c1(al)). Finally, we erase all conversion functions and obtain iterator (al), which is displayed to the user as the Scala code al . iterator ().

8.8 InSynth Implementation and Evaluation

Program	# Loaded Declarations	# Methods in Synthesized Snippets	Time [s]
FileReader	6	4	< 0.001
Map	4	4	< 0.001
FileManager	3	3	< 0.001
Calendar	7	3	< 0.001
FileWriter	320	6	0.093
SwingBorder	161	2	0.016
TcpService	89	2	< 0.001

Figure 8.6: Basic algorithm for synthesizing code snippets

InSynth is implemented in Scala and built on top of the Ensime plugin [Cannon(2011)]. It can therefore directly use program information computed by the Scala compiler, including abstract syntax trees and the inferred types. Furthermore, it can generate an appropriate pop-up window with suggested synthesized snippets and allow the user to interactively select the desired fragment.

Figure 8.6 gives an idea of the performance of the system. We ran all examples on Intel(R) Core(TM) i7 CPU 2.67 GHz with 4 GB RAM. The running times to find the first solution are usually bellow two milliseconds. Our experience suggests that the algorithm scales well. As an illustration, we were able to synthesize a snippet containing six methods in 0.093 seconds from the set of 320 declarations. Times to encode declarations into FOL formulas range from

0.015 (Calendar) to 0.046 (FileWriter) seconds. If the synthesized snippets need to use more methods from imported libraries, the synthesis typically takes longer, but is typically fast enough to be useful.

9 Conclusions

In this dissertation we have explored the use of decision procedures for increasing software reliability. We particularly focused on decision procedures for the verification of data structures, for software synthesis, and for proving program termination. In addition, we also presented a new combination procedure for non-disjoint theories, obtaining a logic in which we can express and verify complex properties of data structures.

Motivated by applications in verification, we introduced an expressive class of constraints on multisets. Our constraints support arbitrary multiset operations, as well as the cardinality operator. We presented a decision procedure for the satisfiability of these constraints, showing that they efficiently reduce to an extension of quantifier-free Presburger arithmetic. For the later problem we presented a decision procedure based on a semilinear set representation of the set of solutions of a Presburger arithmetic formula. We proved that the satisfiability problem for our constraints is an NP-complete problem.

This thesis further contains two extensions of the logic for reasoning about multisets with cardinality constraints. First, we extended the logic to one that allows reasoning about sets, multisets, and fuzzy sets, as well as their cardinality bounds. We developed a decision procedure similar to the original one, which translates a formula into an extension of mixed-integer linear arithmetic (MLIRA). We proved that the star operator can also be eliminated when applied to a MLIRA formula. The second extension that we considered, was adding to the above logic multisets that result from the application of an uninterpreted function to a set. On top of the previous decision procedure, we built a decision procedure for this extended logic. We showed that the satisfiability problem for this new extended logic is NEXPTIME complete.

Next, we presented POSSUM , a new logic and decision procedure for reasoning about multiset orderings. POSSUM can express constraints over complex well-founded orderings, which makes it a useful tool for proving termination. The logic subsumes linear integer arithmetic which has been traditionally used to express ranking functions in automated termination proofs. We established that the satisfiability problem for POSSUM is NP-complete, provided the base theory is in NP. Thus, POSSUM has the same complexity as quantifier-free linear

integer arithmetic. Furthermore, our decision procedure is amenable to a practical implementation. We thus believe that POSSUM provides a valuable tool for extending the scope of existing termination provers.

As a unifying framework for our individual decision procedures, we developed a combination procedure for non-disjoint theories, which share set symbols and operators. The combination is possible if the component theories can be reduced to a common theory, namely to the logic of sets with cardinality constraints. We have shown that the following theories can be combined: 1) Boolean Algebra with Presburger Arithmetic (with quantifiers over sets and integers), 2) weak monadic second-order logic over trees (with monadic second-order quantifiers), 3) two-variable logic with counting quantifiers (ranging over elements), 4) the Bernays-Schönfinkel-Ramsey class of first-order logic with equality (with $\exists^*\forall^*$ quantifier prefix), and 5) the quantifier-free logic of multisets with cardinality constraints.

As for software synthesis, we presented the general idea of turning decision procedures into synthesis procedures. We have explored in greater detail how to do this transformation for theories admitting quantifier elimination, in particular linear arithmetic. We have further transformed a BAPA decision procedure into a synthesis procedure, illustrating, in the process, how to layer multiple synthesis procedures one top of each other. We have pointed out that synthesis can be viewed as a powerful programming language extension. Such an extension can be seamlessly introduced into popular programming languages in the form of a new programming construct.

Finally, we have presented an algorithm for synthesizing snippets of Scala code. The algorithm is based on automated reasoning and implements an intuitionistic calculus based on first-order resolution. The synthesized snippets can combine all declared values, fields, and methods that are in the scope at the current program point, so the problem is closely related to the problem of type inhabitation for type systems. Our system supports parametric polymorphism and uses a theorem prover to find proofs that correspond to code snippets.

To conclude, we believe that the decision procedures presented in this thesis have increased the range of properties of various programming constructs that can be automatically verified. Additionally, synthesis procedures generate code that is correct by construction, thereby rendering obsolete any further need to verify this code. As indicated, we have implemented most of the decision and synthesis procedures presented in this dissertation and explored their practical potential.

9.1 Future Work

We conclude this dissertation by giving an outline of possible directions for future research.

9.1.1 Complete Reasoner for Sets and Multisets

MUNCH is currently implemented as an incomplete tool. We plan to further develop MUNCH so that it will become a complete reasoner for set and multisets. One way to make it complete is by applying the Decomposition Theorem for Polyhedra and using the ideas from Chapter 4. We would compute a semilinear set by relaxing an integer formula with rational constraints. We should then check for the integer vectors that describe the integer solution set. This way we would avoid computing Gröbner bases, which are currently used for computation of semilinear sets. With such an implementation, MUNCH will become the first tool that can compute a semilinear set. Semilinear sets as non-negative solutions of an arithmetic formula F have many intriguing applications that we would like to explore. In order to makes this approach scalable, we plan to investigate how semilinear sets can be optimally represented.

9.1.2 Software Synthesis by Combining Subroutines

The technique of complete functional synthesis described in Chapter 7 generates programs using only built-in programming language constructs. On the other hand, our tool InSynth to derive code snippets also takes into account methods and functions that are defined by APIs or the programmer, in addition to the built-in constructs. We are interested in improving algorithms for complete functional synthesis, so that they can also make calls to custom-made functions and procedures. We believe that the experience that we gained by developing InSynth, will help us to developer a decision procedure that can reason about such subroutines. The most natural way for specifying the behavior of a subroutine is by using pre and postconditions. Since these specifications are given in a formal logic, we should be able to reduce reasoning about subroutines to existing decision procedures.

