Programming with Specifications

THÈSE N^{O} 5581 (2012)

PRÉSENTÉE LE 7 DÉCEMBRE 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ANALYSE ET DE RAISONNEMENT AUTOMATISÉS
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Philippe Paul Henri SUTER

acceptée sur proposition du jury:

Prof. C. Koch, président du jury
Prof. V. Kuncak, directeur de thèse
Dr N. Bjørner, rapporteur
Prof. T. Henzinger, rapporteur
Prof. M. Odersky, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

Acknowledgements

I would like to thank for advisor Viktor Kuncak for five years of guidance throughout my graduate studies. The enthusiasm and energy Viktor devotes to research are particularly contagious. I cannot say with certainty that I would have pursued a PhD, had I not met him in 2007. To this day, I am still equally impressed by his knowledge and humility. It was a real pleasure being his student. I am grateful to Nikolaj Bjørner, Tom Henzinger, Christoph Koch, and Martin Odersky for taking the time to serve on my thesis committee, for their advice on this dissertation, and for their excellent questions during the defense.

Without my co-authors, parts of this dissertation would not exist : I thank Ruzica Piskac and Mikaël Mayer for initiating the research on synthesis procedures, and Swen Jacobs for his contribution to reduction-based synthesis. The combination of these efforts forms the basis of Chapter 6. I thank Ali Sinan Köksal for his help with the development and evaluation of Leon, presented in Chapter 3. I also thank him for the many fruitful discussions on the design of Kaplan, described in Chapter 5, and for his work on the implementation. I thank Robin Steiger for his collaboration on the decision procedure presented in Chapter 4.

My studies gave me the chance to interact with experts from around the world. For their help and inspiration, I thank Sergio Antoy, Nikolaj Bjørner, Rupak Majumdar, Pete Manolios, David Monniaux, J. Strother Moore, and Leonardo de Moura. I owe a particular thank you to Mike Whalen for his detailed feedback on technical aspects of chapters 2 and 3.

I thank the members of the LARA group, past or present, who made my work more enjoyable in many different ways : Ali, Andrej, Etienne, Eva, Filip, Giuliano, Hossein, Mikaël, Mirco, Ravi, Ruzica, Régis, Swen, Tihomir, and Utkarsh. I thank the members of the LAMP group, including Adriaan, Alex, Gilles, Ingo, Iulian, Lukas, Miguel, and Philipp, for their help in taming Scala and its compiler, and for the stimulating conversations at the coffee machine. I thank Danielle Chamberlain and Yvette Gallay for their mastery of the bureaucratic ways, and in particular for their patience when dealing with me. I thank Fabien Salvi for keeping our technical infrastructure running, and for rescuing me whenever my urge to switch to the latest technology proved damaging.

I thank my parents for their continuous support. In particular, my father introducing me to programming on his HP-85 some twenty years ago most certainly has something to do with where I am now.

Finally of course, I thank Violaine.

Lausanne, November 2012 P. S.

Preface

Formal specifications have at some point acquired a bad reputation as being heavy-weight and hardly cost effective. However, new generations of constraint solving and verification tools emerging in recent years are rapidly changing this view. This thesis provides an outlook into the near future where developers will no longer question the usefulness of specifications in software validation. This is not merely a matter of educating programmers, but comes through fundamental improvements to the usability of tools. First of all, the specifications considered here are not in a notation foreign to programmers, but are written in a programming language itself, making it easy for developers to write. Like in model checking for finite-state systems, the tool gives concrete counterexamples in cases of wrong code or wrong specifications. As a result, the specifications and code are guided by each other and help the developer navigate the space of reasonable programs much more effectively. Unlike heuristic bug-finding tools, these approaches can also establish rigorous proofs of correctness, using symbolic reasoning to replace an infinite number of test cases. In particular, a roadblock to reasoning about software correctness is induction. This thesis shows that seemingly inductive properties do not always require explicit induction with heuristically chosen inductive steps to be solved, but can be tackled using systematic approaches of decision procedures.

Rarely do we find results that are non-obvious, but end up having very elegant solutions that work well in practice. Such results can be discovered by opportunistically pursuing natural problems that we do not understand yet and building tools that validate the algorithmic and theoretical ideas. What sets this thesis apart is a remarkable interplay between motivation from concrete programming tasks, theoretical ideas that apply broadly, and experimental validation that shows the algorithms and theory make a real difference in practice.

Once we accept that specifications and programs are merely different pieces in describing the same object —the intended functionality, the desirability of executing and compiling specifications becomes a natural goal. The thesis makes exciting steps into this direction, first showing what expressive constraint programming becomes in the age of modern SMT (satisfiability modulo theory) solvers integrated into high-productivity languages like Scala. The work on the constraint programming system, Kaplan, establishes a bar in what we should expect today : a language that does not compromise on types, higher-order features, interoperability, and a constraint solving technology that leverages advances such as conflict-driven clause learning and domain-specific reasoning in e.g. integer linear programming and reasoning about finitely generated data structures.

As a way to point to the future, one can consider bridging the gap between the formal requirements and efficient code by compiling input/output relations into computable functions that realize them, through the process termed complete functional synthesis. The gap between specifications and efficient code will perhaps never be fully automated, but each aspect of its automation can bring huge savings in software development efforts.
The agenda of going from specifications to executable code is as relevant as ever before, and the tools at our disposal to solve it are starting to be up to the task. This dissertation shows the results that make this feasible.

Lausanne, November 2012 Viktor Kuncak

Abstract

This thesis explores the use of specifications for the construction of correct programs. We go beyond their standard use as run-time assertions, and present algorithms, techniques and implementations for the tasks of 1) program verification, 2) declarative programming and 3) software synthesis. These results are made possible by our advances in the domains of decision procedure design and implementation.

In the first part of this thesis, we present a decidability result for a class of logics that support user-defined recursive function definitions. Constraints in this class can encode expressive properties of recursive data structures, such as sortedness of a list, or balancing of a search tree. As a result, complex verification conditions can be stated concisely and solved entirely automatically. We also present a new decision procedure for a logic to reason about sets and constraints over their cardinalities. The key insight lies in a technique to decompose constraints according to mutual dependencies. Compared to previous techniques, our algorithm brings significant improvements in running times, and for the first time integrates reasoning about cardinalities within the popular DPLL(T) setting. We integrated our algorithmic advances into Leon, a static analyzer for functional programs. Leon can reason about constraints involving arbitrary recursive function definitions, and has the desirable theoretical property that it will always find counter-examples to assertions that do not hold. We illustrate the flexibility and efficiency of Leon through experimental evaluation, where we used it to prove detailed correctness properties of data structure implementations.

We then illustrate how program specifications can be used as a high-level programming construct ; we present Kaplan, an extension of Scala with first-class logical constraints. Kaplan allows programmers to create, manipulate and combine constraints as they would any other data structure. Our implementation of Kaplan illustrates how declarative programming can be incorporated into an existing mainstream programming language. Moreover, we examine techniques to transform, at compile-time, program specifications into efficient executable code. This approach of software synthesis combines the correctness benefits of declarative programming with the efficiency of imperative or functional programming.

Keywords : constraint solving, decision procedures, software verification, declarative programming, software synthesis, synthesis procedures.

Résumé

La présente thèse est consacrée à la construction de programmes corrects. Nous allons au-delà de l'utilisation standard de spécifications en tant qu'assertions, et présentons des techniques, des algorithmes et leurs implémentations pour les questions de 1) la vérification de programmes, 2) la programmation déclarative et 3) la synthèse de programmes. Ces résultats sont rendus possibles par nos avancées dans la conception et l'implémentation de procédures de décision.

Dans la première partie de cette thèse, nous présentons un résultat de décidabilité pour une classe de logiques paramétrisées par des fonctions récursives. Grâce à cette classe, nous pouvons représenter par des contraintes des propriétés expressives liées à des structures de données récursives, comme le fait qu'une liste soit triée, ou qu'un arbre binaire soit équilibré. Ainsi, des conditions de vérification complexes peuvent être exprimées de manière concise et traitées de manière entièrement automatisée. Nous présentons également une nouvelle procédure de décision pour une logique qui permet de raisonner sur les ensembles finis et leur cardinalité. L'élément essentiel de notre approche est une technique de décomposition des contraintes d'après leurs dépendances mutuelles. Par rapport aux algorithmes existants, notre algorithme est remarquablement plus performant. Il intègre également pour la première fois le raisonnement sur les tailles d'ensembles dans le modèle DPLL(T). Nous avons implémenté nos algorithmes dans Leon, un analyseur statique pour programmes fonctionnels. Leon peut résoudre des contraintes impliquant des fonctions récursives arbitraires, et possède la propriété théorique appréciable qu'il reporte systématiquement des contre-exemples s'il en existe. Nous illustrons la flexibilité et l'efficacité de Leon par une évaluation pratique, où nous prouvons des propriétés détaillées pour diverses implémentations de structures de données. Nous illustrons ensuite l'utilisation des spécifications comme construction intégrée à un langage de programmation ; nous présentons Kaplan, une extension du langage de programmation Scala où les contraintes sont des objets de première classe. Kaplan permet à ses utilisateurs de créer, manipuler et combiner les contraintes comme n'importe quelle autre valeur. Notre implémentation de Kaplan illustre comment la programmation déclarative peut se greffer à des langages existants. Finalement, nous examinons des techniques pour transformer des spécifications en code exécutable. Cette dernière approche, la synthèse de programmes, permet de combiner la fiabilité de la programmation déclarative avec la vitesse d'exécution des paradigmes impératifs ou fonctionnels.

Mots-clés : résolution de contraintes, procédures de décision, vérification logicielle, programmation déclarative, synthèse logicielle, procédures de synthèse.

Contents

Acknowledgements **iii**

Preface **v**

Abstract (English/Français) **vii**

List of figures **xiii**

1 Introduction **1**
1.1 Specifications and Implementations 2
1.2 Specifications for Verification 3
1.3 Executing Specifications 5
1.4 Compiling Specifications 6
1.5 A Common Language 7
1.6 Contributions and Outline 8

2 Reasoning with Abstraction Functions **11**
2.1 Introduction 11
2.2 Example 13
2.3 Recursive Abstraction Functions 15
2.3.1 Instances of our Decision Procedure 16
2.4 The Decision Procedure 21
2.4.1 Overview of the Decision Procedure 21
2.4.2 Syntax and Semantics of our Logic 22
2.4.3 Key Steps of the Decision Procedure 22
2.4.4 Soundness of the Decision Procedure 28
2.4.5 Complexity of the Reduction 28
2.5 Completeness 29
2.5.1 Canonical Set Abstraction 29
2.5.2 Infinitely Surjective Abstractions 31
2.5.3 Sufficiently Surjective Abstractions 32
2.5.4 Application to Multisets, Lists, and Sortedness 35
2.5.5 A Note on Surjectivity 36
2.6 Related Work 37

2.7 Conclusions 39

3 Satisfiability Modulo Recursive Functions **41**
3.1 Introduction 41
3.2 Examples 42
3.3 PureScala 46
3.4 A Satisfiability Procedure Modulo Recursive Functions 47
3.4.1 Properties of the Procedure 50
3.5 The Leon Verification System 55
3.6 Experimental Evaluation 56
3.7 Related Work 59

4 Sets with Cardinality Constraints **63**
4.1 Introduction 63
4.2 Example 65
4.3 Decomposition in Solving BAPA Constraints 66
4.3.1 A Simple Decision Procedure for QFBAPA 67
4.3.2 Decomposing Conjunctions of QFBAPA Formulas 68
4.3.3 Hypertree Decompositions 71
4.4 Integration with Z3 73
4.5 Evaluation 76
4.6 Related Work 77

5 Programming with Constraints **79**
5.1 Introduction 79
5.2 Examples and Features 82
5.2.1 First-class Constraints 82
5.2.2 Ordering Solutions 84
5.2.3 User-defined Functions and Datatypes 86
5.2.4 Timeouts 87
5.2.5 Logical Variables 88
5.2.6 Imperative Constraint Programming 89
5.3 Semantics 91
5.4 Implementation 93
5.4.1 Run-Time Library 93
5.4.2 Scala Compiler Plugin 94
5.4.3 Implementation of the Core Solving Algorithms 96
5.5 Advanced Usage Scenarios and Evaluation 100
5.5.1 Enumerating Data Structures for Testing 100
5.5.2 Executable Specifications 101
5.5.3 Counter-example Guided Inductive Synthesis 103
5.5.4 Comparison to Other Systems 103
5.6 Related Work 106

5.7 Conclusion 108

6 Compiling Specifications **111**

6.1 Introduction 111
6.2 Examples 112
6.3 Synthesis using Relation Transformations 115
6.3.1 Theory-Independent Inference Rules 116
6.4 Synthesis for Linear Integer Arithmetic 118
6.4.1 Processing Equalities 119
6.4.2 Processing Inequalities 120
6.4.3 Example 121
6.5 Synthesis for QFBAPA 122
6.6 Synthesis for Term Algebras 123
6.6.1 Pure Term Algebras 123
6.6.2 Reduction to an Interpreted Theory 126
6.7 Implementation 127
6.8 Related Work 128

7 Conclusion **131**

Bibliography **145**

Curriculum Vitæ **147**

List of Figures

2.1 Partial binary search tree implementation of a set. 13
2.2 Graphical depiction of a catamorphism application. 15
2.3 Computing the set of free variables in a λ-calculus term. 17
2.4 Using the minimal element as an abstraction. 18
2.5 A fold that checks that a tree is sorted. 18
2.6 Example instances of our decision procedure for different catamorphisms. . . . 20
2.7 Syntax of the parametric logic. 22
2.8 Semantics of the parametric logic. 22
2.9 Unification Rules. 25

3.1 Pseudo-code of the procedure. 48
3.2 Function definition and its translation into clauses with control literals. 48
3.3 Online interface to the Leon verification system. 57
3.4 Summary of evaluation results. 61

4.1 Grammar of QFBAPA. 66
4.2 Labeling of Venn regions with integer variables. 68
4.3 Independent labeling of Venn regions. 71
4.4 Hypertree decomposition example. 73
4.5 Experimental results. 78

5.1 Small-step semantics. 91
5.2 Term class hierarchy. 95
5.3 Implementation of **assuming**–**otherwise** in terms of lazyFind. 96
5.4 Pseudo-code of the minimization algorithm. 98
5.5 Pseudo-code of the ordered enumeration algorithm. 98
5.6 Evaluation results 1. 100
5.7 Evaluation results 2. 102
5.8 Evaluation results 3. 102
5.9 Counter-example guided inductive synthesis in Kaplan. 104
5.10 Evaluation results 4. 104

6.1 Labeling of Venn regions. 123
6.2 Synthesis times. 128

1 Introduction

Programming is hard. Anyone who ever faced the task of solving a programming task will recognize that fact. The reasons for this difficulty lie in what programming essentially is: a transformation of a set of requirements into an executable implementation. This transformation can be difficult and error-prone, making the construction of correct programs a challenging task. Responses to this challenge can be broadly divided into two categories: the *a priori* and *a posteriori* approaches to correct software developement.

The a priori approach strives to produce software that is correct by construction. One way to achieve this goal is to use programming languages that raise the level of abstraction closer to the language of requirements. This narrows the translation gap between the requirements and the implementation, thus reducing the opportunities for introducing errors [KK71, CKC81, JL87, AH10]. Another way to achieve a priori correctness is to start from the requirements and derive an implementation through successive, provably sound, refinement steps. This process can be manual [Dij75, Mor94, BvW99] or automated [MW80, KMPS10b, GJTV11].

In the a posteriori approach on the other hand, the goal is to supplement the executable implementation with a proof of its correctness. Techniques to derive such proofs automatically exist [CC77, BHJM07] but are in practice limited to low-level specifications such as memory safety. Higher-level specifications typically require that the programmer produces part of the proof [ZKR08, ZKR09, CDH+09, BFL+11].

Each comes with its advantages and drawbacks: the a priori approach leads to software that is easier to construct and reason about but is harder to execute efficiently, while the a posteriori approach leads to efficient programs whose meaning can sometimes be obscured. We thus believe it to be necessary to examine both, and to develop systems that allow for programmers to alternate between the two.

1.1 Specifications and Implementations

A program implementation, in the standard imperative or functional settings, is of an *explicit* nature: the computation steps are spelled out by the programmer, who must understand the execution model. The implementation dictates *how* the computation should be performed. A natural mathematical abstraction for a program implementation is therefore a *function*.

Program specifications, on the other hand, are of an *implicit* nature: they describe the relationship that should hold between the inputs and the outputs of the program. In other words, specifications describe *what* should hold. The natural mathematical abstraction for a specification is a *relation*.

There are several appealing ways in which combining implementations and specifications can help us produce correct programs. Consider a program P consuming inputs $\bar{a}$ and producing outputs $\bar{x}$, such that $P(\bar{a}) = \bar{x}$, and a specification $\phi(\bar{a}, \bar{x})$ relating the inputs and outputs.

- The simplest use of specifications is to perform *run-time checking*: for a specific input $\bar{a}$ and computed output $\bar{x} = P(\bar{a})$, check whether $\phi(\bar{a}, \bar{x})$ holds. As long as ϕ is expressed in a form that can be evaluated, this is straightforward to do and it does not require reasoning about P. Due to their simplicity, run-time checks (or assertions) are commonly used. They provide little guarantees of correctness, though: because assertions are only evaluated on observed executions, the absence or errors offers no assurance about the general case.
- We can ask whether it is the case that $\forall \bar{a} : \phi(\bar{a}, P(\bar{a}))$. This is the problem of *program verification*: given a specification and an implementation, check that the implementation fulfills the specification. Dually, we can ask whether $\exists \bar{a} : \neg\phi(\bar{a}, P(\bar{a}))$. This is the problem of *counter-example generation*: find values for inputs such that the implementation does not meet the specification. Verification is appealing in that it can prove that a program *always* meets a specification, even for executions that have not been observed.
- For a specific input $\bar{a}$, we can also *solve* for $\bar{y}$ such that $\phi(\bar{a}, \bar{y})$ holds: if ϕ is strong enough to encode all the desired properties, $\bar{y}$ can effectively replace the computed value $\bar{x} = P(\bar{a})$. We refer to this as a *constraint solving* approach. Constraint solving can be a useful tool to produce values that meet a given specification.
- We can in some cases raise constraint solving to a higher-level, and solve for a function rather than a single value. In effect, we ask the question $\exists F : \forall \bar{a} : \phi(\bar{a}, F(\bar{a}))$: does there exist a *function* that fulfills the specification? While this question is commonly for the programmer to solve, the goal of *software synthesis* is to use symbolic reasoning to achieve the same automatically.

We note that synthesis is to constraint solving what verification is to run-time checking: a generalization from a problem for a fixed input to the problem for *any* input. Thus, whereas run-time checking and constraint solving take place at execution time, verification and synthesis are by necessity compile-time activities.

This dissertation presents algorithms, techniques and implementations for the activities of

verification, *constraint solving* and *synthesis*, which all arise in software development. We briefly introduce each of these domains through examples.

1.2 Specifications for Verification

Consider this simple programming task:

> *Given a total number of seconds, find the corresponding decompositions into hours, minutes and remaining seconds.*

The resulting program should be a function of one argument that returns three values. One possible implementation, in functional programming style, is to use a loop to subtract from the total and accumulate first whole hours, then whole minutes. In functional style, this loop can be expressed as a recursive function:

```
def secondsToTime(total : Int) : (Int, Int, Int) = {
  def rec(s : Int, h : Int, m : Int) : (Int, Int, Int) = {
    if(s ≥ 3600) {
      rec(s − 3600, h + 1, m)
    } else if(s ≥ 60) {
      rec(s − 60, h, m + 1)
    } else {
      (h, m, s)
    }
  }
  rec(total, 0, 0)
}
```

To decide whether this implementation is correct, one first needs to formally state what correct means. From the original statement, written in English, we can derive a requirement for the computation: the result should be such that when we add the components and multiply each of them by the appropriate factor, we get back the original value. Because the specification refers to the return value of the function (as opposed to for instance intermediate steps of the computation), we can write it as a postcondition:

```
ensuring { case (h, m, s) ⇒ h ∗ 3600 + m ∗ 60 + s == total }
```

This specification does not encode all requirements of the original problem, though. Indeed, another program that meets it is given by:

```
def secondsToTime(total : Int) : (Int, Int, Int) = (0, 0, total)
```

In other words, returning the number of seconds and performing no decomposition satisfies the post-condition. To fully encode the desired requirements, we can strengthen our specification into:

```
ensuring {
```

```
  case (h, m, s) ⇒ h * 3600 + m * 60 + s == total && m ≥ 0 && m < 60 && s ≥ 0 && s < 60
}
```

This illustrates an important fact: *deriving a full specification from a set of requirements is in itself difficult.*

Now that we have a candidate implementation and specification, it remains to show that the former satisfies the later. Applying Leon, our verification system, results in the following output

```
Error: Counter-example found:
total -> -1
```

This illustrates counter-example generation: our implementation of secondsToTime violates the specification when given a negative input. We can choose to handle only positive inputs by writing a precondition:

```
require(total ≥ 0)
```

This new annotation is not sufficient to prove our program correct, though. Indeed, because it involves a recursive function that manipulates values in a non-obvious way, we need to provide additional invariants. A version of the program completed with appropriate annotations and proved correct by Leon is given by:

```
def partial(t: Int, h: Int, m: Int, s: Int) : Boolean = h * 3600 + m * 60 + s == t

def full(t: Int, h: Int, m: Int, s: Int) : Boolean =
  partial(t, h, m, s) && m ≥ 0 && m < 60 && s ≥ 0 && s < 60

def secondsToTime(total : Int) : (Int, Int, Int) = {
  def rec(s : Int, h : Int, m : Int) : (Int, Int, Int) = {
    require(s ≥ 0 && h ≥ 0 && m ≥ 0 && m < 60 &&
            partial(total, h, m, s) &&
            (m == 0 || s + m * 60 < 3600))
    if(s ≥ 3600) {
      rec(s − 3600, h + 1, m)
    } else if(s ≥ 60) {
      rec(s − 60, h, m + 1)
    } else {
      (h, m, s)
    }
  } ensuring { case (h,m,s) ⇒ full(total, h, m, s) }

  rec(total, 0, 0)
}
```

(Note that the functions partial and full encode the partial and full correctness conditions presented before.) Intuitively, the extra annotations encode the fact that the values h, m and s are not arbitrary but rather together maintain an invariant.

Strikingly, the amount of annotations required for the proof to succeed exceeds the size of the implementation itself. *Proving programs correct is difficult.* The efforts are well rewarded, though: the implementation is guaranteed to match the specification for *any* output. Additionally, because the program was crafted manually, it can be executed efficiently.

1.3 Executing Specifications

Our example in the previous section illustrated an important phenomenon; specifications that are precise enough to capture the desired functional properties can approach the size or complexity of the implementation itself. Additionally, proving that an implementation meets them can result in additional work involving the derivation of intermediate, partial specifications. An appealing approach then, is to *execute* specifications; at run-time upon reaching a specification, invoke a constraint solver to compute possible solutions for the result variables.

With such a mechanism available, solving the time decomposition programming task becomes a matter of executing the post-condition. Using Kaplan, our system for declarative programming, a user can state this as:

```
def secondsToTime(total : Int) = ((h : Int, m : Int, s : Int) ⇒
  h ∗ 3600 + m ∗ 60 + s == total && m ≥ 0 && m < 60 && s ≥ 0 && s < 60).solve
)
```

The first benefit of this form of constraint solving is obvious: the programmer needs to provide only one of the implementation or the specification. Another advantage is that, as long as we are willing to trust our constraint solver, we can consider the code to be correct by construction. The most appealing aspect of executing declarative specifications, though, is that of code reuse. Consider for example a set implemented as a binary search tree, built with the algebraic data type

```
sealed abstract class Tree
case class Leaf() extends Tree
case class Node(left : Tree, value : Int, right : Tree) extends Tree
```

and the recursive function computing the set of values in a tree:

```
def content(tree : Tree) : Set[Int] = tree match {
  case Leaf() ⇒ Set.empty
  case Node(l, v, r) ⇒ content(l) ++ Set(v) ++ content(r)
}
```

Let us also assume a function that checks whether the sorting and balancing properties of a search tree are fulfilled:

```
def isValid(tree : Tree) : Boolean = ...
```

Using these functions, we can define for instance insertion:

```
def insert(tree : Tree, value : Int) : Tree =
  ((t : Tree) ⇒ isValid(t) && content(t) == content(tree) ++ Set(value)).solve
```

and defining removal declaratively is a trivial matter of using a different set operation:

```
def insert(tree : Tree, value : Int) : Tree =
  ((t : Tree) ⇒ isValid(t) && content(t) == content(tree) -- Set(value)).solve
```

One can of course not hope that the execution of these declarative specifications will match the execution of carefully written explicit code. Indeed, a programmer can possess insight on efficient methods to achieve a particular subgoal – in our last example, for instance, the knowledge that indices are sorted. A constraint solver, because it needs to be general, will not be tuned for any particular task.

The possibility of programming with just specifications remains appealing, however, for tasks such as rapid prototyping, test-case generation, or cases where no obvious algorithm is available.

1.4 Compiling Specifications

The natural next step after executing specifications is to ask oneself in which circumstances one can compile them. As we pointed out, a specification is naturally viewed as a relation, and executable code as a function. The problem of synthesis from a specification is then the problem of automatically transforming a relation into a function. Coming back to our original programming task and its expression as the execution of the specification

```
def secondsToTime(total : Int) = ((h : Int, m : Int, s : Int) ⇒
  h ∗ 3600 + m ∗ 60 + s == total && m ≥ 0 && m < 60 && s ≥ 0 && s < 60).solve
)
```

we wish to automatically derive a program that, given an input, produces a decomposition that satisfies the full specification. Finding such a program is by nature harder than finding a single value, as in the case of executing specifications. As we show in this thesis, the task remains possible however when the specification falls into a well-defined class. In this case, it is expressed in linear integer arithmetic, a logic that our tool Comfusy supports. Given this input, Comfusy produces the program:

```
def secondsToTime(total : Int) = {
  val loc1 = total / 3600
```

```
  val num2 = total + ((−3600) * loc1)
  val loc2 = min(num2 / 60, 59)
  val loc3 = total + ((−3600) * loc1) + (−60 * loc2)
  (loc1, loc2, loc3)
}
```

Whenever, as here, synthesis is possible, we obtain the combined benefits of program verification and executable specifications. On the one hand and similarly to executable specifications, the programming task is solved simply by stating the requirements to satisfy, and the results can be trusted to be correct. On the other hand, as in the verification approach, the code that is eventually executed is written in an imperative or functional style, and can therefore be executed efficiently.

1.5 A Common Language

We have so far deliberately avoided the question of what exactly we consider as a specification language. Throughout this thesis, we embrace a functional language both as the implementation language and as the specification language. More precisely, we choose to work with a subset of the Scala programming language [OSV11].

Working with a functional language has the advantage that programs are mostly expressions, and these expressions can be very naturally mapped into logical formulas amenable to symbolic reasoning. While techniques have been developed to encode imperative programs into formulas [Hoa72, DL05, Bla12], we choose to side-step that step entirely and focus on a functional language.

Using the same same language for implementations and specifications presents several advantages. Conceptually, it is line with our general claim that specifications should be an integral part of programs. More pragmatically, developers never have to leave the familiar world of Scala; they do not need to learn a new notation for properties, and can use the existing libraries for dynamically checking executable contracts [Ode10] to describe the desired properties. As a result, run-time contract violations can be found through testing, using popular approaches such as QuickCheck [CH00] and bounded-exhaustive testing [BKM02, GGJ+10]. This would be difficult if there was a significant gap between annotation and implementation language [ZKTR07].

Working with executable specifications also allows us to search for counter-examples by simulating executions. This possibility plays a fundamental part in our algorithms for satisfiability, which ensure completeness by systematically exploring executions with iterative deepening.

One last advantage of working with a subset of an existing language is that it provides us with very concrete opportunities for reuse: our systems Leon, Kaplan and Comfusy all rely on parts of the Scala compiler, relieving us from duties such as parsing or type-checking, and ensuring

that our tools can be used within a rich and well-maintained ecosystem.

1.6 Contributions and Outline

This thesis focuses on techniques for the construction of correct software.

– We develop algorithms that extend the reach of automated symbolic reasoning. They allow us to verify complex pieces of software against full functional specifications.
– We design programming language extensions that allow for correct-by-construction software development.
– We illustrate how these extensions can be seamlessly integrated into mainstream programming languages, thus bringing their reliability benefits to the greater number.

The following chapters are organized as follows:

Chapter 2 presents a decidability result for a theory of term algebras augmented with recursive abstraction functions (homomorphisms). These functions are particularly useful in the context of writing specifications, as they can naturally encode high-level attributes of data structures. Among such attributes are, for instance: the elements contained in a data structure, whether a list is sorted, or whether a tree is balanced. By using such abstraction functions, the contracts for operations on data structure can often be stated as simple algebraic laws. Our decidability result ensures that these contracts can be handled using automated tools.

Chapter 3 presents a general technique to reason about formulas with user-defined recursive functions. In particular, it can act as an implementation for the family of decision procedures described in Chapter 2. We implemented this technique as part of Leon, a verification system for a functional subset of the Scala programming language. We show that Leon has the important property that it always finds contract violations whenever they exist. We used our system to prove detailed correctness properties about functional data structures. Our experimental results show that Leon is fast, both to prove valid properties and to produce counter-example to invalid ones.

Chapter 4 describes a new decision procedure for constraints over sets and their cardinalities. Sets with cardinalities provide a relevant abstraction for container data structures, and are therefore a natural candidate co-domain for the abstraction functions described in Chapter 2. The key component of our decision procedure is a technique to automatically decompose conjunctions of constraints according to their co-dependencies. We show that, compared to previous techniques, these decompositions lead to significant improvements in solving times.

Chapter 5 marks the departure from using specification as a safety mechanism into using them as the program itself. We present Kaplan, an extension of Scala with declarative programming features. Kaplan allows programmers to manipulate constraints as first-class values. Whenever a constraint needs to be solved, it invokes Leon to produce a valid model. Our implementation of Kaplan shows that, using existing tools, we can

integrate declarative programming constructs into a standard imperative or functional language. The execution of programs that do not use a declarative style is left unchanged, thus allowing programmers to mix paradigms and reap the benefits of each in a single environment.

Chapter 6 builds on the ideas of using specifications as a programming construct, and examines situations in which they can be turned, at compile time, into executable code. We present a generic framework to reason about sound relational derivations, and show how to apply it to develop synthesis procedures. Synthesis procedures are algorithms that are guaranteed to succeed in transforming a relation from a given class into a function that implements it. As a programming language extension, our algorithms combine the benefits of the expressive power of declarative specifications with the efficiency of imperative code.

2 Reasoning with Abstraction Functions

This chapter introduces a family of decision procedures that extend the decision procedure for quantifier-free constraints on recursive algebraic data types (term algebras) to support recursive abstraction functions. Our abstraction functions are catamorphisms (term algebra homomorphisms) mapping algebraic data type values into values in other decidable theories (e.g. sets, multisets, lists, integers, booleans). Each instance of our decision procedure family is sound. Additionally, we identify a widely applicable many-to-one condition on abstraction functions that implies the completeness. Complete instances of our decision procedure include the following correctness statements: 1) a functional data structure implementation satisfies a recursively specified invariant, 2) such data structure conforms to a contract given in terms of sets, multisets, lists, sizes, or heights, 3) a transformation of a formula (or lambda term) abstract syntax tree changes the set of free variables in the specified way.

2.1 Introduction

While much recent work on automation was invested into imperative languages, it is interesting (50 years after [McC60]) to consider the reach of decision procedures when applied to functional programming languages, which were designed with the ease of reasoning as one of the explicit goals. Researchers have explored the uses of advanced type systems to check expressive properties [DP04, Xi03], and have recently also applied satisfiability modulo theory solvers to localized type system constraints [RKJ08, KRJ09]. Among the recognized benefits of theorem provers is efficient support for propositional operators and arithmetic. This chapter revives another direction where theorem provers can play a prominent role: complete reasoning about certain families of functions operating on algebraic data types.

Purely functional implementations of data structures [Oka98] present a well-defined and interesting benchmark for automated reasoning about functional programs. Data structures come with well-understood specifications: they typically implement ordered or unordered collections of objects, or maps between objects. To express the desired properties of data structures, we often need a rich set of data types to write specifications. In particular, it is

desirable to have in the language not only algebraic data types, but also finite sets and multisets. These data types can be used to concisely specify the observable behavior of data structures with the desired level of under-specification [KLZR06, LKR05, ZX05, Dun07]. For example, if neither the order nor the repetitions of elements in the tree matter, an appropriate abstract value is a set. An abstract description of an `add` operation that inserts into a data structure is then

$$\alpha(\mathsf{add}(e,t)) = \{e\} \cup \alpha(t) \tag{2.1}$$

Here α denotes an abstraction function mapping a tree into the set of elements stored in the tree. Other variants of the specification can use multisets or lists instead of sets.

An important design choice is how to specify such mappings α between the concrete and abstract data structure values. A popular approach [CDH$^+$09, ZKR08] does not explicitly define a mapping α but instead introduces a fresh *ghost* variable to represent the values $\alpha(t)$. It then uses invariants to relate the ghost variable to the concrete value of the data structure. Because developers explicitly specify values of ghost variables, such technique yields simple verification conditions. However, it can impose additional annotation overhead—for the tree example above it would require supplying a set as an additional argument to each algebraic data type constructor. To eliminate this overhead and to decouple the specification from the implementation, we use recursively defined abstraction functions that compute the abstract value for each concrete data structure. As a result, our verification conditions contain user-defined function definitions that manipulate rich data types, along with equations and disequations involving such functions. This chapter introduces decision procedures that can reason about interesting fragments of such a language.

We present decision procedures for reasoning about algebraic data types with user-defined abstraction functions expressed as a fold, or catamorphisms [MFP91], over algebraic data types. These decision procedures subsume approaches for reasoning about algebraic data types [Opp78] and add the ability to express constraints on the abstract view of the data structure. When using sets as the abstract view, our decision procedure can naturally be combined with decision procedures for reasoning about sets of elements in the presence of cardinality bounds [KR07]. (Chapter 4 develops new techniques for this particular target logic.)

Our decision procedures are not limited to using sets as an abstract view of data structures. The most important condition for applicability is that the notion of a collection has a decidable theory in which the fold can be expressed. This includes in particular arrays [BM07, dMB09], multisets with cardinality bounds [PK08a], and even option types over integer elements. Each abstract value provides different possibilities for defining the fold function.

We believe that we have identified an interesting region in the space of automated reasoning approaches, because the technique turned out to be applicable more widely than we had expected. We intended to use the technique to verify the abstraction of values of functional data structures using sets. It turned out that the approach works not only for sets but also for

```
object BSTSet {
  sealed abstract class Tree
  case class Leaf() extends Tree
  case class Node(left: Tree, value: Int, right: Tree) extends Tree

  // abstraction function
  def content(t: Tree): C = t match {
    case Leaf() ⇒ Set.empty
    case Node(l,e,r) ⇒ content(l) ++ Set(e) ++ content(r)
  }

  // returns an empty set
  def empty: Tree = Leaf() ensuring (res ⇒ content(res) == Set.empty)

  // adds an element to a set
  def add(e: Int, t: Tree): Tree = (t match {
    case Leaf() ⇒ Node(Leaf(), e, Leaf())
    case Node(l,v,r) ⇒
      if (e < v) Node(add(e, l), v, r)
      else if (e == v) t
      else Node(l, v, add(e, r))
  }) ensuring (res ⇒ content(res) == content(t) ++ Set(e))

  // user−defined equality on abstract data type (congruence)
  def equals(t1 : Tree, t2 : Tree) : Boolean = (content(t1) == content(t2))
}
```

Figure 2.1 – Partial binary search tree implementation of a set.

lists and multisets, and even for abstractions that encode the truth-values of data structure invariants. Beyond data structures used to implement sets and maps, we have found that computing bound variables, a common operation on the representations of lambda terms and formulas, is also amenable to our approach. We thus expect that our decision procedure can help increase the automation of reasoning about operational semantics and type systems of programming languages.

2.2 Example

Figure 2.1 shows code for a partial implementation of a set of integers using a binary search tree. As this chapter is concerned only with a fixed form of recursive functions, we postpone the presentation of our programming language until 3.3, and focus here on the relevant parts. The class hierarchy

```
sealed abstract class Tree
private case class Leaf() extends Tree
private case class Node(left:Tree, value:Int, right:Tree) extends Tree
```

describes an algebraic data type Tree with the alternatives Node and Leaf. The module BSTSet provides its clients with functions to create empty sets and to insert elements into existing sets. Because the client is not supposed to have information on the type Tree, they use the abstraction function content to view these trees as sets. The abstraction function is declared like any other function and is executable, but we assume that it obeys a syntactic restriction to make it a tree fold, as described in the next section.

Functions empty and add are annotated with postconditions on which the client can rely, without knowing anything about their concrete implementation. These postconditions do not give any information about the inner structure of binary search trees—such information would be useless to a user who has no access to the internal structure. Instead, the postconditions express properties on their result in terms of the abstraction function. The advantages of such an abstraction mechanism are well-known. By separating the specification (functions signatures and contracts) from the implementation, developers obtain better opportunities for code reuse, manual proofs become simpler [Hoa72], and automated analysis of clients becomes more tractable [KLZR06].

The type Set[Int] used in the abstraction function and the specifications, refers to the Scala library class for immutable sets. The operator ++ computes a set consisting of the union of two sets, and the constructor Set(e) constructs the singleton set {e} (containing only the element e). We assume the implementation of the container library to be correct, and map the container operations (e.g. ++) to the corresponding ones in the mathematical theory of finite sets (e.g. $\cup$) when we reason about the programs.

The advantages of using an abstraction function in specifications are numerous, but they also require verification systems that can reason about these user-defined functions—these functions appear in contracts and therefore in verification conditions. For example, consider the function add in Figure 2.1 and apply the standard technique to replace recursive function call with the function contract. The result is a set of verification conditions including (among others) the condition:

$$
\begin{aligned}
&\forall t_1, t_2, t_3, t_4 : \mathsf{Tree},\ e_1, e_2 : \mathsf{Int} \\
&\quad t_1 = \mathsf{Node}(t_2, e_1, t_3) \implies \\
&\quad \mathsf{content}(t_4) = \mathsf{content}(t_2) \cup \{e_2\} \implies \\
&\quad \mathsf{content}(\mathsf{Node}(t_4, e_1, t_3)) = \mathsf{content}(t_1) \cup \{e_2\}
\end{aligned}
\tag{2.2}
$$

This formula is graphically represented in Figure 2.2. It combines constraints over algebraic data types and over finite sets, as well as a non-trivial connection given by the recursively defined abstraction function content. It is therefore beyond the reach of currently known decision procedures. In the following sections, we present a new decision procedure which can handle such formulas.

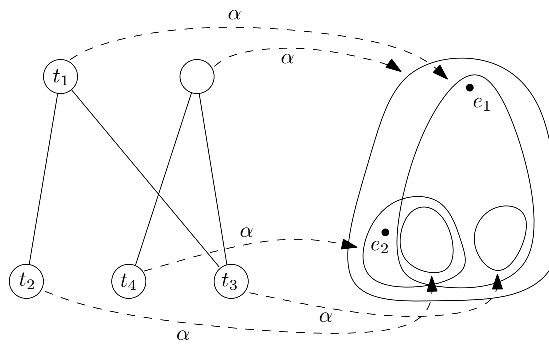

Figure 2.2 – Illustration of (2.2), where edges labeled by α denote applications of the `content` abstraction function.

2.3 Recursive Abstraction Functions

Decision procedures for reasoning about algebraic data types [Opp78, BST07] are concerned with proving and disproving quantifier-free formulas that involve constructors and selectors of an algebraic data type, such as the immutable version of heterogeneous lists in LISP. They generalize the unification algorithms used in theorem proving [Rob65] and Hindley-Milner type inference. Using the terminology of model theory, this problem can be described as the satisfiability of quantifier-free first-order formulas in the theory of term algebras [Hod97]. A term algebra structure has as a domain of interpretation ground terms over some set of function symbols, called *constructors*. The language of term algebras includes application of constructors to build larger terms from smaller ones, and the only atomic formulas are comparing terms for equality.

In this chapter, we extend the decision procedure for such algebraic data types with the ability to specify an *abstract value* of the data type. The abstract value can be, for example, a set, relation, multiset (bag), or a list. A number of decision procedures are known for theories of such abstract values [KR07, PK08a, Mai09, Jaf90] (see also Chapter 4). Such values purposely ignore issues such as tree shape, ordering, or even the exact number of times an element appears in the data structure. In return, they come with powerful algebraic laws and decidability properties that are often not available for algebraic data types themselves, and they often provide the desired amount of under-specification for interfaces of data structures. The decision procedures we describe enable proving formulas that relate data structures implemented as algebraic data types to their abstract values that specify the observable behavior of these data types. They can thus increase the automation when verifying correctness of functional data structures.

2.3.1 Instances of our Decision Procedure

Our decision procedures for different fold functions follow the same pattern, so we talk about them as instances of one generic decision procedure. The choice of the type of data stored in the tree in each decision procedure instance is largely unconstrained; the procedures work for any infinitely countable parameterized data type, which we will denote by $\mathscr{E}$ in our discussion (it could be extended to finite data types using techniques from [KGGT07]). The decision procedure is parameterized by:

1. an element type $\mathscr{E}$,
2. a collection type $\mathscr{C}$, and
3. an abstraction function α

The abstraction function generalizes the content function in Figure 2.1. We require it to be a catamorphism (generalized fold) [MFP91]. In the presentation, we focus on the case of binary trees, so we require an abstraction function of the form

```
def α(t: Tree): 𝒞 = t match {
  case Leaf() ⇒ empty
  case Node(l,e,r) ⇒ combine(α(l), e, α(r))
}
```

for some functions

$$\mathsf{empty} : \mathscr{C} \qquad\qquad \mathsf{combine} : (\mathscr{C}, \mathscr{E}, \mathscr{C}) \to \mathscr{C}$$

The generalization to recursive types other than trees is straightforward. The abstraction function α is given by: for each constant in the term algebra c_i, a function $\mathsf{empty}_i : \mathscr{C}$, and for each constructor $\mathsf{C}(\ldots)_i$, a function $\mathsf{combine}(\alpha(\ldots), \alpha(\ldots), \ldots)_i \to \mathscr{C}$. A natural way to think about catamorphisms is indeed to see their action as transforming their argument by rewriting the constructor applications into function applications.

Figure 2.6 on page 20 summarizes some of the instances of our decision procedure. It shows the type of the abstract value $\mathscr{C}$, the definition of the functions empty and combine that define the catamorphism, some of the operations available on the logic $\mathscr{L}_\mathscr{C}$ of $\mathscr{C}$ values, and points to one of the references that can be used to show the decidability of $\mathscr{L}_\mathscr{C}$. (The decision procedure for $\mathscr{L}_\mathscr{C}$ is invoked as the last step of our decision procedure.) Figure 2.6 shows that our decision procedure covers a wide range of collection abstractions of interest, as well as some other relevant functions definable as folds. We describe some of these cases in more detail.

Set abstractions. The content function in Figure 2.1 is an example of a fold used as an abstraction function. In this case, $\mathsf{empty} = \emptyset$ and $\mathsf{combine}(t_1, e, t_2) = c_1 \cup \{e\} \cup c_2$. We found this example to be particularly useful and well-behaved, so we refer to it as the *canonical set abstraction.*

```
object Lambda {
  sealed abstract class Term
  case class Var(id: ID) extends Term
  case class App(fun: Term, arg: Term) extends Term
  case class Abs(bound: ID, body: Term) extends Term

  def free(t: Term): Set[ID] = t match {
    case Var(id) ⇒ Set(id)
    case App(fun, arg) ⇒ free(fun) ++ free(arg)
    case Abs(bound, body) ⇒ free(body) -- Set(bound)
  }
}
```

Figure 2.3 – Computing the set of free variables in a λ-calculus term.

The canonical set abstraction is not the only interesting abstraction function whose result is a set. Figure 2.3 shows another example, where the fold free computes a set by adding and removing elements as the tree traversal goes. Such abstraction function can then be used to prove that, e.g., a rewriting step on a λ-calculus term does not increase the set of free variables in the term.

Abstractions using multisets and lists. A set abstracts both the order and the multiplicity (the number of occurrences) of elements in the data structure. A more precise abstraction is a multiset (bag) (Figure 2.6), which preserves the multiplicity. Moreover, the decision procedure for multisets [PK08a, PK08b] supports an abstraction function that abstracts a multiset into the underlying set, which enables simultaneous use of trees, multisets and sets in the same specification, giving a decision procedure for an interesting fragment of the tree-list-bag-set hierarchy [HB94]. Even more precise abstractions of trees use lists, supporting any chosen traversal order; they reduce to the decision procedure for the theory of lists (words) with concatenation [Pla04].

Minimal element. Some useful abstractions map trees into a quantity rather than into a collection. findMin in Figure 2.4 for instance is naturally expressed as a fold, and can be used to prove properties of data structures which maintain invariants about the position of certain particular elements (e.g. priority queues).

Sortedness of binary search trees. Fold functions can also compute properties about tree structures which apply to the complete set of nodes and go beyond the expression of a container in terms of another. Figure 2.5 shows the abstraction function sorted which, when applied to a binary tree, returns a triple containing a lower and upper bound on the set of elements, and a boolean indicating whether the tree is sorted. Although alternative specifications of sortedness are possible, this one directly conforms to the form of a fold function; at

```
object MinElement {
  sealed abstract class Tree
  case class Node(left: Tree, value: Int, right: Tree) extends Tree
  case class Leaf() extends Tree

  def findMin(t: Tree): Option[Int] = t match {
    case Leaf() ⇒ None
    case Node(l,v,r) ⇒ (findMin(l),findMin(r)) match {
      case (None,None) ⇒ Some(v)
      case (Some(vl),None) ⇒ Some(min(v, vl))
      case (None,Some(vr)) ⇒ Some(min(v, vr))
      case (Some(vl),Some(vr)) ⇒ Some(min(v, vl, vr))
    }
  }
}
```

Figure 2.4 – Using the minimal element as an abstraction.

```
object SortedSet {
  sealed abstract class Tree
  case class Leaf() extends Tree
  case class Node(left: Tree, value: Int, right: Tree) extends Tree

  def sorted(t: Tree): (Option[Int],Option[Int],Boolean) = t match {
    case Leaf() ⇒ (None, None, true)
    case Node(l, v, r) ⇒ (sorted(l),sorted(r)) match {
      case ((_,_,false),_) ⇒ (None, None, false)
      case (_,(_,_,false)) ⇒ (None, None, false)
      case ((None,None,_),(None,None,_)) ⇒ (Some(v), Some(v), true)
      case ((Some(minL),Some(maxL),_),(None,None,_))
        if (maxL ≤ v) ⇒ (Some(minL),Some(v),true)
      case ((None,None,_),(Some(minR),Some(maxR),_))
        if (minR > v) ⇒
        (Some(v), Some(maxR), true)
      case ((Some(minL),Some(maxL),_), (Some(minR),Some(maxR),_))
        if (maxL ≤ v && minR > v) ⇒ (Some(minL),Some(maxR),true)
      case _ ⇒ (None,None,false)
    }
  }
}
```

Figure 2.5 – A fold that checks that a tree is sorted.

the same time it is efficiently executable.

The code in Figure 2.5 allows for trees with repeated elements. By replacing the occurrences of $\leq$ by the stricter $<$ we obtain the definition of sorted trees with distinct elements, which can also be handled by our decision procedure (the strict inequality turns out to be a more complicated instance of the decision procedure, see Section 2.5.3).

This example also illustrates fold functions that return n-tuples, which is a useful strategy to represent multiple mutually recursive functions. We will therefore assume that we work with a single fold function in our decision procedure.

$\mathscr{C}$	empty	combine(c_1, e, c_2)	abstract operations (apart from $\wedge, \neg, =$)	complexity	follows from
Set	$\emptyset$	$c_1 \cup \{e\} \cup c_2$	$\cup, \cap, \setminus$, cardinality	NP	[KR07]
Multiset	$\emptyset$	$c_1 \uplus \{e\} \uplus c_2$	$\cap, \cup, \setminus, \uplus$, setof, cardinality	NP	[PK08a, PK08b]
$\mathbb{N}$	0	$c_1 + 1 + c_2$ (size)	$+, \leq$	NP	[Pap81]
$\mathbb{Z}$	0	$1 + \max(c_1, c_2)$ (height)	$+, \leq$	NP	[Pap81]
List	List()	*a*) c_1++ List(e)++ c_2 (in-order)	++(concat), List(_)(singleton)	PSPACE	[Pla04]
	List()	*b*) List(e)++c_1++ c_2 (pre-order)			
	List()	*c*) c_1++ c_2++ List(e) (post-order)			
Tree	Leaf	Node(c_2, e, c_1) (mirror)	Node, Leaf	NP	[Opp78]
Option	None	*a*) Some(e)	Some, None	NP	[NO80]
	None	*b*) (computing minimum) see Figure 2.4	Some, None, $+, \leq$, **if**	NP	[NO79, NO80, Pap81]
(Option, Option, Boolean)	(None, None, **true**)	*c*) (checking sortedness) see Figure 2.5			

Figure 2.6 – Example instances of our decision procedure for different catamorphisms.

2.4 The Decision Procedure

To simplify the presentation, we describe our decision procedure for the specific algebraic data type of binary trees, corresponding to the case classes in Figure 2.1. The procedure naturally extends to data types with more constructors.

If t_1 and t_2 denote values of type Tree, by $t_1 = t_2$ we denote that t_1 and t_2 are structurally equal that is, either they are both leaves, or they are both nodes with equal values and equal subtrees.

As far as soundness is concerned, we can leave the collection type $\mathscr{C}$ and the language $\mathscr{L}_\mathscr{C}$ of decidable constraints on $\mathscr{C}$ largely unconstrained. As explained in Section 2.5, the conditions for completeness are relatively easy to satisfy when the image are sets; they become somewhat more involved for e.g. multisets and lists.

In our exposition, we use the notation

$$\mathsf{distinct}(x_1^1, x_2^1, \ldots, x_{I(1)}^1; \ldots; x_1^n, \ldots, x_{I(n)}^n)$$

as a syntactic shorthand for the following conjunction of disequalities

$$\bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} \bigwedge_{k=1}^{I(i)} \bigwedge_{l=1}^{I(j)} x_k^i \neq x_l^j$$

For example, $\mathsf{distinct}(x, y; z)$ means $x \neq z \wedge y \neq z$, whereas $\mathsf{distinct}(x_1; \ldots; x_n)$ means that all x_i are different.

2.4.1 Overview of the Decision Procedure

We first give a high-level description of the decision procedure. For a conjunction ϕ of literals over the theory of trees parametrized by $\mathscr{L}_\mathscr{C}$ and α:

1. apply purification to separate ϕ into $\phi_T \wedge \phi_B \wedge \phi_C$ where:
   - ϕ_T contains only literals over tree terms
   - ϕ_C contains only literals over terms from $\mathscr{L}_\mathscr{C}$
   - ϕ_B contains only literals of the form $c = \alpha(t)$ where c is a variable from $\mathscr{L}_\mathscr{C}$ and t is a tree variable
2. flatten all terms and eliminate the selectors left and right
3. apply unification on the tree terms, detecting possible unsatisfiability within the term algebra theory
4. if unification did not fail, project the constraints on tree terms obtained from unification to the formula ϕ_C in the collection theory, yielding a new formula ϕ'_C
5. establish the satisfiability of ϕ with a decision procedure for $\mathscr{L}_\mathscr{C}$ applied to ϕ'_C

As should be clear from this summary, the overall strategy is to reduce a formula over trees and their abstract $\mathscr{L}_\mathscr{C}$-values to a $\mathscr{L}_\mathscr{C}$ formula, for which a decision procedure is assumed

$$
\begin{array}{rcll}
T & ::= & t \,|\, \mathsf{Leaf} \,|\, \mathsf{Node}(T, E, T) \,|\, \mathsf{left}(T) \,|\, \mathsf{right}(T) & \text{Tree terms} \\
C & ::= & c \,|\, \alpha(t) \,|\, \mathcal{T}_{\mathscr{C}} & \mathscr{C}\text{-terms} \\
F_T & ::= & T = T \,|\, T \neq T & \text{Equations over trees} \\
F_C & ::= & C = C \,|\, \mathcal{F}_{\mathscr{C}} & \text{Formulas of } \mathscr{L}_{\mathscr{C}} \\
E & ::= & \text{variables of type } \mathscr{E} & \\
\phi & ::= & \bigwedge F_T \wedge \bigwedge F_C & \text{Conjunctions} \\
\psi & ::= & \phi \,|\, \neg\phi \,|\, \phi \vee \phi \,|\, \phi \wedge \phi \,|\, \phi \implies \phi \,|\, \phi \iff \phi & \text{Formulas}
\end{array}
$$

$\mathcal{T}_{\mathscr{C}}$ and $\mathcal{F}_{\mathscr{C}}$ represent terms and formulas of $\mathscr{L}_{\mathscr{C}}$ respectively. Formulas are assumed to be closed under negation.

Figure 2.7 – Syntax of the parametric logic.

$$
\begin{array}{rcl}
[\![\mathsf{Node}(T_1, e, T_2)]\!] & = & \mathsf{Node}([\![T_1]\!], [\![e]\!]_{\mathscr{C}}, [\![T_2]\!]) \\
[\![\mathsf{Leaf}]\!] & = & \mathsf{Leaf} \\
[\![\mathsf{left}(\mathsf{Node}(T_1, e, T_2))]\!] & = & [\![T_1]\!] \\
[\![\mathsf{right}(\mathsf{Node}(T_1, e, T_2))]\!] & = & [\![T_2]\!] \\
[\![\alpha(t)]\!] & & \text{given by the catamorphism} \\
[\![T_1 = T_2]\!] & = & [\![T_1]\!] = [\![T_2]\!] \\
[\![T_1 \neq T_2]\!] & = & [\![T_1]\!] \neq [\![T_2]\!] \\
[\![C_1 = C_2]\!] & = & [\![C_1]\!]_{\mathscr{C}} = [\![C_2]\!]_{\mathscr{C}} \\
[\![\mathcal{F}_{\mathscr{C}}]\!] & = & [\![\mathcal{F}_{\mathscr{C}}]\!]_{\mathscr{C}} \\
[\![\neg\phi]\!] & = & \neg[\![\phi]\!] \\
[\![\phi_1 \star \phi_2]\!] & = & [\![\phi_1]\!] \star [\![\phi_2]\!] \\
 & & \text{where } \star \in \{\vee, \wedge, \implies, \iff\}
\end{array}
$$

Figure 2.8 – Semantics of the parametric logic.

to be available. We next define our decision problem more precisely, then present the core steps and show their soundness. Section 2.5 provides remaining subtle steps of the decision procedure and proves its completeness for a certain class of abstraction functions.

2.4.2 Syntax and Semantics of our Logic

Figure 2.7 shows the syntax of our logic. Figure 2.8 describes its semantics. The description refers to the catamorphism α, as well as the semantics $[\![\ ]\!]_{\mathscr{C}}$ of the parameter theory $\mathscr{L}_{\mathscr{C}}$.

2.4.3 Key Steps of the Decision Procedure

We describe a decision procedure for conjunctions of literals in our parametric theory. To lift the decision procedure to formulas of arbitrary boolean structure, we can apply the DPLL(T) approach [GHN+04].

Purification. In the first step of our decision procedure, we separate the conjuncts of our formula into literals over tree terms on one side, literals of $\mathscr{L}_{\mathscr{C}}$ on the other side, and finally the literals containing the catamorphism to connect the two sides. By the syntax of formulas, a literal in the formula can only combine tree terms with terms of $\mathscr{L}_{\mathscr{C}}$ when the tree terms occur as arguments of the abstraction function α. It therefore suffices to replace all such applications by fresh variables of $\mathscr{L}_{\mathscr{C}}$ and add the appropriate binding equalities to the formula:

$$\mathscr{F}_{\mathscr{C}} \leadsto t_F = T \land c_F = \alpha(t_F) \land \mathscr{F}_{\mathscr{C}}[\alpha(T) \mapsto c_F]$$

In the rewrite rule above, T denotes any tree term, c_F and t_F are fresh in the new formula.

Flattening of tree terms. We then flatten tree terms in a straightforward way. If t and t_F denote tree variables, T_1 and T_2 non-variable tree terms and T an arbitrary tree term, we repeatedly apply the following five rewrite rules until none applies ($\doteq$ denotes one of $\{=, \neq\}$):

$$\begin{aligned}
T \doteq \mathsf{Node}(T_1, E, T_2) &\leadsto t_F = T_1 \land T \doteq \mathsf{Node}(t_F, E, T_2) \\
T \doteq \mathsf{Node}(t, E, T_2) &\leadsto t_F = T_2 \land T \doteq \mathsf{Node}(t, E, t_F) \\
T \doteq \mathsf{left}(T_1) &\leadsto t_F = T_1 \land T \doteq \mathsf{left}(t_F) \\
T \doteq \mathsf{right}(T_1) &\leadsto t_F = T_1 \land T \doteq \mathsf{right}(t_F) \\
T_1 \doteq t &\leadsto t \doteq T_1 \\
t \neq T_1 &\leadsto t_F = T_1 \land t \neq t_F
\end{aligned}$$

where t_F is always a fresh variable. It is straightforward to see that this rewriting always terminates.

Elimination of selectors. The next step is to eliminate terms of the form $\mathsf{left}(t)$ and $\mathsf{right}(t)$. We do this by applying the following rewrite rules:

$$\begin{aligned}
t = \mathsf{left}(t_1) &\leadsto t_1 = \mathsf{Node}(t_L, e, t_R) \land t = t_L \\
t = \mathsf{right}(t_1) &\leadsto t_1 = \mathsf{Node}(t_L, e, t_R) \land t = t_R
\end{aligned}$$

Here we use an assumption that the original formula was well-typed, which ensures that selectors are not applied to Leaf nodes. Again, e, t_L and t_R denote fresh variables of the proper types.

These first three steps yield a normalized conjunctive formula where all literals are in exactly one of following three categories:

– literals over tree terms, which are of one of the following forms:

$$t_1 = t_2, \quad t = \mathsf{Node}(t_1, E, t_2), \quad t_1 \neq t_2$$

(Note that disequalities are always between variables.)

– binding literals, which are of the form:

$$c = \alpha(t)$$

– literals over terms of $\mathscr{L}_{\mathscr{C}}$, which do not contain tree variables or applications of α, and whose specific form depends on the parameter theory $\mathscr{L}_{\mathscr{C}}$.

Case splitting. For simplicity of the presentation, we describe our procedure non-deterministically by splitting the decision problem into a collection of problems of simpler structure (this is a non-deterministic polynomial process). Consider the set $\{t_1, \ldots, t_n, \mathsf{Leaf}\}$ of tree variables appearing in the normalized formula, augmented with the constant term Leaf. We solve the decision problem for each possible partitioning of this set into equivalence classes. Let $\sim$ denote an equivalence corresponding to such a partitioning. We generate our subproblem by adding to the original problem, for each pair of terms (T_i, T_j) in the set, the constraint $T_i = T_j$ if $T_i \sim T_j$, and $T_i \neq T_j$ otherwise. Consider now the set $\{e_1, \ldots, e_m\}$ of variables denoting elements of type $\mathscr{E}$. We again decompose our subproblem according to all possible partitionings over this set, adding equalities and disequalities for all pairs (e_i, e_j) in the same way as for tree variables. The original problem is satisfiable if and only if any of these subproblems is satisfiable. The remaining steps of the decision procedure are applied to each subproblem separately.

Unification. At this point, we apply unification on the positive tree literals. Following [BS01], we describe the process using inference rules consisting of transformations on *systems*. A system is the pair, denoted $P; S$, of a set P of equations to unify, and a set S of solution equations. Equations range over tree variables and element variables. The special system $\perp$ represents failure. The set of equations S has the property that it is of the form $\{t_1 = T_1, \ldots, t_n = T_n, e_1 = e_i, \ldots, e_m = e_j\}$, where each tree variable t_i and each element variable e_i on the left-hand side of an equality does not appear anywhere else in S. Such a set is said to be in *solved form*, and we associate to it a substitution function σ_S. Over tree terms, it is defined by $\sigma_S = \{t \mapsto T \mid (t = T) \in S\}$. The definition over element variables is similar. The inference rules are the usual rules for unification adapted to our particular case, and are shown in Figure 2.9.

Any algorithm implementing the described inference system has the property that on a set of equations to unify, it will either fail, or terminate with no more equations to unify and a system $\emptyset; S$ describing a solution and its associate function σ_S.

If for any disequality $t_i \neq t_j$ or $e_i \neq e_j$, we have that respectively $\sigma_S(t_i) = \sigma_S(t_j)$ or $\sigma_S(e_i) = \sigma_S(e_j)$, then our (sub)problem is unsatisfiable. Otherwise, the tree constraints are satisfiable and we move on to the constraints on the collection type $\mathscr{C}$.

Normal form after unification. After applying unification, we can represent the original formula as a disjunction of formulas in a normal form. Let σ_S be the substitution function

$$\textsc{Trivial}\ \frac{T \stackrel{?}{=} T \cup P'; S}{P'; S} \qquad\qquad \textsc{Symbol Clash}\ \frac{\mathsf{Leaf} \stackrel{?}{=} \mathsf{Node}(\ldots) \cup P'; S}{\bot}$$

$$\textsc{Orient}\ \frac{\{T_1 \stackrel{?}{=} t\} \cup P'; S \qquad T_1 \text{ is not a variable}}{\{t \stackrel{?}{=} T_1\} \cup P'; S}$$

$$\textsc{Occurs Check}\ \frac{\{t \stackrel{?}{=} T\} \cup P'; S \qquad t \text{ appears in } T \text{ but } t \neq T}{\bot}$$

$$\textsc{Term Variable Elimination}\ \frac{\{t \stackrel{?}{=} T\} \cup P'; S \qquad t \text{ does not appear in } T}{P'[t \mapsto T]; S[t \mapsto T] \cup \{t = T\}}$$

$$\textsc{Element Variable Elimination}\ \frac{\{e_1 \stackrel{?}{=} e_2\} \cup P'; S}{P'[e_1 \mapsto e_2]; S[e_1 \mapsto e_2] \cup \{e_1 = e_2\}}$$

$$\textsc{Decomposition}\ \frac{\{\mathsf{Node}(T_1, e, T_2) \stackrel{?}{=} \mathsf{Node}(T_1', e', T_2')\} \cup P'; S}{\{T_1 \stackrel{?}{=} T_1', T_2 \stackrel{?}{=} T_2', e \stackrel{?}{=} e'\} \cup P'; S}$$

Figure 2.9 – Unification Rules.

obtained from unification. Let $\bar{t}$ be the vector of n variables t_i for which $\sigma_S(t_i) = t_i$; we call such variables *parameter variables*. Let $\bar{u}$ denote the vector of the remaining m tree variables; for these variable $\sigma_S(u_j)$ is an expression built from $\bar{t}$ variables using Node and Leaf, they are thus uniquely given as a function of parameter variables. By the symbol v_i we denote a term variable that is either a parameter variable t_i or a non-parameter variable u_i. Using this notation, we can represent (a disjunct of) the original formula in the form:

$$\bar{u} = \bar{T}(\bar{t}) \wedge N(\bar{u}, \bar{t}) \wedge M(\bar{u}, \bar{t}, \bar{c}) \wedge F_E \wedge F_C \tag{2.3}$$

where

1. $\bar{T}$ are vectors of expressions in the language of algebraic data types, expressing non-parameter term variables $\bar{u}$ in terms of the parameter variables $\bar{t}$;
2. $N(\bar{u}, \bar{t})$ denotes a conjunction of disequalities of term variables u_i, t_i that, along with $\bar{T}$, completely characterize the equalities and disequalities between the term variables. Specifically, N contains:
   (a) a disequality $t_i \neq t_j$ for every pair of distinct parameter variables;
   (b) a disequality $t_i \neq u_j$ for every pair of a parameter variable and a non-parameter variable for which the term $T_j(\bar{t})$ is not identical to t_i
   (c) a disequality $t_i \neq \mathsf{Leaf}$ for each parameter variable t_i.

   Note that for the remaining pairs of variables u_i and u_j, either the equality holds and $T_i(\bar{t}) = T_j(\bar{t})$ or the disequality holds and follows from the other disequalities and the fact that $T_i \neq T_j$. Note that, if $\bar{u} = u_1, \ldots, u_m$ and $\bar{t} = t_1, \ldots, t_n$, then the constraint $N(\bar{u}, \bar{t})$ can be denoted by $\mathsf{distinct}(u_1, \ldots, u_m; t_1; \ldots; t_n; \mathsf{Leaf})$;
3. $M(\bar{u}, \bar{t}, \bar{c})$ denotes a conjunction of formulas $c_i = \alpha(v_i)$ where v_i is a term variable and c_i is a collection variable;
4. F_E is a conjunction of literals of the form $e_i = e_j$ and $e_i \neq e_j$ for some element variables e_i, e_j;
5. F_C is a formula of the logic of collections (Figure 2.7).

Partial evaluation of the catamorphism. We next partially evaluate the catamorphism α with respect to the substitution σ_S obtained from unification. More precisely, we repeatedly apply the following rewriting on terms to terms contained in the subformula $M(\bar{u}, \bar{t}, \bar{c})$:

$$\begin{aligned} \alpha(u) &\rightsquigarrow \alpha(\sigma_S(u)) \\ \alpha(\mathsf{Node}(t_1, e, t_2)) &\rightsquigarrow \mathsf{combine}(\alpha(t_1), e, \alpha(t_2)) \\ \alpha(\mathsf{Leaf}) &\rightsquigarrow \mathsf{empty} \end{aligned}$$

After this transformation, α applies only to parameter variables. We introduce a variable c_i of $\mathscr{L}_\mathscr{C}$ to ensure that for each parameter t_i we have an equality of the form $c_i = \alpha(t_i)$, unless such conjunct is already present. After adding conjuncts $c_i = \alpha(t_i)$ we can replace all occurrences

of $\alpha(t_i)$ with c_i. We can thus replace, without changing the satisfiability of the formula (2.3), the subformula $M(\bar{u}, \bar{t}, \bar{c})$ with

$$M^1(\bar{t}, \bar{c}) \wedge F_C^1$$

where M^1 contains only conjunctions of the form $c_i = \alpha(t_i)$ and F_C^1 is a formula in $\mathscr{L}_\mathscr{C}$.

Example. This is a crucial step of our decision procedure, and we illustrate it with a simple example. If $\bar{u} = \bar{T}(\bar{t})$ is simply the formula $u = \mathsf{Node}(t_1, e, t_2)$, then a possible formula N is

$$\mathsf{distinct}(t_1; t_2; u; \mathsf{Leaf})$$

A possible formula M is $c = \alpha(u) \wedge c_1 = \alpha(t_1)$. After the partial evaluation of the catamorphism and introducing variable c_2 for $\alpha(t_2)$, we can replace M with

$$c_1 = \alpha(t_1) \wedge c_2 = \alpha(t_2) \wedge c = \mathsf{combine}(c_1, e, c_2)$$

where we denote the first two conjuncts by $M^1(c_1, c_2)$ and the third conjunct by F_C^1. (Here, combine is an expression in $\mathscr{L}_\mathscr{C}$ defining the catamorphism.)

Normal form after evaluating catamorphism. We next replace $\bar{u}$ by $\bar{T}(\bar{t})$ in (2.3) and obtain formula of the form

$$D \wedge E \tag{2.4}$$

where

1. $D \;\equiv\; N(\bar{T}(\bar{t}), \bar{t}) \wedge M^1(\bar{t}, \bar{c})$
2. $E \;\equiv\; F_E \wedge F_C \wedge F_C^1$

Expressing existence of distinct terms. Note that E already belongs to the logic of collection $\mathscr{L}_\mathscr{C}$. To reduce (2.4) to a formula in $\mathscr{L}_\mathscr{C}$, it therefore suffices to have a mapping from D to some $\mathscr{L}_\mathscr{C}$-formula D_M. Observe that by using **true** as D_M we obtain a sound procedure for proving unsatisfiability. While useful, such procedure is not complete. To ensure completeness, we require that D and D_M are equisatisfiable. The appropriate mapping from D to D_M depends on $\mathscr{L}_\mathscr{C}$, and the properties of α. In Section 2.5 we give such mappings that ensure completeness for a number of logics $\mathscr{L}_\mathscr{C}$ and catamorphisms α.

Invoking decision procedure for collections. Having reduced the problem to a formula in $\mathscr{L}_\mathscr{C}$ we invoke a decision procedure for $\mathscr{L}_\mathscr{C}$.

2.4.4 Soundness of the Decision Procedure

We show that each of our reasoning steps results in a logically sound conclusion. The soundness of the purification and flattening steps is straightforward: each time a fresh variable is introduced, it is constrained by an equality, so any model of the original formula will naturally extend to a model for the rewritten formula which contains additional fresh variables. Conversely, the restriction of any model for the rewritten formula to the initial set of variables will be a model for the original formula.

Our decision procedure relies on two case splittings. We will give an argument for the splitting on the partitioning of tree variables. The argument for the splitting on the partitioning of content variables is then essentially the same. Let us call ϕ the formula before case splitting. Observe that for each partitioning, the resulting subproblem contains a strict superset of the constraints of the original problem, that is, each subproblem is expressible as a formula $\phi \wedge \psi$, where ψ does not contain variables not appearing in ϕ. Therefore, if, for any of the subproblems, there exists a model $\mathscr{M}$ such that $\mathscr{M} \models \phi \wedge \psi$, then $\mathscr{M} \models \phi$ and $\mathscr{M}$ is also a model for the original problem. For the converse, assume the existence of a model $\mathscr{M}$ for the original problem. Construct the relation $\sim$ over the tree variables $t_1, \ldots, t_n$ of ϕ as follows:

$$t_i \sim t_j \iff \mathscr{M} \models t_i = t_j$$

Clearly, $\sim$ is an equivalence relation and thus there is a subproblem for which the equality over the tree variables is determined by $\sim$. It is not hard to see that $\mathscr{M}$ is a model for that subproblem. It is therefore sound to reduce the satisfiability of the main problem to the satisfiability of at least one of the subproblems.

Our unification procedure is a straightforward adaptation from a textbook exposition of the algorithm and the soundness arguments can be lifted from there [BS01, Page 451].

The soundness of the evaluation of α follows from its definition in terms of empty and combine. Introducing fresh variables c_i in the form of equalities $c_i = \alpha(t_i)$ is again sound, following the same argument as for the introduction of tree variables during flattening. The subsequent replacement of terms of the form $\alpha(t_i)$ by their representative variable c_i is sound: any model for the formula without the terms $\alpha(t_i)$ can be trivially extended to include a valuation for them. Finally, the replacement of the tree variables $\bar{u}$ by the terms $\bar{T}(\bar{t})$ is sound, because unification enforces that any model for the formula before the substitution must have the same valuation for u_i and the corresponding term T_i. Therefore, there is a direct mapping between models for the formula before and after the substitution.

2.4.5 Complexity of the Reduction

Our decision procedure reduces formulas to normal form in non-deterministic polynomial time because it performs guesses of equivalence relations on polynomially many variables, runs the unification algorithm, and does partial evaluation of the catamorphism at most once

for each appropriate term in the formula. The reduction is therefore in the same complexity class as the pure theory of algebraic data types [BST07]. In addition to the reduction, the overall complexity of the decision problem also depends on the formula D_M, and on the complexity of solving the resulting constraints in the collection theory.

2.5 Completeness

We next describe the strategy for computing the formula D_M from Section 2.4 for a broad class of catamorphisms. We prove that a computation following our strategy results in a sound and *complete* overall decision procedure.

2.5.1 Canonical Set Abstraction

We first give a complete procedure for the canonical set abstraction, where $\mathscr{C}$ is the structure of all finite sets with standard set algebra operations, and α is given by

$$\begin{aligned} \mathsf{empty} &= \emptyset \\ \mathsf{combine}(c_1, e, c_2) &= c_1 \cup \{e\} \cup c_2 \end{aligned}$$

Observations about α. Note that, for each term $t \neq \mathsf{Leaf}$, $\alpha(t) \neq \emptyset$. Let $e \in \mathscr{E}$ and consider the set $S = \alpha^{-1}(\{e\})$ of terms that map to $\{e\}$. Then S is the set of all non-leaf trees that have e as the only stored element, that, is, S is the least set such that

1. $\mathsf{Node}(\mathsf{Leaf}, e, \mathsf{Leaf}) \in S$, and
2. $t_1, t_2 \in S \implies \mathsf{Node}(t_1, e, t_2) \in S$.

Thus, $\alpha^{-1}(\{e\})$ is infinite. More generally, $\alpha^{-1}(c)$ is infinite for every $c \neq \emptyset$, because each tree that maps into a one-element subset of c extends into some tree that maps into c.

Expressing existence of distinct terms using sets. We can now specify the formula D_M that is equisatisfiable with the formula D in (2.4).

Definition 2.1. *If $c_1, \ldots, c_n$ are the free set variables in D, then (for theory $\mathscr{C}$ and α given above) define D_M as*

$$\bigwedge_{i=1}^{n} c_i \neq \emptyset$$

To argue why this simple choice of D_M gives a complete decision procedure, we prove the following.

Lemma 2.2. *Let D_0 be a conjunction of n disequalities of terms built from tree variables $t_1, \ldots, t_m$ and symbols $\mathsf{Node}, \mathsf{Leaf}$. Suppose that D_0 does not contain a trivial disequality $T \neq T$*

for any term T. If $A_1, \ldots, A_m$ are sets of trees such that $|A_j| > n$ for all $1 \le j \le m$, then D_0 has a satisfying assignment such that for each j, the value t_j belongs to A_j.

Proof. We first show that we can reduce the problem to a simpler one where the disequalities all have the form $t_a \neq T_b$, then show how we can construct a satisfying assignment for a conjunction of such disequalities.

We start by rewriting each disequality $T_i \neq T_i{}'$ in the form:

$$\neg \left(\begin{array}{ll} & \ldots \\ \wedge & t_a = C_a(t_{a1}, \ldots, t_{ak_a}) \\ \wedge & \ldots \end{array} \right)$$

where the conjunction of equalities is obtained by unifying the terms T_i and $T_i{}'$. The conjunction is non-empty because the statement of the lemma assumes that T_i and $T_i{}'$ are not syntactically identical. Here, the expressions of the form $C_a(t_{a1}, \ldots, t_{ak_a})$ denote terms built using Node, Leaf and the variables $t_{a1}, \ldots, t_{ak_a}$, where each of the variables appears at least once in the term. After applying this rewriting to all disequalities and converting the resulting formula to disjunctive normal form, we obtain a problem of the form

$$\bigvee_s L_{s1} \wedge \ldots \wedge L_{sn}$$

Note that in each conjunction, there is exactly one conjunct of the form $t_a \neq C_a(t_{a1}, \ldots, t_{ak_a})$ for each of the n disequalities $T_i \neq T_i{}'$ of the original problem. Notice as well that each variable t_a can be on the left-hand side of several disequalities in the same conjunction. From the form of the equations obtained using unification, we know that the set of variables $\{t_{a1}, \ldots, t_{ak_a}\}$ never contains t_a. This formula is logically equivalent to the original one from the statement of the lemma. To show that it is satisfiable, we pick an arbitrary disjunct and show that it is satisfiable.

We construct a satisfying assignment for such a conjunction as follows. For the first step, we start by collecting in a set P_1 all disequalities of the form $t_1 \neq T$, where T is a ground term. (Note that P_1 may be empty.) We pick for t_1 a value T_1 in A_1 different from all such Ts. This is always possible because there are n disequalities in the conjunction and $|A_1| > n$. We substitute in the entire formula T_1 instead of t_1. Because t_1 cannot appear on both left and right-hand side of an equation, in the resulting formula, all ground disequalities result from the grounding of the disequalities in P_1 and reduce to **true**. We eliminate the disequalities of P_1 from the set of disequalities.

For all indices $j \in \{2, \ldots, m\}$ do the following. Collect the set P_j of all disequalities of the form $t_j \neq T$ and $T \neq C(t_j)$ (in the second form, $C(t_j)$ denotes a term built with Leaf, Node, at least one occurrence of t_j, and no other variable). There are clearly no more than n such disequalities in P_j. For each of these disequality literals L_k, there is *at most* one value v_k^j for t_j which contradicts it: it is either the ground term T or its subterm. Because $|A_j| > n$, there

exists a term $T_j \in A_j \setminus \{v^j_k\}_k$. Substitute T_j instead of t_j in the entire conjunction. This ensures that all disequalities in P_j hold. Remove the disequalities in P_j from the conjunction. We then proceed with t_{j+1}. The procedure terminates in m steps with an assignment mapping t_j to T_j for $1 \leq j \leq m$. Moreover, at this point there are no ground equations left and no variables left, so all conjuncts have been eliminated and satisfied. □

Remark. Lemma 2.2 above is a strengthening of the Independence of Disequations Lemma [CD94, Page 178], [Mal71]. Namely, the statement in [CD94, Page 178] requires the sets A_j to be infinite, whereas we showed above (using a new, more complex proof) that it suffices for A_j to have more elements than there are disequalities. While the original weaker version suffices for Lemma 2.3, we need our stronger statement in Section 2.5.3.

Lemma 2.3. *For $\mathscr{C}$ denoting the structure of finite sets and α given as above, $\exists \bar{t}: D$ is equivalent to D_M.*

Proof. Let $\bar{t}$ be $t_1, \ldots, t_n$. Fix values $c_1, \ldots, c_n$. We first show $\exists \bar{t}: D$ implies D_M. Pick values $t_1, \ldots, t_n$ for which D holds. Then $t_i \neq \mathsf{Leaf}$ holds because this conjunct is in D. Therefore, $\alpha(t_i) \neq \emptyset$ by the above observations about α. Because $c_i = \alpha(t_i)$ is a conjunct in D, we conclude $c_i \neq \emptyset$. Therefore, D_M holds as well.

Conversely, suppose D_M holds. This means that $c_i \neq \emptyset$ for $1 \leq i \leq n$. Let $A_i = \alpha^{-1}(c_i)$ for $1 \leq i \leq n$. Then the sets A_i are all infinite by the above observations about α. By Lemma 2.2 there are values $t_i \in A_i$ for $1 \leq i \leq n$ such that the disequalities in $N(\bar{T}(\bar{t}), \bar{t})$ hold. By definition of A_i, $M^1(\bar{t}, \bar{c})$ is also true. Therefore, D is true in this assignment. □

Complexity for the canonical set abstraction. We have observed earlier that the reduction to $\mathscr{L}_{\mathscr{C}}$ is an NP process. There are several decision procedures that support reasoning about sets of elements and support standard set operations. One of the most direct approaches to obtain such a decision procedure [KR05] is to use an encoding into first-order logic, and observe that the resulting formulas belong to the Bernays-Schönfinkel-Ramsey class of first-order logic with a single universal quantifier. Checking satisfiability of such formulas is NP-complete [BGG97]. It is also possible to extend this logic to allow stating that two sets have the same cardinality, and the resulting logic is still within NP [KR07]. Because the reduction, the generation of D_M and the decision problem for $\mathscr{L}_{\mathscr{C}}$ are all in NP, we conclude that the decision problem for algebraic data types with the canonical set abstraction belongs to NP.

2.5.2 Infinitely Surjective Abstractions

The canonical set abstraction is a special case of what we call *infinitely surjective abstractions*, for which we can compute the formula D_M.

Definition 2.4 (Infinitely Surjective Abstraction). *If S is a set of trees, we call a domain $\mathscr{C}$ and a catamorphism α an infinitely surjective S-abstraction if and only if*
- $\alpha^{-1}(\alpha(t))$ *is finite for* $t \in S$,
- $\alpha^{-1}(\alpha(t))$ *is infinite for* $t \notin S$,
- *there exists a formula M in the collection theory such that $M(c)$ iff $\alpha^{-1}(c)$ is infinite.*

The canonical set abstraction is an infinitely surjective {Leaf}-abstraction, with $M \equiv c \neq \emptyset$. Other infinitely surjective {Leaf}-abstractions are the tree size abstraction, which for a given tree computes its size as the number of internal nodes, the tree height abstraction, and the sortedness abstraction of Figure 2.5.

An example of infinitely surjective $\emptyset$-abstractions is the function vars(t) that computes the set of free variables in an abstract syntax tree t representing a lambda expression or a formula. Indeed, for each finite set s of variables (including $s = \emptyset$), there exist infinitely many terms t such that vars(t) = s.

We can compute D_M for an infinitely surjective S-abstraction whenever S is finite. The general idea is to add the elements $T_1, \ldots, T_m$ of S into the unification algorithm and guess arrangements over them. This will ensure that, in the resulting formula, the terms containing variables are distinct from all T_i. The formula D_M then states the condition $\bigwedge_{t \in S} c_i \neq \alpha(t)$. For any term T that is guessed not be equal to any element of S, we add the atom $M(\alpha(T))$ to the formula. We omit the details, as we will later present a more general construction, but we note that the above algorithm for {Leaf}-abstractions also works for $\emptyset$-abstractions.