9.1.3 Additional Theories for Complete Synthesis

We plan to extend the range of theories supported by our tool Comfusy. For instance, reasoning about multisets with cardinality constraints reduces to reasoning in linear integer arithmetic. A synthesis procedure for multisets will, thus, rely on a synthesis procedure for linear integer arithmetic. Since we already provided such a synthesis procedure in this thesis, multisets with cardinalities are an ideal candidate for inclusion into Comfusy. However, unlike sets with cardinality constraints, multisets with cardinalities do not admit quantifier elimination. We therefore need to develop new complete synthesis techniques that go beyond the techniques presented in this thesis.

In general, every decision procedures that outputs a model is amenable to transformation into a synthesis procedure. However, applying this transformation naively, without investigating the structure of the models and developing optimizations, can result in code that is too complex to be of practical use. We therefore plan to develop new algorithms that can give complexity guarantees for the synthesized code.

A Appendix A

To make our results on NP-completeness of logics of multisets more self-contained, this appendix gives the proof of Theorem 2.19.

Theorem 2.19, denoted Theorem 1 (ii) in [Eisenbrand and Shmonin(2006)]. *Let $X \subseteq \mathbb{Z}^d$ be a finite set of integer vectors and let $b \in X^*$. Then there exists a subset $\tilde{X}$ such that $b \in \tilde{X}^*$ and $|\tilde{X}| \leq 2d\log(4dM)$, where $M = \max_{x\in X} ||x||_\infty$.*

For $X \subseteq \mathbb{Z}^d$, a set of integer vectors, and a vector $b \in X^*$, the question is how many vectors from X are needed to generate b? If those were vectors with the real coefficients, Carathéodory theorem states that b is generated with at most d vectors [Schrijver(1998)]. However, in the integer case things are more complicated and the answer to this question was not known until relatively recently. In [Eisenbrand and Shmonin(2006)], Eisenbrand and Shmonin showed that in the integer case the number is not only bounded by d but that the size of the vectors in the set X also influences the bound.

We recall the definition of the norm infinity: for an integer vector $x = (x_1, \ldots, x_n)$, $||x||_\infty = \max\{|x_1|, \ldots, |x_n|\}$. In order to avoid possible confusions, for a set of vectors S we define $M_S = \max_{x\in X} ||x||_\infty$.

The first step towards a proof of the Theorem 2.19 is Lemma A.1. Given a set of integer vectors X and a vector b, Lemma A.1 establishes a bound on the size of the set. If the set is bigger than this bound, there is a smaller subset of X, which also generates b. The proof of the lemma relies only on the combinatorial arguments.

Lemma A.1 *Let $X \subseteq \mathbb{Z}^d$ be a set of integer vectors and let $b \in X^*$. If $|X| > d\log_2(2|X|M_X + 1)$, then there exists a proper subset $\tilde{X} \subset X$ such that $b \in \tilde{X}^*$.*

Proof. The fact that $b \in X^*$ means that $b = \sum_{x\in X} \lambda_x x,\ \lambda_x \geq 0$. Without loss of generality we consider only those λ_i that are non-zero. This way we represent b as $b = \sum_{x\in X} \lambda_x x,\ \lambda_x > 0$.

Let S be a subset of X and consider the vector $s = \sum_{x\in S} x$. Vector s is bounded: $||s||_\infty \leq |X|M_X$

and its coordinates are in the range $\{-|X|M_X,\ldots,|X|M_X\}$. This implies that the number of different vectors which are representable as the sum of vectors of $S \subseteq X$ is bounded by $(2|X|M_x+1)^d$.

The lemma assumption is that $2^{|X|} > (2|X|M_x+1)^d$. As $2^{|X|}$ is the number of the subsets of X, there are two different subsets, A and B, such that $\sum_{x\in A} x = \sum_{x\in B} x$. If A and B are not disjoint, we define new sets $A' = A \setminus (A\cup B)$ and $B' = B \setminus (A\cup B)$. Note that $\sum_{x\in A'} x = \sum_{x\in B'} x$.

Up to this point, using combinatorial arguments, we showed that there are two disjoint subsets $A, B \subseteq X$ such that $\sum_{x\in A} x = \sum_{x\in B} x$. Having A and B and the assumption $b = \sum_{x\in X} \lambda_x x,\ \lambda_x > 0$, we define value $\lambda = \min\{\lambda_x \mid x \in A\}$. We can now rewrite b by using λ.

$$\begin{aligned} b &= \sum_{x\in X} \lambda_x x = \sum_{x\in X\setminus A} \lambda_x x + \sum_{x\in A} \lambda_x x = \sum_{x\in X\setminus A} \lambda_x x + \sum_{x\in A} (\lambda_x-\lambda)x + \lambda \sum_{x\in A} x \\ &= \sum_{x\in X\setminus A} \lambda_x x + \sum_{x\in A} (\lambda_x-\lambda)x + \lambda \sum_{x\in B} x \end{aligned}$$

We define the new coefficients μ_x as follows: $\mu_x = \begin{cases} \lambda_x, & x \in X\setminus(A\cup B) \\ \lambda_x - \lambda, & x \in A \\ \lambda_x + \lambda, & x \in B \end{cases}$

Note that at least one of μ_x is zero and all of them are non-negative. We can further rewrite b as:

$$\begin{aligned} b &= \sum_{x\in X\setminus(A\cup B)} \lambda_x x + \sum_{x\in B} \lambda_x x + \sum_{x\in A} (\lambda_x-\lambda)x + \lambda \sum_{x\in B} x \\ &= \sum_{x\in X\setminus(A\cup B)} \lambda_x x + \sum_{x\in A} (\lambda_x-\lambda)x + \sum_{x\in B} (\lambda_x+\lambda) = \sum_{x\in X} \mu_x x \end{aligned}$$

This way we managed to show that b is a linear combination of vectors in X where at least one vector does not appear in that combination. We define set $\tilde{X} = \{x \in X \mid \mu_x > 0\}$. Note that $\tilde{X} \subset X$ and $b \in \tilde{X}^*$ which proves the lemma.

This result was crucial for the proof of Theorem 2.19:

Proof. [Theorem 2.19] Let $\tilde{X}$ be a minimal subset such that $b \in \tilde{X}^*$ and let us assume that $|\tilde{X}| > 2d\log_2(4dM_x)$. In the following Lemma A.2 we show that $|\tilde{X}| > 2d\log_2(4dM_x)$ implies $|\tilde{X}| > d\log_2(2|X|M_x+1)$. We apply previous Lemma A.1 and conclude that there exist X_1, a proper subset of $\tilde{X}$ such that $b \in X_1^*$. This contradicts the minimality of $\tilde{X}$, so we conclude that $|\tilde{X}| \leq 2d\log_2(4dM_x)$.