2.5.3 Sufficiently Surjective Abstractions

We next present a more general completeness result, which requires collections to be classified either as being an image of sufficiently many terms, or as having one of finitely many shapes.

Definition 2.5 (Tree Shape and Size). *Let* SLeaf *be a new constant symbol and* SNode(t_1, t_2) *a new constructor symbol. The* shape *of a tree t, denoted $\check{s}(t)$, is a ground term built from* SLeaf *and* SNode(_,_) *as follows:*

$$\check{s}(\mathsf{Leaf}) = \mathsf{SLeaf}$$
$$\check{s}(\mathsf{Node}(T_1, e, T_2)) = \mathsf{SNode}(\check{s}(T_1), \check{s}(T_2))$$

We define the size *of a shape as:*

$$\mathsf{size}(\mathsf{SLeaf}) = 0$$
$$\mathsf{size}(\mathsf{SNode}(s_1, s_2)) = 1 + \mathsf{size}(s_1) + \mathsf{size}(s_2)$$

By extension, we define the size of a tree t to be the size of its shape.

Definition 2.6 (Shape Instantiation). *The* instantiation *of the shape of a tree t produces a copy*

of t where the values stored in the nodes are replaced by fresh variables:

$$\begin{aligned} \mathrm{inst}(t,i) &= \mathrm{inst}'(t,i,1) \\ \mathrm{inst}'(\mathsf{SNode}(s_1,s_2),i,j) &= \mathsf{Node}(\mathrm{inst}'(s_1,i,1+j), v^i_j, \mathrm{inst}'(s_2,i,1+j+\mathrm{size}(s_1))) \\ \mathrm{inst}'(\mathsf{SLeaf},i,j) &= \mathsf{Leaf} \end{aligned}$$

In the instantiation function, i determines the names of the fresh variables: the variables introduced by the instantiation $\mathrm{inst}(s,i)$ range from v^i_1 to $v^i_{\mathrm{size}(s)}$. Consequently, if $i \neq j$, then the terms $\mathrm{inst}(t,i)$ and $\mathrm{inst}(t,j)$ have no common variables. Note that for an abstraction function α and a tree shape s, the term $\alpha(\mathrm{inst}(s,i))$ contains no tree variables, so it can be rewritten (by completely evaluating α) into a term in the collection theory with the free variables $v^i_1,\ldots,v^i_{\mathrm{size}(s)}$. Note finally that for every tree term T, the formula $\mathrm{inst}(\check{s}(T),i) = T$ is satisfiable.

Definition 2.7 (Sufficient Surjectivity)**.** *We call an abstraction function* sufficiently surjective *if and only if, for each natural number $p > 0$ there exist, computable as a function of p,*
– *a finite set of shapes S_p*
– *a closed formula M_p in the collection theory such that $M_p(c)$ implies $|\alpha^{-1}(c)| > p$*
such that, for every term t, $M_p(\alpha(t))$ or $\check{s}(t) \in S_p$.

Note that M_p can introduce fresh variables as long as it is existentially closed and the decision procedure for the collection theory can handle positive occurrences of existential quantifiers.

The definition above implies:

$$\lim_{p\to\infty} \inf_{\check{s}(t)\notin S_p} |\alpha^{-1}(\alpha(t))| = \infty$$

We now show how we can build a formula D_M equisatisfiable with the formula D of (2.4) provided the aforementioned assumptions hold. We keep the notational convention that the parameter variables $\bar{t}$ range from t_1 to t_n and that the terms $\bar{T}(\bar{t})$ built around them range from $T_1(\bar{t})$ to $T_m(\bar{t})$. We also assume that for all variables t_i, the conjunct $c_i = \alpha(t_i)$ is present in D. This is consistent with the normal form we presented earlier, up to renaming of the variables.

In the following we take $p = \binom{n}{2} + n \cdot m$, where n is the dimension of the vector of term variables in $\bar{t}$ in D, and m is the dimension of the vector $\bar{T}(\bar{t})$ for derived terms in D. Consider the formula:

$$\begin{aligned} P \equiv & \bigwedge_{i=1}^{n} \left(M_p(c_i) \vee \bigvee_{s\in S_p} \mathrm{inst}(s,i) = t_i \right) \\ \wedge & \bigwedge_{j=1}^{m} \left(M_p(\alpha(T_j(\bar{t}))) \vee \bigvee_{s\in S_p} \mathrm{inst}(s,j+n) = T_j(\bar{t}) \right) \end{aligned}$$

Note that $\alpha(T_j(\bar{t}))$ can be rewritten as a term in the collection theory using the variables $\bar{c}$. Note that existentially quantifying P over the variables introduced by inst gives a formula that is always true, by the assumptions on M_p and S_p. Let P' be the disjunctive normal form of P. For every disjunct P^d of P', observe that for each t_i, either $M_p(\alpha(t_i))$ is a conjunct of P^d, or $\text{inst}(s,i) = t_i$ is a conjunct for exactly one s in S_p. The same observation holds for the terms $T_j(\bar{t})$.

We proceed as follows for each disjunct P^d of P'. We run unification over the equalities between terms. This can either result in a clash (because the shape assigned to a term $T_j(\bar{t})$ is in contradiction with the shapes assigned to the variables of $\bar{t}$), or produce new equalities between the freshly introduced element variables v. If there was a clash, we simply replace P^d by **false** and eliminate it from the formula. Otherwise, we add to P^d the new equalities produced by unification, yielding a disjunct P^d_U.

We next add additional conjuncts to P^d_U to obtain a formula D^d equisatisfiable with $D \wedge P^d_U$, as follows. Recall that D contains conjuncts of the forms:

- $t_i \neq t_j$ as part of $N(\bar{T}(\bar{t}), \bar{t})$,
- $t_i \neq T_j(\bar{t})$ as part of $N(\bar{T}(\bar{t}), \bar{t})$, and
- $c_i = \alpha(t_i)$ as part of $M^1(\bar{t}, \bar{c})$.

Initially, we set D^d to be the formula P^d_U. Then, for each disequality $T \neq T'$ in D (where T and T' can represent either variables or constructed terms), if in P^d_U we have $\text{inst}(s,i) = T$ and $\text{inst}(s,j) = T'$ for the same shape s, we add as a conjunct to D^d the disjunction $\bigvee_{1 \leq k \leq \text{size}(s)} v^i_k \neq v^j_k$. Finally, we replace in D^d all the equalities of the form $\text{inst}(s,i) = T$ by $\alpha(\text{inst}(s,i)) = \alpha(T)$. As we already observed, $\alpha(\text{inst}(s,i))$ can always be rewritten to a term in the collection theory by evaluating α. In the case where T is a variable t_i, $\alpha(T)$ is simply c_i. If it is a term $T_j(\bar{t})$, $\alpha(T)$ can be rewritten in terms of $\bar{c}$ by partially evaluating α.

The resulting formula D_M is $\bigvee_d D^d$. We claim that D_M is equisatisfiable with D.

Proof. (Preliminary transformations) Conjoining P to D does not change the satisfiability of the formula, and neither does the transformation to disjunctive normal form, so D is equisatisfiable with $D \wedge \bigvee_p P^d$. The unification procedure is equivalence preserving, so the formula after unification is still equisatisfiable. It therefore suffices to show that $D \wedge P^d_U$ is equisatisfiable with D^d.

(From trees to collections) First, observe that D^d is a consequence of $D \wedge P^d_U$. Indeed, $\bigvee v^i_k \neq v^j_k$ follows from $T \neq T'$, $\text{inst}(s,i) = T$, and $\text{inst}(s,j) = T'$. Also, $\alpha(\text{inst}(s,i)) = \alpha(T)$ follows from $\text{inst}(s,i) = T$, and partial evaluation of α is equivalence preserving. Therefore, if $D \wedge P^d_U$ has a model, then D^d as a consequence holds in this model. Thus D^d has a model. It remains to show the converse.

(From collections to trees) Assume $\mathscr{M}$ is a model for D^d, which specifies the values for element

and collection variables. We construct an extension of $\mathcal{M}$ with values for tree variables $\bar{t}$ such that $D \wedge P_U^d$ holds.

For those terms T for which P_U^d contains a conjunct $\mathrm{inst}(s,i) = T$, we assign T to be the value of $\mathrm{inst}(s,i)$ in $\mathcal{M}$ (indeed, $\mathcal{M}$ specifies the values of all free variables v_j^i in $\mathrm{inst}(s,i)$). In this assignment, the literals in P_U^d of the form $\alpha(T) = c_i$ are true for such terms T. Furthermore, all disequalities between such terms hold. Indeed, terms of different shape are distinct, and for terms of equal shape the formula D^d contains a disjunct $\bigvee v_k^i \neq v_k^j$ ensuring that the terms differ in at least one element.

It remains to define values for the trees T for which P_U^d does not contain a conjunct of the form $\mathrm{inst}(s,i) = T$ in such a way that the literals containing these trees are true. These are disequality literals, as well as literals of the form $\alpha(T) = c_i$, when T is a variable t_i. For each such tree T, the formula P_U^d contains the conjunct $M_p(\alpha(T))$, by construction of the disjunctive normal form. From the assumptions on M_p, from $M_p(\alpha(T))$ we conclude $|\alpha^{-1}(\alpha(T))| > p$. Therefore, there are at least $p+1$ trees T_k such that $\alpha(T_k) = \alpha(T)$. The number of disequalities in D is at most $\binom{n}{2} + n \cdot m$. Because $p = \binom{n}{2} + n \cdot m$, we can apply Lemma 2.2 to choose values for (at most) n trees satisfying (at most) p disequations from sets of size at least $p+1$. This choice of trees completes the assignment for the remaining tree variables such that all conjuncts of D^d hold. □

Model construction. Lemma 2.2 is constructive, so the proof above also gives model construction whenever 1) the underlying decision procedure for the collection provides model construction, and 2) there is an algorithm to compute, for each c where $M_p(c)$, a finite set of containing p elements t such that $\alpha(t) = c$.

Worst-case complexity of the decision problem. The reduction from the starting formula to the theory of collections is a non-deterministic polynomial-time algorithm that invokes the computation of the set S_p and the formula M_p. When S_p and M_p can be computed in polynomial time, then each of the disjuncts considered is of polynomial size. Our decision procedure is this case an NP reduction. This case applies to the three examples below. When the satisfiability for the collection theory is in NP (e.g. for multisets and sortedness), the overall satisfiability problem is also in NP.

2.5.4 Application to Multisets, Lists, and Sortedness

We now show that the list and multiset abstraction are sufficiently surjective abstractions, as is the sortedness abstraction for trees with distinct elements. (The set of these examples is not meant to be exhaustive.) In the following, let C_n denote the number of binary trees with n elements, and let K_m denote a its inverse, that is, the smallest natural number n such that $C_n > m$. The functions C_n and K_m are monotonic and computable.

Lists. Consider the catamorphism for infix traversal of the tree, for which we have $\mathsf{empty} = \mathsf{List}()$ and $\mathsf{combine}(c_1, e, c_2) = c_1 {+}{+} \mathsf{List}(e) {+}{+} c_2$. (Catamorphisms for pre-order and post-order traversal can be handled analogously.) We can use the following definitions for S_p and M_p:
– $S_p = \{s \mid \text{size}(s) < K_p\}$
– $M_p(c) \equiv \exists e_1, \ldots, e_{K_p} : \exists c' : \ c = \mathsf{List}(e_1, \ldots, e_{K_p}) \text{ ++ } c'$
S_p is the set of shapes with less than K_p nodes, while $M_p(c)$ expresses that the list c has at least K_p elements, so clearly for any tree t, either its shape $\check{s}(t)$ in is S_p, or it has more than K_p nodes and therefore $M_p(\alpha(t))$ holds. Finally, observe that for a list c of n elements, α maps exactly C_n distinct trees to c. Therefore, for any c such that $M_p(c)$ holds, we have $|\alpha^{-1}(c)| = C_{K_p}$, and $C_{K_p} > p$ by construction. Therefore, the infix traversal abstraction is sufficiently surjective and our completeness argument applies.

Multisets. Consider the multiplicity-preserving multiset abstraction, which is given by $\mathsf{empty} = \emptyset$ and $\mathsf{combine}(c_1, e, c_2) = c_1 \uplus \{e\} \uplus c_2$. We then take
– $S_p = \{s \mid \text{size}(s) < K_p\}$
– $M_p(c) \equiv \exists e_1, \ldots, e_{K_p} : \exists c' : \ c = \{e_1, \ldots, e_{K_p}\} \uplus c'$
For a multiset c with n elements (counting repetitions), there are *at least* C_n trees mapped by α to c, so the same argumentation as for lists applies.

Sortedness. Finally, consider the abstraction function described in Section 2.3.1 that checks the sortedness of trees. We mentioned in Section 2.5.2 that the version which allowed repeated elements is infinitely surjective. In contrast, in the case where the elements of the trees have to be distinct, it is not infinitely surjective. The reason is that the catamorphism also computes the minimal and maximal elements of the tree, and there are only finitely many sorted trees with distinct elements between a given minimum and maximum. The catamorphism is nevertheless sufficiently surjective. Indeed, we can take
– $S_p = \{s \mid \text{size}(s) < K_p\}$
– $M_p((a, b, \varsigma)) \equiv 1 + b - a \geq K_p$
where (a, b, ς) is the triple of the minimum, the maximum, and the sortedness of the tree, as computed by the catamorphism. Here, M_p essentially says that the range of values in the tree is sufficiently wide, so that enough distinct trees mapping to (a, b, ς) can be constructed.

In conclusion, the catamorphisms that map trees into lists, multisets, or sortedness property are also instances for which our decision procedure is complete.

2.5.5 A Note on Surjectivity

We should note that, despite their names, definitions 2.4 and 2.7 do not actually imply that α is a surjective function.

As an example, let α be the height abstraction, mapping trees into integers ($\mathbb{Z}$). From our

definitions, height is an infinitely surjective {Leaf}-abstraction, with $M \equiv c \geq 0$. Just like any infinitely surjective abstraction, height is also sufficiently surjective. We can take, for any p, $S_p \equiv$ SLeaf and $M_p \equiv M$. However, height is not surjective with respect to $\mathbb{Z}$, as there exist integers outside of the image of α. Indeed, no tree is mapped by α into a negative number.

The decidability result presented in this chapter is valid independently of the actual surjectivity of α. Intuitively, by introducing M_p, the decision procedure essentially limits the search for models to collection elements that are within the image of α.

2.6 Related Work

The results in this chapter were first described in [SDK10].

We have presented a non-deterministic description of our decision procedure. Our own implementation of the procedure does not follow the exact steps described in sections 2.4 and 2.5, although the completeness arguments are important to understand why the implementation is a valid (see Chapter 3). The group of Michael Whalen at the University of Minnesota have independently implemented a version of our procedure as an extension of the OpenSMT solver [HSWP12, BPST10], and used it to prove properties about network guards, applications that filter messages based on declarative rules.

One reason why we find our result useful is that it can leverage a number of existing decidability results. In [BST07], the authors present an abstract approach that can be used to obtain efficient strategies for reasoning about algebraic data types (without abstraction functions). For reasoning about sets and multisets one expressive approach is the use of the decidable array fragment [BM07]. Optimal complexity bounds for reasoning about sets and multisets in the presence of cardinality constraints have been established in [KR07, PK08b]. Building on these results, extensions to certain operations on vectors has been presented in [Mai09]. Reasoning about lists with concatenation can be done using Makanin's algorithm [Mak77] and its improvements [Pla04]. A different class of constraints uses rich string operations but imposes bounds on string length [BTV09]. Researchers have identified a number of laws in the area of manual program derivation, including laws that relate trees, lists, bags, and sets [HB94]. Our work can be viewed as a step towards automating some of these laws. Another example in this direction is the combinatory array logic [dMB09], which supports map operations on arrays but does not support the cardinality operator.

Our parameterized decision procedure is an example of an approach to combine logics (e.g. the logic of algebraic data types and a logic of collections). Standard results in this field are Nelson-Oppen combination [NO79]. Nelson-Oppen combination is not sufficient to encode catamorphisms because the disjointness conditions are not satisfied, but is very useful in obtaining interesting decidable theories to which the catamorphism can map an algebraic data type; such compound domains are especially of interest when using catamorphisms to encode invariants. There are combination results that lift the stable infiniteness restriction of

the Nelson-Oppen approach [TZ03, KGGT07, Fon07] as well as disjointness condition subject to a local finiteness condition [Ghi04]. An approach that allows theories to share set algebra with cardinalities is presented in [WPK09]. None of these results by itself handles the problem of reasoning about a catamorphism from the theory of algebraic data types. That said, our canonical set abstraction of algebraic data types is a new BAPA-reducible theory that fits into the framework [WPK09].

A technique for connecting two theories through homomorphic functions has been explored in [BG05]. We were not able to derive our decision procedure from [BG05], because the combination technique in [BG05] requires the homomorphism to hold between two copies of some shared theory Ω_0 that is locally finite, but our homomorphisms (i.e. catamorphisms) are defined on term algebras, which are not locally finite.

Related to our partial evaluation of the catamorphism is the phenomenon of local theory extensions [IJSS08], where axioms are instantiated only to terms that already exist syntactically in the formula. In our case of tree data types, the decision procedure must apply the axioms also to some consequences of the formula, obtained using unification, so the extented version of the local theory framework is needed. Concurrently with our result, the machinery of local theory extensions has been extended to certain homomorphisms in term algebras [SS09], although without considering homomorphisms that compute sets, multisets, or lists. We plan to investigate unifying the results in [SS09] with our notion of sufficiently surjective abstraction.

The proof decidability for term powers [KR03] introduces homomorphic functions that map a term into 1) a simplified "shape" term that ignores the stored elements and 2) the set of elements stored in the term. However, this language was meant to address reasoning about structural subtyping and not transformation of algebraic data types. Therefore, it does not support the comparison of the set of elements stored in distinct terms, and it would not be applicable to the verification conditions we consider in this thesis. Furthermore, it does not apply to multisets or lists.

In [ZSM06] researchers describe a decision procedure for algebraic data types with size constraints and in [MSZ07] a decision procedure for trees with numeric constraints that model invariants of red-black trees. Our decision procedure supports reasoning about not only size, but also the content of the data structure. We remark that [ZSM06] covers also the case of a finite number of atoms, whereas we have chosen to focus on the case of infinite set of elements $\mathscr{E}$. Term algebras have an extensively developed theory, and enjoy many desirable properties, including quantifier elimination [Mal71, Hod93]; quantifier elimination also carries over to many extensions of term algebras [CD94, KR03, RV01, ZSM06, ZSM05]. Note that in examples such as multisets with cardinality, we cannot expect quantifier elimination to hold because the quantified theory is undecidable [PK08a, Section 6].

Some aspects of our decision procedure are similar to folding and unfolding performed when using types to reason about data structures [RKJ08, KRJ09, WM00, NDQC07, Xi03]. One of

our goals was to understand the completeness or possible sources of incompleteness of such techniques. We do not aim to replace the high-level guidance available in such successful systems, but expect that our results can be used to further improve such techniques.

Several decision procedures are suitable for data structures in imperative programs [LQ06, LQ08, MN05, WKL+06, MS01]. These logics alone fail to describe algebraic data types because they cannot express extensional equality and disequality of entire tree data structure instances, or the construction of new data structures from smaller ones. Even verification systems reason about imperative programs [BFL+11, BHS07, Kun07, ZKR08] typically use declarative constructs and data types within specification annotations. Our decision procedure extends the expressive power of imperative specifications that can be handled in a predictable way.

The SMT-LIB standard [BdMR+10] for SMT provers currently does not support algebraic data types, even though several provers support it in their native input languages [BT07, dMB08b]. By providing new opportunities to use decision procedures based on algebraic data types, our results present a case in favor of incorporating such data types into standard formats. Our new decidability results also support the idea of using rich specification languages that admit certain recursively defined functions.

2.7 Conclusions

This chapter introduced a decision procedure that extends the well-known decision procedure for algebraic data types. The extension enables reasoning about the relationship between the values of the data structure and the values of a recursive function (catamorphism) applied to the data structure. The presence of catamorphisms gives great expressive power and provides connections to other decidable theories, such as sets, multisets, lists. It also enables the computation of certain recursive invariants. Our decision procedure has several phases: the first phase performs unification and solves the recursive data structure parts, the second applies the recursive function to the structure generated by unification. The final phase is more subtle, is optional from the perspective of soundness, but ensures completeness of the decision procedure.

Automated decision procedures are widely used for reasoning about imperative programs. Functional programs are claimed to be more amenable to automated reasoning—this was among the original design goals of functional programming, and has been supported by experience from type systems and interactive proof assistants. Our decision procedure further supports this claim, by showing a wide range of properties that can be predictably proved about functional data structures.

3 Satisfiability Modulo Recursive Functions

This chapter introduces a semi-decision procedure for checking satisfiability of expressive correctness properties of recursive first-order functional programs. Our procedure is sound for counterexamples and for proofs of terminating functions. It is terminating and thus complete for many important classes of specifications, including all satisfiable formulas and all formulas where recursive functions are surjective and satisfy the criteria described in Chapter 2. We implemented our procedure and integrated it with the Z3 SMT solver and the Scala compiler. Using our system, Leon, we verified detailed correctness properties of functional data structure implementations, as well as syntax tree manipulations. We have found Leon to be fast both for finding counterexamples and for finding correctness proofs. The algorithm presented in this chapter is also at the core of Kaplan, our declarative programming language described in Chapter 5.

3.1 Introduction

This chapter explores the problem of reasoning about functional programs. We reduce this problem to solving constraints representing precisely program semantics.

Our technique extends SMT solvers with recursive function definitions, so it can be used for all tasks where SMT solvers are used, including verification of functional and imperative programs, synthesis, and test generation. In this chapter we introduce this technique and its concrete implementation as a verifier, named Leon, for a functional subset of Scala [OSV11]. Leon enables the developer to state the properties as pure Scala functions and compile them using the standard Scala compiler.

Leon generates verification conditions that enforce:

1. that the functions meet their contracts,
2. that the preconditions at all function invocations are met, and that
3. the pattern-matching is complete for given preconditions.

Note that, to define an external property of a set of functions, the developer can write a

boolean-valued test function that invokes the functions of interest, and state a contract that the function always returns **true**. This approach is similar to, for instance, the one taken in the ACL2 system [KMM00, CDMV11]. Leon searches in parallel for proofs and counterexamples for all generated verification conditions.

The technical core of this chapter is an algorithm for combined proof and counterexample search, as well as a theoretical and experimental analysis of its effectiveness. We establish that our satisfiability procedure is:

1. sound for models: every model it returns makes the formula true,
2. terminating for all formulas that are satisfiable,
3. sound for proofs: if it reports UNSAT, then there are no models,
4. terminating for sufficiently surjective abstractions, whenever they are actually surjective, or when their co-domain can be described effectively,
5. satisfying the above properties if the declared functions are always terminating; more generally, UNSAT implies no "terminating models", moreover, each returned model leads to **true**.

In addition to the presentation of our satisfiability procedure, we describe the implementation of our system, called Leon, as a plugin for the Scala compiler. The system integrates with the SMT solver Z3 [dMB08b]. We present our results in verifying over 60 functions manipulating integers, sets, and algebraic data types, with detailed invariants of complex data structures such as red-black trees and amortized heap, and user code such as syntax tree manipulation. Leon verified detailed correctness properties about the content of data as well as completeness of pattern-matching expressions.

We have found that Leon is fast for finding both counterexamples and proofs for verification conditions. We thus believe that the algorithm holds great promise for practical verification of complex properties of computer systems. Leon and all benchmarks are available from http://lara.epfl.ch. An additional testament to the applicability of Leon is its use as part of Kaplan, described in Chapter 5: our experiences with Kaplan directly rely on the efficiency of Leon.

3.2 Examples

We now illustrate how Leon can be used to prove interesting properties about functional programs. Consider the following recursive datatype that represents formulas of propositional logic:

```
sealed abstract class Formula
case class And(lhs: Formula, rhs: Formula) extends Formula
case class Or(lhs: Formula, rhs: Formula) extends Formula
case class Implies(lhs: Formula, rhs: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class PropVar(id: Int) extends Formula
```

We can write a recursive function that simplifies a formula by rewriting implications into disjunctions as follows:

```
def simplify(f: Formula): Formula = (f match {
  case And(lhs, rhs) ⇒ And(simplify(lhs), simplify(rhs))
  case Or(lhs, rhs) ⇒ Or(simplify(lhs), simplify(rhs))
  case Implies(lhs, rhs) ⇒ Or(Not(simplify(lhs)), simplify(rhs)) // note the replacement
  case Not(f) ⇒ Not(simplify(f))
  case PropVar(_) ⇒ f
}) ensuring(isSimplified(_))
```

Here, isSimplified is a function that checks whether a given formula contains an implication as a subformula, and use it in a contract. The **ensuring** statement in the example is a postcondition written in Scala notation, stating that the function isSimplified evaluates to **true** on the result. We define isSimplified recursively as follows:

```
def isSimplified(f: Formula): Boolean = f match {
  case And(lhs, rhs) ⇒ isSimplified(lhs) && isSimplified(rhs)
  case Or(lhs, rhs) ⇒ isSimplified(lhs) && isSimplified(rhs)
  case Implies(_,_) ⇒ false
  case Not(f) ⇒ isSimplified(f)
  case PropVar(_) ⇒ true
}
```

Note that one would also typically write such an executable specification function for testing purposes. Using our procedure for satisfiability modulo computable functions, Leon can prove that the postcondition of simplify is satisfied for *every* input formula f.

Such subsets of values denoted by algebraic data types are known as refinement types [FP91]. Refinement types that are defined using functions such as isSimplified are sufficiently surjective abstractions, as we have seen in Chapter 2, and our satisfiability procedure acts as a *decision* procedure for such constraints (see Section 3.4.1).

Suppose now that we wish to prove that simplifying a simplified formula does not change it further. In other words, we want to prove that the property simplify(simplify(f))==simplify(f) holds for all formulas f. Because our programming and specification languages are identical, we can write such universally quantified statements as functions that return a boolean and whose postcondition is that they always return true. In this case, we would write:

```
def simplifyIsStable(f: Formula) : Boolean = {simplify(simplify(f)) == simplify(f)} holds
```

Because such specifications are common, we use the notation holds instead of the more verbose postcondition stating that the returned result should be an identity function with boolean argument **ensuring**(res⇒res). Our verification system proves this property instantly.

Another application for our technique is verifying that pattern-matching expressions are defined for all cases. Pattern-matching is a very powerful construct commonly found in

functional programming languages. Typically, evaluating a pattern-matching expression on a value not covered by any case raises a runtime error. Because checking that a **match** expression never fails is difficult in non-trivial cases (for instance, in the presence of guards), compilers in general cannot statically enforce this property. For instance, consider the following function that computes the set of variables in a propositional logic formula, assuming that the formula has been simplified:

```
def vars(f: Formula): Set[Int] = {
  require(isSimplified(f))
  f match {
    case And(lhs, rhs) ⇒ vars(lhs) ++ vars(rhs)
    case Or(lhs, rhs) ⇒ vars(lhs) ++ vars(rhs)
    case Not(f) ⇒ vars(f)
    case PropVar(i) ⇒ Set[Int](i)
  }
}
```

Although it is implied by the precondition that all cases are covered, the Scala compiler on this example will issue the warning:

```
Logic.scala: warning: match is not exhaustive!
missing combination        Implies
```

Previously, researchers have developed specialized analyses for checking such exhaustiveness properties [DSK08, Fer10]. Our system generates verification conditions for checking the exhaustiveness of all pattern-matching expressions, and then uses the same procedure to prove or disprove them as for the other verification conditions. It quickly proves that this particular example is exhaustive by unrolling the definition of isSimplified sufficiently many times to conclude that t can never be an Implies term. Note that Leon will also prove that all recursive calls to vars satisfy its precondition; it performs sound assume-guarantee reasoning.

Consider now the following function, that supposedly computes a variation of the negation normal form of a formula f:

```
def nnf(formula: Formula): Formula = formula match {
  case And(lhs, rhs) ⇒ And(nnf(lhs), nnf(rhs))
  case Or(lhs, rhs) ⇒ Or(nnf(lhs), nnf(rhs))
  case Implies(lhs, rhs) ⇒ Implies(nnf(lhs), nnf(rhs)) // problematic
  case Not(And(lhs, rhs)) ⇒ Or(nnf(Not(lhs)), nnf(Not(rhs)))
  case Not(Or(lhs, rhs)) ⇒ And(nnf(Not(lhs)), nnf(Not(rhs)))
  case Not(Implies(lhs, rhs)) ⇒ And(nnf(lhs), nnf(Not(rhs)))
  case Not(Not(f)) ⇒ nnf(f)
  case Not(PropVar(_)) ⇒ formula
  case PropVar(_) ⇒ formula
}
```

From the supposed roles of the functions simplify and nnf, one could conjecture that the operations are commutative. Because of the treatment of implications in the above definition of nnf, though, this is not the case. We can disprove this property by finding a counterexample to

```
def wrongCommutative(f: Formula) : Boolean = { nnf(simplify(f)) == simplify(nnf(f)) } holds
```

On this input, Leon reports

```
Error: Counter-example found:
f -> Implies(Not(And(PropVar(48), PropVar(47))),
                And(PropVar(46), PropVar(45)))
```

A consequence of our algorithm is that Leon never reports false positives (see Section 3.4.1). In this particular case, the counterexample clearly shows that there is a problem with the treatment of implications whose left-hand side contains a negation. Counterexamples such as this one are typically short and Leon finds them quickly.

As a final example of the expressive power of our system, we consider the question of showing that an implementation of a collection implements the proper interface. Consider the implementation of a set as red-black trees defined as follows:

```
sealed abstract class Color
case class Red() extends Color
case class Black() extends Color

sealed abstract class Tree
case class Empty() extends Tree
case class Node(color: Color, left: Tree, value: Int, right: Tree) extends Tree
```

To specify the operation on the trees in terms of the set interface that they are supposed to implement, we define an abstraction function in the style of those introduced in Chapter 2 that computes from a tree the set it represents:

```
def content(t : Tree) : Set[Int] = t match {
  case Empty() ⇒ Set.empty
  case Node(_, l, v, r) ⇒ content(l) ++ Set(v) ++ content(r)
}
```

Note that this is again a function one would write for testing purposes. The specification of insertion using this abstraction becomes very natural, despite the relative complexity of the operations:

```
def ins(x: Int, t: Tree): Tree = (t match {
  case Empty() ⇒ Node(Red(),Empty(),x,Empty())
  case Node(c,a,y,b) if x SLT y ⇒ balance(c, ins(x, a), y, b)
```

```
  case Node(c,a,y,b) if x EQ y ⇒ Node(c,a,y,b)
  case Node(c,a,y,b) if x SGT y ⇒ balance(c, a, y, ins(x, b))
}) ensuring (res⇒ content(res) == content(t) ++ Set(x))
```

where balance is defined as:

```
def balance(c: Color, a: Tree, x: Int, b: Tree): Tree = ((c,a,x,b) match {
  case (Black(),Node(Red(),Node(Red(),a,xV,b),yV,c),zV,d) ⇒
    Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),Node(Red(),a,xV,Node(Red(),b,yV,c)),zV,d) ⇒
    Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),Node(Red(),b,yV,c),zV,d)) ⇒
    Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),b,yV,Node(Red(),c,zV,d))) ⇒
    Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (c,a,xV,b) ⇒ Node(c,a,xV,b)
}) ensuring (res ⇒ content(res) == content(Node(c,a,x,b)))
```

For full-functional correctness, we can also write functions that check whether a tree is balanced and whether it satisfies the coloring properties. We used these checks to specify insertion and balancing operations. Leon proves all these properties of red-black tree operations. We present more such results in Section 3.6.

3.3 PureScala

We briefly describe the language that Leon supports for writing implementations and specifications. The language is a strict, purely functional, subset of Scala, and as such is executable and deterministic. We call it PureScala.

Data Types. We give an inductive definition of the set of data types $\mathcal{T}$ supported in PureScala:

$$\mathsf{Int} \in \mathcal{T} \qquad \mathsf{Boolean} \in \mathcal{T} \qquad \frac{T \in \mathcal{T}}{\mathsf{Set}[T] \in \mathcal{T}} \qquad \frac{T_1 \in \mathcal{T} \qquad T_2 \in \mathcal{T}}{\mathsf{Map}[T_1, T_2] \in \mathcal{T}} \qquad \frac{T_1 \in \mathcal{T} \qquad T_2 \in \mathcal{T}}{T_1 \Rightarrow T_2 \in \mathcal{T}}$$

$$\frac{\text{sealed abstract class } C}{C \in \mathcal{T}} \qquad \frac{\text{case class } C(\bar{n} : \bar{D}) \text{ extends } E \qquad \bar{D} \in \mathcal{T} \qquad E \in \mathcal{T}}{C \in \mathcal{T}}$$

The types Set and Map correspond to the standard Scala implementations of finite sets and maps respectively, as defined in the package `scala.collection.immutable`. An important feature is the ability to define (recursive) algebraic data types. In Scala, these types are defined using a hierarchy of special **case** classes. Algebraic data types are typically manipulated with pattern-matching, as shown in many examples throughout this thesis. A current limitation is that these user-defined types cannot take type parameters. The Map type represents finite,

immutable maps, which can thus be viewed as partial functions. The operations available on maps in PureScala are isDefinedAt, updated, which returns a new map with an additional key/pair value, and (...) (apply), which returns the value associated to a given key. The function type ⇒ should be viewed, in the context of PureScala, as a type of infinite map.

Because it supports unbounded data types and arbitrary recursive functions, PureScala is itself Turing-complete. This can be viewed both as an advantage and an inconvenient: on one hand, this expressive power guarantees that just about any constraint is expressible, but on the other hand, standard incompleteness theorems predict that some constraints cannot be shown to have no solutions. In practice, we have found that constraints that come up in programming tasks such as the ones presented in this chapter and in Chapter 5 (as opposed to theoretically hard ones) are handled well.

Expressions and function definitions. PureScala expressions can contain all standard arithmetic operators, map applications and updates, set operators and membership tests, function applications (of user-defined or anonymous functions), and constructor and selectors from user-defined data types. Expression can also contain **val**s to factor out common subexpressions. Indeed, a Scala block

```
{ val x1 = e2
  ...
  val xn = en
  e }
```

is to be understood as **let** x1 = e2 **in**...**in** **let** xn = en **in** e. Expressions can contain pattern-matching on user-defined data types. Any sub-pattern can be bound to a variable and used accordingly on the right-hand side of patterns. Furthermore, patterns can contain arbitrary guards.

A PureScala function body is defined by a single expression whose free variables are the arguments of the function. Functions can optionally be annotated with a pre- and post-conditions. We note that all examples in Scala encountered so far in this thesis are valid PureScala functions.

3.4 A Satisfiability Procedure Modulo Recursive Functions

In this section, we describe our algorithm for checking the satisfiability of formulas modulo recursive functions. We start with a description of the supported class of formulas. Let $\mathscr{L}$ be a *base theory* (logic) with the following properties:

- $\mathscr{L}$ is closed under propositional combination and supports boolean variables
- $\mathscr{L}$ supports uninterpreted function symbols
- there exists a complete decision procedure for $\mathscr{L}$

Note that the logics supported by DPLL(T) SMT solvers naturally have these properties.

```
def solve(φ, Π) {
  (φ, B) = unrollStep(φ, Π, ∅)
  while(true) {
    decide(φ ∧ ⋀_{b∈B} b) match {
      case "SAT" ⇒ return "SAT"
      case "UNSAT" ⇒ decide(φ) match {
        case "UNSAT" ⇒ return "UNSAT"
        case "SAT" ⇒ (φ, B) = unrollStep(φ, Π, B) }}}}
```

Figure 3.1 – Pseudo-code of the solving algorithm. The decision procedure for the base theory is invoked through the calls to decide.

```
size(lst) = lst match {
  case Nil ⇒ 0
  case Cons(_, xs) ⇒ 1 + size(xs)
}
```

$$\begin{aligned}\psi \equiv\ & \mathsf{size}(\mathsf{lst}) = t_{\mathsf{f}} \\ \wedge\ & b_{\mathsf{f}} \iff \mathsf{lst} = \mathsf{Nil} \\ \wedge\ & b_{\mathsf{f}} \implies t_{\mathsf{f}} = 0 \\ \wedge\ & \neg b_{\mathsf{f}} \implies t_{\mathsf{f}} = 1 + \mathsf{size}(\mathsf{lst.tail})\end{aligned}$$

Figure 3.2 – Function definition and its translation into clauses with control literals.

Let $\mathscr{L}^{\Pi}$ be the extension of $\mathscr{L}$ with *interpreted* function symbols defined in a program Π. The interpretation is given by an expression in $\mathscr{L}^{\Pi}$ (the *implementation*). To facilitate proofs and the description of program invariants, functions in Π can also be annotated with a *pre-* and *postconditions*. We denote the implementation, pre- and postcondition of a function f in Π by impl^{Π}_{f}, prec^{Π}_{f} and post^{Π}_{f} respectively. The free variables of these expressions are the arguments of f denoted args^{Π}_{f}, as well as, for the postcondition, a special variable ρ that denotes the result of the computation.

Figure 3.1 shows the pseudo-code of our algorithm. It is defined in terms of two subroutines, decide, which invokes the decision procedure for $\mathscr{L}$, and unrollStep, whose description follows. Note that the algorithm maintains, along with a formula ϕ, a set B of boolean literals. We call these *control literals* and their role is described below.

At a high-level, the role of unrollStep is to give a partial interpretation to function invocations, which are treated as uninterpreted in $\mathscr{L}$. This is achieved in two steps:

1. introduce definitions for one unfolding of the (uninterpreted) function invocations in ϕ,
2. generate an assignment of control literals to guard introduced function invocations

As an example, consider a formula ϕ that contains the term size(lst), where lst is a list and size is the usual recursive definition of its length. Figure 3.2 shows on the left the definition of size(lst) and on the right its encoding into clauses with fresh variables.

This encoding into clauses is obtained by recursively introducing, for each if-then-else term, a boolean variable representing the truth value of the branching condition, and another variable representing the value of the if-then-else term.

In addition to conjoining ψ to ϕ, unrollStep would produce the set of literals $\{b_\mathsf{f}\}$. The set should be understood as follows: the decision procedure for the base theory $\mathscr{L}$, which treats size as an uninterpreted function symbol, if it reports SAT, can only be trusted when b_f is set to true. Indeed, if b_f is false, the value used for t_f and hence for the term size(lst) may depend on size(lst.tail), which is undefined (because its definition has not yet been introduced). A subsequent call to unrollStep on $\phi \wedge \psi$ would introduce the definition of size(lst.tail). When unrolled functions have a precondition, the definitions introduced for their body and postcondition are guarded by the precondition. This is done to prevent an inconsistent function definition with a precondition equivalent to **false** from making the formula unsatisfiable.

Let us denote by ϕ^Π The formula in $\mathscr{L}$ without the control literals can be seen as an *over-approximation* of the formula with the semantics of the program defining $\mathscr{L}^\Pi$, in that it accepts all the same models plus some models in which the interpretation of some invocations is incorrect. The formula with the control literals is an *under-approximation*, in the sense that it accepts only the models that do not rely on the guarded invocations. This explains why the UNSAT answer can be trusted in the first case and the SAT case in the latter.

To come back to our example, if we reason in terms of models, we have the following chain of implications:

$$(\exists \mathscr{M}_1 : \mathscr{M}_1 \models \phi \wedge \psi \wedge b_\mathsf{f}) \implies (\exists \mathscr{M} : \mathscr{M} \models^\Pi \phi) \implies (\exists \mathscr{M}_2 : \mathscr{M}_2 \models \phi \wedge \psi)$$

where $\models^\Pi$ denotes the interpretation with respect to the full program semantics. The first implication follows from the fact that, given that $\mathscr{M}_1 \models b_\mathsf{f}$, the uninterpreted invocation of size does not participate in the satisfying assignment. The second implication follows from the fact that, from a true model with the full semantics, one can always construct a partial model for the uninterpreted functions defined only for the arguments that occur in the evaluation.

Fairness in unrollings. In Figure 3.1, the third argument of calls to unrollStep denotes the set of control literals introduced at previous steps. An invocation of unrollStep will insert definitions for *all* or only *some* function terms that are guarded by such control literals, and the returned set will contain all literals that were not released as well as the newly introduced ones. From an abstract point of view, when a new control literal is created, it is inserted in a global priority queue with all the function invocations that it guards. An important requirement is that the dequeuing must be *fair*: any control literal that is enqueued must eventually be dequeued. This fairness condition guarantees that our algorithm is complete for satisfiable formulas (see Section 3.4.1). Using a first-in first-out policy, for instance, is enough to guarantee fairness and thus completeness. Finally, note that unrollStep not only adds definitions for the implementation of the function calls, but also for their postcondition when it exists. We discuss the issue of reliably using these facts in Section 3.4.1.

Implementation notes. While the description of solve suggests that we need to query the solver twice in each loop iteration, we can in practice use the solver's capability to output *unsat cores* to detect with a single query whether the conjunction of control literals $\bigwedge_{b \in B} b$ played any role in the unsatisfiability. Similarly, when adding the new constraints obtained from unrollStep, we can use the solver's incremental reasoning and push the new constraints directly, rather than building a new formula and issuing a new query. SMT solvers can thus exploit at any time facts learned in previous iterations.

Finally, although we noted that we cannot in general trust the underlying solver when it reports SAT for the formula ϕ without control literals, it could still be that the assignment the solver guessed for the uninterpreted function symbols is valid. Because testing an assignment is fast (it amounts to executing the specification), we can therefore sometimes report SAT early and save iterations.

3.4.1 Properties of the Procedure

The properties of our procedure rely on the following two assumptions.

Termination: All precondition computations terminate for all values. Each function in the program Π terminates on each input for which the precondition holds, and similarly for each postcondition. Tools such as [GTSKF04, AAG+07] or techniques developed for ACL2 [MT09] could be used to establish this property.

Base theory solver soundness: The underlying solver is complete and sound for the (quantifier-free) formulas in the base theory. The completeness means that each model that the solver reports should be a model for the conjunction of all constraints passed to the solver. Similarly, soundness means that whenever the solver reports unsatisfiability, **false** can be logically concluded modulo the solver's theories from these constraints.

We rely on the above assumptions throughout this section. Note, however, that even without the termination assumption, a counterexample reported by Leon is never a counterexample that generates a terminating execution of the property resulting in the value **true**, so it is a counterexample worth inspecting.

Soundness for proofs. Our algorithm reports unsatisfiability if and only if the underlying solver could prove unsatisfiable the problem given to it *without* the control literals. Because the control literals are not present, some function applications are left uninterpreted, and the conclusion that the problem is unsatisfiable therefore applies to *any* interpretation of the remaining function application terms, and in particular to the one conforming to the correct semantics.

From the assumption that the underlying solver only produces sound proofs, it suffices to show that all the axioms communicated to the solver as a result of the unrollStep invocations are obtained from sound derivations. These are correct by definition: they are logical consequences obtained by the definition of functions, and these definitions are conservative when the functions are terminating.

An important consideration when discussing soundness of the *post* axioms is that any proof obtained with our procedure can be considered valid only when the following properties about the functions of Π have been proved: [1]

1. for each function f of Π, the following formula must hold:

$$\mathsf{prec}_f^\Pi \implies \mathsf{post}_f^\Pi\left[\mathsf{impl}_f^\Pi/\rho\right]$$

2. for each call in f to a function f_2 (possibly f itself), the precondition $\mathsf{prec}_{f_2}^\Pi$ must be implied by the path condition
3. for each pattern-matching expression, the patterns must be shown to cover all possible inputs under the path condition.

The above conditions guarantee the absence of runtime errors, and they also allow us to prove the overall correctness by induction on the call stack, as is standard in assume-guarantee reasoning for sequential procedures without side effects [Gri81, Chapter 12].

The first condition shows that all postconditions are logical implications of the function implementations under the assumption that the preconditions hold. The second condition shows that all functions are called with arguments satisfying the preconditions. Because all functions terminate, it follows that we can safely assume that postconditions always hold for all function calls. This justifies the soundness of axioms *post* in the presence of ϕ and Π.

Soundness for models. Our algorithm reports satisfiability when the solver reports that the unrolled problem augmented with the control literals is satisfiable. By construction of the set of control literals, it follows that the solver can only have used values for function invocations whose definition it knows. As a consequence, every model reported by the solver for the problem augmented with the control literals is always a true model of the original formula. We mentioned in Section 3.4 that we can also check other satisfying assignments produced by the solver. In this second case, we use an evaluator that complies with the semantics of the program, and therefore the validated models are true models as well.

Termination for satisfiable formulas. Our procedure has the property that it finds a model whenever the model for a formula exists.

1. When proving or disproving a formula ϕ modulo the functions of Π, it is in fact sufficient that the three properties hold only for all functions in ϕ and those that can be called (transitively) from them or from their contracts.

We define a model as an assignment to the free variables of the formula such that evaluating it under that assignment terminates with the value **true**. An assignment that leads to a diverging evaluation is not considered to be a proper model.

To understand why our procedure always finds models when they exist, consider a counterexample for the specification. This counterexample is an assignment of integers and algebraic data types to variables of a function $f(\bar{x})$ being proved. This evaluation specifies concrete inputs $\bar{a}$ for f such that evaluating $f(\bar{a})$ yields a value for which the postcondition of f evaluates to false (the case of preconditions or pattern-matching is analogous). Consider the computation tree arising from (call-by-value) evaluation of f and its postcondition. This computation tree is finite. Consequently, the tree contains finitely many unrollings of function invocations. Let us call K the maximum depth of that tree. Consider now the execution of the algorithm in Figure 3.1; because we assume that any function invocation that is guarded by a control literal is eventually accessible, we can safely conclude that every function application in the original formula will eventually be unrolled. By applying this reasoning inductively, we conclude that eventually, all function applications up to nesting level $K+1$ will be unrolled.

This means that the computation tree corresponding to $f(\bar{a})$ has also been explored. By the completeness of the solver for the base theory and the consistency of a satisfying specification, it means that the solver reports a counterexample (either $\bar{a}$ itself or another counterexample).

Termination for sufficiently surjective abstractions. Our procedure always terminates and is therefore a decision procedure in the case of a recursive function that is a *sufficiently surjective catamoprhism*, as defined in Section 2.5.3, provided that the function is annotated with a postcondition describing its co-domain (or, as a special case, that it is actually surjective). In fact, it also serves as the first implementation of our decision procedure. Leon thus shows that the procedure can be made fast in practice for an important subclass of problems. Moreover, by interleaving unrolling and satisfiability checking, it addresses in practice the problem of determining the maximal amount of unrolling needed for a user-defined function.

We have already established termination in the case of satisfiable instances. We therefore focus here on the case of an unsatisfiable formula. To simplify the presentation, we assume, as in Chapter 2, that the catamorphism is defined over a binary tree data type.

Recall that the general form for such catamorphisms is:

```
def α(t: Tree): 𝒞 = t match {
  case Leaf() ⇒ empty
  case Node(l,e,r) ⇒ combine(α(l), e, α(r))
}
```

for some functions $\mathsf{empty} : \mathscr{C}$ and $\mathsf{combine} : (\mathscr{C}, \mathscr{E}, \mathscr{C}) \to \mathscr{C}$.

A key observation is that for such functions, unrolling the definition α is exactly equivalent to

inspecting trees of greater and greater height. Indeed, consider a formula containing a term $\alpha(t)$. Unrolling the definition of α once produce the clauses:

$$\begin{aligned} &\alpha(\mathsf{t}) = t_0 \\ &\quad \wedge b_0 \iff t = \mathsf{Leaf}() \\ &\quad \wedge b_0 \implies t_0 = \mathsf{empty} \\ &\quad \wedge \neg b_0 \implies t_0 = \mathsf{combine}(\alpha(\mathsf{t.left}), \mathsf{t.e}, \alpha(\mathsf{t.right})) \end{aligned}$$

Further unrolling the definition of e.g. t.left produces clauses of the similar structure: the formula is identical, with the following rewritings

$$\mathsf{t} \rightsquigarrow \mathsf{t.left} \qquad b_0 \rightsquigarrow b_1 \qquad t_0 \rightsquigarrow t_1$$

Similarly, for t.right, with the rewritings

$$\mathsf{t} \rightsquigarrow \mathsf{t.right} \qquad b_0 \rightsquigarrow b_2 \qquad t_0 \rightsquigarrow t_2$$

(Meaning that the terms t.left.left, t.left.right, etc. appear in these clauses.) From the fair unrolling property, we thus know that our satisfiability procedure will examine all trees in order of increasing size. Another observation is that, under the assumption that the underlying decision procedure for data types is complete, the chains of selectors introduced by the unrolling correspond exactly to the fresh variables introduced through *shape instantiation*.

Let us now consider an unsatisfiable formula ϕ, in which α is the only recursive function. From the two observations above, it follows that, using the terminology of Definition 2.7, our satisfiability procedure eventually considers, for each tree term argument of α, each shape of the set S_p. (Note also that a complete solver will automatically apply unification and partial evaluation as the context requires.)

It remains to show that, upon concluding that none of the shapes in S_p can serve as the basis for a satisfying assignment, the procedure successfully concludes (without control literals) that ϕ admits no model. Equivalently, we show that if the unrolled formula without control literals admits a model, then it is a true model.

Call ϕ_p the formula obtained by applying the procedure until, for each argument of α in ϕ, all shapes in S_p have been considered. Observe that at any point in the procedure, the SMT solver maintains for each invocation $\alpha(t)$ an equality of the form:

$$\alpha(t) = C(s_1, \ldots, s_i, \alpha(s_{i+1}), \ldots, \alpha(s_n)) \tag{3.1}$$

where the s variables denote chained selector applications on t, and C denotes the term obtained by nesting applications of combine and empty. Note in particular that in this term, the variables appearing as arguments of α appear nowhere else.

Assume now that α is actually surjective. That is

$$\forall c : \exists t : \alpha(t) = c$$

Then, any model $\mathcal{M}_p$ for ϕ_p can be lifted to a model for ϕ. The reconstruction works as follows: for each application $\alpha(s_j)$ uninterpreted in ϕ_p, pick for s_j a value such that $\alpha(s_j) = \alpha(s_j)^{\mathcal{M}_p}$ and such that all disequalities between terms in ϕ are satisfied. That this is always possible is guaranteed by the surjectivity of α, and the assumptions that 1) α is sufficiently surjective and 2) the tree terms are not described by S_p.

We conclude by considering the case where α is not surjective, but otherwise matches the requirements of Definition 2.7. In such situations, if we can characterize the co-domain of α by a unary predicate $P(c)$ in the collection theory, we can add $P(c)$ as a postcondition of α and retrieve completeness. To see why this works, observe that the procedure will inline this postcondition for each invocation of α in (3.1). This in turn will guarantee that in any given model $\mathcal{M}_p$, the valuations for uninterpreted instances of α will be invertible, as in the surjective case.

Concrete examples of such functions are given by catamorphisms that map into natural numbers, such as size or height, because SMT solvers typically make no distinction between naturals and integers. By annotating them with a postcondition **ensuring** { res⇒res > 0} we achieve completeness nevertheless.

Non-terminating functions. We conclude this section with some remarks on the behavior of our procedure in the presence of functions that do not terminate on all their inputs. We are interested in showing that if for an input formula the procedure returns UNSAT, then there are indeed no models whose evaluation terminates with **true**. (The property that all models are true models is not affected by non-terminating functions.) Note that it may still be the case that the procedure returns UNSAT when there is an input for which the evaluation doesn't terminate.

To see why the property doesn't immediately follow from the all-terminating case, consider the definition:

```
def f(x : Int) = f(x) + 1
```

Unrolling this function introduces a mathematical contradiction f(x)= f(x)+ 1 and makes the formula immediately unsatisfiable, thus potentially masking a true satisfying assignment. However, because all introduced definitions are guarded by a control literal, the contradiction will only prevent those literals from being true that correspond to executions leading to a diverging execution.

3.5 The Leon Verification System

We now present some of the characteristics of the implementation of Leon, our verification system that has at its core an implementation of the procedure presented in the previous sections. Leon takes as an input a program written in PureScala and produces verification conditions for all specified postconditions, calls to functions with preconditions, and match expressions in the program.

Front-end. We wrote a plugin for the official Scala compiler to use as the front-end of Leon. The immediate advantage of this approach is that all programs are parsed and type-checked before they are passed to Leon. This also allows users to write expressive programs concisely, thanks to type-inference and the flexible syntax of Scala. As mentioned in Section 3.3, the subset we support allows for definitions of recursive data types as well as arbitrarily complex pattern-matching expressions over such types. The other admitted types are integers and sets and maps, which we found to be particularly useful for specifying properties with respect to an abstract interface.

Conversion of pattern-matching. We transform all pattern-matching expressions into equivalent expressions built with if-then-else terms. For this purpose, we use predicates that test whether their argument is of a given subtype (this is equivalent to the method `.isInstanceOf[T]` in Scala). The translation is relatively straightforward, and preserves the semantics of pattern-matching. In particular, it preserves the property that cases are tested in their definition order. To encode the error that can be triggered if no case matches the value, we return for the default case a fresh, uninterpreted value. This value is therefore guarded by the conjunction of the negation of all matching predicates. Recall that we separately prove that all match expressions are exhaustive. When these proofs succeed, we effectively rule out the possibility that the unconstrained error value affects the semantics of the expression. (It remains to be seen whether more sophisticated encodings of pattern-matching [Wad87, Mar08] can positively impact the solving times of the resulting formulas.)

Proofs by induction. To simplify the statement and proof of some inductive properties, we defined an annotation `@induct`, that indicates to Leon that it should attempt to prove a property by induction on the arguments. This works only when proving a property over a variable that is of a recursive type; in these cases, we decompose the proof that the postcondition is always satisfied into subproofs for the alternatives of the datatype. For instance, when proving by induction that a property holds for all binary trees, we generate a verification condition for the case where the tree is a leaf, then for the case where it is a node, assuming that the property holds for both subtrees.

Communicating with the solver. Leon uses Z3 version 3.2 [dMB08b] as the SMT solver at the core of our solving procedure. As described in Section 3.4, we use Z3's support for incremental reasoning to avoid solving a new problem at each iteration of our top-level loop.

Interpretation of selectors as total functions. We should note that the interpretation of selector functions in Z3 is different than in Scala, since they are considered to be total, but uninterpreted when applied to values of the wrong type. For instance, the formula Nil.head = 5 is considered in Z3 to be satisfiable, while taking the head of an empty list has no meaning in Scala (if not a runtime error). This discrepancy does not affect the correctness of Leon, though, as the type-checking algorithm run by the Scala compiler succeeds only when it can guarantee that the selectors are applied only to properly typed terms.

Online demonstration version. While the default way to interact with Leon is via the command line, we also developed a web front-end. At the webpage http://lara.epfl.ch/leon/, users can type in Leon benchmarks and verify them or find counter-examples. Figure 3.3 shows the interface as it was in September 2012.

3.6 Experimental Evaluation

We here report results of Leon on proving correctness properties for a number of functional programs, and discovering counterexamples when functions did not meet their specification. A summary of our evaluation can be seen in Figure 3.4 on page 61.

In this figure, LOC denotes the number of lines of code, #p. denotes the number of verification conditions for function invocations with preconditions, #m. denotes the number of conditions for showing exhaustiveness of pattern-matchings, V/I denotes whether the verification conditions were valid or invalid, U denotes the maximum depth for unrolling function definitions, and T denotes the total running time in seconds to verify all conditions for a function. The benchmarks were run on a computer equipped with two Intel Core 2 processors running at 2.66 GHz and 3.2 GB of RAM, using Z3 version 3.2. We verified over 60 functions, with over 600 lines of compactly written code and properties that often relate multiple function invocations. This includes a red-black tree set implementation including the height invariant (which most reported benchmarks for automated systems omit); amortized queue data structures, and examples with syntax tree refinement that show Leon to be useful for checking user code, and not only for data structures. [2]

The ListOperations benchmark contains a number of common operations on lists. Leon proves, e.g., that a tail-recursive version of size is functionally equivalent to a simpler version. For append, reverse and concat, we prove that the content is as expected after the operations. We

2. All benchmarks and the sources of Leon are available from http://lara.epfl.ch.

LeonOnline

Leon is developed by the LARA group at EPFL. A good starting point is to copy-paste the source from an example in this list and to click the *Verify !* button.

```
// <<insert element x into the tree t>>
def ins(x: Int, t: Tree): Tree = {
  require(redNodesHaveBlackChildren(t) && blackBalanced(t))
  t match {
    case Empty() => Node(Red(),Empty(),x,Empty())
    case Node(c,a,y,b) =>
      if      (x < y)  balance(c, ins(x, a), y, b)
      else if (x == y) Node(c,a,y,b)
      else             balance(c,a,y,ins(x, b))
  }
} ensuring (res => content(res) == content(t) ++ Set(x)
                 && size(t) <= size(res) && size(res) <= size(t) + 1
                 && redDescHaveBlackChildren(res)
                 && blackBalanced(res))

def makeBlack(n: Tree): Tree = {
  require(redDescHaveBlackChildren(n) && blackBalanced(n))
  n match {
    case Node(Red(),l,v,r) => Node(Black(),l,v,r)
    case _ => n
  }
} ensuring(res => redNodesHaveBlackChildren(res) && blackBalanced(res))

def add(x: Int, t: Tree): Tree = {
  require(redNodesHaveBlackChildren(t) && blackBalanced(t))
  makeBlack(ins(x, t))
} ensuring (res => content(res) == content(t) ++ Set(x) &&
redNodesHaveBlackChildren(res) && blackBalanced(res))
```

Verify !

```
Node(c, a, x, b).right.value, Node(c, a, x, b).right.right))
    } else {
      if ((Node(c, a, x, b).isInstanceOf[Node] ∧ (Node(c, a, x, b).color.isInstanceOf[Black] ∧ (Node(c, a,
x, b).right.isInstanceOf[Node] ∧ (Node(c, a, x, b).right.color.isInstanceOf[Red] ∧ (Node(c, a, x,
b).right.right.isInstanceOf[Node] ∧ Node(c, a, x, b).right.right.color.isInstanceOf[Red]))))) {
        Node(Red(), Node(Black(), Node(c, a, x, b).left, Node(c, a, x, b).value, Node(c, a, x,
b).right.left), Node(c, a, x, b).right.value, Node(Black(), Node(c, a, x, b).right.right.left, Node(c, a,
x, b).right.right.value, Node(c, a, x, b).right.right.right))
      } else {
        error("non-exhaustive match")[Tree]
      }
    }
  }
}) == content(Node(c, a, x, b)))
x -> 0
a -> Node(Red(), Node(Red(), Empty(), 6, Empty()), 4, Empty())
c -> Red()
b -> Node(Red(), Node(Red(), Empty(), 3, Empty()), 2, Node(Red(), Empty(), 1, Empty()))
==== INVALID ====
.
 Summary
 redNodesHaveBlackChildren match.    (35,56) valid     Z3-f   0.041
 blackHeight               match.    (51,39) valid     Z3-f   0.002
 ins                       precond.  (63,40) valid     Z3-f   0.016
 ins                       precond.  (65,43) valid     Z3-f   0.012
```

LeonOnline is powered by: Scala Z3 Lift CodeMirror EPFL

Figure 3.3 – Online interface to the Leon verification system.

(inductively) show that append is associative by expressing the property using the function appendAssoc:

```
@induct
def appendAssoc(xs : List, ys : List, zs : List) : Boolean = {
  append(append(xs, ys), zs) == append(xs, append(ys, zs))
} holds
```

In a similar fashion, the function sizeAppend proves that the result of appending two lists has size equal to the sum of the sizes of input lists. As a final example for list functions, we consider zip, which requires that the two input lists l1 and l2 have equal size, and never attempts to match l2 as an empty list if l1 is matched as nonempty. It is defined as follows:

```
def zip(l1: List, l2: List) : IntPairList = {
  require(size(l1) == size(l2))
  l1 match {
    case Nil() ⇒ IPNil()
    case Cons(x, xs) ⇒ l2 match {
      case Cons(y, ys) ⇒ IPCons(IP(x, y), zip(xs, ys))
    }
  }
} ensuring(iplSize(_) == size(l1))
```

Our system is able to prove that the pattern matching expression in this example is complete.

For associative lists, Leon proves that updating a list l1 with all mappings from another list l2 yields a new associative list whose domain is the union of the domains of l1 and l2. It proves the *read-over-write* property, which states that looking up the value associated with a key gives the most recently updated value. We express this property simply as:

```
def readOverWrite(l : List, e : Pair, k : Int) : Boolean = (e match {
  case Pair(key, value) ⇒
    find(updateElem(l, e), k) == (if (k == key) Some(value) else find(l, k))
}) holds
```

Leon proves properties of insertion sort such as the fact that the output of the function sort is sorted, and that it has the same size and content as the input list. The function buggySortedIns is similar to sortedIns, and is responsible for inserting an element into an already sorted list, except that the precondition that the list should be sorted is missing.

On the RedBlackTrees benchmark, Leon proves that the tree implements a set interface and that balancing preserves the "red nodes have no black children" and "every simple path from the root to a leaf has the same number of black nodes" properties as well as the contents of the tree. In addition to proving correctness, we also seeded two bugs (forgetting to paint a node black and missing a case in balancing); Leon found a concise counterexample in each case. The PropositionalLogic benchmark contains functions manipulating abstract syntax trees of

boolean formulas. Leon proves that, e.g., applying a negation normal form transformation twice is equivalent to applying it once.

Further benchmarks are taken from the Verified Software Competition [VSC10]: For example, in the AmortizedQueue benchmark Leon proves that operations on an amortized queue implemented as two lists maintains the invariant that the size of the "front" list is always larger than or equal to the size of the "rear" list, and that the function front implements an abstract queue interface given as a sequence.

We also point out that, apart from the parameterless @induct hint for certain functions, there are no other hint mechanisms used in Leon: the programmer simply writes the code, and boolean-valued functions that describe the desired properties (as they would do for testing purposes). We thus believe that Leon is easy and simple to use, even for programmers that are not verification experts.

3.7 Related Work

Leon and our associated procedure for satisfiability modulo recursive functions were originally presented in [SKK11]. We conclude this chapter by comparing our approach to the most closely related techniques.

Interactive verification systems. The practicality of computable functions as an executable logic has been demonstrated through a long line of systems, notably ACL2 [KMM00] and its predecessors. These systems have been applied to a number of industrial-scale case studies in hardware and software verification [KMM00, Moo10]. Recent systems based on functional programs include VeriFun [WS03] and AProVE [GTSKF04]. Moreover, computable specifications form important parts of many case studies in proof assistants Coq [BC04] and Isabelle [NPW02]. These systems support more expressive logics, with higher-order quantification, but provide facilities for defining executable functions and generating the corresponding executable code in functional programming languages [HN07]. When it comes to reasoning within these systems, they offer varying degrees of automation. What is common is the difficulty of predicting when a verification attempt will succeed. This is in part due to possible simplification loops associated with the rewrite rules and tactics of these provers. Moreover, for performance and user-interaction reasons, interactive proofs often fail to fully utilize aggressive case splitting that is at the heart of modern SMT solvers.

Inductive generalizations vs. counterexamples. Existing interactive systems such as ACL2 are stronger in automating induction, whereas our approach is complete for finding counterexamples. We believe that the focus on counterexamples will make our approach very appealing to programmers that are not theorem proving experts. The HMC verifier [JMR11] and DSolve [RKJ08, KRJ09] can automatically discover inductive invariants, so they have more

automation, but it appears that certain properties involving multiple user-defined functions are not expressible in these systems. Recent results also demonstrate inference techniques for higher-order functional programs [KTU10, JMR11]. These approaches hold great promise for the future, but the programs on which those systems were evaluated are smaller than our benchmarks. Leon focuses on first-order programs and is particularly effective for finding counterexamples. Our experience suggests that Leon is more scalable than the alternative systems that can deal with this expressive properties. Counterexample generation has been introduced into Isabelle through tools like Nitpick [BN10]. Further experimental comparisons would be desirable, but these techniques do not use theory solvers and appear slower than Leon on complex functional programs. Counterexample generation has been recently incorporated into ACL2 Sedan [CDMV11]. This techniques is tied to the sophisticated ALC2 proof mechanism and uses proof failures to find counterexamples. Although it appears very useful, it does not have our completeness guarantees.

Counterexample finding systems for imperative code. Researchers have explored the idea of iterative function and loop unfolding in a number of contexts. Among well-known tools is CBMC [CKL04]; techniques to handle procedures include [Tag04, BKW07, Sin10]. The use of imperative languages in these systems makes their design more complex and limits the flexibility of the counterexample search. Thanks to a direct encoding into SMT and the absence of side-effects, Leon can prove more easily properties that would be harder to prove using imperative semantics. As a result, we were able to automatically prove detailed functional correctness properties as opposed to only checking for errors such as null dereferences. Both [Tag04] and [Sin10] only find counterexampels, while we were additionally able to *prove* several non-trivial properties (and still benefit from counterexample finding to debug the code and the properties).

Satisfiability modulo theory solvers. SMT solvers behave as (complete) decision procedures on certain classes of quantifier-free formulas containing theory operations and uninterpreted functions. However, they do not support user-defined functions, such as functions given by recursive definitions. An attempt to introduce them using quantifiers leads to formulas on which the prover behaves unpredictably for unsatisfiable instances, and is not able to determine whether a candidate model is a real one. This is because the prover has no way to determine whether universally quantified axioms hold for all of the infinitely many values of the domain. Leon uses terminating executable functions, whose definitions are a well-behaved and important class of quantified axioms, so it can check the consistency of a candidate assignment. A high degree of automation and performance in Leon comes in part from using state-of-the-art SMT solver Z3 [dMB08b] to reason about quantifier-free formula modulo theories, as well as to perform case splitting along with automated lemma learning. Other SMT solvers, such as CVC3 [BT07] or OpenSMT [BPST10] could also be used.

Benchmark	p	m	s	U	T	function	#p.	#m.	V/I	U	T
ListOperations *(107)*											
size	0	1	V	1	0.12	sizeTailRecAcc	1	1	V	1	0.01
sizesAreEquiv	0	0	V	2	<0.01	sizeAndContent	0	0	V	1	<0.01
reverse	0	0	V	2	0.02	reverse0	0	1	V	2	0.04
append	0	1	V	1	0.03	nilAppend	0	0	V	1	0.03
appendAssoc	0	0	V	1	0.03	sizeAppend	0	0	V	1	0.04
concat	0	0	V	1	0.04	concat0	0	2	V	2	0.29
zip	1	2	V	2	0.09	sizeTailRec	1	0	V	1	<0.01
content	0	1	V	0	<0.01						
AssociativeList *(50)*											
update	0	1	V	1	0.03	updateElem	0	2	V	1	0.05
readOverWrite	0	1	V	1	0.10	**domain**	0	1	V	0	0.05
find	0	1	V	1	<0.01						
InsertionSort *(99)*											
size	0	1	V	1	0.06	sortedIns	1	1	V	2	0.24
buggySortedIns	1	1	I	1	0.08	sort	1	1	V	1	0.03
contents	0	1	V	0	<0.01	isSorted	0	1	V	1	<0.01
RedBlackTree *(117)*											
ins	2	1	V	3	2.88	makeBlack	0	0	V	1	0.02
add	2	0	V	2	0.19	buggyAdd	1	0	I	3	0.26
balance	0	1	V	3	0.13	buggyBalance	0	1	I	1	0.12
content	0	1	V	0	<0.01	size	0	1	V	1	0.11
blackChld1	0	1	V	1	<0.01	blackChld2	0	1	V	0	<0.01
blackHeight	0	1	V	1	<0.01						
PropositionalLogic *(86)*											
simplify	0	1	V	2	0.84	nnf	0	1	V	1	0.37
commuteBug	0	0	I	3	0.44	simplifyBug	0	0	I	1	0.28
nnfIsStable	0	0	V	1	0.17	simplifyIsStable	0	0	V	1	0.12
isSimplified	0	1	V	0	<0.01	isNNF	0	1	V	1	<0.01
vars	6	1	V	1	0.13						
SumAndMax *(46)*											
max	2	1	V	1	0.13	sum	0	1	V	0	<0.01
allPos	0	1	V	0	<0.01	size	0	1	V	1	<0.01
prop0	1	0	V	1	0.02	property	1	0	V	1	0.11
SearchLinkedList *(48)*											
size	0	1	V	1	0.11	contains	0	1	V	0	<0.01
firstZero	0	1	V	1	0.03	firstZeroAtPos	0	1	V	0	<0.01
AmortizedQueue *(124)*											
size	0	1	V	1	0.14	content	0	1	V	0	<0.01
asList	0	1	V	0	<0.01	concat	0	1	V	1	0.04
isAmortized	0	1	V	0	<0.01	isEmpty	0	1	V	0	<0.01
reverse	0	1	V	3	0.20	amortizedQueue	0	0	V	2	0.05
enqueue	0	1	V	1	<0.01	front	0	1	V	3	0.01
tail	0	1	V	3	0.15	propFront	1	1	V	3	0.07
enqueue0	1	0	V	4	0.21	enqDeqThrice	5	0	V	5	2.48

Figure 3.4 – Summary of evaluation results.

4 Sets with Cardinality Constraints

We have seen in the previous chapter that abstraction functions mapping recursive structures into collection types can be very useful to write precise specifications. We have also seen how to reduce reasoning about such recursive functions to reasoning about the collection theory. In this chapter, we focus on a specific collection theory supporting sets with cardinality constraints, Boolean Algebra with Presburger Arithmetic (BAPA), and more precisely on its quantifier-free fragment (QFBAPA). In contrast to many other NP-complete problems (such as quantifier-free first-order logic or linear arithmetic), the applications of QFBAPA to a broader set of problems has so far been hindered by the lack of an efficient implementation that can be used alongside other efficient decision procedures.

We overcome these limitations by extending the efficient SMT solver Z3 with the ability to reason about cardinality (QFBAPA) constraints. Our implementation uses the DPLL(T) mechanism of Z3 to reason about the top-level propositional structure of a QFBAPA formula, improving the efficiency compared to previous implementations. Moreover, we present a new algorithm for automatically decomposing QFBAPA formulas. Our algorithm alleviates the exponential explosion of considering all Venn regions, significantly improving the tractability of formulas with many set variables. Because it is implemented as a theory plugin, our implementation enables Z3 to prove formulas that use QFBAPA constructs with constructs from other theories that Z3 supports, as well as with quantifiers. We have applied our implementation to the verification of functional programs; we show it can automatically prove formulas that no automated approach was reported to be able to prove before.

4.1 Introduction

Sets naturally arise in software that performs discrete computation, as a built-in data type [DGL+79], as container libraries, or inside program specification constructs [LKR05, ZKR08]. An intrinsic part of reasoning about sets is reasoning about sizes of sets, with well-known associated laws such as the inclusion-exclusion principle $|A \cup B| = |A| + |B| - |A \cap B|$. A natural decidable logic that supports set operations (union, intersection, complement) as well as a

cardinality operator is a logic we call BAPA, for Boolean Algebra with Presburger Arithmetic [FV59, KNR06].

We here consider the quantifier-free fragment of BAPA, denoted QFBAPA. QFBAPA was shown to be NP-complete using a particular encoding into quantifier-free Presburger arithmetic that exploits an integer analogue of Carathéodory theorem [KR07]. We thus think of QFBAPA as a generalization of SAT that is similar to SAT from a high-level complexity-theory point of view. The richness of QFBAPA is reflected in the fact that, being propositionally closed, it subsumes SAT. Moreover, unlike SAT, no encoding is needed to represent integer linear arithmetic in BAPA. Crucially, BAPA supports set operations and cardinality, whose polynomial encoding into SAT is possible but non-trivial [KR07]. Strikingly, a number of expressive logics can be naturally reduced to QFBAPA [WPK09, KPSW10]. This enables combination of formulas from non-disjoint theory signatures that share set operations, and goes beyond current disjoint theory combinations.

However, although the QFBAPA satisfiability problem is NP-complete, the resulting algorithm has proven difficult for unsatisfiable QFBAPA instances; its improvements have primarily helped dealing with satisfiable instances[KR07]. This empirical observation is in contrast to the situation for propositional logic, for which the community developed SAT solvers that are effective in showing unsatisfiability for large industrial benchmarks. In this chapter we present a QFBAPA implementation that is effective both for satisfiable and unsatisfiable instances.

Our implementation incorporates an important new algorithmic component: a decomposition of constraints based on the hypergraphs of common set variables. This component analyzes the variables occurring in different atomic formulas within a QFBAPA formula and uses the structural property of the formula to avoid generating all Venn regions. Our implementation is integrated into the state-of-the-art SMT solver Z3, whose important feature is efficient support for linear arithmetic [dMB08a]. Efficient integration with Z3 was made possible by the recently introduced theory plugin architecture of Z3, as well as by an incremental implementation of our algorithm. In this integration, Z3 processes top level propositional structure of the formula, providing QFBAPA solver with conjunctions of QFBAPA constraints. Our solver generates lemmas in integer linear arithmetic and gives them back to Z3, which incorporates them with other integer constraints. At the same time, Z3 takes care of equality constraints. The net result is (1) dramatic improvement of efficiency compared to previously reported QFBAPA implementations (2) the ability to use QFBAPA cardinality operation alongside all other operations that Z3 supports. We illustrate the usefulness of this approach through experimental results that prove the validity of complex verifications condition arising from the verification of imperative as well as functional programs.

Contributions. In summary, this chapter makes the following contributions:

– decomposition theorems and algorithms for efficient handling of QFBAPA cardinality constraints, often avoiding the need for exponentially many Venn regions as in [KNR06],

and avoiding complex conditional sums as in [KR07];
- an incremental algorithm to analyze the structure of QFBAPA formulas and generate the integer constraints following the decomposition theorems;
- an implementation of these algorithms as a theory plugin for the Z3 solver, with support for detecting equalities entailed by QFBAPA constraints and therefore precise combination with other theories supported by Z3;
- encouraging experimental results for benchmarks arising from verification of imperative and functional programs.

4.2 Example

We next illustrate the expressive power of the SMT prover we obtained by incorporating our QFBAPA decision procedure into Z3. Given a list datatype, consider the question of proving that the set of elements contained in the list has a cardinality always less than or equal to the length of the list. The set of elements contained in the list and the length of the list are computed using catamorphisms such as the ones introduced in Chapter 2:

```
def content(list: List) : Set[Int] = list match {
  case Nil() ⇒ ∅
  case Cons(x, xs) ⇒ {x} ∪ content(xs)
}
def length(list: List) : Int = list match {
  case Nil() ⇒ 0
  case Cons(x, xs) ⇒ 1 + length(xs)
}
```

As an example, consider proving the property that the size of the set returned by the content abstraction is always less than the size computed by length (because of repeated elements). This can be stated as proving that the following function always computes true:

```
def setSmallerThanLength(list : List) : Boolean = {
  content(list).size ≤ length(list)
} ensuring (res ⇒ res == true)
```

or in mathematical notation, that the following formula holds:

$$\forall list : \mathsf{List[Int]} : \ |\mathsf{content}(list)| \leq \mathsf{length}(list)$$

In Chapter 3, we will present a general technique to handle such recursive functions. In the present chapter, however, we will simply assume that we proceed by unfolding the recursive

$$
\begin{aligned}
\phi &::= A \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid \neg\phi \\
A &::= S_1 = S_2 \mid S_1 \subseteq S_2 \mid T_1 = T_2 \mid T_1 \leq T_2 \\
S &::= s \mid \emptyset \mid \mathcal{U} \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 \setminus S_2 \mid S^{\mathsf{c}} \\
T &::= i \mid K \mid T_1 + T_2 \mid K \cdot T \mid |S| \\
K &::= \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots
\end{aligned}
$$

Figure 4.1 – Quantifier-Free Formulas of Boolean Algebra with Presburger Arithmetic (QFBAPA). S^{c} denotes the complement of the set S with respect to the universe $\mathcal{U}$, that is, $\mathcal{U} \setminus S$.

definitions sufficiently many times to obtain the following verification condition:

$$
\begin{aligned}
& \quad list \neq \mathsf{Nil} \implies list = \mathsf{Cons}(x, xs) \\
& \land \mathsf{length}(\mathsf{Nil}) = 0 \land \mathsf{length}(\mathsf{Cons}(x, xs)) = 1 + \mathsf{length}(xs) \\
& \land \mathsf{content}(\mathsf{Nil}) = \emptyset \land \mathsf{content}(\mathsf{Cons}(x, xs)) = \{x\} \cup \mathsf{content}(xs) \\
& \land |\mathsf{content}(xs)| \leq \mathsf{length}(xs) \\
\implies & |\mathsf{content}(list)| \leq \mathsf{length}(list)
\end{aligned}
$$

Note that the formula includes reasoning about function symbols, integers, algebraic data types, sets, and cardinalities of sets. To the best of our knowledge, the implementation we present in this chapter is the only one that is complete for proving the validity of formulas with such operators. The proof is found using Z3's DPLL(T) algorithm, which, among others, performs cases analysis on whether $list$ is empty. It relies on Z3 to support arithmetic, congruence properties of functions, and algebraic data types. Finally, it crucially relies on invocations of our QFBAPA plugin. In response to currently asserted QFBAPA constraints, our plugin generates integer constraints on Venn regions that are chosen to ensure that the relationships between sets are handled in a complete way. The entire theorem proving process takes negligible time (see Section 4.5 for experimental results).

4.3 Decomposition in Solving BAPA Constraints

In this section, we consider formulas over finite sets of uninterpreted elements from a domain $\mathcal{E}$. We show in Section 4.4 how we combined QFBAPA with other theories to obtain a theory of sets of interpreted elements.

Syntax. Figure 4.1 presents the syntax of QFBAPA. We use vars(ϕ) to denote the set of free set variables occurring in ϕ.

Definition 4.1 (Venn regions)**.** *One central notion throughout the presentation of the theorems*

and the decision procedure is the notion of Venn region. *A Venn region of n sets* $S = \{S_1, \ldots, S_n\}$ *is one of the* 2^n *set expressions described by* $\bigcap_{i=1}^{n} S_i^{\alpha_i}$ *where* $S_i^{\alpha_i}$ *is either* S_i *or* S_i^c*. By construction, the* 2^n *regions form a partition of* $\mathcal{U}$*. We write venn*(S) *to denote the set of all Venn regions formed with the sets in* S*.*

Semantics. An interpretation $\mathcal{M}$ of a QFBAPA formula ϕ is a map from the set variables of ϕ to finite subsets of $\mathcal{E}$ and from the integer variables of ϕ to values in $\mathbb{Z}$. It is a model of ϕ, denoted $\mathcal{M} \models \phi$ if the following conditions are satisfied:

- $\mathcal{U}^{\mathcal{M}}$ is a finite subset of $\mathcal{E}$ and $\emptyset^{\mathcal{M}}$ is the empty set
- for each set variable S of ϕ, $S^{\mathcal{M}} \subseteq \mathcal{U}^{\mathcal{M}}$
- for each integer variable k of ϕ, $k^{\mathcal{M}} \in \mathbb{Z}$
- when $=, \subseteq, \emptyset, \cup, \cap, \setminus$ and the cardinality function $|\cdot|$ are interpreted as expected, ϕ evaluates to true in $\mathcal{M}$

Definition 4.2. *We define* $\sim_V$ *to be the equivalence relation on interpretations, parametrized by a set of set variables* V*, such that:*

$$\mathcal{M}_1 \sim_V \mathcal{M}_2 \iff \forall v \in venn(V) : |v^{\mathcal{M}_1}| = |v^{\mathcal{M}_2}|$$

Definition 4.3. *Let* $\mathcal{M}$ *be an interpretation and* $f : \mathcal{E} \to \mathcal{E}$ *a bijection from the interpretation domain to itself (a permutation function). We denote by* $f[\mathcal{M}]$ *the interpretation such that:*

- $\mathcal{U}^{f[\mathcal{M}]} = \{f(u) \mid u \in \mathcal{U}^{\mathcal{M}}\}$
- *for each set variable* S *interpreted in* $\mathcal{M}$, $S^{f[\mathcal{M}]} = \{f(u) \mid u \in S^{\mathcal{M}}\}$
- *for each integer variable* k *interpreted in* $\mathcal{M}$, $k^{f[\mathcal{M}]} = k^{\mathcal{M}}$

Theorem 4.4. *Let* ϕ *be a* QFBAPA *formula,* $\mathcal{M}$ *a model of* ϕ *and* $f : \mathcal{E} \to \mathcal{E}$ *a bijection, then* $f[\mathcal{M}] \models \phi$*.*

Proof. Prove by induction that $t^{f[\mathcal{M}]} = f[t^{\mathcal{M}}]$ for every set algebra term t. By bijectivity of f, $|t^{f[\mathcal{M}]}| = |f[t^{\mathcal{M}}]|$, so the values of all integer-valued terms remain invariant under f. Finally, note that $S_1 \subseteq S_2$ reduces to $|S_1 \setminus S_2| = 0$ whereas $S_1 = S_2$ reduces to $|S_1 \setminus S_2| = 0 \wedge |S_2 \setminus S_1| = 0$. □

4.3.1 A Simple Decision Procedure for QFBAPA

A simple technique for solving QFBAPA formulas is to reduce the problem to integer linear arithmetic as follows. Introduce a fresh integer variable for each Venn region in venn(vars(ϕ)). Rewrite each constraint of the form $S_1 = S_2$ as $S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$, and each constraint of the form $S_1 \subseteq S_2$ as $|S_1 \setminus S_2| = 0$. Finally, use sums over the integer variables representing the Venn regions to rewrite the cardinality constraints.