The last missing peace in the proof is to show that, if

$$|X| > 2d\log_2(4dM_x)$$

then $|X| > d\log_2(2|X|M_x+1)$. We prove that in the following lemma.

Lemma A.2 *Let $X \subseteq \mathbb{Z}^d$ be a set of integer vectors and let $M_x = \max_{x \in X} ||x||_\infty$. Suppose that $|X| > 2d \log_2(4dM_x)$. Then $|X| > d \log_2(2|X|M_x + 1)$.*

Proof. From the fact that $|X| > 2d \log_2(4dM_x)$ using simple rewriting follows that

$$M_x < 2^{|X|/(2d)}/(4d)$$

We multiply the both sides by $2|X|$ and add 1 afterwards, so we obtain: $2|X|M_x + 1 < |X|/(2d) * 2^{|X|/(2d)} + 1$. Since $1 \leq 2^{|X|/(2d)}$ we obtain:

$$2|X|M_x + 1 \leq 2^{|X|/(2d)}(|X|/(2d) + 1)$$

We first apply the $\log_2$ function to the both side, and then multiply by d. This results in $d \log_2(2|X|M_x + 1) < |X|/2 + d \log_2(|X|/(2d) + 1)$. For every number $y \geq 1, \log_2(y+1) \leq y$. From the fact that $|X| > 2d \log_2(4dM_x)$, we conclude that $|X| \geq 2d$, i.e. $|X|/(2d) \geq 1$. Combining all those facts together, we obtain formula:

$$d \log_2(2|X|M_x + 1) < |X|/2 + d * |X|/(2d) = |X|$$

This way we completed the proof of Theorem 2.19.

Bibliography

[Anand et al.(2008)Anand, Godefroid, and Tillmann] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[Andrews(2002)] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Springer (Kluwer), 2nd edition, 2002. ISBN ISBN 1402007639.

[Asarin et al.(1995)Asarin, Maler, and Pnueli] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20, 1995.

[Baader and Nipkow(1998)] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bachmair and Ganzinger(2001a)] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier, 2001a.

[Bachmair and Ganzinger(2001b)] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. MIT Press, 2001b.

[Ball et al.(2001)Ball, Majumdar, Millstein, and Rajamani] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[Ball et al.(2002)Ball, Podelski, and Rajamani] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS'02*, volume 2280 of *LNCS*, page 158, 2002.

[Ball et al.(2004)Ball, Cook, Levin, and Rajamani] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. Technical report, MSR-TR-2004-08, 2004.

[Balser et al.(2000)Balser, Reif, Schellhorn, Stenzel, and Thums] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *FASE*, number 1783 in LNCS, 2000.

[Banâtre and Métayer(1993)] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Commun. ACM*, 36(1):98–111, 1993. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/151233.151242.

[Banerjee(1988)] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988. ISBN 0898382890.

[Barnett et al.(2004a)Barnett, DeLine, Fähndrich, Leino, and Schulte] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004a.

[Barnett et al.(2004b)Barnett, Leino, and Schulte] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004b.

[Barnett et al.(2005)Barnett, Chang, DeLine, Jacobs, and Leino] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.

[Barrett and Tinelli(2007)] Clark Barrett and Cesare Tinelli. CVC3. In *CAV*, volume 4590 of *LNCS*, 2007.

[Basin and Friedrich(2000)] David Basin and Stefan Friedrich. Combining WS1S and HOL. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[Berdine et al.(2006)Berdine, Cook, Distefano, and O'Hearn] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, pages 386–400, 2006.

[Berdine et al.(2011)Berdine, Cook, and Ishtiaq] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[Berezin et al.(2003)Berezin, Ganesh, and Dill] Sergey Berezin, Vijay Ganesh, and David L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *TACAS*, 2003.

[Beyer et al.(2007)Beyer, Henzinger, Jhala, and Majumdar] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6): 505–525, 2007.

[Boigelot et al.(2005)Boigelot, Jodogne, and Wolper] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Trans. Comput. Logic*, 6(3):614–633, 2005. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/1071596.1071601.

[Börger et al.(1997)Börger, Grädel, and Gurevich] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[Bouajjani et al.(2011)Bouajjani, Drăgoi, Enea, and Sighireanu] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 578–589, 2011.

[Boyer and Moore(1988)] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence*, volume 11, pages 83–124. Oxford University Press, 1988.

[Bradley and Manna(2007)] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer, 2007.

[Breazu-Tannen et al.(1991)Breazu-Tannen, Coquand, Gunter, and Scedrov] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93:172–221, July 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90055-7.

[Bruttomesso et al.(2008)Bruttomesso, Cimatti, Franzén, Griggio, and Sebastiani] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4smt solver. In *CAV*, pages 299–303, 2008.

[Bruttomesso et al.(2010)Bruttomesso, Pek, Sharygina, and Tsitovich] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015, pages 150–153, Paphos, Cyprus, 2010. Springer, Springer. URL http://dx.doi.org/10.1007/978-3-642-12002-2_12.

[Bryant(1986)] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Cannon(2011)] Aemon Cannon. Ensime. https://github.com/aemoncannon/ensime/, 2011. Last checked August 2011.

[Clarke et al.(2004)Clarke, Kroening, and Lerda] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *TACAS*, pages 168–176, 2004.

[Colón and Sipma(2001)] Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *TACAS*, pages 67–81, 2001.

[Cook et al.(2005)Cook, Podelski, and Rybalchenko] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.

[Cook et al.(2006)Cook, Podelski, and Rybalchenko] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV*, pages 415–418, 2006.

[Cooper(1972)] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972.

[Cormen et al.(2001)Cormen, Leiserson, Rivest, and Stein] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.

[Damas and Milner(1982)] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, 1982.

[de Moura and Bjørner(2008a)] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008a. URL http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[de Moura and Bjørner(2008b)] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008b.

[Deng and Sangiorgi(2006)] Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Inf. Comput.*, 204(7):1045–1082, 2006.

[Dershowitz(1979)] Nachum Dershowitz. Orderings for term-rewriting systems. In *Symposium on Foundations of Computer Science (SFCS)*, pages 123–131, 1979.

[Dershowitz and Manna(1979)] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359138.359142.

[Dewar(1979)] Robert K. Dewar. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM TOPLAS*, July 1979.

[Dick et al.(1990)Dick, Kalmus, and Martin] Jeremy Dick, John Kalmus, and Ursula Martin. Automating the Knuth Bendix Ordering. *Acta Inf.*, 28(2):95–119, 1990.

[Dijkstra(1976)] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., 1976.

[Dutertre and de Moura(2006a)] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*, 2006a.

[Dutertre and de Moura(2006b)] Bruno Dutertre and Leonardo de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, 2006b.

[Eisenbrand and Shmonin(2008)] Friedrich Eisenbrand and Gennady Shmonin. Parametric integer programming in fixed dimension. *Math. Oper. Res.*, 33(4):839–850, 2008.

[Eisenbrand and Shmonin(2006)] Friedrich Eisenbrand and Gennady Shmonin. Carathéodory bounds for integer cones. *Operations Research Letters*, 34(5):564–568, September 2006. http://dx.doi.org/10.1016/j.orl.2005.09.008.

[Emir et al.(2007)Emir, Odersky, and Williams] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP*, 2007.

[Feferman and Vaught(1959)] S. Feferman and R. L. Vaught. The first order properties of products of algebraic systems. *Fundamenta Mathematicae*, 47:57–103, 1959.

[Ferrante and Rackoff(1979)] Jeanne Ferrante and Charles W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979.

[Filliâtre and Marché(2007)] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007. URL http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf.

[Flanagan et al.(2002)Flanagan, Leino, Lilibridge, Nelson, Saxe, and Stata] Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002.

[Floyd(1967)] Robert W. Floyd. Assigning meanings to programs. In *Proc. Amer. Math. Soc. Symposia in Applied Mathematics*, volume 19, pages 19–31, 1967.

[Fontaine(2007)] Pascal Fontaine. Combinations of theories and the bernays-schönfinkel-ramsey class. In *VERIFY*, 2007.

[Fontaine(2009)] Pascal Fontaine. Combinations of decidable fragments of first-order logic. In *FroCoS*, 2009.

[Ford and Havas(1996)] David Ford and George Havas. A new algorithm and refined bounds for extended gcd computation. In *ANTS*, pages 145–150, 1996.

[Gabbay and Ohlbach(1992)] Dov M. Gabbay and Hans Jürgen Ohlbach. Quantifier elimination in second-order predicate logic. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Principles of Knowledge Representation and Reasoning (KR92)*, pages 425–435. Morgan Kaufmann Publishers, Inc., 1992.

[Ge et al.(2007)Ge, Barrett, and Tinelli] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE*, 2007.

[Ghilardi(2005)] Silvio Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 33(3-4):221–249, 2005.

[Giles and Pulleyblank(1979)] F. R. Giles and W. R. Pulleyblank. Total dual integrality and integer polyhedra. *Linear Algebra and its Applications*, 25:191 – 196, 1979. ISSN 0024-3795.

[Ginsburg and Spanier(1964)] S. Ginsburg and E. Spanier. Bounded algol-like languages. *Transactions of the American Mathematical Society*, 113(2):333–368, 1964.

[Ginsburg and Spanier(1966)] S. Ginsburg and E. Spanier. Semigroups, Presburger formulas and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

[Givan et al.(2002)Givan, McAllester, Witty, and Kozen] Robert Givan, David McAllester, Carl Witty, and Dexter Kozen. Tarskian set constraints. *Inf. Comput.*, 174(2):105–131, 2002. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.2001.2973.

[Grädel et al.(1997)Grädel, Otto, and Rosen] Erich Grädel, Martin Otto, and Eric Rosen. Two-variable logic with counting is decidable. In *LICS*, 1997. URL http://www-mgi.informatik.rwth-aachen.de/Publications/pub/graedel/gorc2.ps.

[Green(1969)] Cordell Green. Application of theorem proving to problem solving. In *Proc. Int'l. Joint Conf. Artificial Intelligence*, pages 219–239. Morgan Kaufmann, 1969.

[Gvero et al.(2011)Gvero, Kuncak, and Piskac] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[Hodges(1993)] Wilfrid Hodges. *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993.

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

[Jacobs(2009)] Swen Jacobs. Incremental instance generation in local reasoning. In *CAV*, pages 368–382, 2009.

[Jaffar and Maher(1994)] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[Jha et al.(2010)Jha, Gulwani, Seshia, and Tiwari] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, 2010.

[Jobstmann and Bloem(2006)] Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *FMCAD*, 2006.

[Jobstmann et al.(2007)Jobstmann, Galler, Weiglhofer, and Bloem] Barbara Jobstmann, Stefan Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In *CAV*, 2007.

[Jones et al.(1993)Jones, Gomard, and Sestoft] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. http://www.dina.kvl.dk/ sestoft/pebook/pebook.html, 1993.

[Jones and group of authors(2010)] Simon Peyton Jones and group of authors. Haskell 98 language and libraries: The revised report, November 2010. URL http://haskell.org/onlinereport.

[Jovanovic and Barrett(2010)] Dejan Jovanovic and Clark Barrett. Polite theories revisited. In *LPAR (Yogyakarta)*, pages 402–416, 2010.

[Klaedtke(2003)] Felix Klaedtke. On the automata size for presburger arithmetic. Technical Report 186, Institute of Computer Science at Freiburg University, 2003.

[Klaedtke and Rueß(2003)] Felix Klaedtke and Harald Rueß. Monadic second-order logics with cardinalities. In *ICALP*, volume 2719 of *LNCS*, 2003.

[Klaedtke and Rueß(2002)] Felix Klaedtke and Harald Rueß. Parikh automata and monadic second-order logics with linear cardinality constraints. Technical Report 177, Institute of Computer Science at Freiburg University, 2002.

[Klarlund and Møller(2001)] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[Klarlund and Schwartzbach(1993)] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[Köksal et al.(2011)Köksal, Kuncak, and Suter] Ali Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of z3: Integrating smt and programming. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer Berlin / Heidelberg, 2011.

[Korovin(2009)] Konstantin Korovin. Instantiation-based automated reasoning: From theory to practice. In *CADE*, pages 163–166, 2009.

[Korovin and Voronkov(2001)] Konstantin Korovin and Andrei Voronkov. Knuth-bendix constraint solving is np-complete. In *ICALP*, pages 979–992, 2001.

[Kroening and Strichman(2008)] Daniel Kroening and Ofer Strichman. *Decision Procedures – an Algorithmic Point of View*. EATCS. Springer, 2008.

[Krstic et al.(2007)Krstic, Goel, Grundy, and Tinelli] Sava Krstic, Amit Goel, Jim Grundy, and Cesare Tinelli. Combined satisfiability modulo parametric theories. In *TACAS*, volume 4424 of *LNCS*, pages 602–617, 2007.

[Kukula and Shiple(2000)] James H. Kukula and Thomas R. Shiple. Building circuits from relations. In *CAV*, 2000.

[Kuncak(2007)] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[Kuncak and Rinard(2003)] Viktor Kuncak and Martin Rinard. On the theory of structural subtyping. Technical Report 879, LCS, Massachusetts Institute of Technology, 2003.

[Kuncak and Rinard(2007)] Viktor Kuncak and Martin Rinard. Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In *CADE-21*, 2007.

[Kuncak and Wies(2009)] Viktor Kuncak and Thomas Wies. On set-driven combination of logics and verifiers. Technical Report LARA-REPORT-2009-001, EPFL, February 2009.