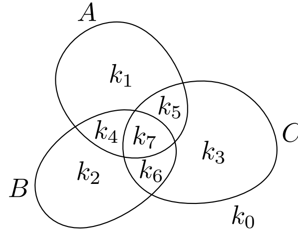

Figure 4.2 – Naming of the Venn regions of set variables A, B, and C using integer variables.

As an example, consider the formula

$$|A| > 1 \land A \subseteq B \land |B \cap C| \leq 2$$

and the naming of the Venn regions venn($\{A, B, C\}$) shown in Figure 4.2. Rewrite the constraints as

$$k_1 + k_4 + k_5 + k_7 > 1 \land k_1 + k_5 = 0 \land k_7 + k_6 \leq 2 \land k_i \geq 0 \text{ for } i \in \{0, \ldots, 7\}$$

A model for this integer formula is, for example,

$$k_4 = 1, k_7 = 1, k_i = 0 \text{ for } i \notin \{4, 7\}$$

From the model for integer variables we can build a model for the QFBAPA formula by picking distinct elements from $\mathscr{E}$ for each Venn region. In this case we can construct, for example,

$$\mathscr{U} = \{e_1, e_2\}, A = \{e_1, e_2\}, B = \{e_1, e_2\}, C = \{e_2\}$$

This reduction is simple to understand and to implement, but always requires 2^N integer variables, where N is the number of elements in vars(ϕ). In the following, we show how to reduce this number considerably in many cases that arise in practice.

Note that what we describe below is largely orthogonal to the NP procedure in [KR07]. Our results should in principle apply both to the naive procedure sketched above, and to the sparse encoding procedure in [KR07]. Note that the NP procedure from [KR07], although theoretically optimal, has so far been shown to be practically useful only for satisfiable cases. For unsatisfiable cases the sparse encoding generates quantifier-free Presburger arithmetic formulas that are difficult for current SMT solver implementations.

4.3.2 Decomposing Conjunctions of QFBAPA Formulas

The decision procedure presented in Section 4.3.1 requires many integer variables because it uses a variable for the intersection of *every* combination of set variables. The intuition behind

the following result is that permutations of elements within sets that are not related by a constraint in a formula do not affect the satisfiability of that formula.

Theorem 4.5. *Let ϕ_1 and ϕ_2 be two* QFBAPA *formulas. Let V_1 and V_2 denote the sets of set variables appearing in ϕ_1 and ϕ_2, respectively. Let $V_S = V_1 \cap V_2$ be the set of shared variables. Let K_S represent the set of shared integer variables. The* QFBAPA *formula $\phi \equiv \phi_1 \wedge \phi_2$ is satisfiable if and only if there exists a model $\mathscr{M}_1$ for ϕ_1 and a model $\mathscr{M}_2$ for ϕ_2 such that, for each integer variable $k \in K_S$, $k^{\mathscr{M}_1} = k^{\mathscr{M}_2}$, and such that $\mathscr{M}_1 \sim_{V_S} \mathscr{M}_2$.*

Proof. We construct a model $\mathscr{M}$ for ϕ by extending $\mathscr{M}_1$ to the variables in $V_2 \setminus V_S$. (The other direction is immediate.) We show that there exists a permutation f on $\mathscr{E}$ such that $f[v^{\mathscr{M}_2}] = v^{\mathscr{M}_1}$ for each Venn region $v \in \text{venn}(V_S)$. In other words, f is a bijection that projects the interpretation in $\mathscr{M}_2$ of all intersections of the shared variables to their interpretation in $\mathscr{M}_1$. We construct f as follows: for each Venn region $v \in \text{venn}(V_S)$, let f_v be a bijection from $v^{\mathscr{M}_2}$ to $v^{\mathscr{M}_1}$. Note that f_v always exists because $v^{\mathscr{M}_1}$ and $v^{\mathscr{M}_2}$ have the same cardinality, by $\sim_{V_S}$. Let $f^\star$ be $\bigcup_{v \in \text{venn}(V_S)} f_v$. Observe that $f^\star$ is a bijection from $\mathscr{U}^{\mathscr{M}_2}$ to $\mathscr{U}^{\mathscr{M}_1}$, because venn$(V_S)$ forms a partition of $\mathscr{U}$ in both models. To obtain the desired f, we can extend $f^\star$ to the domain and range $\mathscr{E}$ by taking its union with any bijection from $\mathscr{E} \setminus \mathscr{U}^{\mathscr{M}_2}$ to $\mathscr{E} \setminus \mathscr{U}^{\mathscr{M}_1}$. The model $\mathscr{M} = \mathscr{M}_1 \cup f[\mathscr{M}_2]$ is a model of ϕ_1 (trivially) and of ϕ_2 (by Theorem 4.4), therefore, it is a model of ϕ. □

The following result is a generalization of Theorem 4.5 for a conjunction of arbitrarily many constraints.

Theorem 4.6. *Let $\phi_1, \ldots, \phi_n$ be n* QFBAPA *formulas. Let V_i denote vars(ϕ_i) for $i \in \{1, \ldots, n\}$. Let*

$$V_S = \bigcup_{1 \le i < j \le n} V_i \cap V_j$$

be the set of all variables that appear in at least two sets V_i and V_j. The formula $\phi_1 \wedge \ldots \wedge \phi_n$ is satisfiable if and only if: 1) there exist models $\mathscr{M}_1, \ldots, \mathscr{M}_n$ such that, for each i, $\mathscr{M}_i \models \phi_i$ and 2) there exists an interpretation $\mathscr{M}$ of the variables V_S such that $\mathscr{M} \sim_{V_S \cap V_i} \mathscr{M}_i$ for each $1 \le i \le n$.

(Note that the conditions imply that $|\mathscr{U}^{\mathscr{M}}| = |\mathscr{U}^{\mathscr{M}_i}|$ for each i.) The proof follows the idea of the proof of Theorem 4.5: it suffices to show that one can extend the model $\mathscr{M}$ to each model $\mathscr{M}_i$ by finding a bijection f_i. Note that f_i is guaranteed to exist because $\mathscr{M} \sim_{V_S \cap V_i} \mathscr{M}_i$.

Remark. It is not sufficient that for each $0 \le i < j \le n$, $\mathscr{M}_i \sim_{(V_i \cap V_j)} \mathscr{M}_j$, that is, that there exists bijections between all pairs of models $\mathscr{M}_i$ and $\mathscr{M}_j$. A simple counter-example is given

by the (unsatisfiable) constraints:

$$\phi_1 \equiv |A| = 1 \land |B| = 1 \land |A \cap B| = 1$$
$$\phi_2 \equiv |A| = 1 \land |C| = 1 \land |A \cap C| = 1$$
$$\phi_3 \equiv |B| = 1 \land |C| = 1 \land |B \cap C| = 0$$

Although any conjunction $\phi_i \land \phi_j$ can be shown satisfiable using Theorem 4.5, the conjunction $\phi_1 \land \phi_2 \land \phi_3$ is unsatisfiable, because a common model $\mathcal{M}$ cannot be built from models for each three constraints.

Lemma 4.7. *Let $\phi_1, \ldots, \phi_n$ be n QFBAPA formulas, and let $V_1, \ldots, V_n, V_S$ be defined as above. Assume there exist $\mathcal{M}, \mathcal{M}_1, \ldots, \mathcal{M}_n$ satisfying the conditions of Theorem 4.6 and the additional condition that:*

$$|\mathcal{U}^{\mathcal{M}}| \geq |\bigcup_{S \in V_S} S^{\mathcal{M}}| + \sum_{i=1}^{n} |\bigcup_{S \in V_i \setminus V_S} S^{\mathcal{M}_i}|$$

Then there exists a model $\mathcal{M}_d$ such that, for any two sets $S_i \in V_i \setminus V_S$ and $S_j \in V_j \setminus V_S$ (with $i \neq j$), if (1) $\forall S \in V_S \cap V_i \,.\, \mathcal{M}_i \models S_i \neq S$ and (2) $\forall S \in V_S \cap V_j \,.\, \mathcal{M}_j \models S_j \neq S$, and (3) either $\mathcal{M}_i \models S_i \neq \emptyset$ or $\mathcal{M}_j \models S_j \neq \emptyset$, then $\mathcal{M}_d \models S_i \neq S_j$.

Intuitively, Lemma 4.7 states that for two sets variables that appear in different constraints and that are not shared, if there exists a model in which these sets are not empty and are not equal to some shared set variable, then there exists a model in which the two sets are not equal. The correctness follows from the fact that with $\mathcal{U}^{\mathcal{M}}$ sufficiently large, it is always possible to find bijections f_i such that for each i the non-shared sets are mapped to a different subset of $\mathcal{U}$. We omit the details of the proof in the interest of space.

Lemma 4.8. *The additional conditions of Lemma 4.7 are fulfilled when $\mathcal{E}$ is infinite and $|\mathcal{U}|$ is not constrained in any formula ϕ_i.*

This result is important for the combination of QFBAPA with other theories, as explained in Section 4.4. The proof follows from the fact that with an infinite domain $\mathcal{E}$ and an unconstrained universe $\mathcal{U}$, if there exists a model, we can extend it to a model $\mathcal{M}$ where $|\mathcal{U}^{\mathcal{M}}|$ is sufficiently large.

A decision procedure based on decompositions. Theorem 4.6 yields a decision procedure for the satisfiability of conjunctions of QFBAPA constraints $\phi_1 \land \ldots \land \phi_n$: independently for ϕ_1 to ϕ_n, introduce integer variables for the regions of venn(V_1), ..., venn(V_n), respectively, then introduce fresh variables for the regions of venn(V_S), where V_S is computed as in the theorem. Finally, constrain the sums of variables representing the same regions to be equal. (This ensures that the bijections f_i can be constructed in the satisfiable case.) As an example,

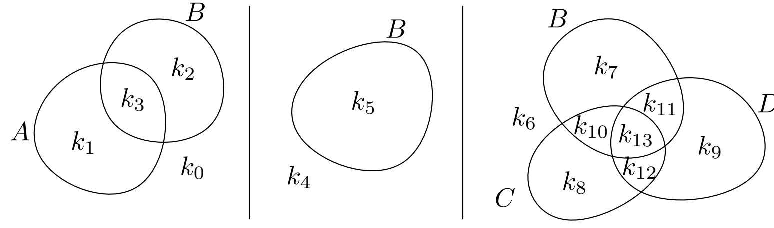

Figure 4.3 – Independent naming of the Venn regions of set variables $V_1 = \{A, B\}$, $V_S = \{B\}$, and $V_2 = \{B, C, D\}$.

consider the formula

$$|A \setminus B| > |A \cap B| \land B \cap C \cap D = \emptyset \land |B \setminus D| > |B \setminus C|$$

Let ϕ_1 be the first conjunct and ϕ_2 the other two. We have $V_1 = \{A, B\}$ and $V_2 = \{B, C, D\}$. Using the naming for regions shown in Figure 4.3, we obtain the integer constraints

$$\begin{array}{lr} k_1 > k_3 & \text{reduction of } \phi_1 \\ \land\ k_{13} = 0 \land k_7 + k_{10} > k_7 + k_{11} & \text{reduction of } \phi_2 \\ \land\ k_4 = k_0 + k_1 = k_6 + k_8 + k_9 + k_{12} & \text{representations of } |B^{\mathsf{c}}| \\ \land\ k_5 = k_2 + k_3 = k_7 + k_{10} + k_{11} + k_{13} & \text{representations of } |B| \end{array}$$

A possible satisfying assignment for the integer variables is $k_4 = 2$, $k_0 = k_1 = k_2 = k_5 = k_8 = k_9 = k_{10} = 1$, and $k_3 = k_6 = k_7 = k_{11} = k_{12} = k_{13} = 0$. From these values, we can build the assignment to set variables

$$\begin{array}{lr} \mathcal{U}_1 = \{e_1, e_2, e_3\}, A = \{e_1\}, B = \{e_2\} & \text{for } \phi_1 \\ \mathcal{U}_2 = \{e_4, e_5, e_6\}, B = \{e_4\}, C = \{e_4, e_5\}, D = \{e_6\} & \text{for } \phi_2 \end{array}$$

Finally, to build a model for $\phi_1 \land \phi_2$, we need to construct two bijections f_1 and f_2 such that $f_1(e_4) = f_2(e_2)$ and $f_1[\{e_5, e_6\}] = f_2[\{e_1, e_3\}]$. This is always possible in this decision procedure, because we constrain the sizes of the Venn regions of shared variables to be equal. Intuitively, the freedom in the construction for this example comes from the fact that when we merge the models, the overlapping between C or D and A is unconstrained: we can choose to map either e_5 or e_6 to the same element as e_1, in which case either C or D will share an element with A. So one possible model for ϕ is $A = \{e_1\}, B = \{e_2\}, C = \{e_1, e_2\}, D = \{e_3\}$. (Here we chose the identity function for f_1.) Note that this model also satisfies the property of models described in Lemma 4.7, since, in this interpretation, $A \neq C$ and $A \neq D$.

4.3.3 Hypertree Decompositions

In general, and as in the previous example, the constraints ϕ_i in Theorem 4.6 can contain top-level conjunctions. It is thus in principle possible to decompose them further. In this section,

we introduce a hypergraph representation of constraints from which it is straightforward to apply the decision procedure presented above recursively.

Definition 4.9 (Hypertree Decomposition). *Let ϕ be a* QFBAPA *formula and let $V = vars(\phi)$. Let $\phi_1,\dots,\phi_n$ be top-level conjuncts of ϕ and $V_1,\dots,V_n$ the sets of variables appearing in them, respectively. We assume without loss of generality that $V_i \neq V_j$ for $i \neq j$. (If two constraints ϕ_i and ϕ_j have the same set of variables, consider instead the constraint $\phi_i \wedge \phi_j$.) Let $\bar{V}$ denote $\{V_1,\dots,V_n\}$. Let $\mathscr{H} = (\bar{V},\bar{E})$ be a hypergraph on the set of sets of variables $\bar{V}$ (so $\bar{E} \subseteq 2^{\bar{V}}$). Let $l : \bar{E} \to 2^V$ be a labeling function that associates to each hyperedge a subset of the set variables of ϕ (that subset need not be one of V_i). We call $\mathscr{H}$ a* hypertree decomposition of ϕ*, if the following properties hold:*

$(H1)$ $\forall V_i, V_j \,.\, V_i \cap V_j \neq \emptyset \implies \exists E \in \bar{E} :\; V_i \in E \wedge V_j \in E \wedge (V_i \cap V_j) \subseteq l(E)$

$(H2)$ $\forall E \in \bar{E}, \forall V \in \bar{V} \,.\, V \in E \iff V \cap l(E) \neq \emptyset$

$(H3)$ *$\mathscr{H}$ is acyclic*[1]

Property $(H1)$ states that two nodes that share at least one variable must be connected with an hyperedge whose label contains at least their shared variables, and Property $(H2)$ states that a node is contained in a hyperedge if and only if the label of that hyperedge shares at least one set variable with the node. Note that a formula ϕ admits in general many distinct hypertree decompositions. For instance, the hypergraph $\mathscr{H} = (\bar{V}, E)$ with a single hyperedge $E = \bar{V}$ and $l(E) = \text{vars}(\phi)$ always satisfies Definition 4.9. To illustrate hypertree decompositions, consider the (unsatisfiable) formula

$$|A \cup B| \leq 3 \wedge C \subseteq B \wedge |(C \cap D) \setminus F| = 2 \wedge |(C \cap G) \setminus D| = 2 \wedge H \subseteq G \tag{4.1}$$

If we number the conjuncts from 1 to 5, we have $V_1 = \{A,B\}$, $V_2 = \{B,C\}$, $V_3 = \{C,D,F\}$, $V_4 = \{C,D,G\}$, $V_5 = \{G,H\}$. Figure 4.4 shows a possible hypertree decomposition of (4.1). The intuition behind hypertree decompositions is that hyperedges represent set of constraints, and that models for these sets need to agree only on the variables they share in their common nodes.

Reduction to integer arithmetic from hypertrees. We show now how to adapt the decision procedure presented at the end of Section 4.3.2 to hypertree decompositions. We proceed as follows: for each set of set variables V_i (each node in the hypertree), we introduce integer variables for all regions venn(V_i). Then, for each hyperedge E, we add variables for the regions venn$(l(E))$. Finally, for each node V_i and each hyperedge E such that $V_i \in l(E)$, we constrain the sums of variables describing the same Venn regions to be equal. For each $v \in \text{venn}(V_i \cap l(E))$,

1. We define as a cycle any set of at least two hyperedges such that there exists a node reachable from itself by traversing all hyperedges in the set. This implies in particular that no two hyperedges intersect on more that one node. Note that this is more restrictive than the definition of a hypertree as a hypergraph that admits a tree as a host graph.

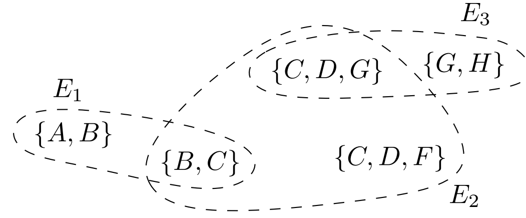

Figure 4.4 – A hypertree decomposition of (4.1). The labeling is given by: $l(E_1) = \{B\}$, $l(E_2) = \{C, D\}$, $l(E_3) = \{G\}$.

we generate the constraint:

$$\sum_{\substack{v_1 \in \text{venn}(V_i) \\ \text{s.t.} v_1 \subseteq v}} k_{V_i}(v_1) = \sum_{\substack{v_2 \in \text{venn}(l(E)) \\ \text{s.t.} v_2 \subseteq v}} k_{l(E)}(v_2)$$

where $k_{V_i}(\cdot)$ and $k_{l(E)}(\cdot)$ denote the integer variable representing a Venn region in the naming scheme for the Venn regions of V_i and $l(E)$, respectively.

This decision procedure has the same worst-time complexity as the one presented in Section 4.3.1; indeed, it is possible to construct a formula ϕ with a hypertree decomposition that leads to a reduction that uses up to 2^{N+1} integer variables, where N is the number of set variables in ϕ. We have found however that the reduction works well in practice (see Section 4.5).

4.4 Integration with Z3

We implemented our new decision procedure for QFBAPA constraints based on the results of Section 4.3 as an extension to the state-of-the-art SMT solver Z3 [dMB08b]. In this extension, sets do not range over uninterpreted elements but over the integers. The resulting prover can thus handle constraints such as

$$(S_1 = \{1, 2, 3, 4\} \land S_2 \subseteq S_1 \land S_2 \neq \emptyset) \implies |S_2| \in S_1$$

and such as our example from the introduction. Although we are thus in effect performing non-disjoint theory combination, we were able to implement our techniques using only the mechanism of theory plugin extensions to interact with the prover. Because such extensions can add arbitrary axioms (i.e. that are not restricted to theory elements) to the logical context of Z3 at any time, this was indeed not a major limitation. We wrote our extension in Scala and used the Java Native Interface to access the C API of Z3 [KKS11]. We did not observe any significant overhead in the forwarding of function calls from and to the Java virtual machine.

Reduction to the integers. We handle constraints on sets with cardinalities by reducing them to integer linear arithmetic using the translation presented in Section 4.3.3. Assume for now a fixed naming of Venn regions (we describe below how we maintain a hypertree and thus a naming in the presence of incremental reasoning). For each set constraint $S_1 \subseteq S_2$ or $S_1 = S_2$, we add an axiom constraining the integer representation of the set variables in the atom. For instance, using the naming of Venn regions in Figure 4.3, we would add for the constraint $B \cup C \subseteq D$ the implication $B \cup C \subseteq D \implies k_7 + k_8 + k_{10} = 0$. Similarly, for each cardinality term $|S|$, we add an axiom expressing the cardinality in terms of integer variables. Using the same naming as above, we would for instance add for the term $|C \setminus B|$ the equality $|C \setminus B| = k_8 + k_{12}$. These axioms enforce that whichever boolean value the SMT solver assigns to the set predicate, it will be forced to maintain for the integer variables an assignment that is consistent with the constraints on the sizes of Venn regions.

Incremental introduction of Venn regions. Our theory extension receives messages from the core solver whenever a constraint on sets or an application of the cardinality operator is added to the logical context. We maintain at all times a hypertree decomposition $\mathscr{H}$ of the conjunction of all constraints that are on the stack. Whenever a new constraint ϕ_i is pushed to the stack, we apply the following steps:

1. We compute $V_i = \text{vars}(\phi_i)$.
2. If there is a node in $\mathscr{H}$ labeled with V_i, we use the naming of Venn regions in that node to generate the axiom expressing the reduction of ϕ_i and skip the following steps.
3. Else we introduce a new node V_i and we create fresh integer variables for each region of $\text{venn}(V_i)$.
4. For each node V_j in $\mathscr{H}$ such that $V_i \cap V_j \neq \emptyset$, we add a hyperedge $E = \{V_i, V_j\}$ labeled with $l(E) = V_i \cap V_j$.
5. In the new graph, we collapse any introduced cycle as follows: let $E_1, \ldots, E_n$ be the hyperedges that participate in the cycle. We introduce a new hyperedge E labeled with $\bigcup_{i \in \{1,\ldots,n\}} l(E_i)$ and whose content is the union of the contents of $E_1, \ldots, E_n$. We then remove from the graph the edges E_1 to E_n.
6. For each newly created edge E in the obtained, acyclic, hypergraph we introduce fresh integer variables to denote the regions of $\text{venn}(l(E))$.
7. Finally, for each node in the new edge, we constrain the sums of integer variables that denote the same Venn regions to be equal.

Note that all introduced integer variables are additionally constrained to be non-negative. When constraints are popped from the stack:

1. We remove from $\mathscr{H}$ all nodes that were created for these constraints, if any.
2. For each hyperedge E that contained at least one removed variable, we check whether it now contains only one node, in which case we delete it.

Removing such nodes and edges provides two benefits: constraints added in a different search branch may generate a smaller (less connected) hypertree, and Z3 has the opportunity to remove from its clause database reduction axioms that have become useless. We efficiently detect cycles by maintaining equivalence classes between nodes that correspond to the connected components in the hypertree. We can then check, when we introduce a new hyperedge, whether it connects nodes from different equivalence classes, which corresponds to our definition of a cycle. We use a union-find data-structure to maintain the equivalence classes.

Interpreted elements. When considering sets in combination with other theories, it is natural to allow the use of the predicate $e \in S$ denoting that a given set contains a particular (interpreted) element, or constant sets $\{e_1, e_2\}$. For the simplicity of the argument, we assume here that all sets have the same underlying element sort $\mathscr{E}$. We handle interpreted elements such as e, e_1 and e_2 by adding to the logic two connecting functions, $\mathsf{singleton} : \mathscr{E} \to \mathsf{Set}$ and $\mathsf{element} : \mathsf{Set} \to \mathscr{E}$. We create for each interpreted element that appears in a predicate $e \in S$ or in a constant set a singleton set term $\mathsf{singleton}(e)$, and we add the axioms $|\mathsf{singleton}(e)| = 1$ and $\mathsf{element}(\mathsf{singleton}(e)) = e$. So singleton is interpreted as a constructor that builds a singleton set from an element, and element as a function that for singleton sets computes its element, and that is uninterpreted otherwise. We can thus rewrite $e \in S$ as $\mathsf{singleton}(e) \subseteq S$ and constant sets as unions of singletons. The connecting functions are used to propagate equalities: when two elements are found to be equal in their theory, congruence closure will conclude that their singleton sets (if they exist) need to be equal as well. The other direction is covered in the following paragraph.

Communicating equalities. The reduction to integer linear arithmetic ensures that if two sets must be equal, then the sizes of the Venn regions of their symmetric difference will be constrained to be 0. However, for completeness we also need to detect that if their symmetric difference is forced to be empty by some integer constraints, then the sets must be equal. To enforce this, we could, for each pair of set variables (S_1, S_2) generate an axiom equivalent to

$$(|S_1 \setminus S_2| = \emptyset \wedge |S_2 \setminus S_1| = \emptyset) \implies S_1 = S_2$$

where the cardinality terms would be rewritten using the naming of set regions for S_1 and S_2. However, because not every pair of set variable appears together in some constraint, our hypertree decomposition does not in general define names for the intersections of any variables S_1 and S_2. Our solution is to generate such axioms for all pairs of variables that appear together in a node of the hypergraph. Additionally, for each set variable we generate the axiom $|S| = 0 \iff S = \emptyset$. We argue that, for sets that range over an infinite domain (such as the integers), this is sufficient: Consider two set variables S_1 and S_2 that do not appear in a common node. From lemmas 4.7 and 4.8, we have that as long as S_1 or S_2 is non-empty or distinct from a shared set variable, there exists a model in which $S_1 \neq S_2$. Both cases are covered by the introduced axioms. Indeed, even if the two variables are equal to a different

shared variable, by transitivity of the equality, their equality will be propagated. We thus have the property that any two set variables that are not constrained to be equal can be set to be distinct in a model. This is consistent with the theory combination approach taken in Z3[dMB08a]. When we construct the hypergraph as presented above, we also make sure that all terms representing singleton sets (that is, applications of the singleton(·) function) share a common hyperedge, even if they don't syntactically appear in a common constraint. This ensures that all equalities between finite sets induced by cardinality constraints will result in equalities between their elements through Z3's congruence closure algorithm, and will be communicated to the proper theory solver.

4.5 Evaluation

We evaluated our implementation using benchmarks from two verification sources. Figure 4.5 shows our experimental results. [2]

Jahob benchmarks. We included all benchmarks from [KR07] in our evaluation. These formulas express verification conditions generated with the Jahob system [ZKR08] for programs manipulating (abstractions of) pointer-based data-structures such as linked lists. In [KR07], the benchmarks were used to compare the efficiency of the sparse encoding into linear arithmetic with the explicit one (as in Section 4.3.1). We indicate for these benchmarks the previously best time using either method. It is important to note that it was shown in [KR07] that the two methods were complementary: the sparse encoding outperformed the explicit one for sat instances and vice-versa. We do not claim that the absolute difference in time is a significant measure of our algorithmic improvements, but rather provide these numbers to illustrate that our new techniques can be used to efficiently handle sat and unsat instances.

Functional programs. We also included new benchmarks consisting of verification conditions for Scala functions without side effects. Additionally to sets and elements, these examples contain constraints on algebraic data types and functions symbols. Each verification condition contains at least one (recursive) function call. We first treated these function symbols as uninterpreted. We then used universally quantified axioms to define the interpretation of the functions and used Z3's pattern-based instantiation mechanism to apply the axioms. Without the axioms, all formulas were invalid (sat), and with the axioms, they were all proved valid. While we could in principle have used our Leon verification system (presented in Chapter 3), the option of using Z3's quantifier instantiation mechanism presented the advantage that all constraints are solved within a single invocations, as opposed to a succession of round-trips.

To the best of our knowledge, the only previous implementation that is complete for QFBAPA and reports performance on benchmarks is [KR07]. We show significant improvement over

2. All benchmarks and the set of axioms we used are available from http://lara.epfl.ch/~psuter/bapaz3/.

the existing benchmarks. While results in [KR07] were unable to handle unsatisfiable formulas with 16 variables, we report success on formulas with 29 variables, which were automatically generated from verification of functional programs.

4.6 Related Work

The algorithm presented in this chapter was first described in [SSK11]. The complexity of QFBAPA was settled in [KR07] to be NP. The complexity of the quantified case as well as the first quantifier elimination implementation was described in [KNR06]. The decidability of the logic itself dates back at least to [FV59].

The work of combination of QFBAPA with other decidable theories has so far included implementation as part of the Jahob system, which requires manual decomposition steps to be complete [ZKR08, ZKR09]. A complete methodology for using QFBAPA as a glue logic for non-disjoint combination was introduced in [WPK09], with additional useful cases introduced in [SDK10, YPK10], some of which are surveyed in [KPSW10]. The combination method we describe is simple, in that it does not require exchanging set constraints between different theories that share sets of objects. In that sense, it corresponds to the multi-sorted combination setup of [Zar02], which introduces a non-deterministic procedure that was, to the best of our knowledge, not implemented. Combinations of theories that have finite domains has been explored in [KGGT07]. In our presentation and implementation, we have focused on combinations with integers, but we believe that our approach can be adapted to more general cases.

Our decomposition of formulas was inspired by the algorithms for bounded (hyper)tree width from constraint satisfaction [GGM09], although we do not directly follow any particular decomposition algorithm from the literature. These algorithms are typically used to reduce subclasses of NP-hard constraint satisfaction problems over finite domains to polynomial-time algorithms. To the best of our knowledge, they have not been applied before to satisfiability of sets with cardinality operators. Our results suggest that this approach is very promising and we expect it to extend to richer logics containing QFBAPA, such as [YPK10].

Research in program analysis has used cardinality constraints in abstract domains [GLAS09, PRS09], typically avoiding the need for a full-fledged QFBAPA decision procedure. Thanks to our efficient implementation of QFBAPA, we expect that precise predicate abstraction approaches [BHJM07] will now also be able to use QFBAPA constraints.

Benchmark	status	sets	cons.	Venn regs.	prop.	prev. best	our time
CADE07-1	unsat	1	1	2	1.00	< 0.1	< 0.1
CADE07-2	unsat	2	1	4	1.00	< 0.1	< 0.1
CADE07-2a	unsat	6	5	28	0.44	1.8	< 0.1
CADE07-2b	sat	6	5	28	0.44	< 0.1	< 0.1
CADE07-3	unsat	2	1	4	1.00	< 0.1	< 0.1
CADE07-3a	unsat	5	4	16	0.50	0.4	< 0.1
CADE07-3b	sat	5	4	16	0.50	< 0.1	< 0.1
CADE07-4	unsat	5	5	80	2.50	0.5	< 0.1
CADE07-4b	sat	5	5	76	2.38	0.1	< 0.1
CADE07-5	unsat	7	5	168	1.31	13.6	< 0.1
CADE07-5b	sat	7	5	168	1.31	0.4	< 0.1
CADE07-6	unsat	6	6	32	0.50	0.4	< 0.1
CADE07-6a	unsat	16	20	596	0.01	> 100	0.3
CADE07-6b	sat	16	22	1120	0.02	0.8	0.3
CADE07-6c	sat	16	20	596	0.01	0.9	0.4
listContent	sat	3	4	14	1.75		< 0.1
listContent-ax	unsat	4	5	16	1.00		< 0.1
listReverse	sat	6	6	26	0.41		< 0.1
listReverse-ax	unsat	9	11	308	0.60		0.2
setToList	sat	6	8	76	1.19		< 0.1
setToList-ax	unsat	7	9	114	0.89		< 0.1
listConcat	sat	11	19	54	0.03		0.2
listConcat-ax	unsat	28	33	268	< 0.01		0.4
treeContent	sat	4	5	24	1.50		0.1
treeContent-ax	unsat	6	7	28	0.44		0.1
treeMirror	sat	6	6	20	0.31		< 0.1
treeMirror-ax	unsat	11	11	572	0.28		0.3
treeToList	sat	8	10	24	0.09		0.1
treeToList-ax	unsat	29	35	2362	< 0.01		1.7

Figure 4.5 – Experimental results. The column “sets” is the number of set variables, “cons.” is the maximal number of constraints on the stack, “Venn regs.” is the maximal number of distinct Venn regions created from the hypertree structure, “prop.” is the proportion of created Venn regions compared to the 2^N naive naming scheme and is a measure of the efficiency of our decomposition technique, “prev. best” indicates the previously best solving time, and “time” is the running time with the new implementation. All times are in seconds. The experiment was conducted using a 2.66GHz Quad-core machine, and using Z3 2.11.

5 Programming with Constraints

In this chapter, we move on from the use of specifications as a support for verification to their use as a computation mechanism. We present an extension of Scala that supports constraint programming over bounded and unbounded domains. The resulting language, Kaplan, provides the benefits of constraint programming while preserving the existing features of Scala. Kaplan integrates constraint and imperative programming by using constraints as an advanced control structure; the developers use the monadic 'for' construct to iterate over the solutions of constraints or branch on the existence of a solution. The constructs we introduce have simple semantics that can be understood as explicit enumeration of values, but are implemented more efficiently using symbolic reasoning.

Kaplan programs can manipulate constraints at run-time, with the combined benefits of type-safe syntax trees and first-class functions. The language of constraints is a functional subset of Scala, supporting arbitrary recursive function definitions over algebraic data types, sets, maps, and integers.

Our implementation runs on a platform combining a constraint solver with a standard virtual machine. For constraint solving we use Leon, presented in Chapter 3. We evaluate Kaplan on examples ranging from enumeration of data structures to execution of declarative specifications. We found Kaplan promising because it is expressive, supporting a range of problem domains, while enabling full-speed execution of programs that do not rely on constraint programming.

5.1 Introduction

Modern mainstream languages incorporate advances in memory safety, type systems, metaprogramming and modularity. However, the sequential control in widely used systems remains largely unchanged compared to some of the earliest imperative and functional language designs. Although this simplicity has advantages in terms of predictable compilation, it prevents languages from approaching the abstraction level used in software design.

The idea of Logic Programming [KK71, CKC81] is to allow programs that are non-deterministic and specify relations (predicates) between values. The original inspiration came from a restricted form of resolution as the execution mechanisms, which uses unification of logical variables ranging over finite trees. Functional logic programming extends logic programming with the narrowing technique and with benefits of functional programming [AH10]. Constraint Logic Programming, CLP [JL87] and later generations of Prolog [CKC81] extend logic programming with the ability to perform constraint solving not only over trees but also over numerical domains. These paradigms hold great promise to raise the abstraction level of software. They are related to program synthesis research [MW80, PR88, SLTB+06, KMPS10b, GJTV11], which can be viewed as a compilation mechanism for declarative specifications.

We believe that key obstacles preventing broader use of non-deterministic declarative programming constructs include:
– the difficulty of solving declarative constraints and
– the difficulty of incorporating these constructs into existing languages and platforms.
We next briefly outline the approach that our system uses to meet these challenges.

Efficient interoperability with existing platforms. In this chapter we present Kaplan, a system that supports Constraint Programming on top of Scala. We address the difficulty of incorporating into existing platforms by choosing not to modify the semantics of the core Scala language, but instead use the flexibility of for-comprehensions in Scala. Because for-comprehensions present a syntax for monads in Scala, this aspect of our approach is related to using non-determinism monads in Haskell [FKcS09]. An important difference is that our starting language is not purely functional, but can have side effects.

Our solution is therefore to identify a functional Turing-complete sublanguage for constraints, and use it locally for declarative programming within the full Scala language (which has features of object-oriented, imperative and functional languages). The declarative control that we explore is based on describing iterations as a search process. The iteration ranges are specified as the set of solutions to constraints. Our constraints are declared using first-class functions and familiar combinators, such as &&. From a language point of view the integration is appealing because users hardly need to learn any new notation.

One approach to implementation of non-deterministic languages is to deploy a virtual machine that supports backtracking, such as Warren's Abstract Machine for Prolog [AK91] or the Java Pathfinder model checker and its extensions [GGJ+10]. Unfortunately, such an approach is costly in terms of both performance and engineering effort. We propose instead to encapsulate any need for backtracking into a for-comprehension that iterates over solution spaces. In addition, we allow constraint programming in code outside of loops. In such cases, logical variables and a constraint store help ensure that a program can find a solution without backtracking over the host imperative program, keeping backtracking within a specialized constraint solver.

Efficient solving of declarative constraints. The first implementations of declarative paradigms predate algorithmic advances of modern SAT solvers [MS96, ZMMM01]. The advances in SAT led to a paradigm shift in solving combinatorial problems, where it became often profitable to outsource constraint solving tasks to dedicated implementations, instead of relying on a programmer to hard-code a search strategy.

Fueled by the developments of SAT solvers, a more expressive technology emerged as the field of Satisfiability Modulo Theories (SMT), with a number of efficient implementations available today [BT07, DdM06, dMB08b, BPST10]. SMT techniques combine SAT solvers with decision procedures and their combination methods [DNS05, NO80, GHN+04] that were and remain motivated by program verification tasks [Nel81, Kin71].

Our aim is to leverage the remarkable progress in dedicated constraint solvers by deploying an SMT solver as a part of the run-time of a programming language. Whereas the developers of Prolog were influenced by the state-of-the-art theorem proving technology of their time (resolution for first-order logic), we aim to explore the potential of SMT. The fact that SMT solvers were developed to model programming language constructs makes them a particularly appropriate choice. However, the original motivation for SMT solvers was program verification, whereas we aim to use them as an execution mechanism for declarative constraints.

Many logical theories are natively supported by modern solvers, including algebraic data types, uninterpreted functions, linear arithmetic, and arrays. Nonetheless, we believe that constraint solving can reach its full potential only if users can define their own classes of constraints. We therefore choose to work with Leon, which supports a rich logic with user-defined recursive functions and recursive data types. We thus trade performance and decidability for greater expressive power and flexibility.

Our work as presented in this chapter makes a concrete and implemented proposal for incorporating constraint programming into an underlying stateful language through several individual contributions:
- first-class constraints, which can be generated at run-time and which carry type and free-variable information; the constraints can use unrestricted operations on booleans (including negation), as well as built-in and user-defined recursive operations on integers, sets, maps, and trees;
- a programming model for constraint programming based on creating and iterating over a stream of solutions of a constraint using explicit control constructs; our programming model (unlike most solutions with backtrackable virtual machines) has no penalty for the code that does not use constraint solving;
- logical variables of numeric and symbolic types that postpone constraint solving steps, often substantially reducing the size of the search space;
- the use of fair function unrolling and SMT solving technology to provide expressive power, predictability, and efficiency of solving in many domains of interest;
- implementation of the system (Kaplan is publicly available from http://lara.epfl.ch);

– evaluation of the system on examples from several domains, such as executing declaratively specified data structure operations, software testing, and counterexample-driven construction of functions of a given template and a given specification.

The rest of this chapter is organized as follows. The next section presents features of Kaplan through an extensive set of examples. Section 5.3 gives the semantics of key Kaplan constructs. Section 5.4 outlines main implementation aspects of Kaplan. We present further evaluation and illustrate use cases of Kaplan in Section 5.5. We finally review the remaining related work and conclude.

5.2 Examples and Features

We present some of the features of Kaplan through examples. The simpler examples can be tried out directly in a Scala shell, provided that it is launched with the Kaplan plugin.

5.2.1 First-class Constraints

```
val c1: Constraint2[Int,Int] = ((x: Int, y: Int) ⇒ 2*x + 3*y == 10 && x ≥ 0 && y ≥ 0)
```

This first command declares a constraint with two free variables. Note that the representation of the constraint is a lambda term. The only difference between a declaration of a constraint and an anonymous function is the type of the expression. As an alternative to explicitly declaring the value to be of a constraint type, one can also append to the function literal a method call `.c`, so the following declaration is identical to the previous one:

```
val c1 = ((x: Int, y: Int) ⇒ 2*x + 3*y == 10 && x ≥ 0 && y ≥ 0).c
```

The type of `c1` is in this case determined by type inference, and this second way is generally shorter. In Kaplan, constraints are in fact extensions (in the object-oriented sense) of functions, and can thus be evaluated, given some values for their argument variables:

```
scala> c1(2,1)
result: false
scala> c1(5,0)
result: true
```

As is common in lambda calculus, the names given to the bound variables play no role; the constraint `((x: Int)⇒x≥0)` is equivalent to the constraint `((y: Int)⇒y≥0)`. One should think of constraints being defined over (typed) De Bruijn indices rather than named variables.