[Kuncak et al.(2005)Kuncak, Nguyen, and Rinard] Viktor Kuncak, Hai Huu Nguyen, and Martin Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *20th International Conference on Automated Deduction, CADE-20*, Tallinn, Estonia, July 2005.

[Kuncak et al.(2006)Kuncak, Nguyen, and Rinard] Viktor Kuncak, Hai Huu Nguyen, and Martin Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. http://dx.doi.org/10.1007/s10817-006-9042-1.

[Kuncak et al.(2010a)Kuncak, Mayer, Piskac, and Suter] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A tool for complete functional synthesis. In *CAV*, pages 430–433, 2010a.

[Kuncak et al.(2010b)Kuncak, Mayer, Piskac, and Suter] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010b.

[Kuncak et al.(2010c)Kuncak, Piskac, and Suter] Viktor Kuncak, Ruzica Piskac, and Philippe Suter. Ordered sets in the calculus of data structures. In *CSL*, pages 34–48, 2010c.

[Kuncak et al.(2010d)Kuncak, Piskac, Suter, and Wies] Viktor Kuncak, Ruzica Piskac, Philippe Suter, and Thomas Wies. Building a calculus of data structures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2010d.

[Lahiri and Seshia(2004)] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID decision procedure. In *CAV'04*, 2004.

[Lahiri et al.(2009)Lahiri, Qadeer, Galeotti, Voung, and Wies] Shuvendu K. Lahiri, Shaz Qadeer, Juan P. Galeotti, Jan W. Voung, and Thomas Wies. Intra-module inference. In *CAV*, pages 493–508, 2009.

[Lee et al.(2001)Lee, Jones, and Ben-Amram] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

[Lewis(1980)] Harry R. Lewis. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci.*, 21(3):317–353, 1980.

[Lugiez(2005)] D. Lugiez. Multitree automata that count. *Theor. Comput. Sci.*, 333(1-2):225–263, 2005. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2004.10.023.

[Lugiez and Zilio(2002)] Denis Lugiez and Silvano Dal Zilio. Multitrees Automata, Presburger's Constraints and Tree Logics. Research report 08-2002, LIF, Marseille, France, June 2002. http://www.lif-sud.univ-mrs.fr/Rapports/08-2002.html.

[Luo(2008)] Zhaohui Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008.

[Mandelin et al.(2005)Mandelin, Xu, Bodík, and Kimelman] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, 2005.

[Manna and Waldinger(1980)] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357084.357090.

[Manna and Waldinger(1971)] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

[Martín-Mateos et al.(2005)Martín-Mateos, Ruiz-Reina, Alonso, and Hidalgo] Francisco-Jesús Martín-Mateos, José-Luis Ruiz-Reina, José-Antonio Alonso, and María-José Hidalgo. Proof Pearl: A Formal Proof of Higman's Lemma in ACL2. In *TPHOLs*, pages 358–372, 2005.

[Matiyasevich(1970)] Yuri V. Matiyasevich. Enumerable sets are Diophantine. *Soviet Math. Doklady*, 11(2):354–357, 1970.

[McLaughlin et al.(2006)McLaughlin, Barrett, and Ge] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *PDPAR*, volume 144(2) of *ENTCS*, 2006.

[Monniaux(2009)] David P. Monniaux. Automatic modular abstractions for linear constraints. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 140–151, 2009.

[Moskal(2009)] Michał Moskal. *Satisfiability Modulo Software*. PhD thesis, University of Wrocław, 2009.

[Narendran et al.(1998)Narendran, Rusinowitch, and Verma] Paliath Narendran, Michaël Rusinowitch, and Rakesh M. Verma. RPO Constraint Solving is in NP. In *CSL*, pages 385–398, 1998.

[Nelson and Oppen(1979)] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357073.357079.

[Nelson and Oppen(1980)] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322186.322198.

[Nguyen et al.(2007)Nguyen, David, Qin, and Chin] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape, size and bag properties via separation logic. In *VMCAI*, 2007.

[Nieuwenhuis(1993)] Robert Nieuwenhuis. Simple LPO constraint solving methods. *Inf. Process. Lett.*, 47(2):65–69, 1993. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(93)90226-Y.

[Nipkow(2008)] Tobias Nipkow. Linear quantifier elimination. In *IJCAR*, 2008.

[Nipkow et al.(2005)Nipkow, Wenzel, Paulson, and Voelker] Tobias Nipkow, Markus Wenzel, Lawrence C Paulson, and Norbert Voelker. Multiset theory version 1.30 (Isabelle distribution). http://isabelle.in.tum.de/dist/library/HOL/Library/Multiset.html, 2005.

[Odersky et al.(2008)Odersky, Spoon, and Venners] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[Owre et al.(1992)Owre, Rushby, and Shankar] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th CADE*, volume 607 of *LNAI*, pages 748–752, jun 1992.

[Pacholski et al.(2000)Pacholski, Szwast, and Tendera] Leszek Pacholski, Wieslaw Szwast, and Lidia Tendera. Complexity results for first-order two-variable logic with counting. *SIAM J. on Computing*, 29(4):1083–1117, 2000.

[Papadimitriou(1981)] Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322276.322287.

[Parikh(1966)] Rohit J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321356.321364.

[Piskac and Kuncak(2008a)] Ruzica Piskac and Viktor Kuncak. Decision procedures for multisets with cardinality constraints. In *VMCAI*, number 4905 in LNCS, 2008a.

[Piskac and Kuncak(2008b)] Ruzica Piskac and Viktor Kuncak. Fractional collections with cardinality bounds. In *Computer Science Logic (CSL)*, 2008b.

[Piskac and Kuncak(2008c)] Ruzica Piskac and Viktor Kuncak. Linear arithmetic with stars. In *CAV*, 2008c.

[Piskac and Kuncak(2010)] Ruzica Piskac and Viktor Kuncak. MUNCH - automated reasoner for sets and multisets. In *IJCAR*, number 6173 in LNCS, pages 149–155, 2010.

[Piskac and Wies(2011)] Ruzica Piskac and Thomas Wies. Decision procedures for automating termination proofs. In *VMCAI*, pages 371–386, 2011.

[Piterman et al.(2006)Piterman, Pnueli, and Sa'ar] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, 2006.

[Pnueli and Rosner(1989)] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: http://doi.acm.org/10.1145/75277.75293.

[Podelski and Rybalchenko(2004a)] Andreas Podelski and Andrey Rybalchenko. A complete method for synthesis of linear ranking functions. In *VMCAI'04*, 2004a.

[Podelski and Rybalchenko(2004b)] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS'04*, 2004b.

[Podelski and Rybalchenko(2007a)] Andreas Podelski and Andrey Rybalchenko. Armc: The logical choice for software model checking with abstraction refinement. In *PADL*, pages 245–259, 2007a.

[Podelski and Rybalchenko(2007b)] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. *ACM TOPLAS*, 29(3):15, 2007b.

[Podelski and Wies(2010)] Andreas Podelski and Thomas Wies. Counterexample-guided focus. In *POPL*, pages 249–260, 2010.

[Pottier(1991)] Loïc Pottier. Minimal solutions of linear diophantine systems: Bounds and algorithms. In *RTA*, volume 488 of *LNCS*, 1991.

[Pratt-Hartmann(2005)] Ian Pratt-Hartmann. Complexity of the two-variable fragment with counting quantifiers. *Journal of Logic, Language and Information*, 14(3):369–395, 2005.

[Pratt-Hartmann(2004)] Ian Pratt-Hartmann. Complexity of the two-variable fragment with (binary-coded) counting quantifiers. *CoRR*, cs.LO/0411031, 2004.

[Presburger(1929)] Mojzesz Presburger. Über die vollständigkeit eines gewissen systems der aritmethik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du premier Congrès des Mathématiciens des Pays slaves, Warsawa*, pages 92–101, 1929.

[Pugh(1992)] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/135226.135233.

[Ramsey(1930)] F. P. Ramsey. On a problem of formal logic. *Proc. London Math. Soc.*, s2-30: 264–286, 1930. doi:10.1112/plms/s2-30.1.264.

[Rehof and Urzyczyn(2011)] Jakob Rehof and Pawel Urzyczyn. Finite combinatory logic with intersection types. In *TLCA*, pages 169–183, 2011.

[Reynolds(2002)] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th LICS*, pages 55–74, 2002.

[Reynolds(1980)] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, pages 211–258, 1980.

[Riazanov and Voronkov(2002)] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2-3):91–110, 2002.

[Robinson(1965)] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12, January 1965.

[Sahavechaphan and Claypool(2006)] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. In *OOPSLA*, 2006. ISBN 1-59593-348-4.

[Schrijver(1998)] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.

[Schulz(2002)] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[Schwartz(1973)] J. T. Schwartz. On programming: An interim report on the SETL project. Technical report, Courant Institute, New York, 1973.

[Sekar et al.(1995)Sekar, Ramesh, and Ramakrishnan] R.C. Sekar, R. Ramesh, and I.V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal on Computing*, 24:1207–1234, December 1995.

[Sharir(1982)] Micha Sharir. Some observations concerning formal differentiation of set theoretic expressions. *Transactions on Programming Languages and Systems*, 4(2), April 1982.

[Sofronie-Stokkermans(2005)] Viorica Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *CADE*, pages 219–234, 2005.

[Sofronie-Stokkermans and Ihlemann(2007)] Viorica Sofronie-Stokkermans and Carsten Ihlemann. Automated reasoning in some local extensions of ordered structures. In *ISMVL*, 2007.

[Solar-Lezama et al.(2006)Solar-Lezama, Tancau, Bodík, Seshia, and Saraswat] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[Solar-Lezama et al.(2007)Solar-Lezama, Arnold, Tancau, Bodík, Saraswat, and Seshia] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, 2007.

[Solar-Lezama et al.(2008)Solar-Lezama, Jones, and Bodík] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

[Srivastava et al.(2010)Srivastava, Gulwani, and Foster] Saurabh Srivastava, Sumit Gulwani, and Jeff Foster. From program verification to program synthesis. In *POPL*, 2010.

[Statman(1979)] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67 – 72, 1979.

[Suter et al.(2010)Suter, Dotta, and Kuncak] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2010.

[Syme et al.(2007)Syme, Granicz, and Cisternino] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

[Thatcher and Wright(1968)] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, August 1968.

[Tinelli and Ringeissen(2003)] Cesare Tinelli and Christophe Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, 290(1):291–353, January 2003. URL ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/TinRin-TCS-03.ps.gz.

[Tinelli and Zarba(2003)] Cesare Tinelli and Calogero Zarba. Combining non-stably infinite theories. In I. Dahn and L. Vigneron, editors, *Proceedings of the 4th International Workshop on First Order Theorem Proving, FTP'03 (Valencia, Spain)*, volume 86.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[Tinelli and Zarba(2005)] Cesare Tinelli and Calogero Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3), 2005.

[Turing(1949)] Alan M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Computation, June 1949*, pages 67–69. University Mathematical Laboratory, Cambridge University, 1949.

[Typesafe(2011)] Typesafe. The Scala programing language. http://www.scala-lang.org, 2011. Last accessed August 2011.

[Vechev et al.(2007)Vechev, Yahav, Bacon, and Rinetzky] Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzky. Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI*, pages 456–467, 2007. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250787.

[Vechev et al.(2009)Vechev, Yahav, and Yorsh] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Inferring synchronization under limited observability. In *TACAS*, 2009.

[Venkataraman(1987)] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM (JACM)*, 34(2):492–510, 1987. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/23005.24037.

[Weidenbach et al.(2009)Weidenbach, Dimova, Fietzke, Kumar, Suda, and Wischnewski] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In Renate Schmidt, editor, *Automated Deduction – CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer Berlin / Heidelberg, 2009.

[Weispfenning(1997)] Volker Weispfenning. Complexity and uniformity of elimination in presburger arithmetic. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 48–53. ACM Press, 1997. ISBN 0-89791-875-4. doi: http://doi.acm.org/10.1145/258726.258746.

[Wies(2009)] Thomas Wies. *Symbolic Shape Analysis*. PhD thesis, University of Freiburg, 2009.

[Wies et al.(2006)Wies, Kuncak, Lam, Podelski, and Rinard] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.

[Wies et al.(2009)Wies, Piskac, and Kuncak] Thomas Wies, Ruzica Piskac, and Viktor Kuncak. Combining theories with shared set operations. In *FroCoS: Frontiers in Combining Systems*, 2009.

[Yessenov et al.(2010)Yessenov, Piskac, and Kuncak] Kuat Yessenov, Ruzica Piskac, and Viktor Kuncak. Collections, cardinalities, and relations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2010.

[Zadeh(1965)] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[Zarba(2002a)] Calogero G. Zarba. Combining multisets with integers. In *CADE-18*, 2002a.

[Zarba(2002b)] Calogero G. Zarba. A tableau calculus for combining non-disjoint theories. In *TABLEAUX '02: Proc. Int. Conf. Automated Reasoning with Analytic Tableaux and Related Methods*, pages 315–329, 2002b.

[Zarba(2004)] Calogero G. Zarba. A quantifier elimination algorithm for a fragment of set theory involving the cardinality operator. In *18th International Workshop on Unification*, 2004.

[Zarba(2005)] Calogero G. Zarba. Combining sets with cardinals. *J. of Automated Reasoning*, 34(1), 2005.

[Zee et al.(2008)Zee, Kuncak, and Rinard] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2008.