Constraints can be queried for a single solution or for a stream of solutions by calling appropriate methods:

```
scala> c1.solve
result: (5,0)
scala> c1.findAll
```

```
result: non−empty iterator
```

The solve method computes a single solution to the constraint, while findAll returns an iterator over all solutions. The iterator computes and returns solutions on demand. (As a result of displaying the iterator in the console, a search for the first solution is triggered.) When the set of solutions is finite, one can compute it for instance as follows:

```
scala> c1.findAll.toList
result: List((5,0),(2,2))
```

The most general way to iterate over solutions is using a for-comprehension:

```
for(s ← c1.findAll) {
  println(s)
}
```

Constraints are first-class members of Kaplan, and can be created and manipulated as such. The following function, for instance, generates constraints describing an integer within bounds; if the second argument is true, the bounds are given by $[-m; m]$, else by $[0; m]$.

```
def bounded(m: Int, neg: Boolean = true) = {
  val basis: Constraint1[Int] = ((x: Int) ⇒ x ≤ m)
  basis && (if(neg)
    ((x: Int) ⇒ x ≥ −m)
  else
    ((x: Int) ⇒ x ≥ 0))
}
```

In this example, && is a call to a method defined as part of constraint classes, and which expects as argument a constraint of the same arity and with the same types of parameters as the receiver. This ensures that constraints can only be combined when the number and the types of their De Bruijn indices are compatible. Note that, thanks to type inference, we need not explicitly state that the two anonymous functions represent constraints. We can now use the function bounded to produce constraints:

```
scala> bounded(3, true).findAll.toList
result: List(0, −1, −2, −3, 1, 2, 3)
scala> bounded(3, false).findAll.toList
result: List(0, 1, 2, 3)
```

Another convenient constraint combinator is the product method, which combines two constraints into a new one whose solution set is the Cartesian product of the original two:

```
scala> (bounded(1, true) product bounded(1,false)).findAll.toList
result: List((−1,0),(0,0),(1,1),(−1,1),(1,0),(0,1))
```

Constructing constraints using combinators is a very concise way to solve general problems. For instance, consider the problem of solving a CNF SAT instance defined in a way similar to the standard DIMACS format, where the input

```
val p1 = Seq(Seq(1,−2,−3), Seq(2,3,4), Seq(−1,−4))
```

represents the problem

$$(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_4)$$

The following function defines a solver for such problems:

```
def satSolve(problem : Seq[Seq[Int]]) : Option[Map[Int,Boolean]] =
  problem.map(clause ⇒ clause.map(lit ⇒ {
    val variable = scala.math.abs(lit)
    val isPos = lit > 0
    ((m : Map[Int,Boolean]) ⇒ m(variable) == isPos).c
  }).reduceLeft(_ || _)).reduceLeft(_ && _).find
}
```

Note that in this case we used the find method on the final constraint rather than solve. They provide the same functionality, but find returns its result in an Option[_] type, where None corresponds to an unsolvable constraint, while solve throws an exception if the constraint has no solution. Another observation is that we use here a constraint over a single variable of Map type to encode a constraint over an unknown number of boolean variables. We found this to be a convenient pattern when the more rigid syntax of anonymous functions with explicit variable naming does not apply. Because all variables are of the same type, the approach works in our type-safe framework. We can now solve SAT problems:

```
scala> satSolve(p1)
result: Some(Map(2 → true, 3 → false, 1 → false, 4 → false))
scala> satSolve(Seq(Seq(1,2), Seq(−1), Seq(−2)))
result: None
```

5.2.2 Ordering Solutions

As illustrated by some of the previous examples, the findAll method generates solutions in no particular order. This corresponds to the intuition that it enumerates the unordered set of solutions to a constraint. Similarly, the semantics of calls to solve or find are simply that if a value is produced, then it is an element of that solution set. Two invocations of solve on the same constraint may or may not result in the same solution.

It is sometimes desirable, though, to enumerate solutions in a defined order. To this end, Kaplan constraints support two methods; minimizing and maximizing. These methods take as argument an objective function. Just like constraints, objective functions are represented using an anonymous function, in this case one that returns an integer. The minimizing and

maximizing methods ensure that the De Bruijn indices' types match those of the constraint. From the user's point of view, these functions are typed as IntTerms. [1]

Consider the knapsack problem: given a maximum weight and a set of items, each with an associated value and weight, find a subset of the items for which the sum of values is maximal while the sum of weights is less or equal to the maximum. The following code produces and solves an instance of the problem, where the solution is represented as a map from the item indices to booleans indicating which should be picked.

```
def solveKnapsack(vals : List[Int], weights : List[Int], max : Int) = {
  def conditionalSumTerm(vs : List[Int]) = {
    vs.zipWithIndex.map(pair ⇒ {
      val (v,i) = pair
      ((m : Map[Int,Boolean]) ⇒ (if(m(i)) v else 0)).i
    }).reduceLeft(_ + _)
  }
  val valueTerm = conditionalSumTerm(vals)
  val weightTerm = conditionalSumTerm(weights)
  val answer = ((x : Int) ⇒ x ≤ max).compose0(weightTerm)
    .maximizing(valueTerm)
    .solve
}
```

We briefly explain the code. A solution to a knapsack instance is a map indicating a choice of which objects should be picked. The conditionalSumTerm function builds, from a list of integers, an integer term parameterized by a choice map and representing a sum of values. The map defines whether each element in the list participates or not to the sum. We use this function twice, to produce the two terms that, given a choice of items, encode the total value and the weight respectively. Observe that we build the final constraint using a function composition: we start the construction of the constraint as (x : Int)⇒x≤max and compose it with the weight term to produce the constraint that the weight should not exceed the maximum. Function composition is a general and type-safe way to build constraints from smaller terms, as demonstrated in this example. We can now find optimal choices for instances of the knapsack problem:

```
scala> val vals : List[Int] = List(4, 2, 2, 1, 10)
scala> val weights : List[Int] = List(12, 1, 2, 1, 4)
scala> val max : Int = 15
scala> solveKnapsack(vals, weights, max)
result: Map(0 → false, 1 → true, 2 → true, 3 → true, 4 → true)
```

1. The relationship from an implementation point of view between constraints and terms is explained in more details in Section 5.4.

5.2.3 User-defined Functions and Datatypes

An important feature of Kaplan is the ability to perform constraint solving in the presence of user-defined functions and data types. Consider the following functions which compute, for a 2×2 matrix $\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$ represented as a tuple(a, b, c, d), its determinant and whether it is unimodular, respectively:

```
@spec def det(a: Int, b: Int, c: Int, d: Int): Int = a*d − b*c
@spec def unimodular(a: Int, b: Int, c: Int, d: Int): Boolean = {
  val dt = det(a, b, c, d)
  dt == 1 || dt == −1
}
```

The @spec annotation indicates that the user wishes to use the functions as part of constraints, and the Kaplan compiler enforces that such functions are written in PureScala (see Section 3.3). We can now characterize unimodular matrices with small elements as

```
def boundedUnimodular(m: Int) = {
  val b = bounded(m, false)
  (b product b product b product b) && (unimodular _)
}
```

(The underscore after unimodular indicates to the Scala compiler that it should apply an η-conversion to produce an anonymous function, and ultimately a constraint, from the definition.) We can use these new definitions to generate unimodular matrices:

```
scala> boundedUnimodular(2).findAll.take(4).toList
result: List((0,1,1,0), (0,1,1,2), (0,1,1,1), (1,1,2,1))
```

Perhaps more interestingly, the @spec functions can be mutually recursive. As an example, consider the following declarative definition of prime numbers:

```
@spec def noneDivides(from : Int, j : Int) : Boolean = {
  from == j || (j % from != 0 && noneDivides(from+1, j))
}
@spec def isPrime(i : Int) : Boolean = (i ≥ 2 && noneDivides(2, i))
val primes = ((isPrime(_:Int)) minimizing ((x:Int) ⇒ x)).findAll
```

which we can subsequently enumerate:

```
scala> primes.take(10).toList
result: List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

Kaplan users can also define their own algebraic data types. The following code declares two such types for red-black trees:

```
@spec sealed abstract class Color
@spec case class Black() extends Color
@spec case class Red() extends Color
```

```
@spec sealed abstract class Tree
@spec case class Node(c : Color, l : Tree, v : Int, r : Tree) extends Tree
@spec case class Leaf() extends Tree
```

Algebraic data types are best manipulated using recursive functions and pattern-matching. Kaplan supports such function definitions:

```
@spec def size(tree : Tree) : Int = (tree match {
  case Leaf() ⇒ 0
  case Node(_, l, _, r) ⇒ 1 + size(l) + size(r)
}) ensuring(result ⇒ result ≥ 0)
```

This function also illustrates the use of post-conditions. In this case, the post-condition states that the result of size can never be negative. Such annotations can help the constraint solver discard parts of the search space. Kaplan uses Leon to prove, at compile time, that the annotations are valid, so that it can then rely on them at run-time for constraint solving. We can similarly define recursive functions that compute whether a red-black tree contains sorted keys or has the right coloring properties. In the interest of readability, we omit the complete definitions.

```
@spec def orderedKeys(t : Tree) : Boolean = ...
@spec def validColoring(t : Tree) : Boolean = ...
@spec def validTree(t : Tree) = orderedKeys(t) && validColoring(t)
@spec def valsWithin(t : Tree, min : Int, max : Int) : Boolean = ...
```

We expect that what these functions compute is clear from their name. With them, we can for instance count the number of red-black trees with a given number of elements:

```
scala> (for(i ← (0 to 7)) yield ((t : Tree) ⇒ validTree(t) &&
   valsWithin(t, 0, i) && size(t) == i ).findAll.size).toList
result: List(1, 1, 2, 2, 4, 8, 16, 33)
```

More advanced applications of data structure enumeration are presented in Section 5.5, including testcase generation for sorted list and tree manipulations.

5.2.4 Timeouts

The language of constraints supported in Kaplan is very expressive. For that reason, one cannot expect that any constraint will be solvable. As an example, given the definition (valid in Kaplan)

```
@spec def pow(x : Int, y : Int) : Int = if(y == 0) 1 else x * pow(x, y − 1)
```

it would be unreasonable to expect the system to find a solution to the constraint

```
val fermat = ((x : Int, y : Int, z : Int, b : Int) ⇒ b > 2 && pow(x,b) + pow(y,b) == pow(z,b)).c
```

As we have seen in Chapter 3, the search algorithm of Leon which forms the core of Kaplan is a semi-decision procedure: if a solution exists, then that solution will be found eventually. If there are no solutions, the procedure can discover this fact and stop, or loop forever. The solve, find and findAll methods all take a parameter describing the timeout strategy. That parameter is optional, so by default no timeout is used. Because it is also implicit, a timeout strategy can be defined for an entire scope by a single definition that the Scala compiler then automatically inserts at each call site:

```
implicit val timeoutStrategy = Timeout(1.0)
```

Given the above declarations, the following attempt to confirm Fermat's last theorem results in an exception after 1 second:

```
scala> fermat.find
result: TimeoutReachedException: No solution after 1.0 second(s)
  at .solve(<console>)
  at .<init>(<console>)
  ...
```

5.2.5 Logical Variables

All the examples so far have illustrated *eager* solution enumeration, where solving constraints immediately produces concrete values. While this by itself is a convenient facility, much of the power of constraint programming in general and of Kaplan in our case comes from the ability to produce logical variables, which represent the *promise* of a solution. Logical variables in turn can be used to control the execution flow of the program in novel ways. We start by describing some of their basic properties. More advanced examples of programming with logical variables follow.

In Kaplan, logical variables are always produced as the result of *lazily* solving a constraint. This is done by calling lazySolve, lazyFind or lazyFindAll instead of solve, find or findAll respectively. Consider the constraint we defined earlier:

```
val c1 = ((x: Int, y: Int) ⇒ 2*x + 3*y == 10 && x ≥ 0 && y ≥ 0).c
```

We produce logical variables representing a solution as follows:

```
scala> val (x,y) = c1.lazySolve; println((x,y))
result: (L(?),L(?))
```

Notice that the result is not a pair of integers, as in the case of solve, but a pair of objects of type L[Int] representing the promise of integers. The question mark indicates that the value has not yet been fixed.

Logical variables in Kaplan have *singular* semantics [RRH10], meaning that a given logical variable will always represent the same concrete value, even when it is copied or passed as an

argument to a function. The identity of a logical variable is determined by the identity of the instance of the L[_] class. The value of a logical variable is fixed as soon as it is queried, which can be done with the .value method:

```
scala> x.value
result: 5
scala> println((x,y))
result: (L(5),L(?))
```

(Note that even though the y can at this point only hold the value 0, the solver has not necessarily detected that the solution is unique and thus still considers the value not to be fixed.) Fixing a value needs not always be done explicitly. The Kaplan library defines two implicit conversions between concrete and logical values:

```
implicit def concretize[T](l : L[T]) : T = l.value
implicit def lift[T](v : T) : L[T] = new FixedL[T](v)
```

The first one simply fixes the value whenever a function expects a concrete value and receives a logical variable. The second one converts a concrete value into a specialized representation of a logical variable. It is used to provide a common representation for concrete and logical values: because Kaplan is built as a thin layer over Scala and most code must execute as usual, we cannot afford to treat every value as logical. Instead, users decide which function can handle logical values by using L[_] types in their signature. Because concrete values can be lifted to logical status, such functions work indifferently with standard and logical variables. Logical variables created through this lifting mechanism are treated specially and do not add any complexity to the solving process.

5.2.6 Imperative Constraint Programming

Scala being a multi-paradigm language, users can alternate between different programming styles depending on their preferences and on the task at hand. The same is true for Kaplan; while eager solution enumeration is best used within functional style for-comprehensions, logical variables can be used rather naturally in an imperative style, thanks to novel control constructs. The Kaplan library defines the **assuming-otherwise** branching construct. It is similar in nature to if-then-else, except for an important difference: rather than strictly evaluating the branching condition, assuming-otherwise blocks check whether the condition is *feasible* and, if so, constrain the logical variables in the environment so that their values satisfy the branching constraint. If the condition contains no logical variable (or only logical variables that have already been fixed), then assuming-otherwise behaves exactly like if-then-else.

As an example, consider the classical puzzle that consists in finding distinct values for the

letters representing digits in the following addition such that the sum is valid:

		s	e	n	d
+		m	o	r	e
=	m	o	n	e	y

We now present a solution in Kaplan written in an imperative style.

```
val anyInt = ((x : Int) ⇒ true).c
val letters @ Seq(s,e,n,d,m,o,r,y) = Seq.fill(8)(anyInt.lazySolve)
```

This second line uses pattern-matching syntax to achieve two things at once: 1) the eight letter variables are bound to eight independent (lazy) representations of a solution to the trivial anyInt constraint and 2) the variable letters is bound to the sequence of all letter variables. At this point, we have 8 unconstrained integer logical variables. We can imperatively add constraints:

```
for(l ← letters) asserting(l ≥ 0 && l ≤ 9)
```

If this loop terminates without any error, then at the end of it, the 8 variables each represent a number between 0 and 9. (Calling asserting(cond) is equivalent to **assuming**(cond){} **otherwise** { error }, just like **assert**(cond) is (conceptually at least) equivalent to **if**(cond){} **else** { error }. More details are given in Section 5.4.1.) We further constrain the letters representing most-significant digits:

```
asserting(s > 0 && m > 0)
```

We can now perform symbolic arithmetic using the existing and new logical variables. We define a new variable for the sum of each line and constrain it to the expected value:

```
val fstLine = anyInt.lazySolve
asserting(fstLine == 1000*s + 100*e + 10*n + d)
val sndLine = anyInt.lazySolve
asserting(sndLine == 1000*m + 100*o + 10*r + e)
val total = anyInt.lazySolve
asserting(total == 10000*m + 1000*o + 100*n + 10*e + y)
```

At this point, the value of the letters is still not fixed, however Kaplan ensures that all variables admit solutions satisfying the asserted constraints. Finally, we check for a solution to the puzzle using one last assuming-otherwise block:

```
scala> assuming(
  distinct(s,e,n,d,m,o,r,y) && fstLine + sndLine == total) {
    println("Solution: " + letters.map(_.value))
  } otherwise {
    println("The puzzle has no solution.")
  }
result: Solution: List(9, 5, 6, 7, 1, 0, 8, 2)
```

$$\textsc{Host}\ \frac{t_1 \mid \mu_1 \ \rightarrow_H \ t_2 \mid \mu_2}{t_1 \mid \langle \mu_1, \kappa \rangle \ \rightarrow \ t_2 \mid \langle \mu_2, \kappa \rangle}$$

$$\textsc{S-Sat}\ \frac{\mathcal{M} \models \phi}{(\lambda \bar{x}.\phi(\bar{x})).\mathsf{find} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathsf{Some}(\mathcal{M}(\bar{x})) \mid \langle \mu, \kappa \rangle}$$

$$\textsc{S-Unsat}\ \frac{\neg \exists \mathcal{M} : \mathcal{M} \models \phi}{(\lambda \bar{x}.\phi(\bar{x})).\mathsf{find} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathsf{None} \mid \langle \mu, \kappa \rangle}$$

$$\textsc{L-Sat}\ \frac{\mathcal{M} \models \kappa \wedge \phi[\bar{x} \mapsto \bar{l}_f] \qquad \bar{l}_f \text{ fresh in } \kappa}{(\lambda \bar{x}.\phi(\bar{x}, \bar{l})).\mathsf{lazyFind} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathsf{Some}(\bar{l}_f) \mid \langle \mu, \kappa \wedge \phi[\bar{x} \mapsto \bar{l}_f] \rangle}$$

$$\textsc{L-Unsat}\ \frac{\neg \exists \mathcal{M} : \mathcal{M} \models \kappa \wedge \phi[\bar{x} \mapsto \bar{l}_f] \qquad \bar{l}_f \text{ fresh in } \kappa}{(\lambda \bar{x}.\phi(\bar{x}, \bar{l})).\mathsf{lazyFind} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathsf{None} \mid \langle \mu, \kappa \rangle}$$

$$\textsc{Value}\ \frac{\mathcal{M} \models \kappa}{l.\mathsf{value} \mid \langle \mu, \kappa \rangle \ \rightarrow \ \mathcal{M}(l) \mid \langle \mu, \kappa \wedge (l = \mathcal{M}(l)) \rangle}$$

$$\textsc{Lift}\ \frac{c \text{ is a constant} \qquad l_f \text{ fresh in } \kappa}{c.\mathsf{lift} \mid \langle \mu, \kappa \rangle \ \rightarrow \ l_f \mid \langle \mu, \kappa \wedge (l_f = c) \rangle}$$

Figure 5.1 – Small-step semantics of Kaplan-specific constructs.

We mentioned that assuming-otherwise is conceptually close to if-then-else, and in fact equivalent in the absence of logical variables. One important difference is that the construct is asymmetrical; while

```
if(cond) thenExpr else elseExpr
```

is equivalent to

```
if(!cond) elseExpr else thenExpr
```

the same transformation cannot be applied to assuming-otherwise blocks; indeed, the semantics are that the control will attempt to satisfy the branching condition, so whether the positive or negative condition is tested has an impact on the rest of the execution. Section 5.4.1 discusses the implementation of assuming-otherwise blocks in terms of lazyFind.

5.3 Semantics

In this section, we present an overview of the semantic aspects of Kaplan through operational semantic rules, shown in Figure 5.1. All features of Kaplan can be implemented in terms of the two constructs find and lazyFind, as well as conversions between concrete values and logical

variables (denoted by l). Section 5.4 describes how to use the host language (Scala) to reduce other constructs (including solve, findAll, lazyFindAll) to this core.

A state consists of a triple $\mathsf{expr}|\langle\mu,\kappa\rangle$, where expr is the expression under evaluation, μ encodes the part of the state that is proper to Scala, and κ is a *constraint store*. A constraint store is conceptually a formula whose free variables correspond to all logical variables used since the beginning of the computation.

The HOST rule captures the intuition that in the absence of invocations to the constraint solver, Kaplan behaves exactly like Scala: we assume the existence of a transition relation $\rightarrow_H$ describing the execution of normal Scala code, and which is lifted to Kaplan through the HOST rule. As one would expect, applying such transitions leaves the constraint store unchanged.

The rules S-SAT and S-UNSAT describe the possible results of an invocation of find, the simplest form of constraint solving. We use $(\lambda\bar{x}.\phi(\bar{x}))$ to denote a constraint that ranges over the variables $\bar{x}$ and that does not refer to logical variables. By $\mathcal{M}$ we denote a map from variables to constants. The condition $\mathcal{M} \models \phi$ denotes that $\mathcal{M}$ is a valid model of ϕ, i.e., a mapping of variables $\bar{x}$ of ϕ to constants such that $\phi[\bar{x} \mapsto \mathcal{M}(\bar{x})]$ holds. Examining the rules for find, S-SAT and S-UNSAT, we note that find has no impact and no dependency on the constraint store, which is consistent with its eager semantics.

The rules L-SAT and L-UNSAT for lazyFind are analogous, but with crucial differences: 1) logical variables can be present in the constraints to solve, which we therefore denote $(\lambda\bar{x}.\phi(\bar{x},\bar{l}))$, and 2) the L-SAT rule does not produce concrete values, but rather fresh logical variables.

The LIFT rule applies whenever a constant value needs to be used in place of a logical variable. Conceptually, it adds to the constraint store a new logical variable whose value is immediately constrained to be the constant. Finally, the VALUE rule specifies how concrete values can be extracted from a satisfying assignment (model) for the constraint store. Observe that when the rule applies, the value of the logical variable is then fixed for the rest of the program execution by the added equality to κ. This ensures singular semantics [RRH10]: in any execution trace, all applications of the VALUE rule for a given logical variable l will produce the same value; a different value would contradict the premise that $\mathcal{M}$ is a valid model for the store.

We now show that execution never gets stuck in one of the Kaplan-specific rules. Because we are examining the behavior of our constructs taking a constraint solver as a parameter, we deliberately ignore termination properties of the constraint solver and assume that the outcome of the constraint solver call, denoted $\models$, becomes immediately available to test the applicability of a rule.

Theorem 5.1. *Suppose the constraint store is satisfiable in the initial state and that the $\rightarrow_H$ relation is total. Then the transition relation given by Figure 5.1 is also total.*

Proof. The case of HOST is clear from the hypothesis that $\rightarrow_H$ is total. For evaluations of find, we observe that the rules S-SAT and S-UNSAT have complementary premises, and therefore

one of them necessarily applies. Similarly for lazyFind, L-SAT and L-UNSAT are complementary, regardless of the value of κ. LIFT has no precondition, so it remains to show that VALUE cannot get stuck, i.e. that it cannot be the case that there is no model $\mathscr{M}$ for the constraint store κ.

Using the assumption that in the initial state the constraint store is satisfiable, it is sufficient to show that no transition will make it unsatisfiable. Observe that only three rules affect the constraint store; VALUE, LIFT, and L-SAT. The addition of the equality $(l_f = c)$ in LIFT clearly does not affect satisfiability, because l_f is fresh. Similarly, the addition of $(l = \mathscr{M}(l))$ in VALUE preserves the model $\mathscr{M}$ by definition. Finally, L-SAT is guarded by a satisfiability check on precisely the next state of the store, so the model obtained in the premise is a valid model for the next state. □

The proof suggests an implementation strategy: preserve at all times, alongside the constraint store, a satisfying assignment. When L-SAT is applied, cache the model $\mathscr{M}$ obtained from the satisfiability check. When LIFT applies, augment the cached model with the appropriate value for l_f. Finally, to apply VALUE, use $\mathscr{M}$ from the cache.

It is not hard to see that this strategy is a valid refinement of the presented rules, and it is, in fact, the strategy we have implemented in Kaplan, as explained in the next section.

5.4 Implementation

We implemented our extension to Scala as a combination of a run-time library and a compiler plugin, both implemented in Scala. In this section, we present the implementation aspect of these parts, along with the interaction with the underlying solver Leon.

5.4.1 Run-Time Library

First-class constraints. In Kaplan, first-class constraints are implemented as a hierarchy of Term classes, as shown in Figure 5.2. The base Term class represents a lambda expression and is parameterized by its argument types and return type. A constraint is simply a Term instance where the return type is instantiated as boolean. We define subclasses of Term for each arity, generating them automatically, as it is the case for the tuple and function definitions in the Scala library. The base class defines methods common to terms of all arities, such as the solve, find and findAll methods for querying constraints. We ensure that these methods are only applicable when the return type is boolean by constraining it using an implicit parameter asBool. We use the same technique to guarantee that term instances are combined in a fully type safe way. We use the c method to trigger an implicit conversion from lambda expressions to Term instances.

Each term subclass extends the corresponding function class in Scala, and uses the original Scala code for the lambda expression to define function application. These classes define

optimization methods (minimizing and maximizing) to obtain optimization constraints. The implementation of the optimization procedures is discussed later in this section.

The creation of terms from Scala lambda expressions relies on compile-time transformations. The transformations are implemented in a plugin for the official Scala compiler. We describe how the compile-time transformations work in Section 5.4.2.

Logical variables. In Kaplan, logical variables are instances of the L class, which is parameterized by the type of the symbolic value that it encapsulates. We define LIterator classes that extend the Iterator trait of Scala and that enumerate tuples of L values. We discuss the implementation of logical variables in Section 5.4.3.

The assuming-otherwise construct. We can define the assuming-otherwise construct naturally at the library level in Kaplan: it boils down to checking the satisfiability of a constraint of arity 0, and can therefore be implemented in terms of lazyFind. Figure 5.3 shows the code for this construct in Kaplan. The assuming block creates an Assuming instance. We rely on the implicit conversion mechanism of Scala for implementing the construct: if the optional **otherwise** block is not defined, the type checking phase will insert a call to assuming2value, which will trigger a conversion from the Assuming instance to the value it encapsulates.

There exists a difference in the treatment of the optional second part of the built-in if-then-else construct and our library extension for assuming-otherwise: without an optional **else** block, an **if**(...){ ... } block is always type-checked as Unit. This is built in in the Scala compiler. We cannot achieve the same effect with assuming-otherwise without making deep changes to the compiler, so in our case, an **assuming**(...){ ... } block will throw an exception at run-time if the following three conditions occur: 1) the **assuming** test fails, 2) no **otherwise** block is defined, and 3) an expression of a type different from Unit is required.

5.4.2 Scala Compiler Plugin

The compile-time transformations of Kaplan programs include the following:

1. extracting user-defined specification functions and algebraic data types;
2. generating methods to allow conversion between values of these data types and their representation in our solver Leon;
3. transforming implicit calls to conversion methods in order to instantiate Term instances.

These transformations are implemented as a compiler plugin that constitutes a compiler phase that follows the type checking phase. Functions and classes that carry the @spec annotation are expected to be in the PureScala constraint sublanguage, presented in Section 3.3. Alternatively, developers can group these specification functions and data types in annotated Scala objects, instead of annotating each of them. These specifications are extracted during compilation to

```
abstract class Term[T,R] { self ⇒
  def find(implicit isBool: R =:= Boolean): Option[T] = ...
  def solve(implicit isBool: R =:= Boolean): T =
    this.find.getOrElse(throw new UnsatException)
  def findAll(implicit isBool: R =:= Boolean) : Iterator[T] = ...
  def c(implicit isBool: R =:= Boolean): self.type = this
  ...
}

class Term0[R] extends Term[Unit,R] with Function0[R] {
  ...
}
class Term1[T1,R] extends Term[T1,R] with Function1[T1,R] {
  ...
}
class Term2[T1,T2,R] extends Term[(T1,T2),R]
    with Function2[T1,T2,R] {
  def ||(other: Term2[T1,T2,Boolean])
    (implicit isBool: R =:= Boolean): Term2[T1,T2,Boolean] = ...
  def &&(other: Term2[T1,T2,Boolean])
    (implicit isBool: R =:= Boolean): Term2[T1,T2,Boolean] = ...
  def unary_!(implicit isBool: R =:= Boolean):
    Term2[T1,T2,Boolean] = ...

  def compose0[A1](other: Term1[A1,T1]): Term2[A1,T2,R] = ...
  def compose1[A1](other: Term1[A1,T2]): Term2[T1,A1,R] = ...
  def compose0[A1,A2](other: Term2[A1,A2,T1]):
    Term3[A1,A2,T2,R] = ...
  def compose1[A1,A2](other: Term2[A1,A2,T2]):
    Term3[T1,A1,A2,R] = ...
  ...
  def product1[A1](other: Term1[A1,Boolean])
    (implicit isBool: R =:= Boolean):
      Term3[T1,T2,A1,Boolean] = ...
  def product2[A1,A2](other: Term2[A1,A2,Boolean])
    (implicit isBool: R =:= Boolean):
      Term4[T1,T2,A1,A2,Boolean] = ...

  def minimizing(objective: Term2[T1,T2,Int])
    (implicit asBool : R =:= Boolean):
      MinConstraint2[T1,T2] = ...
  ...
}
...
type Constraint[T] = Term[T,Boolean]
type Constraint0 = Term0[Boolean]
type Constraint1[T1] = Term1[T1,Boolean]
...
```

Figure 5.2 – Term class hierarchy.

```
def assuming[A](cond: Constraint0)(block: ⇒ A): Assuming[A] = {
  val v: Option[A] = cond.lazyFind match {
    case Some(_) ⇒ Some(block)
    case None ⇒ None
  }
  new Assuming(v)
}

final class Assuming[A](val thenResult: Option[A]) {
  def otherwise(elseBlock: ⇒ A): A = thenResult match {
    case None ⇒ elseBlock
    case Some(tr) ⇒ tr
  }
}

implicit def assuming2value[A](a: Assuming[A]): A = {
  a.thenResult match {
    case Some(tr) ⇒ tr
    case None ⇒
      throw new Exception("otherwise block not defined")
  }
}
```

Figure 5.3 – Implementation of **assuming**–**otherwise** in terms of lazyFind.

obtain a representation that we use in solving. In addition, method definitions are generated and inserted into the code in order to convert between these types and their representation. [2]

We rely on the Scala compiler to signal to us the locations where a function literal needs to be lifted to a constraint literal. The type checking phase, which runs before ours, does so by surrounding such function literals by a call to an implicit function2term conversion function. These functions are defined in the Kaplan library but have no implementation: they simply serve as a guide to indicate to the type checker that the conversion is legal. (The effect of compiling code written for Kaplan without the Kaplan plugin is thus that all constraint manipulation operations result in a run-time NotImplemented exception.)

5.4.3 Implementation of the Core Solving Algorithms

Our implementation leverages the SMT solver Z3 that we use extensively through Leon. In the following we refer simply to Z3, as the algorithms we cover in this section would remain the same if we used Z3 alone; what we gain by using Leon is the additional expressive power of recursive functions within constraints. We now describe the interactions with the solver that allow us to put into practice features such as enumeration, minimization, and logical variables.

2. We note that many of these operations could now be implemented in a simpler way using the macro language introduced in Scala 2.10.

Solution enumeration. find and lazyFind are used to implement findAll and lazyFindAll, respectively, through the use of an iterator. The iterator maintains a constraint at all times, starting with the original one. Each time a new solution is required, the iterator destructively updates the constraint by adding to it the negation of the previous solution, thus ensuring that all following solutions will be different. To make this process efficient, Kaplan relies on the incremental reasoning capabilities of Z3 (and thus Leon) to avoid solving the entire constraint each time. The implementation of lazyFindAll is conceptually identical, with the difference that it returns logical variables instead of concrete values. These logical variables are constrained in the store to be all-different. For an enumeration using lazyFindAll to terminate, Leon must therefore prove that the constraint has finitely many solutions.

Optimization constraints. Our procedure for optimizing a constraint with respect to an objective function can be seen as a generalization of binary search over the range of values that the objective can take. Let us consider the case of minimization (the maximization procedure is analogous). The pseudo-code for the algorithm can be seen in Figure 5.4.[3] It starts by attempting to find a satisfying assignment for the constraint. It then repeatedly looks for a model in which the objective is smaller than the last satisfying value, by exponentially increasing the difference until a lower bound is found. It further reduces the interval until the optimal value for the objective is found. The procedure maintains the invariants that: 1) lo is always less than any satisfying assignment to t_m; and 2) there always exists a satisfying assignment to t_m which is less than hi.

Ordered enumeration. Having defined the solving procedure that minimizes a given term, we can now compose it with solution enumeration to obtain ordered enumeration. We present in Figure 5.5, a recursive algorithm that will enumerate solutions to ϕ, ordered by the value of t_m, which should be minimized.

In this pseudo-code, we use findAll to get an iterator of all values satisfying the given predicate, and ++ to concatenate iterators.

Handling logical variables. We use a global context to keep track of the constraints associated with logical variables. Given a logical variable l and the constraint c_l associated with it, we create a guard (a boolean literal) g_l, denoting the *liveness* of the logical variable, i.e. whether its value has not been fixed yet. Throughout the execution, we maintain in the global context the set of guards that are still alive. Upon creation of the variable l, we add g_l to the set of alive variables and we assert c_l, guarded by the disjunction of all the guards in the set G, defined as

3. We use pseudo-code for clarity, but it is not hard to see that it can be implemented using only standard Scala along with find or lazyFind.

```
def solveMinimizing(φ, t_m) {
  solve(φ) match {
    case ("SAT", m) ⇒
      model = m
      v = modelValue(m, t_m)
      pivot = v−1
      lo = null
      hi = v+1
      while (lo == null ∨ hi−lo>2) {
        solve(φ ∧ t_m ≤ pivot) match {
          case ("SAT", m) ⇒
            model = m
            if (lo == null) {
              pivot = pivot≥0 ? −1 : pivot×2
              hi = pivot + 1
            } else {
              l_v = modelValue(m, t_m)
              pivot = l_v + (pivot+1 − lv)/2
              hi = pivot+1
            }
          case ("UNSAT", _) ⇒
            pivot = pivot+(hi−pivot)/2
            lo = pivot
        }
      }
      return ("SAT", model)
    case ("UNSAT", _) ⇒ return ("UNSAT", null)
  }
}
```

Figure 5.4 – Pseudo-code of the solving algorithm with minimization. We invoke our base satisfiability procedure via calls to solve.

```
def orderedEnum(φ, t_m) {
  solveMinimizing(φ) match {
    case ("SAT", m) ⇒
      v_m = modelValue(m, t_m)
      findAll(φ ∧ t_m = v_m) ++ orderedEnum(φ ∧ t_m > v_m, t_m)
    case ("UNSAT", _) ⇒ return Iterator.empty
  }
}
```

Figure 5.5 – Pseudo-code of the ordered enumeration algorithm.

the set containing g_l and the guards associated with all the logical variables that appear in c_l:

$$\bigvee_{g \in G} g \Rightarrow c_l$$

The reason for considering the guards associated with the other logical variables is to ensure the single value semantics of these. Consider the following simple example that illustrates the situation:

```
for (x ← ((x: Int) ⇒ x ≥ 0 && x < 4).lazyFindAll) {
  val y = ((y : Int) ⇒ y == x).lazyFind
  assuming (x ≥ 2) {
    ... // first block
  }
  assuming (y < 2) {
    ... // second block
  }
}
```

In each iteration of the outer for-comprehension, if the first block is entered, we do not want to enter the second one, as this would violate the single value semantics for the variable x. Since the constraint y == x will be enforced as long as either x or y has not been fixed, the conflicting situation is correctly avoided.

In the terminology of Section 5.3, g_l denotes whether the VALUE rule has been used for l (true means it has not), c_l is the constraint on which L-SAT was applied to introduce l into κ (as part of $\bar{l}_f$) and the logical variables represented by the set of guards G are those that contributed to the constraint (the $\bar{l}$ variables in L-SAT). Note that using individual guards for logical variables is not required by the semantics, but rather is an optimization that allows us to reduce the size of the constraint store when some values become known. When the value v_l of the variable l is set, we remove g_l from the set of alive guards, and we assert the following:

$$\neg g_l \wedge l = v_l$$

When all of the literals guarding a constraint c_l are removed from the alive set, the formula $\bigvee_{g \in G} g \Rightarrow c_l$ that was asserted becomes trivially true, and the constraint can be discarded by Z3. Another source of optimization is that we override the finalize method of the L instances such that, even if their value is never fixed, their guard is removed from the set of alive variables when they are being considered for garbage collection by the JVM. This is again not strictly speaking required by the semantics, but helps reducing the overhead of tracking all logical variables.

Invocations of the solver. We summarize this section by presenting a list of all the places where an invocation of the solver occurs:

example	time (s)
first examples	0.16
unimodular matrices	0.76
sat solving	0.05
knapsack	0.18
prime numbers	0.80
all red-black trees up to 7 nodes	27.45
send+more=money	1.17

Figure 5.6 – Evaluation results for the examples presented in Section 5.2.

- calls to find, as they search for a solution eagerly,
- calls to lazyFind, as they check whether a satisfying assignment exists for logical variables,
- calls to the hasNext method of the iterators returned by calls to findAll and lazyFindAll, which are then translated into calls to find and lazyFind respectively, and
- evaluation of the constraint of assuming blocks, as they indirectly invoke the solver by invoking lazyFind on the constraint.

Note that, as the proof in Section 5.3 suggested, calls to the value method of logical variables do not trigger a solver invocation.

5.5 Advanced Usage Scenarios and Evaluation

In this section, we present an experimental evaluation of our constraint programming system by considering a number of examples.

As a first overview of the performance of our implementation, we present the running times for the examples that were introduced in Section 5.2. The results of our evaluation can be seen in Figure 5.6. We observe that most problems are solved almost instantly, with the exception of the red-black tree enumeration. We discuss the difficulty of enumerating data structures satisfying an invariant in the subsequent examples.

5.5.1 Enumerating Data Structures for Testing

One use of the findAll construct consists in enumerating data structures that satisfy given invariants. This is a problem that has been studied previously, and was motivated by [BKM02]. Subsequent work [GGJ$^+$10] presents a Java-based language with non-deterministic choice operators that can be used for enumerating linked data structures.

We describe our experience in using Kaplan to enumerate functional data structures to find input values that violate function contracts. We consider a functional specification of red-black trees. We enumerate solutions to function preconditions and check whether the postconditions hold. As in [BKM02, GGJ$^+$10], we enumerate the data structures while bounding the range of values than can be stored in nodes. A red-black tree is a binary search tree character-

ized by the following additional properties: 1) each node is either red or black; 2) all leaves are black; 3) both children of every red node are black; and 4) every simple path from the root to any leaf contains the same number of black nodes.

The first method we consider is balance, which defines one of the cases erroneously to duplicate one of the subtrees and forgetting another while rebalancing the tree. This results in violating the post-condition of the add method as the result tree does not have the expected content. In this case, an example violating the post-condition is found after enumerating twelve trees. Our test harness consists of enumerating trees that satisfy the precondition of add and calling the method in the body of the loop.

We then consider the case where the add method has a missing precondition, namely that the tree must be black-balanced. In this case, the precondition to a method that is called within add fails, and we find a bug-producing value using a similar harness after enumerating four trees, in 0.187 seconds. We argue that a random test-case generation approach would be insufficient in enumerating such data structures that satisfy complex invariants. While the results using constraint solving is not as fast as the specialized solving in UDITA [GGJ+10], we should keep in mind that this is an experiment in using a general-purpose constraint solving engine. The generality of Kaplan is in contrast to previously proposed solutions for data structure enumeration, which rely on specialized techniques for linked heap data structures, or even techniques specialized to a particular data structure [BHR+00]. In the light of the generality of our technique and the overall difficulty of the problem, we consider these to be good results.

5.5.2 Executable Specifications

As another application of implicit computation, we now explore examples that consist of the “execution” of declarative specifications instead of explicit computation. Execution of specifications is the approach taken in Plan B [SAM10], in which specifications are used as a fallback mechanism upon contract violations.

Consider, for instance, a function that computes the last element of a given list. We can define this function declaratively, by stating that the input list is equal to some list concatenated with the list that has only the element that we are looking for:

```
def last(list : List) : Int = {
  val (elem, _) = ((e: Int, zs: List) ⇒ concat(zs, Cons(e, Nil())) == list).solve
  elem
}
```

As a more elaborate example, consider adding an element to (or removing an element from) a red-black tree. The explicit insertion and removal have to consider multiple cases in order to keep the invariants that red-black trees should satisfy, and are known to be tricky to implement. On the other hand, these methods can be stated succinctly in a declarative manner, using

list size	20	40	60	80	100
time (s)	0.24	0.45	0.56	0.72	0.98

Figure 5.7 – Evaluation results for declarative last method.

size	list add	list remove	RBT add	RBT remove
0	0.07	0.02	0.03	0.02
1	0.08	0.02	0.10	0.05
2	0.12	0.05	0.14	0.09
3	0.16	0.10	0.55	0.41
4	0.24	0.18	0.66	0.76
5	0.39	0.38	1.07	0.91
6	0.55	0.45	1.51	1.56
7	0.97	0.67	9.32	13.09
8	1.48	1.09	11.13	18.80
9	2.27	1.80	24.49	25.79
10	3.32	2.22	11.51	20.55

Figure 5.8 – Evaluation results for declarative add and remove, for red-black trees and for sorted lists. “size” is the size of the structure without the element. All times are in seconds.

functions that check if a given tree is indeed a red-black tree, along with functions computing the content of the tree as a set:

```
def addDeclarative(x: Int, tree: Tree) : Tree =
  ((t: Tree) ⇒ isRedBlackTree(t) && content(t) == content(tree) ++ Set(x)).solve
def removeDeclarative(x: Int, tree: Tree) : Tree =
  ((t: Tree) ⇒ isRedBlackTree(t) && content(t) == content(tree) −− Set(x)).solve
```

The performance of the above methods is presented in Figure 5.8. We also show the results for the similar case of inserting into and removing from a sorted list. Note that we encounter essentially the same running times for the problem of declaratively sorting a list, by asking for a sorted list of the same content as a given list.

The above examples show that it is possible to replace the explicit computation for a method by its purely declarative specification, even for sophisticated contracts such as the ones on red-black trees. The declarative variants are notably slower than the imperative implementations, but it is likely preferable to rely on these executable specifications instead of simply crashing when the imperative version violates the contract.

As a final example, let us consider the implicit computation that gives the integer square root $\lfloor\sqrt{i}\rfloor$ of a positive integer i. This is concisely stated as following:

```
def sqrt(i : Int) : Int =
  ((res: Int) ⇒ res > 0 && res ∗ res ≤ i && (res + 1) ∗ (res + 1) > i).solve
```

Our implementation can handle numbers as large as hundreds of thousands, performing

under 0.3 seconds in all cases.

5.5.3 Counter-example Guided Inductive Synthesis

Counter-example guided inductive synthesis (CEGIS) is a method for effectively solving $\exists\forall$ constraints using SMT solvers. Such constraints are particularly important for program synthesis, identified as the class of $\forall\exists$ synthesis problems in [PR88]. The counter-example guided approach for this problem was successfully applied to software synthesis in the algorithm for combinatorial sketching using SAT solvers [SLTB$^+$06]. This technique was later applied with an SMT solver as the satisfiability engine to synthesize loop-free bit-vector code fragments [GJTV11].

The technique starts by choosing some initial set of values for the universally quantified variables, then solving the constraint for the existentially quantified variables. If the values for the existentially quantified variables work for all values of the universally quantified variables, a solution has been found. Otherwise, there is a counterexample which is some valuation of the universally quantified variables. This counterexample is added to the set of values for universally quantified variables and the procedure is repeated until a solution is found.

This synthesis loop can be expressed in our system succinctly, using first-class constraints. As an example we consider the following: *Are there integers a and b such that, for every integer x, $a \cdot (x-1) < b \cdot x$?* In Kaplan we can describe the CEGIS approach as in Figure 5.9. When executed, the program finds correct values for a and b in two iterations of the loop. The output of the program is:

```
Initial x: 0
candidate parameters a = 1, b = 0
counterexample for x: 1
candidate parameters a = 1, b = 1
proved!
```

The example generalizes to arbitrary constraints (which, in this example, are cnstrGivenX and cnstrGivenParams), keeping the same simple structure. Note that we here explicitly constructed increasingly stronger first-class constraints; we can instead use logical variables and incrementally augment the constraint store.

5.5.4 Comparison to Other Systems

Providing a fair comparison of running times of Kaplan against competing systems is difficult because Kaplan covers many areas for which specialized tools have been developed. We do not expect to match performance of each of these specialized systems. At the same time, there is no single system that subsumes Kaplan. This section illustrates how other tools can solve some

```
var continue = true
val initialX = ((x: Int) ⇒ true).solve
def cnstrGivenX(x0: Int): Constraint2[Int,Int] =
  ((a: Int, b: Int) ⇒ a * (x0 − 1) < b * x0)
def cnstrGivenParams(a0: Int, b0: Int): Constraint1[Int] =
  ((x: Int) ⇒ a0 * (x − 1) < b0 * x)
var currentCnstr = cnstrGivenX(initialX)
while (continue) {
  currentCnstr.find match {
    case Some((a, b)) ⇒ {
      println("candidate parameters a = " + a + ", b = " + b)
      (! cnstrGivenParams(a, b)).find match {
        case None ⇒
          println("proved!")
          continue = false
        case Some(ce) ⇒
          println("counterexample for x: " + ce)
          currentCnstr = currentCnstr && cnstrGivenX(ce)
      }
    }
    case None ⇒
      println("cannot prove property!")
      continue = false
  }
}
```

Figure 5.9 – Counter-example guided inductive synthesis in Kaplan.

list size	time in Kaplan (s)	time in Curry (s)
10000	< 0.01	0.21
100000	< 0.01	2.41
1000000	< 0.01	23.88

Figure 5.10 – Evaluation results for functional last method.

of the problems we presented in this chapter and solved using Kaplan. The running times should not be taken as a definitive statement of the relative merits of the systems, but rather as a guide to understanding key differences of the underlying constraint solving techniques.

Last element of the list. We first compare the performance of Kaplan and Curry on computing the last element of a list, both in a declarative way as presented in Section 5.5.2, and as a tail-recursive functional method. In Curry, the declarative version can be expressed as:

```
last xs | concat ys [x] =:= xs
      = x where x,ys free
```

The performance of the Curry implementation surpasses the performance of Kaplan that we reported in Figure 5.7, running under 0.01 seconds in lists of size up to 100. However, in

Kaplan we also have the freedom of writing the constraint directly as a functional program. In such style, Curry performs at about the same speed as for a constraint-based definition, so Kaplan outperforms Curry on longer lists. Figure 5.10 shows the evaluation results for this case. It is of course the Scala compiler and the Java Virtual Machine that take credit for the good performance of the functional implementation; to the credit of Kaplan is simply that it does not lose any of this underlying efficiency. This basic consequence of Kaplan design is very important from a practical point of view. In principle, a static analysis could be used to recognize among logical constraints special classes that do not require constraint solving (consider, for example, mode analysis of Mercury [OSS02]). However, this approach clearly requires significant compilation effort to merely recover the baseline performance of a functional or imperative code, whereas in Kaplan it follows by construction.

Generating data structures with complex invariants. We now report on our experience in comparing Kaplan with ScalaCheck [Nil12] for generating red-black trees (results were similar for sorted lists). ScalaCheck is a tool for producing test cases using random test generation, similar to QuickCheck [CH00] for Haskell. We implemented basic generators for lists and trees using generator combinators in ScalaCheck. The problem with random generation is that, for many classes of properties, the probability of a random structure satisfying it may tend to zero as the structure size grows [BGK85], making test cases vacuous for larger structures. To see this in practice, consider first our own performance: Figure 5.6 shows that Kaplan takes 27.45 seconds in total to generate *all* red-black trees of size n containing elements 1 to n, for all n from 0 to 7. We therefore defined a basic ScalaCheck generator that, with equal probability, generates an empty or a non-empty tree. We ran it for 29.1 seconds, generating 200'000 trees. Although 62% of all of them were red-black, that is because 104'544 were, in fact, empty. Of the rest, 18'191 had size one, 934 size two, and there were no red-black trees of larger size. In this experiment, we gave both Kaplan and ScalaCheck only the constraints, without any additional insight, which makes the comparison fair. Experienced users could write better ScalaCheck generators, but they could similarly write better Kaplan checkers. Like UDITA [GGJ+10], Kaplan supports a full spectrum of intermediate approaches, where one puts more attention into either writing better generators, to increase the ratio of valid testcases, or into manually decomposing the implicit computation to reduce the search space.

Constraint satisfaction problems. Tools for constraint solving and optimization over finite domains have developed somewhat independently of SMT and the related technologies [NSB+07, AW07, GJM06, Tac09]. Our anecdotal experience suggests that, for problems of moderate size over finite domains, the two technologies yield comparable results. A full experimental comparison between the state of the art of these two communities is beyond the scope of our work. Note that we could incorporate into Kaplan solvers from either class, as long as they support our domains of interest (including, for example, algebraic data types).

5.6 Related Work

We conclude this chapter by surveying related work. Kaplan itself was first presented in [KKS12].

Existing languages. Functional logic programming [AH10] amalgamates the functional programming and logic programming paradigms into a single language. Functional logic languages, such as Curry, benefit from efficient demand-driven term reduction strategies proper to functional languages, as well as non-deterministic operations of logic languages, by using a technique called *narrowing*, a combination of term reduction and variable instantiation. Instantiation of unbound logic variables occur in constructive guessing steps, only to sustain computation when a reduction needs their values. The performance of non-deterministic computations depends on the evaluation strategy, which are formalized using definitional trees [Ant92]. Applications using functional logic languages include programming of graphical and web user interfaces [HK09, Han06] as well as providing high-level APIs for accessing and manipulating databases [BHM08]. Our work can be seen as the practical experiment of building from existing components a system close in functionality to Curry: we proceeded by extending an existing language while preserving its execution model, rather than starting from scratch. As such, we lose the luxury of a total integration of paradigms; for instance, users of Kaplan need to specify which functions can handle logical variables. On the other hand, such a separation of features comes with benefits: we can readily use the full power of Z3 and the Leon verification system for constraint solving, and the execution efficiency of the Java virtual machine for regular code. Implementations of functional logic languages, on the other hand, must typically focus on efficiently executing either the logical or the functional components of the code.

The Oz language and the associated Mozart Programming System is another admirable combination of multiple paradigms [vR99], with applications in functional, concurrent, and logic programming. In particular, Oz supports a form of logical variables, and logic programming is enabled through unification. One limitation is that one cannot perform arithmetic operations with logical variables (which we have demonstrated in several of our examples), because unification only applies to constructor terms.

Some of the earlier efforts to integrate constraint and imperative programming resulted in the languages Kaleidoscope [LFBB94], Alma-0 [ABPS98] and Turtle [GH03]. This line of research put emphasis on designing novel ways to define constraints, ideally to resemble imperative-style programming. Kaleidoscope, for instance, promotes the integration of constraints with objects; programmers can define instance constraints that relate various members of an object, and constraints can be reassigned, just like mutable variables. Constraints are also associated to a *duration* (*once* or *always*), specifying intuitively the scope in which they can affect variables. Kaplan currently does not support constraints over mutable data types and could benefit from the integration of such features.

The integration of constraints as first-class members of the language is tighter in Kaplan than in any of the languages discussed above. For example, in Curry programmers can define constraints as functions that return the special type Success (which is *not* identical to Boolean). However, such functions (or rather predicates) in Curry can only be built from equality predicates and cannot be combined freely: while conjunction and disjunction are valid combinators, negation, for instance, is not. In Kaplan, on the other hand, any predicate expressed in PureScala can be used as a constraint, and terms of other types can also be manipulated and composed freely.

Another distinguishing feature of Kaplan is the native support for many theories (integers, sets, maps, data types) that comes from using Leon and Z3 as the underlying constraint solver.

Language design. Monadic constraint programming [SSW09] integrates constraint programming into purely functional languages by using monads to define solvers. The authors define monadic search trees, corresponding to a base search, that can be transformed by the use of *search transformers* in a composable fashion to increase performance. Our system differs from this work in its use of SMT solvers for search, and a more flexible way of mixing constraints with imperative programming.

The work on uniform reduction to bit-vector arithmetic [MJ10] (URBiVA) proposes a C-like language for specifying constraints. It uses symbolic execution to encode problems into bit-vector arithmetic and invokes one of the underlying bit-vector and SAT solvers. The system allows for the comparison of different solvers on examples involving solution enumeration and functional equivalence checking. The use of symbolic values in conditional statements and array indexing is not permitted. Because it uses native enumeration capability of solvers such as CLASP, it can be faster than our system on some of the benchmarks. URBiVA does not support unbounded domains; we are aware of no techniques to enumerate solutions for unbounded domains more efficiently than in Kaplan.

SMT as a programming platform. The Dminor language [BGHL10] introduces the idea of using an SMT solver to check subtyping relations between refinement types; in Dminor, all types are defined as logical predicates, and subtyping thus consists of proving an implication between two such predicates. The authors show that an impressive number of common types (including for instance algebraic data types) can be encoded using this formalism. In this context, generating values satisfying a predicate is framed as the *type inhabitation* problem, and the authors introduce the expression elementof T to that end. It is evaluated by invoking Z3 at run-time and is thus conceptually comparable to our find construct but without support for recursive function unfolding. We have previously found that recursive function unfolding works better as a mechanism for satisfiability checking than using quantified axiomatization of recursive functions. In general, we believe that our examples are substantially more complex than the experiences with elementof in the context of Dminor.

The Jeeves programming language [YYSL12] is another recent example of integration of SMT as an executable component. Jeeves proposes to let users define privacy policies declaratively, while enforcing them at run-time through the appropriate constraints. To this end, Jeeves relies on symbolic variables, similar in design to the logical variables of Kaplan.

The Scala^{Z3} library [KKS11] is our earlier effort to integrate invocations to Z3 into a programming language. Because it is implemented purely as a library, we were then not able to integrate user-defined recursive functions and data types into constraints, so the main application is to provide an embedded domain-specific language to access the constraint language of Z3 (but not to extend it).

Applications of declarative programming. One approach in using specifications for software reliability is data structure repair [DR03, EK08], where the goal is to recover from corrupted data structures by transforming states that are erroneous with respect to integrity constraints into valid ones, performing local heuristic search. [ZK10] uses method contracts instead of data structure integrity constraints to be able to support rich behavioral specifications. While the primary goal is to perform run-time recovery of data structures, recent work [MSK11] extends the technique for debugging purposes, by abstracting concrete repair actions to program statements performing the same actions. Data structure repair differs from our system in that it can perform local search to modify existing states, while we do not currently do so. We do not expect that a general-purpose constraint solving such as ours can immediately compare with these dedicated techniques.

The idea to use specifications as a fall-back mechanism, as in one of our application examples, was adopted in [SAM10]. Similarly to our setting, dynamic contract checking is applied and, upon violations, specifications can be *executed*. The technique ignores the erroneous state and computes output values for methods given concrete input values and the method contract. The implementation uses a relational logic similar to Alloy [Jac02] for specifications, and deploys the Kodkod model finder [TJ07]. A related tight integration between Java and the Kodkod engine is presented in [MRYJ11]. In both cases, due to the finite bound on the search space, a satisfying answer may not always be found, which makes the techniques incomplete. We have shown that executing declarative specifications is possible in our setting. We expect that a constraint solver such as the one deployed in Leon will ultimately be better suited than an approach based on KodKod, due to the presence of unbounded or large data types such as integers and recursive structures.

5.7 Conclusion

We presented Kaplan, an extension to the multi-paradigm Scala language that integrates constraint programming while preserving the existing functional and object-oriented features of the host language. The behavior and performance of Kaplan is identical to Scala in the absence of declarative constraint solving. Kaplan integrates constraints as first-class objects

in Scala, and logical variables for solving constraints lazily. It allows for solving optimization problems, as well as enumerating solutions in a user-specified order. Kaplan is implemented as a combination of a compiler plugin and a run-time library, and allows users to express constraints in a powerful and expressive logic. Kaplan relies on Leon to solve these constraints.

We evaluated our system by considering applications such as solution enumeration, execution of declarative specifications, test-case generation for bug-finding and counterexample guided inductive synthesis, as well as numerous smaller, yet expressive, snippets. We observed that our model for introducing non-determinism using for-comprehensions integrates well with Scala.

Based on our experience with the Z3 SMT solver and the Leon verification system in constraint programming, we found that a number of features, if natively supported by solvers, could directly bring benefits to constraint programming. These include 1) support for enumeration of theory models and 2) solving constraints while minimizing/maximizing a given term. Overall, we believe there is great potential in extending standard programming languages with constraint solving capabilities. Many interesting problems are left open, both in language design and in constraint solving; a system such as Kaplan that integrates state-of-the-art tools from both domains is likely to benefit from progress made in each of them.

6 Compiling Specifications

In the previous chapter, we showed how specifications can be used directly as a programming construct, thus bypassing the tedious and error-prone task of translating them into imperative or functional code. While we have demonstrated that some programming tasks can thus be expressed very concisely, we have also noted that the main barrier to wider adoption remains performance. In this chapter, we look at techniques to automatically transform, at *compile*-time, specifications into executable code. This approach promises to combine the advantages of declarative specifications with the execution efficiency of standard programming paradigms.

Our starting point are decision procedures, which we propose to systematically generalize into *synthesis procedures*. Synthesis procedures are predictable, because they are guaranteed to find code that satisfies the specification whenever such code exists. To illustrate our method, we derive synthesis procedures by extending quantifier elimination algorithms for integer arithmetic, term algebras, and set data structures. As a result, any relation expressed in (a combination of) these logics can be translated into executable code. We show that an implementation of such synthesis procedures can extend a compiler to support implicit value definitions and advanced pattern matching.

6.1 Introduction

Synthesis of software from specifications [MW71, Gre69] promises to make declarative programming more efficient. Despite substantial recent progress in techniques that generate short instruction sequences [JNZ06] and program fragments [SLTB$^+$06, SGF10], synthesis is limited to small pieces of code. We expect that this will continue to be the case for some time in the future: synthesis is algorithmically a difficult problem and requires detailed specifications, which for large programs become difficult to write.

We therefore expect that practical applications of synthesis (and more generally declarative programming) lie in its integration into the compilers of general-purpose programming lan-

guages. To make this integration feasible, we aim to identify well-defined classes of expressions and synthesis algorithms for given classes of expressions. Our starting point for such synthesis algorithms are *decision procedures*.

We have already seen in Chapter 5 how to deploy decision procedures at run-time to solve declarative tasks. Our goal in this chapter is to provide similar benefits in a more controller 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 than a generic decision procedure. Another benefit of examining specifications at compile time is that we can provide static feedback to developers by checking the conditions under which the specifications have solutions and whether they are unique. We can thus provide a form of static debugging for specifications.

The input to a synthesis procedure is only the desired constraint, and not necessarily any bounds on values or on the structure of the synthesized code as in sketching [SLTB+06] and resource-bounded synthesis [SGF10]. This makes a synthesis procedure highly automated. It also means that its implementation cannot rely only on searching an obviously finite state space; it must instead use insights from the underlying decision procedure. The use of decision procedures for particular theories also differentiates our work from the earlier work based on first-order logic [Gre69, JNZ06].

We demonstrate our approach by outlining synthesis algorithms for three unbounded domains: linear arithmetic, term algebras, and QFBAPA. We have implemented and deployed synthesis as a tool called Comfusy, a compiler extension for Scala. In our experience, the synthesis times were acceptable and the running times were similar to equivalent hand-written code. Comfusy can thus bring the combined advantages of declarative specifications as in Kaplan, with the correctness and efficiency of manually written and verified code.

6.2 Examples

We first illustrate the use of the synthesis procedure for integer linear arithmetic as it is implemented in Comfusy. Recall, from Chapter 1, the declarative code to decompose a number of seconds into appropriate components:

```
def secondsToTime(total : Int) = ((h : Int, m : Int, s : Int) ⇒
  h ∗ 3600 + m ∗ 60 + s == total && m ≥ 0 && m < 60 && s ≥ 0 && s < 60).solve
)
```

Comfusy succeeds, because the specification is written in linear arithmetic. It reports no warning and generates the following code:

```
def secondsToTime(total : Int) = {
  val loc1 = total div 3600
  val num2 = total + ((−3600) ∗ loc1)
  val loc2 = min(num2 div 60, 59)
```

```
  val loc3 = total + ((−3600) ∗ loc1) + (−60 ∗ loc2)
  (loc1, loc2, loc3)
}
```

The absence of warnings guarantees that the solution always exists and that it is unique. Suppose that we replace in the specification the predicate s < 60 by s≤60. The synthesizer still succeeds, however it 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 s are not strict. The warning can indicate to a programmer that the declarative code is underspecified, leading potentially to undesired results. Playing with our example further, suppose that the developer imposes the constraint

```
val (hrs, mns, scs) = ((h: Int, m: Int, s: Int) ⇒
  h ∗ 3600 + m ∗ 60 + s == totsec &&
  0 ≤ h < 24 &&
  0 ≤ m && m < 60 &&
  0 ≤ s && s < 60).solve
```

Our system then 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.

In addition to solving constraints directly, programmers can use synthesis for more flexible pattern matching on integers. In existing deterministic programming languages, matching on integers either tests on constant types, or, in the case of Haskell's $(n+k)$ patterns, on some very special forms of patterns. Our approach supports a much richer set of patterns, as illustrated by the following fast exponentiation code that does case analysis on whether the argument is zero, even, or odd:

```
def pow(base : Int, p : Int) = {
  def fp(m : Int, b : Int, i : Int) = i match {
    case 0 ⇒ m
    case 2∗j ⇒ fp(m, b∗b, j)
    case 2∗j+1 ⇒ fp(m∗b, b∗b, j)
  }
  fp(1,base,p)
}
```

In this example, the system uses synthesis to, e.g., compute j from the constraint $i = 2 * j + 1$. The correctness of the function follows from the observation that $\mathsf{fp}(m,b,i){=}m \cdot b^i$, which we can prove by induction. For the case i = 2 * j+1 we observe:

$$\begin{aligned} \mathsf{fp}(m,b,i) &= \mathsf{fp}(m,b,2j+1) = \mathsf{fp}(mb,b^2,j) \\ \text{(by induct. hypothesis)} &= mb(b^2)^j = mb^{2j+1} = mb^i \end{aligned}$$

Note how the pattern matching on integer arithmetic expressions exposes the equations that make the inductive proof clearer. To support this construct, our compiler extension generates the code that selects the appropriate case and decomposes i into the appropriate new exponent j. Moreover, it checks that the pattern matching is exhaustive. The construct supports arbitrary expressions of linear integer arithmetic (it can prove, for example, that the set of patterns 2*k, 3*k, 6*k−1, 6*k+1 is exhaustive). Comfusy also accepts implicit definitions, such as

```
val 42 * x + 5 * y = z
```

The system ensures that the above definition matches every integer z (because 42 and 5 are mutually prime), and emits the code to compute x and y from z.

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) = ((a1:Set, a2:Set) ⇒
  a1 union a2 == s && a1 intersect a2 == empty &&
  a1.size == a2.size).solve
```