[Zhang et al.(2005)Zhang, Sipma, and Manna] Ting Zhang, Henny B. Sipma, and Zohar Manna. The decidability of the first-order theory of knuth-bendix order. In *CADE*, pages 131–148, 2005.

Curriculum Vitae

Ruzica Piskac

address: EPFL
School of Computer & Communications Sciences
Station 14
CH-1015 Lausanne
Switzerland

phone: +41 21 69 31221
fax: +41 21 69 36660
email: ruzica.piskac@epfl.ch
web: http://icwww.epfl.ch/~piskac/

Research Interests

Formal methods, decision procedures for software verification and code synthesis, automated reasoning, theorem proving, semantic web

Education

- [09/2007 – 12/2011] Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland
  - PhD student at EPFL, Laboratory for Automated Reasoning and Analysis (LARA) group.
  - thesis title: "Decision Procedures for Program Synthesis and Verification"
  - thesis advisor: Viktor Kuncak
- [10/2002 – 08/2006] Max-Planck-Institut für Informatik, Saarbrücken, Germany
  - first a master student and afterwards a PhD student at MPII, Programming Logics Group, advisors: Harald Ganzinger[†], Andreas Podelski, Hans de Nivelle
  - Master's thesis: "Formal Correctness of Result Checking for Priority Queues"
  - finished Master program with the best grades (1.0/1.0)
- [09/1995 – 12/2000] University of Zagreb, Croatia
  - student of mathematics at the University of Zagreb, Croatia, advisor: Robert Manger.
  - Diploma thesis: "Parallel Algorithms for Sorting and Merging" (in Croatian).
  - Dipl.Ing. degree with Honors, graduated as the first in the class of 81 students

Work Experience

- [June – September 2008] Microsoft Research, Redmond, WA, USA
  - summer internship, mentor: Nikolaj Bjørner
- [08/2006 – 08/2007] Digital Enterprise Research Institute (DERI), Innsbruck, Austria
  - junior researcher at DERI, Intelligent Reasoning for Integrated Systems (IRIS) cluster
  - research topic: developing a reasoner for a language that uniformly supports ontologies and rules.
  - working in the following EU- or Austrian-funded projects: SEKT (EU IST IP 2003-506826), RW2 and Knowledge Web FP6-507482
- [12/2000 – 09/2002] University of Zagreb, Croatia
  - junior researcher and teaching assistant at the University of Zagreb
  - collaboration with CERN: participating in ALICE project (A Large Ion Collider Experiment)
  - managing a cluster for scientific computing
- [06/2000 – 09/2000] Tourist Office Dubrovnik, Croatia
  - certified tourist guide

Honors and Awards

- The Google Anita Borg Memorial Scholarship 2010 winner
- Grace Hopper Conference attendance full scholarship recipient 2008 and 2007
- [2002 – 2006] IMPRS Fellowship for both Master and Ph.D. studies in Computer Science. Max-Planck Institut für Informatik, Saarbrücken, Germany
- Won the most prestigious student's award of Croatia: Chancellor's Award ("Rektorova nagrada"), 2000. for the work in the area of decision making theory.

Publications

1. T. Gvero, V. Kuncak, R. Piskac. **Interactive Synthesis of Code Snippets**. *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, Springer, LNCS Volume 6806, p. 418-423.
2. R. Piskac, T. Wies. **Decision Procedures for Automating Termination Proofs**. *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, Springer, LNCS Volume 6538, p. 371-386.
3. V. Kuncak, R. Piskac, P. Suter. **Ordered Sets in the Calculus of Data Structures**. *Proceedings of the 19th Annual Computer Science Logic (CSL 2010)*, Springer, LNCS Volume 6247, p. 34-48.
4. R. Piskac, V.Kuncak. **MUNCH - Automated Reasoner for Sets and Multisets**. *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR 2010)*, Springer, LNAI 6173, p. 149-155.
5. V. Kuncak, M. Mayer, R. Piskac, P. Suter. **Comfusy: A Tool for Complete Functional Synthesis**. *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010)*, Springer, LNCS Volume 6174, p. 430-433.
6. V. Kuncak, M. Mayer, R. Piskac, P. Suter. **Complete Functional Synthesis**. *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI)*, p. 316-329.
7. R. Piskac, L. de Moura, N. Bjørner. **Deciding Effectively Propositional Logic Using DPLL and Substitution Sets** , Journal of Automated Reasoning, DOI: 10.1007/s10817-009-9161-6, Volume 44, Number 4, p. 401-424, April 2010.
8. V. Kuncak, R. Piskac, P. Suter, T. Wies: **Building a Calculus of Data Structures**. *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2010)*, Springer, LNCS Volume 5944, p. 26-44.
9. K. Yessenov, R. Piskac, V. Kuncak. **Collections, Cardinalities, and Relations**. *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2010)*, Springer, LNCS Volume 5944, p. 380-395.
10. T.Hillenbrandt, R.Piskac, U. Waldmann, C. Weidenbach. **From Search to Computation: Redundancy Criteria and Simplification at Work**. Accepted for *Harald Ganzinger Memorial volume*. To appear in 2010.
11. T. Wies, R. Piskac, V. Kuncak. **Combining Theories with Shared Set Operations**. *Proceedings of the 7th International Symposium on Frontiers of Combining Systems (FroCoS 2009)*, Springer, LNCS Volume 5749, p. 366-382.
12. R. Piskac, L. de Moura, N. Bjørner. **Deciding Effectively Propositional Logic with Equality**, Technical Report, MSR-TR-2008-181, December 2008.
13. R. Piskac, V. Kuncak. **Fractional Collections with Cardinality Bounds**. *Proceedings of the 17th Annual Computer Science Logic (CSL 2008)*, Springer, LNCS Volume 5213, p. 124-138.
14. R. Piskac, V. Kuncak. **Linear Arithmetic with Stars**. *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, Springer, LNCS 5123, p. 268-280.
15. R. Piskac, V. Kuncak. **Decision Procedures for Multisets with Cardinality Constraints**. *Proceedings of he Ninth International Conference on Verification, Model Checking and Abstract Interpretation - VMCAI 2008*, Springer, LNCS Volume 4905, p. 218-232.
16. H. de Nivelle, R. Piskac. **Verification of an Off-Line Checker for Priority Queues**.*Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, Koblenz, IEEE computer society press, Washington, 2005, 210-219.
17. P. Saiz, L. Aphecetche, P. Buncic, R. Piskac, J.-E. Revsbech, V. Sego. 2003. **AliEn-ALICE environment on the GRID**.*Nuclear Instruments and Methods in Physics Research* Section A, Volume 502, Issue 2-3, p. 437-440.
18. L. Caklovic, R. Piskac, V. Sego. 2001. **Improvement of AHP method**. *Mathematical Communications* - Supplement No.1 (2001), 13-21.

Proceedings Editor

1. R. Piskac, F. van Harmelen, N. Zhong. Proceedings of the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference Workshop on New forms of reasoning for the Semantic Web: scalable, tolerant and dynamic, Busan, Korea, November 2007.
2. F. van Harmelen, A. Herzig, P. Hitzler, Z. Lin, R. Piskac, G. Qi. Proceedings of the 5th European Semantic Web Conference Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (AReа-2008), Tenerife, Spain, June, 2008.
3. R. Piskac, E. Simperl. Proceedings of the Doctoral Consortium of the 3rd Future Internet Symposium 2010, Berlin, Germany, September 23-24, 2010.

Professional Activities

- Organizer and PC co-Chair
    1. New forms of reasoning for the Semantic Web: scalable, tolerant and dynamic, workshop co-located with the 6th International Semantic Web Conference and the 2nd Asian Semantic Web Conference, Busan, Korea, November 2007. Co-organizer with F. van Harmelen and N. Zhong
    2. Advancing Reasoning on the Web: Scalability and Commonsense, workshop co-located with the 5th European Semantic Web Conference, Tenerife, Spain, June 2008. Co-organizer with F. van Harmelen, A. Herzig, P. Hitzler, Z. Lin and G. Qi
    3. Doctoral Consortium of the 3rd Future Internet Symposium 2010, Berlin, Germany, September 2010. Education co-Chair together with E. Simperl
- Program Committees
    1. The 18th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-18) - Program Committee Member
    2. STI International Symposium 2010 - Organizing Committee Member
    3. The 4th Asian Semantic Web Conference (ASWC 2009) - Program Committee Member
    4. The 3rd Asian Semantic Web Conference (ASWC 2008) - Program Committee Member
    5. The Second International Workshop on New forms of reasoning for the Semantic Web: scalable, tolerant and dynamic (NeForS08) - Program Committee Member
- Reviewer for
    - Journal of Symbolic Computation, STTT Journal, International Conference on Automated Deduction (CADE 2011), European Symposium on Programming (ESOP2011), Symposium on Principles of Programming Languages (POPL 2011), International Conference on Computer Aided Verification (CAV 2010, CAV 2011), Computer Science Symposium in Russia (CSR 2010), Static Analysis Symposium (SAS 2009, SAS 2011), Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009), Asian Semantic Web Conference (ASWC 2009, ASWC 2008), Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-08, LPAR-07), International Joint Conference on Automated Reasoning (IJCAR-08), Workshop on Web Semantics (WebS2007), International World Wide Web Conference (WWW2007), Conference on Information Integration and Web-based Applications & Services (iiWAS2006)
- [September 2008 - present] Managing editor of the CEUR-WS.org proceedings series (`http://CEUR-WS.org/`)
    - over 800 published conference and workshop proceedings

Presentations

- Conference Presentations
    - CAV 2011, SMT 2011, POPL 2011 (student session), IJCAR 2010, CSL 2008, CAV 2008, VMCAI 2008
- Invited Talks
    - gave talks about research results at the Rich models toolkit meeting, Turin (2011), Workshop on Synthesis, Verification, and Analysis of Rich Models - SVARM, Saarbrücken (2011), New York University (2011), Caltech (2011), Microsoft Research Cambridge (2011), Max-Planck Institute for Software Systems (2011), Seminar on "Deduction at Scale 2011" at the Ringberg Castle (2011), RiSE Seminar, TU Vienna (2011), Student

Seminar, University of Freiburg (2011), LogicBlox, Atlanta (2011), Alpine Verification Meeting, Lugano (2010), Dagstuhl seminar 10161 (2010), Université Libre de Bruxelles (2010), Research seminar of the STI International Symposium (2010), Workshop on Formal and Automated Theorem Proving and Applications, Belgrade (2010), The IMDEA Software Institute, Madrid (2010), Meeting of the Action IC0701, Eindhoven (2009), Colorado University, Boulder (2008), Microsoft Research, Redmond (2008), SRI International (2008), Dagstuhl seminar 07401 (2007), University of Zagreb (2007), University of Freiburg (2006), University of Innsbruck (2006), Deduktionstreffen, Koblenz (2005), Dagstuhl seminar 05431 (2005), Deduktionstreffen, Saarbrücken (2004), Logic Seminar at Schloss Ringberg (2003)

- Poster Presentations
  - presented posters at EPFL Research Day (2010 and 2008), Microsoft Research Summer School, Cambridge (2010), The Technical Poster Competition of The Grace Hopper Celebration (2008) - one of four posters that was qualified in the finals, all together more than 100 submissions; "Femmes de Sciences" at EPFL (2008)

Teaching and Supervising Experience

- Teaching
  - lecturer at the *First International SAT/SMT Solver Summer School 2011*, MIT, June 2011.
  - *Vertiefungsseminar*, on the topic of Linked Open Data, University of Innsbruck, Spring 2011 - scientific organizer [3)]
  - *Seminar on Automated Reasoning*, EPFL, Fall 2010 - head teaching assistant [1)]
    Additional duties included preparing course materials, teaching and student supervision
  - *Synthesis, Analysis, and Verification*, EPFL, Spring 2010, 2009, 2008 - head teaching assistant [1)]
    (Spring 2008: the course received the highest grade among master courses, student evaluation: 5.4/6.0)
  - *Vertiefungsseminar*, University of Innsbruck, Spring 2010 - scientific organizer [3)]
  - *Compiler Construction*, EPFL, Fall 2008/09 - head teaching assistant [1)]
  - *Seminar "Master Seminar"*, University of Innsbruck, Fall 2007/08 - scientific organizer [3)]
  - *Seminar "Semantic Systems"*, University of Innsbruck, Spring 2007 - scientific organizer [3)]
  - *Seminar "The Role of Computer Science in Science"*, University of Innsbruck, Fall 2006/07 - scientific organizer [3)]
  - *Introduction to Proof Theory*, University of Saarbrücken, Fall 2005/06 - head teaching assistant [1)]
  - *Verification*, University of Saarbrücken, Fall 2004/05 - head teaching assistant [1)]
  - *Automated Reasoning*, University of Saarbrücken, Spring 2004 - teaching assistant [2)]
    The course was awarded the Teaching Award of the Computer Science Students Association for the summer semester 2004. Duties included grading and leading weekly exercises
  - Explanations:
    [1)] duties of a head teaching assistant include designing and leading weekly exercise sessions, grading
    [2)] duties of a teaching assistant include leading weekly exercise sessions, grading
    [3)] duties of a scientific organizer include topic assignment, student supervision, grading written reports, event organization

- Supervised Students
  - Mikaël Mayer: "Complete Program Synthesis for Linear Arithmetics", Master Thesis, EPFL, 2010

Language Skills

- Croatian, English, German, Russian

References

- References available upon request