This specification is written in QFBAPA and can thus be handled by our synthesis procedure. For the code above, Comfusy 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) = ((a1:Set, a2:Set) ⇒
  a1 union a2 == s && a1 intersect a2 == empty &&
  a1.size − a2.size ≤ 1 &&
  a2.size − a1.size ≤ 1).solve
```

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 such that if there is one solution, then there are many solutions. Our synthesizer resolves these choices at compile time. The resulting generated code is therefore deterministic.

6.3 Synthesis using Relation Transformations

We start by introducing a general framework for synthesis based on the notion of relation transformation. We then use it to develop synthesis procedures for linear integer arithmetic, QFBAPA, and term algebras

A *synthesis problem* is a triple

$$[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$$

where $\bar{a}$ is a set of *input* variables, $\bar{x}$ is a set of *output* variables and ϕ is a formula whose free variables are a subset of $\bar{a} \cup \bar{x}$. A synthesis problem denotes a binary relation $\{(\bar{a}, \bar{x}) \mid \phi\}$ between inputs and outputs. The goal of synthesis is to transform such relations until they become executable programs. Programs correspond to formulas of the form $P \wedge (\bar{x} = \bar{T})$ where $\text{vars}(P) \cup \text{vars}(\bar{T}) \subseteq \bar{a}$. We denote programs

$$\langle P \mid \bar{T}\rangle$$

We call the formula P a *precondition* and call the term $\bar{T}$ a *program term*.

We use $\vdash$ to denote the transformation on synthesis problems, so

$$[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash [\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!] \tag{6.1}$$

means that the problem $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$ can be transformed into the problem $[\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!]$. The variables on the right-hand side are always the same as on the left-hand side. Our goal is to compute, given $\bar{a}$, one value of $\bar{x}$ that satisfies ϕ. We therefore define the soundness of (6.1) as a process that refines the binary relation given by ϕ into a smaller relation given by ϕ', without reducing its domain. Expressed in terms of formulas, the conditions become the following:

$$\begin{array}{rcll} \phi' & \models & \phi & \text{refinement} \\ \exists\bar{x}: \phi & \models & \exists\bar{x}: \phi' & \text{domain preservation} \end{array}$$

In other words, $\vdash$ denotes domain-preserving refinement of relations. Note that the dual entailment $\exists\bar{x}: \phi' \models \exists\bar{x}: \phi$ also holds, but it follows from *refinement*. Note as well that $\vdash$ is transitive.

Equivalences in the theory of interest immediately yield useful transformation rules: if ϕ and ϕ' are equivalent, (6.1) is sound. We can express fact as the following *inference rule*:

$$\frac{\models \phi_1 \iff \phi_2}{[\![\bar{a}\ \langle\phi_1\rangle\ \bar{x}]\!] \vdash [\![\bar{a}\ \langle\phi_2\rangle\ \bar{x}]\!]} \tag{6.2}$$

In most cases we will consider transformations whose result is a program:

$$[\![\bar{a} \ \langle\phi\rangle \ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle$$

The correctness of such transformations reduces to

$$\begin{array}{rcll} P & \models & \phi[\bar{x} \mapsto \bar{T}] & \text{refinement} \\ \exists\bar{x} : \phi & \models & P & \text{domain preservation} \end{array}$$

A *synthesis procedure* for a theory $\mathcal{T}$ is given by a set of inference rules and a strategy for applying them such that every formula in the theory is transformed into a program.

6.3.1 Theory-Independent Inference Rules

We next introduce inference rules for a logic with equality. These rules are generally useful and are not restricted to a particular theory.

EQUIVALENCE. From the transitivity of $\vdash$ and (6.2), we can derive a rule for synthesizing programs from equivalent predicates.

$$\frac{[\![\bar{a} \ \langle\phi_1\rangle \ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad \models \phi_1 \iff \phi_2}{[\![\bar{a} \ \langle\phi_2\rangle \ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle}$$

GROUND. In the case where no input variables are given, a synthesis problem is simply a satisfiability problem.

$$\frac{\mathcal{M} \models \phi}{[\![\varnothing \ \langle\phi\rangle \ \bar{x}]\!] \vdash \langle \top \mid \mathcal{M}\rangle} \qquad \frac{\neg\exists\mathcal{M} : \mathcal{M} \models \phi}{[\![\varnothing \ \langle\phi\rangle \ \bar{x}]\!] \vdash \langle \bot \mid \bot\rangle}$$

(In these rules $\mathcal{M}$ is a *model* for ϕ and should be thought of as a tuple of ground terms.) Note that the second rule can be generalized: even in the presence of input variables, if the synthesis predicate ϕ is unsatisfiable, then the generated program must be $\langle\bot \mid \bot\rangle$.

ASSERTIONS. Parts of a formula that only refer to input variables are essentially assertions and can be moved to the precondition.

$$\frac{[\![\bar{a} \ \langle\phi_1\rangle \ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad \text{vars}(\phi_2) \subseteq \bar{a}}{[\![\bar{a} \ \langle\phi_1 \land \phi_2\rangle \ \bar{x}]\!] \vdash \langle \phi_2 \land P \mid \bar{T}\rangle}$$

CASE SPLIT. A top-level disjunction in the formula can be handled by deriving programs for

both disjuncts and combining them with an if-then-else structure.

$$\frac{[\![\bar{a}\ \langle \phi_1 \rangle\ \bar{x}]\!] \vdash \langle P_1 \mid \bar{T}_1 \rangle \qquad [\![\bar{a}\ \langle \phi_2 \rangle\ \bar{x}]\!] \vdash \langle P_2 \mid \bar{T}_2 \rangle}{[\![\bar{a}\ \langle \phi_1 \lor \phi_2 \rangle\ \bar{x}]\!] \vdash \langle P_1 \lor P_2 \mid \mathsf{if}(P_1)\ \{\bar{T}_1\}\ \mathsf{else}\ \{\bar{T}_2\} \rangle}$$

Given that the generated code executes both branches in a certain order, we can also strengthen ϕ_2 into $\phi_2 \land \neg P_1$ in the second part of the premise. While this will not change the precondition of the final program, it can trigger simplifications in $\phi_2 \land \neg P_1$. Intuitively, this relieves the second program from the burden of analyzing inputs that the first can always handle.

UNUSED INPUT. Input variables that are not used in the constraint can be discarded.

$$\frac{[\![\bar{a}\ \langle \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad a_0 \notin \mathrm{vars}(\phi)}{[\![a_0 ; \bar{a}\ \langle \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle}$$

UNCONSTRAINED OUTPUT. Output variables that are not constrained by ϕ can be assigned any value.

$$\frac{[\![\bar{a}\ \langle \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \mathrm{vars}(\phi)}{[\![\bar{a}\ \langle \phi \rangle\ x_0 ; \bar{x}]\!] \vdash \langle P \mid \mathsf{any} ; \bar{T}(\bar{a}) \rangle}$$

In the program, `any` denotes a nullary function that returns an arbitrary of the appropriate type.

ONE-POINT. Whenever the value of an output variable is uniquely determined by an equality atom, it can be eliminated by a simple substitution.

$$\frac{[\![\bar{a}\ \langle \phi[x_0 \mapsto t] \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \mathrm{vars}(t)}{[\![\bar{a}\ \langle x_0 = t \land \phi \rangle\ x_0 ; \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ \bar{x} := \bar{T}\ \mathsf{in}\ t ; \bar{x} \rangle}$$

DEFINITION. The definition rule is in a sense dual to ONE-POINT, and is convenient to give a name to a subterm appearing in a formula. Typical applications include purification and flattening of terms.

$$\frac{[\![\bar{a}\ \langle x_0 = t \land \phi[t \mapsto x_0] \rangle\ x_0 ; \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \mathrm{vars}(\phi)}{[\![\bar{a}\ \langle \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ (x_0 ; \bar{x}) := \bar{T}\ \mathsf{in}\ \bar{x} \rangle}$$

SEQUENCING. The sequencing rule allows us to synthesize values for two groups of variables one after another. It fixes the values of some of the output variables, treating them temporarily

as inputs, and then continues with the synthesis of the remaining ones.

$$\frac{[\![\bar{a};\bar{x}\ \langle\phi\rangle\ \bar{y}]\!] \vdash \langle P_1 \mid \bar{T}_1\rangle \qquad [\![\bar{a}\ \langle P_1\rangle\ \bar{x}]\!] \vdash \langle P_2 \mid \bar{T}_2\rangle}{[\![\bar{a}\ \langle\phi\rangle\ \bar{x};\bar{y}]\!] \vdash \langle P_2 \mid \mathsf{let}\ \bar{x} := \bar{T}_2\ \mathsf{in}\ \bar{x};\bar{T}_1\rangle}$$

STATIC COMPUTATION. A basic rule is to perform computational steps when possible.

$$\frac{[\![a_0;\bar{a}\ \langle\phi[t \mapsto a_0]\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad \mathrm{vars}(t) \subseteq \bar{a}}{[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle \mathsf{let}\ a_0 := t\ \mathsf{in}\ P \mid \mathsf{let}\ a_0 := t\ \mathsf{in}\ \bar{T}\rangle}$$

VARIABLE TRANSFORMATION. The $\vdash$ transformation preserves the variables. To show how we can change the set of variables soundly, we next present in our framework *variable transformation* by a computable function ρ [KMPS12], as an inference rule on two $\vdash$ transformations.

$$\frac{[\![\bar{a}\ \langle\phi[\bar{x} \mapsto \rho(\bar{x}')]\rangle\ \bar{x}']\!] \vdash \langle P \mid \bar{T}\rangle}{[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \rho(\bar{T})\rangle}$$

Slightly more generally, we have the following.

$$\frac{[\![\bar{a}\ \langle\phi'\rangle\ \bar{x}']\!] \vdash \langle P \mid \bar{T}\rangle \qquad \exists\bar{x} : \phi \models \exists\bar{x}' : \phi' \qquad \phi' \models \phi[\bar{x} \mapsto \rho(\bar{x}')]}{[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \rho(\bar{T})\rangle}$$

EXISTENTIAL PROJECTION. This rule is a special case of variable transformation, where ρ simply projects out some of the variables.

$$\frac{[\![\bar{a}\ \langle\phi\rangle\ \bar{x};\bar{x}']\!] \vdash \langle P \mid \bar{T}\rangle}{[\![\bar{a}\ \langle\exists\bar{x}' : \phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ (\bar{x};\bar{x}') := \bar{T}\ \mathsf{in}\ \bar{x}\rangle}$$

6.4 Synthesis for Linear Integer Arithmetic

This section presents a synthesis procedure for integer linear arithmetic (or Presburger arithmetic) using the formalism developed in Section 6.3. We assume without loss of generality that the synthesis problem is given as a conjunction of positive literals, with the only two relations being = and ≤. The overall strategy is to, for a conjunction of literals, first eliminate as many variables as possible using the equalities, then to compute witness terms for the remaining variables based on the inequalities. The reader will observe that our description reads like a description of quantifier elimination.

6.4.1 Processing Equalities

A practical way of solving a system of integer arithmetic equalities is to rewrite it into Hermite normal form (an integer analogous to echelon normal form) [Coh93]. Given a system

$$M\bar{x} = \bar{t}(\bar{a})$$

where M is an integer matrix and $\bar{t}(\bar{a})$ denotes terms built over input variables and constants, find two matrices U and H such that $H = MU$, U is unimodular and H is triangular. One can then apply a variable transformation to simplify the system as

$$H\bar{x}' = \bar{t}(\bar{a}) \qquad\qquad \text{with } \bar{x} = U\bar{x}'$$

Since H is triangular, computing solutions for $\bar{x}'$ is straightforward: it suffices to solve variables one by one and propagate their solution term. Because we are solving problems over integers, though, we must take care that all divisions are valid. To this effect, we record the divisibility constraints along the way. We denote these two operations by $\mathsf{sol}(H, \bar{x}', \bar{t}(\bar{a}))$ and $\mathsf{div}(H, \bar{x}', \bar{t}(\bar{a}))$ respectively. As an example, consider the system

$$\begin{pmatrix} 5 & 2 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

Solving for x'_2 gives us $\frac{a_2}{3}$ as a term, and the divisibility constraint $3 \mid a_2$. Propagating this, we obtain for x'_1 the divisibility constraint $5 \mid \frac{3a_1 - 2a_2}{3}$ and the term $\frac{3a_1-2a_2}{15}$. We denote this by

$$\mathsf{sol}(\begin{pmatrix} 5 & 2 \\ 0 & 3 \end{pmatrix}, \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix}, \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}) = \left(\frac{3a_1 - 2a_2}{15}, \frac{a_2}{3}\right) \qquad \text{and}$$

$$\mathsf{div}(\begin{pmatrix} 5 & 2 \\ 0 & 3 \end{pmatrix}, \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix}, \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}) = 5 \mid \frac{3a_1 - 2a_2}{3} \wedge 3 \mid a_2$$

Note that the first one is a vector, while the second one is a formula. In both cases, the free variables could in principle range over $\{x'_1, x'_2, a_1, a_2\}$. Except in degenerate cases, however, the free variables contain only a subset of the x variables. These two functions allow us to express the resolution of a system of equalities as a rewrite rule:

$$\frac{H = MU \qquad [\![\bar{a}\ \langle \mathsf{div}(H, \bar{x}', \bar{t}(\bar{a})) \wedge \phi[\bar{x} \mapsto U \cdot \mathsf{sol}(H, \bar{x}', \bar{t}(\bar{a}))] \rangle\ \bar{x}']\!] \vdash \langle P \mid \bar{T} \rangle}{[\![\bar{a}\ \langle M\bar{x} = \bar{t}(\bar{a}) \wedge \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}[\bar{x}' \mapsto \bar{x}] \rangle}$$

Note that this rule is in fact a special case of VARIABLE TRANSFORMATION, where the multiplication by U serves the role of ρ.

Intuitively, applying this rule corresponds to eliminating as many variables as the system of equations permits. The remaining variables correspond to the dimensions of the solution space for the equations, and we use them to rewrite the rest of the problem, applying the

variable substitution principle.

6.4.2 Processing Inequalities

Contrary to the case of equalities, where the most productive approach is to consider the entire system at once, in the case of inequalities we treat variables one by one. We assume the use of the SEQUENCING rule and thus present all transformations as applied to a single output variable x. We therefore work with a synthesis problem of the form:

$$\left[\!\left[\bar{a} \left\langle \bigwedge_i L_i \leq l_i x \;\wedge\; \bigwedge_j u_j x \leq U_j \right\rangle x \right]\!\right]$$

where all l_i and u_j are integer constants, and L_i and U_j are terms whose free variables are in $\bar{a}$ (lower bounds and upper bounds). Possible witnesses for terms for x are:

$$L = \max_i \left(\left\lceil \frac{L_i}{l_i} \right\rceil \right) \qquad \text{and} \qquad U = \min_i \left(\left\lfloor \frac{U_i}{u_i} \right\rfloor \right)$$

(Note that x may not be bounded on both sides, but we can assume it has at least one bound, for otherwise we would simply apply the UNCONSTRAINED OUTPUT rule.)

For the witness to be valid, though, the inequality $L \leq U$ needs to hold, or in expanded form:

$$\bigwedge_{i,j} \left\lceil \frac{L_i}{l_i} \right\rceil \leq \left\lfloor \frac{U_i}{u_i} \right\rfloor \tag{6.3}$$

We cannot simply output this conjunction as the precondition, as it is not expressed and integer linear arithmetic, and may thus prevent us from processing further variables.

Let

$$m = \text{lcm}(\text{lcm}_i\, l_i, \text{lcm}_j\, u_j)$$

the least common multiple of all multiplying constants l_i and u_j. For each lower bound term L_i, let $L'_i = \frac{m}{l_i} L_i$, and similarly for upper bounds. For a given i and j, we have the equivalences

$$\left\lceil \frac{L_i}{l_i} \right\rceil \leq \left\lfloor \frac{U_j}{u_j} \right\rfloor \iff \left\lceil \frac{L'_i}{m} \right\rceil \leq \left\lfloor \frac{U'_j}{m} \right\rfloor \iff \frac{L'_i}{m} \leq \frac{U'_j - U'_j \bmod m}{m}$$
$$\iff U'_j \bmod m \leq U'_j - L'_i \iff \exists n_j, k_j :\; U'_j = m \cdot n_j + k_j \wedge k_j \leq U'_j - L'_i$$

Using these equivalences, formula (6.3) can be rewritten as

$$\exists \bar{m}, \bar{k} : \bigwedge_j \left(U'_j = m \cdot n_j + k_j \;\wedge\; \bigwedge_i \left(k_j \leq U'_j - L'_i \right) \right)$$

We could in principal handle the newly introduced m_j and k_j variables using the EXISTENTIAL PROJECTION rule. If we call $|J|$ the number of j indices in (6.3), we observe that our rewriting introduces $|J|$ equalities and $2 \cdot |J|$ new variables. Our rules for processing equalities thus cannot guarantee to always them all.

We observe however that whenever there is solution, there is also one where all k_j take values between 0 and $m-1$. This implies that a sound synthesis schema for the k_j variables is to apply case splitting on all $m \cdot |J|$ possible assignments.

6.4.3 Example

We now present an example of our synthesis procedure. Consider the synthesis problem:

$$[\![a \ \langle x_1 + x_2 = a \wedge x_1 + 5x_3 = 2x_2 \wedge x_1 \geq 0 \wedge x_2 \geq 0 \wedge x_3 \geq 1 \rangle \ x_1, x_2, x_3]\!]$$

As described in the previous sections, we first handle the equalities. The corresponding system is:

$$M\bar{x} = \bar{t}(\bar{a}) \equiv \begin{pmatrix} 1 & 1 & 0 \\ 1 & -2 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} a \\ 0 \end{pmatrix}$$

The Hermite normal form of M is given by:

$$H = MU \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & -2 & 5 \end{pmatrix} \begin{pmatrix} -1 & -3 & -5 \\ 2 & 3 & 5 \\ 1 & 2 & 3 \end{pmatrix}$$

In this case, H is already in completely solved form, and there are no divisibility constraints. We have:

$$\mathsf{sol}(H, \begin{pmatrix} x_1' \\ x_2' \\ x_3' \end{pmatrix}, \begin{pmatrix} a \\ 0 \end{pmatrix}) = \begin{pmatrix} a \\ 0 \\ x_3' \end{pmatrix} \qquad \text{and} \qquad \mathsf{div}(H, \begin{pmatrix} x_1' \\ x_2' \\ x_3' \end{pmatrix}, \begin{pmatrix} a \\ 0 \end{pmatrix}) = \top$$

By multiplying the solution vector by U, we obtain our variable substitution:

$$x_1 \mapsto -a - 5x_3' \qquad x_2 \mapsto 2a + 5x_3' \qquad x_3 \mapsto a + 3x_3'$$

We apply this substitution to the inequalities of the original problem to obtain a new problem:

$$[\![a \ \langle -2a \leq 5x_3' \wedge 1 - a \leq 3x_3' \wedge 5x_3' \leq -a \rangle \ x_3']\!]$$

Because we have no more variables to eliminate, we do not need to eliminate the max, $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ operators and can keep as a precondition the computable expression:

$$\max\left(\left\lceil\frac{-2a}{5}\right\rceil, \left\lceil\frac{1-a}{3}\right\rceil\right) \leq \left\lfloor\frac{-a}{5}\right\rfloor$$

We can choose as a witness term, e.g. the upper bound. Our final program reads as follows:

$$\begin{array}{l}
\mathsf{let}\ \ x_3' := \frac{-a}{5}\ \ \mathsf{in} \\
\quad \mathsf{let}\ \ x_1 := -a - 5x_3'\ \ \mathsf{in} \\
\quad \mathsf{let}\ \ x_2 := 2a + 5x_3'\ \ \mathsf{in} \\
\quad \mathsf{let}\ \ x_3 := a + 3x_3'\ \ \mathsf{in} \\
\qquad (x_1, x_2, x_3)
\end{array}$$

6.5 Synthesis for QFBAPA

In this section, we briefly describe how we can leverage synthesis for integer linear arithmetic to obtain a synthesis procedure for QFBAPA. In Chapter 4, we have presented a decision procedure that works by reducing constraints over sets and their cardinalities to constraints in integer arithmetic. The general idea of synthesis for QFBAPA is to use a similar reduction through a variable substitution. The principal difference with the procedure of Chapter 4 is that our synthesis rules must also account for model reconstruction.

We illustrate how to handle this through an example. Consider the synthesis problem of splitting a set S into partitions of desired sizes:

$$[\![S\ \langle A \cup B = S \land A \cap B = \emptyset \land |A| + 5x = 2|B| \land x \geq 1\rangle\ \ A, B, x]\!]$$

As in Chapter 4, we start by labeling the sizes of the Venn regions of A, B and S with integer variables k_i, as displayed in Figure 6.1.

We now rewrite our synthesis problem using these new variables:

$$\begin{array}{rcl}
A \cup B = S & \leadsto & k_1 = k_2 = k_3 = k_4 = 0 \\
A \cap B = \emptyset & \leadsto & k_3 = k_7 = 0 \\
|A| + 5x = 2|B| & \leadsto & k_1 + k_3 + k_5 + k_7 + 5x = 2(k_2 + k_3 + k_6 + k_7) \\
x \geq 1 & \leadsto & x \geq 1
\end{array}$$

We additionally encode the cardinality of the input set variable S as equality $s = k_4 + k_5 + k_6 + k_7$, where s is a fresh (integer) input variable. Finally, we constrain all variables k_i to be positive. After simplification, the problem reduces to

$$[\![s\ \langle k_5 + k_6 = s \land k_5 + 5x = 2k_6 \land k_5 \geq 0 \land k_6 \geq 0 \land x \geq 1\rangle\ \ k_5, k_6, x]\!]$$

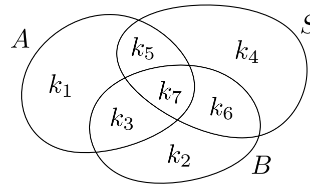

Figure 6.1 – Labeling of Venn regions of A, B, and S.

Up to renaming, this problem is equivalent to the one we presented in Section 6.4.3. The synthesis for integers thus yields the first part of the final program, to compute k_5 and k_6.

It remains to show how we can reconstruct a solution for the set variables from a solution for k_5 and k_6. (In the terminology of VARIABLE TRANSFORMATION, we need to identify ρ). For this, we rely on the existence of any computable function $\mathsf{pick}(i, T)$ that computes a subset of size i of a set T. For instance, $\mathsf{pick}(i, T)$ can pick the top i elements according to some ordering. We reconstruct A and B as follows:

$$\begin{array}{l} \mathsf{let} \quad A = \mathsf{pick}(k_5, S) \quad \mathsf{in} \\ \mathsf{let} \quad B = \mathsf{pick}(k_6, S \setminus A) \quad \mathsf{in} \\ \qquad (A, B) \end{array}$$

By construction, the sets from which subsets are selected are guaranteed to be of sufficient size.

6.6 Synthesis for Term Algebras

We present in this section a synthesis procedure for formulas over term algebras. We start by assuming a pure term algebra, and later extend the system to algebras with elements from parameter theories. In both cases, we present a series of normal forms and inference rules, and argue that together they can be used to build a synthesis procedure.

6.6.1 Pure Term Algebras

The grammar of atoms over our term algebra is given by the following two production rules, where c and F denote a constant and a function symbol from the algebraic signature, respectively:

$$\begin{array}{lcl} A & ::= & T = T \mid T \neq T \mid \mathrm{is}_c(T) \mid \mathrm{is}_F(T) \\ T & ::= & x \mid c \mid F(\bar{T}) \mid F_i(T) \end{array}$$

In the following we assume that the algebra defines at least one constant and one non-nullary constructor function. Formulas are built from atoms with the usual propositional connectives. We use an extension of the standard theory of term algebras. The extension defines additional

unary *tester* functions $\mathrm{is}_c(\cdot)$ and $\mathrm{is}_F(\cdot)$ for constant and functions in the algebraic signature respectively, and unary *selector* functions $F_i(\cdot)$, with $1 \leq i \leq n$ where n is the arity of F. These extra symbols form a definitional extension [Hod97] given by the axioms:

$$\forall x:\ \mathrm{is}_c(x) \iff x = c \tag{6.4}$$

$$\forall x:\ \mathrm{is}_F(x) \iff \exists \bar{y}:\ x = F(\bar{y}) \tag{6.5}$$

$$\forall x, y:\ F_i(x) = y \iff (\exists \bar{y}:\ y = \bar{y}[i] \land F(\bar{y}) = x) \lor \neg(\exists \bar{y}:\ F(\bar{y}) = x) \land x = y \tag{6.6}$$

Note that the case analysis in (6.6) is required only to make the selector functions total. In practice, we are only interested in cases where the selectors are applied to arguments of the proper type. We will therefore assume in the following that each selector application $F_i(x)$ is accompanied with a side condition $\mathrm{is}_F(x)$.

Rewriting of tester and selector functions. By applying the axioms (6.4) and (6.5), we can rewrite all applications of a tester function into an existentially quantified equality over terms. We can similarly eliminate applications of testers by existentially quantifying over the arguments of the corresponding constructors. Using the EXISTENTIAL PROJECTION rule, we can in turn consider the obtained synthesis problem as a quantifier-free one.

Elimination through unification. We can at any point apply *unification* to a conjunction of equalities over terms. Unification rewrites a conjunction of term equations into either $\bot$, if the equations contain a cycle or an equality involving incompatible constructors, or into an equivalent conjunction of atoms $\bigwedge_i v_i = t_i$, where v_i is a variable and t_i is a term. This set of equations has the additional property that

$$\left(\bigcup_i \{v_i\}\right) \cap \left(\bigcup_i \mathrm{vars}(t_i)\right) = \emptyset$$

In other words, it defines a set of variables as a set of terms built from another disjoint set of variables [BS01]. This form is particularly suitable for applications of the ONE-POINT rule: indeed, whenever v_i is an output variable, we can apply it, knowing that v_i does not appear in t_i (or in any other equation).

Dual view. Unification allows us to eliminate output variables that are to the left of an equality. When instead an input variable appears in such position, we can resort to a dual form to eliminate output variables appearing in the right-hand side. We obtain the dual form by applying as much as possible the following two rules to term equalities:

$$\frac{t = c}{\mathrm{is}_c(t)} \qquad \frac{t = F(t_1, \ldots, t_n)}{F_1(t) = t_1 \land \ldots \land F_n(t) = t_n \land \mathrm{is}_F(t)}$$

Note that these are rewrite rules for formulas. Because they preserve the set of variables and equisatisfiability, they can be lifted to inference rules for programs using the EQUIVALENCE rule. Observe that at saturation, the generated atoms are of two kinds: 1) applications of tester predicates and 2) equalities between two terms, each containing at most one variable. In particular, all equalities between an output variable and a term are amenable to applications of the ONE-POINT rule.

Disequalities. Finally, we introduce a dedicated rule for the treatment of disequalities between terms. The rule is defined for disequalities over variables and constants in *conjunctive normal form* (CNF). From a conjunction of disequalities over terms, we can obtain CNF by applying the following rewrite rules until saturation:

$$\frac{F(\bar{t}_1) \neq G(\bar{t}_2) \qquad F \neq G}{\top} \qquad\qquad \frac{F(\bar{t}_1) \neq F(\bar{t}_2)}{t_1^1 \neq t_1^n \vee \ldots \vee t_2^1 \neq t_2^n}$$

Intuitively, the first rule captures the fact that terms built with distinct constructors are trivially distinct (note that this also captures distinct constants, which are nullary constructors). The second rule breaks down a disequality into a disjunction of disequalities over subterms.

To obtain witness terms from the CNF, it suffices to satisfy one disequality in each conjunct. We achieve this by eliminating one variable after another, applying for each a diagonalization principle, as follows. In the following rule ϕ_{CNF} denotes the part of the CNF formula over atomic disequalities that does not contain a given variable of interest x_0.

$$\frac{[\![\bar{a}\ \langle \phi_{\text{CNF}} \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \phi_{\text{CNF}}}{\left[\!\!\left[\bar{a} \left\langle \begin{matrix} & (x_0 \neq t_1 \vee \ldots) \\ \wedge & \ldots \\ \wedge & (x_0 \neq t_n \vee \ldots) \\ \wedge & \phi_{\text{CNF}} \end{matrix} \right\rangle x_0 ; \bar{x} \right]\!\!\right] \vdash \langle P \mid \mathsf{let}\ \bar{x} := \bar{T}\ \mathsf{in}\ (\Delta(t_1, \ldots, t_n) ; \bar{T}) \rangle}$$

In the generated program, Δ denotes an n-ary computable function that returns, at run time, a term distinct from all its arguments. Such a value is guaranteed to exist, since the term algebra is assumed to have at least one constructor. This function runs in time polynomial in the number of its arguments.

Synthesis Procedure

We now argue that the reductions to normal forms and the rules presented above are sufficient to form a complete synthesis procedure for a given pure term algebra. The procedure is given by the following steps:

1. Reduce an arbitrary propositional structure to a conjunction through applications of the CASE SPLIT rule.

2. Remove selectors and testers through rewrites and applications of EXISTENTIAL PROJECTION.
3. Apply unification to all equalities, then apply ONE-POINT as often as possible. As a result, the only equalities remaining have the form $a = t$, where a is an input variable and $a \notin \text{vars}(t)$.
4. Rewrite into dual form, then apply ONE-POINT as much as possible. After applying ASSERTIONS, the problem is reduced to a conjunction of disequalities, each involving at least one output variable.
5. Transform the conjunction into CNF and eliminate all remaining variables by successive applications of the diagonalization rule.

Given a conjunction of literals, the generated program runs in time polynomial in the size of the input terms: it consists of a sequence of assignments, one for each output variable, and each term has polynomial size.

6.6.2 Reduction to an Interpreted Theory

We now consider the case of a term algebra defined over an interpreted theory $\mathcal{T}$. A canonical example is the algebra of integer lists, where $\mathcal{T}$ is the theory of integers, and defined by the constant $\mathsf{Nil} : \mathsf{List}$ and the constructor $\mathsf{Cons} : \mathbb{Z} \times \mathsf{List} \to \mathsf{List}$. In this theory, the selector function $\mathsf{Cons}_1(\cdot)$, for instance, is of type $\mathsf{List} \to \mathbb{Z}$. We show how to reduce a synthesis problem in the combination of theories to a synthesis problem in $\mathcal{T}$. We focus on the important differences with the previous case.

Purification. We can assume without loss of generality that constructor terms contain no subterms from $\mathcal{T}$ other than variables. Indeed one can always apply the DEFINITION rule to separate such terms.

Unification. Applying unification can result in derived equalities between variables of $\mathcal{T}$. These should simply be preserved in the reduced problem.

Dual view. Applying the rewriting into the dual view can result in derived equalities of the form $x = t$, where x is a variable from $\mathcal{T}$ and t is an application of selectors to an input variable. Because $\mathcal{T}$ cannot handle these selectors, we need to rewrite t into a simple variable. By using the DEFINITION and SEQUENCING rules, we make this variable an input of the problem in $\mathcal{T}$.

Disequalities. Contrary to the pure case, we cannot always eliminate all conjuncts in the CNF by applying a diagonalization; we can eliminate variables that belong to the term algebra, but not variables of $\mathcal{T}$. Instead, for each disequality $v \neq w$ over $\mathcal{T}$ in the CNF, we introduce at

the top-level a disjunction $v = w \vee v \neq w$, and apply the CASE SPLIT rule to encode a guess. This in essence compiles the guessing of the partitioning of shared variables that is traditionally introduced in a Nelson-Oppen setting [NO79]. Because CASE SPLIT preserves the relation entirely, this is a sound and complete reduction step.

Once all the disequalities have been handled, either through diagonalization if they are over algebraic terms, or by case-splitting if they are over $\mathcal{T}$ variables, we have entirely reduced the synthesis problem into a synthesis problem for $\mathcal{T}$.

6.7 Implementation

We have implemented our synthesis procedures for integer linear arithmetic and QFBAPA as a tool called Comfusy (Complete Functional Synthesis). [1] Comfusy is a plugin for the Scala compiler, and can synthesize code for all the examples presented in this chapter, including pattern-matching expressions involving arithmetic constraints. Since it works at an early compilation, all subsequent optimization are applied to our generated code as well. We used Z3 [dMB08b] to handle the compile-time checks. (Although we could in principle have used synthesis to solve these satisfiability problems, our implementation is far slower for such tasks than Z3.)

Compilation times. Figure 6.2 shows the compile times for a set of benchmarks, with and without our plugin. Without the plugin, the code is of no use (the invocations to synthesis, when not rewritten, simply throw an exception), but the difference between the timings indicates how much time is spent generating the synthesized code. We also measured how much time is used for the compile-time checks for satisfiability and uniqueness. The examples *SecondsToTime*, *FastExponentiation*, and *SplitBalanced* were presented in Section 6.2. *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*.

Our numbers show that the additional time required for the code synthesis is minimal. Moreover, 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 synthesis invocations.

Code size. An older version of Comfusy generated if-then-else statements that correspond to large disjunctions that appear when handling inequalities, as explained at the end of Section 6.4.2. 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.

1. The synthesis procedure for term algebras has been implemented independently and is available from http://lara.epfl.ch/~psuter/spt/.

	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

Figure 6.2 – Measurement of synthesis 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.

Comfusy is available for download from http://lara.epfl.ch.

6.8 Related Work

The theory and results in this chapter were originally presented in a series of papers [KMPS10b, KMPS10a, KMPS12, JKS13].

Early work on synthesis [MW71, MW80] focused on synthesis using expressive and undecidable logics, such as first-order logic and logic containing the induction principle. The Deductive Tableau methodology [MW92, AB01], for instance, can synthesize interesting programs containing recursion. There are two features that distinguish our work from such approaches. First, we can synthesize programs even when the output does not exist for any input. In fact, the precondition computed by synthesis procedures exactly characterizes the acceptable input domain. Second, our procedures provide completeness and termination guarantees, just like decision procedures, while tableau-based techniques do not.

Recent work on synthesis [SGF10] 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 [SLTB+06, SLAT+07]. The algorithms employed in sketching are typically focused on appropriately guided search over the syntax tree of the synthesized program. 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.

Quantifier elimination decision procedures directly support parameterized problems, so they are a particularly convenient starting point for our method. Other decision procedures are also suitable, but may require more design and implementation effort to be turned into interesting synthesis procedures. In particular, an alternative automata-theoretic approach to synthesis for integer arithmetic with bitvectors was subsequently developed [HJK10]; this approach tends to generate larger but more efficient code. Term algebras admit quantifier elimination [SW02, Hod97] and thus are suitable candidates. Our synthesis procedure is similar to quantifier elimination when it comes to eliminating variables that are constrained by an equality, with the additional requirement that the witness term be stored to serve in the program. However the treatment of disequalities is simplified: elimination procedures typically rewrite a disequality between a variable and a term into a disjunction of equalities between the same variable and terms constructed with different constructors [Hod97, p.63sqq]. This has the advantage that the language of formulas needs not be extended, allowing for nested quantifiers to be eliminated one after the other. In our synthesis setting, this is not necessary, as we can rely on additional computable functions, as we have illustrated with the use of Δ, thus greatly simplifying the resulting program. A related area of research is compilation of unification in Prolog [AK91]. Note, however, that this process typically does not require handling of disequalities, so it deals with a simpler language.

Pattern-matching compilation is a task for which specialized procedures for term algebras have been developed [Wad87, Mar08]. When viewed through the prism of synthesis procedures, these algorithms can be thought of as procedures that are specialized for disjunctions of term equalities, and where the emphasis is put on code reuse. We expect that using a combination of our synthesis procedures and common subexpression elimination techniques, one should be able to derive pattern-matching compilation schemes that would support, e.g., disjunctive patterns, non-linear patterns, and could take into account guards referring to integer predicates.

Specialization of decision procedure for the purpose of predicate abstraction was considered in [LBC05]. In addition to covering a different set of theories, our results are broader because our process generates not only a satisfiability check but also the values of variables.

Finally, our approach can be viewed as sharing some of the goals of partial evaluation [JGS93]. 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.

7 Conclusion

We have presented in this dissertation new theoretical results as well as concrete system implementations. We have developed decidability results for broad classes of logic, and demonstrated their applicability to software verification. We have developed and implemented an extension to Scala that integrates declarative programming with the more standard functional and imperative paradigms, demonstrating the feasibility and the benefits of this approach. Moreover, we have designed and implemented techniques to automatically derive correct program implementations from specifications.

Taken individually, our contributions represent advances in the fields of decision procedures, program verification, programming language design and synthesis respectively. Taken together, we believe they form conclusive evidence that program specifications have an integral role to play alongside implementations in software development.

We conclude this dissertation by highlighting future research directions of interest.

Quantifier elimination for term algebras with catamorphisms. Our results in chapters 2 and 3 apply to *quantifier-free* formulas. While positive results already exist for particular instances of the problem [ZSM05], it would be interesting to characterize when exactly the first-order theory is decidable for the combination of term algebras, a catamorphism and the image theory. One immediate application would be to devise a synthesis procedure for such a theory, thus extending the applicability of the framework presented in Chapter 6.

Decision procedures for sets with homomorphisms. We have presented in Chapter 4 a decision procedure for sets with the cardinality operator. Our technique relies on decomposing constraints by identifying and organizing absolute and relative independences. One intuitive explanation for why this works is that models for BAPA formulas have a high degree of symmetry: if there exists one model, then there exist many others, obtained by permuting the elements and preserving the sizes of the Venn regions.

The cardinality operator on sets is a particular instance of a set homomorphism. A desirable result would be to establish that, for any set homomorphism α, the theory of one-level interpreted sets with α is decidable and amenable to hypertree decomposition. This would give us for instance an efficient algorithm for the theory of sets with minimal and maximal elements [KPS10]. We expect that such a general result may require that the homomorphism be accompanied with a unary predicate characterizing its co-domain, not unlike the condition under which the satisfiability procedure of Chapter 3 is complete for tree homomorphisms.

Synthesis beyond decidable domains. We have used in Chapter 6 the formalism of inference rules to describe synthesis procedures as application strategies that always succeed. While we find it remarkable that such procedures exist for interesting classes of synthesis predicates, bridging the gap between what can be solved at runtime and what can be synthesized will necessarily involve going beyond these domains. An appealing direction is to define *speculative* rules, that apply transformations which may or may not make progress towards a solution. As an example, we could devise a rule that corresponds to guessing that the solution requires a recursive function, and that decomposes the problem into a base case and a recursive case. In the recursive case, it would be free to assume that the synthesis predicate holds for the recursive call. Speculative rules, when they reach a goal, would still be required to generate correct programs by construction, and in particular to satisfy the relation refinement and domain preservation conditions. A strategy for synthesis in the presence of speculative rules would be to perform synthesis procedures steps whenever applicable, and to interleave them with guesses if no immediate progress is possible. When the resulting search space becomes too large to be searched practically, a promising possibility is to rely on interaction with the user, in a way similar to how proof assistants work: synthesis could become a process where the computer performs as many steps as is possible in a short amount of time, and relies on the user to break down harder problems.

Integrating synthesis and runtime solving. Finally, we believe new exciting directions will arise from the combination of the runtime approach described in Chapter 5 and the synthesis approach from Chapter 6. These two approaches come with different trade-offs: the runtime approach can handle general constraints that need not be known entirely at compile time, hardly generates any code, but can be slow. The synthesis approach on the other hand requires that the predicate is completely known at the time of synthesis, works for well-defined classes, and generates very efficient, although sometimes large, code. The first natural direction in combining these two approaches would be to compile specifications whenever they fit into synthesizable classes. Simply detecting this at compile time is unlikely to be enough, though, as constraints are typically built dynamically. A more promising solution would be to invoke synthesis procedures at run time, resulting in an architecture similar to just-in-time compilers in virtual machines. Another avenue for combination would be to allow for synthesis rule to generate code that invokes constructs such as `find`, resulting in a form of code generation where programs are only partially synthesized. This is particularly appealing for problems

that admit a simple solution in a general case, but require complex programs for a few corner cases.

Such advances would be an important step towards programming languages that integrate specifications and code. It is our hope that the availability of such tools will impact the way we approach program development, and constructing correct programs will no longer be the exception, but become the norm.

Bibliography

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *FMCO*, pages 113–132, 2007.

[AB01] Abdelwaheb Ayari and David A. Basin. A higher-order interpretation of deductive tableau. *JSC*, 31(5):487–520, 2001.

[ABPS98] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-O: An imperative language that supports declarative programming. *TOPLAS*, 20(5):1014–1066, 1998.

[AH10] Sergio Antoy and Michael Hanus. Functional logic programming. *CACM*, 53(4):74–85, 2010.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[Ant92] Sergio Antoy. Definitional trees. In *ALP*, pages 143–157, 1992.

[AW07] Krzysztof R. Apt and Mark Wallace. *Constraint logic programming using Eclipse.* Cambridge University Press, 2007.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

[BdMR+10] Clark Barrett, Leonardo M. de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The SMT-LIB initiative and the rise of SMT. In *Haifa Verification Conference*, page 3, 2010.

[BFL+11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *CACM*, 54(6):81–91, 2011.

[BG05] Franz Baader and Silvio Ghilardi. Connecting many-sorted theories. In *CADE*, pages 278–294, 2005.

[BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem.* Springer-Verlag, 1997.

[BGHL10] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. In *ICFP*, pages 105–116, 2010.

[BGK85] Andreas Blass, Yuri Gurevich, and Dexter Kozen. A zero-one law for logic with a fixed-point operator. *Information and Control*, 67(1-3):70–90, 1985.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[BHM08] Bernd Braßel, Michael Hanus, and Marion Müller. High-level database programming in Curry. In *PADL*, pages 316–332, 2008.

[BHR+00] Thomas Ball, Daniel Hoffman, Frank Ruskey, Richard Webber, and Lee J. White. State generation and automated class testing. *STVR*, 10(3):149–170, 2000.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[BKW07] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. A complete bounded model checking algorithm for pushdown systems. In *Haifa Verification Conference*, pages 202–217, 2007.

[Bla12] Régis Blanc. Verification of imperative programs in Scala. Master's thesis, EPFL, July 2012.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer, 2007.

[BN10] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *ITP*, pages 131–146, 2010.

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In *TACAS*, pages 150–153, 2010.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.

[BST07] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. *ENTCS*, 174(8):23–37, 2007.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In *CAV*, pages 298–302, 2007.

[BTV09] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321, 2009.

[BvW99] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - a systematic introduction*. Undergraduate texts in computer science. Springer, 1999.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252, 1977.

[CD94] Hubert Comon and Catherine Delor. Equational formulae with membership constraints. *IANDC*, 112(2):167–216, 1994.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, pages 23–42, 2009.

[CDMV11] Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. The acl2 sedan theorem proving system. In *TACAS*, pages 291–295, 2011.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[CKC81] Alain Colmerauer, Henry Kanoui, and Michel Van Caneghem. Last steps towards an ultimate PROLOG. In *IJCAI*, pages 947–948, 1981.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1993.

[DdM06] Bruno Dutertre and Leonardo M. de Moura. The Yices SMT solver, 2006.

[DGL+79] Robert B. K. Dewar, Arthur Grand, Ssu-Cheng Liu, Jacob T. Schwartz, and Edmond Schonberg. Programming by refinement, as exemplified by the setl representation sublanguage. *TOPLAS*, 1(1):27–49, 1979.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, 1975.

[DL05] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 3 2005.

[dMB08a] Leonardo M. de Moura and Nikolaj Bjørner. Model-based theory combination. *ENTCS*, 198(2):37–49, 2008.

[dMB08b] Leonardo M. de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[dMB09] Leonardo M. de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52, 2009.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *JACM*, 52(3):365–473, 2005.

[DP04] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *POPL*, pages 281–292, 2004.

[DR03] Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

[DSK08] Mirco Dotta, Philippe Suter, and Viktor Kuncak. On static analysis for expressive pattern matching. Technical Report LARA-REPORT-2008-004, EPFL, 2008.

[Dun07] Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.

[EK08] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: a tool for repairing complex data structures. In *ICSE*, pages 855–858, 2008.

[Fer10] Pietro Ferrara. Static type analysis of pattern matching by abstract interpretation. In *FMOODS/FORTE*, pages 186–200, 2010.

[FKcS09] Sebastian Fischer, Oleg Kiselyov, and Chung chieh Shan. Purely functional lazy non-deterministic programming. In *ICFP*, pages 11–22, 2009.

[Fon07] Pascal Fontaine. Combinations of theories and the bernays-schönfinkel-ramsey class. In *VERIFY*, 2007.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI*, pages 268–277, 1991.

[FV59] Solomon Feferman and Robert L. Vaught. The first order properties of products of algebraic systems. *Fundamenta Mathematicae*, 47:57–103, 1959.

[GGJ$^+$10] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *ICSE*, pages 225–234, 2010.

[GGM09] Georg Gottlob, Gianluigi Greco, and Bruno Marnette. HyperConsistency width for constraint satisfaction: Algorithms and complexity results. In *Graph Theory, Computational Intelligence and Thought*, pages 87–99, 2009.

[GH03] Martin Grabmüller and Petra Hofstedt. Turtle: A constraint imperative programming language. In *Innovative Techniques and Applications of Artificial Intelligence*, 2003.

[Ghi04] Silvio Ghilardi. Model-theoretic methods in combined constraint satisfiability. *J. Autom. Reasoning*, 33(3-4):221–249, 2004.

[GHN$^+$04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[GLAS09] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251, 2009.

[Gre69] C. Cordell Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–240, 1969.

[Gri81] David Gries. *The Science of Programming.* Springer, New York, 1981.

[GTSKF04] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with AProVE. In *RTA*, pages 210–220, 2004.

[Han06] Michael Hanus. Type-oriented construction of web user interfaces. In *PPDP*, pages 27–38, 2006.

[HB94] Paul F. Hoogendijk and Roland Carl Backhouse. Relational programming laws in the tree, list, bag, set hierarchy. *Sci. Comput. Program.*, 22(1-2):67–105, 1994.

[HJK10] Jad Hamza, Barbara Jobstmann, and Viktor Kuncak. Synthesis for regular specifications over unbounded domains. In *FMCAD*, pages 101–109, 2010.

[HK09] Michael Hanus and Christof Kluß. Declarative programming of user interfaces. In *PADL*, pages 16–30, 2009.

[HN07] Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. In *TPHOL*, 2007.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *ACTA*, 1:271–281, 1972.

[Hod93] Wilfrid Hodges. *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993.

[Hod97] Wilfrid Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[HSWP12] David Hardin, Konrad Slind, Michael W. Whalen, and Tuan-Hung Pham. The Guardol language and verification system. In *TACAS*, pages 18–32, 2012.

[IJSS08] Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. On local reasoning in verification. In *TACAS*, pages 265–281, 2008.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *TOSEM*, 11(2):256–290, 2002.

[Jaf90] Joxan Jaffar. Minimal and complete word unification. *JACM*, 37(1):47–85, 1990.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[JKS13] Swen Jacobs, Viktor Kuncak, and Philippe Suter. Reductions for synthesis procedures. In *VMCAI*, 2013.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL*, pages 111–119, 1987.

[JMR11] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV*, pages 470–485, 2011.

[JNZ06] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *TOPLAS*, 28(6):967–989, 2006.

[KGGT07] Sava Krstić, Amit Goel, Jim Grundy, and Cesare Tinelli. Combined satisfiability modulo parametric theories. In *TACAS*, pages 602–617, 2007.

[Kin71] James C. King. A program verifier. In *IFIP Congress*, pages 234–249, 1971.

[KK71] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.

[KKS11] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: Integrating SMT and programming. In *CADE*, pages 400–406, 2011.

[KKS12] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *POPL*, pages 151–164, 2012.

[KLZR06] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin C. Rinard. Modular pluggable analyses for data structure consistency. *TSE*, 32(12):988–1005, 2006.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[KMPS10a] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A tool for complete functional synthesis. In *CAV*, pages 430–433, 2010.

[KMPS10b] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[KMPS12] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *CACM*, 55(2):103–111, 2012.

[KNR06] Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. Deciding boolean algebra with presburger arithmetic. *JAR*, 36(3):213–239, 2006.

[KPS10] Viktor Kuncak, Ruzica Piskac, and Philippe Suter. Ordered sets in the calculus of data structures. In *CSL*, pages 34–48, 2010.

[KPSW10] Viktor Kuncak, Ruzica Piskac, Philippe Suter, and Thomas Wies. Building a calculus of data structures. In *VMCAI*, pages 26–44, 2010.

[KR03] Viktor Kuncak and Martin C. Rinard. Structural subtyping of non-recursive types is decidable. In *LICS*, pages 96–107, 2003.

[KR05] Viktor Kuncak and Martin C. Rinard. Decision procedures for set-valued fields. *ENTCS*, 131:51–62, 2005.

[KR07] Viktor Kuncak and Martin C. Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *CADE*, pages 215–230, 2007.

[KRJ09] Ming Kawaguchi, Patrick M. Rondon, and Ranjit Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.

[KTU10] Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *POPL*, pages 495–508, 2010.

[Kun07] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, Massachusetts Institute of Technology, 2007.

[LBC05] Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. In *CAV*, pages 24–38, 2005.

[LFBB94] Gus Lopez, Bjørn N. Freeman-Benson, and Alan Borning. Implementing constraint imperative programming languages: The Kaleidospace'93 virtual machine. In *OOPSLA*, pages 259–271, 1994.

[LKR05] Patrick Lam, Viktor Kuncak, and Martin C. Rinard. Generalized typestate checking for data structure consistency. In *VMCAI*, pages 430–447, 2005.

[LQ06] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL*, 2006.

[LQ08] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*, pages 171–182, 2008.

[Mai09] Patrick Maier. Deciding extensions of the theories of vectors and bags. In *VMCAI*, pages 245–259, 2009.

[Mak77] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, pages 129–198, 1977. (In AMS, (1979)).

[Mal71] Anatolii Ivanovic Mal'cev. Chapter 23: Axiomatizable classes of locally free algebras of various types. In *The Metamathematics of Algebraic Systems*, volume 66. North Holland, 1971.

[Mar08] Luc Maranget. Compiling pattern matching to good decision trees. In *ML*, pages 35–46, 2008.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *CACM*, 3(4):184–195, 1960.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, pages 124–144, 1991.

[MJ10] Filip Maric and Predrag Janicic. URBiVA: Uniform reduction to bit-vector arithmetic. In *IJCAR*, pages 346–352, 2010.

[MN05] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.

[Moo10] J. Strother Moore. Theorem proving for verification: The early days. In *LICS*, page 283, 2010.

[Mor94] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, second edition, 1994.

[MRYJ11] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520, 2011.

[MS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[MS01] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.

[MSK11] Muhammad Zubair Malik, Junaid Haroon Siddiqui, and Sarfraz Khurshid. Constraint-based program debugging using data structure repair. In *ICST*, pages 190–199, 2011.

[MSZ07] Zohar Manna, Henny B. Sipma, and Ting Zhang. Verifying balanced trees. In *LFCS*, pages 363–378, 2007.

[MT09] Panagiotis Manolios and Aaron Turon. All-termination(t). In *TACAS*, pages 398–412, 2009.

[MW71] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *CACM*, 14(3):151–165, 1971.

[MW80] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *TOPLAS*, 2(1):90–121, 1980.

[MW92] Zohar Manna and Richard J. Waldinger. Fundamentals of deductive program synthesis. *TSE*, 18(8):674–704, 1992.

[NDQC07] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.

[Nel81] Greg Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981. Revised version of Nelson's PhD Thesis, Stanford 1980.

[Nil12] Rickard Nilsson. ScalaCheck user guide. http://code.google.com/p/scalacheck/wiki/UserGuide, 9 2012.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2):245–257, 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–364, 1980.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL Tutorial Draft*, March 8 2002.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.

[Ode10] Martin Odersky. Contracts for scala. In *RV*, pages 51–57, 2010.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Opp78] Derek C. Oppen. Reasoning about recursively defined data structures. In *POPL*, pages 151–157, 1978.

[OSS02] David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of mercury. In *PPDP*, pages 109–120, 2002.

[OSV11] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc., second edition, 2011.

[Pap81] Christos H. Papadimitriou. On the complexity of integer programming. *JACM*, 28(4):765–768, 1981.

[PK08a] Ruzica Piskac and Viktor Kuncak. Decision procedures for multisets with cardinality constraints. In *VMCAI*, pages 218–232, 2008.

[PK08b] Ruzica Piskac and Viktor Kuncak. Linear arithmetic with stars. In *CAV*, pages 268–280, 2008.

[Pla04] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *JACM*, 51(3):483–496, 2004.

[PR88] Amir Pnueli and Roni Rosner. A framework for the synthesis of reactive modules. In *Concurrency*, pages 4–17, 1988.

[PRS09] Juan Antonio Navarro Pérez, Andrey Rybalchenko, and Atul Singh. Cardinality abstraction for declarative networking applications. In *CAV*, pages 584–598, 2009.

[RKJ08] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

[Rob65] John A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

[RRH10] Adrián Riesco and Juan Rodríguez-Hortalá. Programming with singular and plural non-deterministic functions. In *PEPM*, pages 83–92, 2010.

[RV01] Tatiana Rybina and Andrei Voronkov. A decision procedure for term algebras with queues. *TOCL*, 2(2):155–181, 2001.

[SAM10] Hesam Samimi, Ei Darli Aung, and Todd D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[SDK10] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210, 2010.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[Sin10] Nishant Sinha. Modular bug detection with inertial refinement. In *FMCAD*, pages 199–206, 2010.

[SKK11] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.

[SLAT+07] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[SS09] Viorica Sofronie-Stokkermans. Locality results for certain extensions of theories with bridging functions. In *CADE*, pages 67–83, 2009.

[SSK11] Philippe Suter, Robin Steiger, and Viktor Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In *VMCAI*, pages 403–418, 2011.

[SSW09] Tom Schrijvers, Peter J. Stuckey, and Philip Wadler. Monadic constraint programming. *JFP*, 19(6):663–697, 2009.

[SW02] Thomas Sturm and Volker Weispfenning. Quantifier elimination in term algebras: The case of finite languages. In *Computer Algebra in Scientific Computing*, pages 285–300, 2002.

[Tac09] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation.* PhD thesis, Saarland University, 2009.

[Tag04] Mana Taghdiri. Inferring specifications to detect errors in code. In *ASE*, pages 144–153, 2004.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647, 2007.

[TZ03] Cesare Tinelli and Calogero G. Zarba. Combining non-stably infinite theories. *ENTCS*, 86(1):35–48, 2003.

[vR99] Peter van Roy. Logic programming in Oz with Mozart. In *ICLP*, pages 38–51, 1999.

[VSC10] VSComp: The Verified Software Competition. http://www.macs.hw.ac.uk/vstte10/Competition.html, 2010.

[Wad87] Philip Wadler. *The Implementation of Functional Programming Languages: Efficient Compilation of Pattern-matching*, chapter 5, pages 78–103. Prentice Hall, 1987.

[WKL+06] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin C. Rinard. Field constraint analysis. In *VMCAI*, pages 157–173, 2006.

[WM00] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *Types in Compilation*, pages 177–206, 2000.

[WPK09] Thomas Wies, Ruzica Piskac, and Viktor Kuncak. Combining theories with shared set operations. In *FroCos*, pages 366–382, 2009.

[WS03] Christoph Walther and Stephan Schweitzer. About VeriFun. In *CADE*, pages 322–327, 2003.

[Xi03] Hongwei Xi. Dependently typed pattern matching. *Journal of Universal Computer Science*, 9(8):851–872, 2003.

[YPK10] Kuat Yessenov, Ruzica Piskac, and Viktor Kuncak. Collections, cardinalities, and relations. In *VMCAI*, pages 380–395, 2010.

[YYSL12] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, pages 85–96, 2012.

[Zar02] Calogero G. Zarba. Combining sets with integers. In *FroCos*, pages 103–116, 2002.

[ZK10] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. Contract-based data structure repair using Alloy. In *ECOOP*, pages 577–598, 2010.

[ZKR08] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.

[ZKR09] Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351, 2009.

[ZKTR07] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. Runtime checking for program verification. In *RV*, pages 202–213, 2007.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

[ZSM05] 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.

[ZSM06] Ting Zhang, Henny B. Sipma, and Zohar Manna. Decision procedures for term algebras with integer constraints. *IANDC*, 204(10):1526–1574, 2006.

[ZX05] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *PADL*, pages 83–97, 2005.

Philippe Suter

Curriculum Vitæ

Education

EPFL, Lausanne, Switzerland	PhD Computer Science (expected)	2008 – 2012
EPFL, Lausanne, Switzerland	MS Computer Science	2006 – 2008
MIT, Cambridge, United States	Visiting student	02–08/2008
EPFL, Lausanne, Switzerland	BS Computer Science	2003 – 2006
McGill University, Montréal, Canada	Exchange student	2005 – 2006

- *Excellency Scholarship at the Master level.* Awarded for outstanding academic record, 2006 – 2008.

Work Experience

Research Assistant, EPFL 2008 – 2012

PhD studies under the guidance of my advisor Viktor Kuncak. My research focused on verification and synthesis of functional programs. It featured two core aspects. **Design and implementation of decision and synthesis procedures:** I worked on algorithms that are relevant to constraint solving and to software synthesis. This included identifying new decidable logical theories, designing efficient algorithms to solve them, and implementing them. See [5, 7, 10, 13, 14]. **Programming language design:** I studied the integration of constraint solving capabilities into mainstream programming languages. The techniques used range from library design to advanced compilation mechanisms for decidable constraint languages. See [2, 4, 6, 12].

Research Intern, ETH Zürich 08–10/2006

Worked in the group of Prof. Thomas Gross. I investigated bottlenecks and optimizations for Jython, a compiler for Python on the Java Virtual Machine. In particular, I looked at preventing unneeded boxing of primitive types into generic Python value wrappers, and moving local values from the simulated stack memory to JVM registers. All experiments were conducted with the Jikes Research Virtual Machine.

Internet Software Developer, Net Oxygen Sàrl, Gland, Switzerland 2000 – 2006

Worked part-time. I developed several database-oriented websites and web-based business solutions, both for clients and for internal use.

Publications

Conference proceedings and journal articles.

[1] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. "Functional Synthesis for Linear Arithmetic and Sets". In: *International Journal on Software Tools for Technology Transfer (STTT)* (2013). To appear.

[2] Swen Jacobs, Viktor Kuncak, and Philippe Suter. "Reductions for Synthesis Procedures". In: *Proceedings of the 14th international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2013.

[3] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. "Software Synthesis Procedures". In: *Communications of the ACM* (Feb. 2012), pp. 103–111.

[4] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. "Constraints as Control". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*. 2012, pp. 151–164.

[5] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. "Satisfiability Modulo Recursive Programs". In: *Proceedings of the 18th International Static Analysis Symposium (SAS)*. 2011, pp. 298–315.

[6] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. "Scala to the Power of Z3: Integrating SMT and Programming". In: *Proceedings of the 23rd International Conference on Automated Deduction (CADE)*. 2011, pp. 400–407.

[7] Philippe Suter, Robin Steiger, and Viktor Kuncak. "Sets with Cardinality Constraints in Satisfiability Modulo Theories". In: *Proceedings of the 12th international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2011, pp. 403–418.

[8] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. "Phantm: PHP Analyzer for Type Mismatch". In: *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE)*. 2010, pp. 373–374.

[9] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. "Runtime Instrumentation for Precise Flow-Sensitive Type Analysis". In: *Proceedings of the 1st international conference on Runtime Verification (RV)*. 2010, pp. 300–314.

[10] Viktor Kuncak, Ruzica Piskac, and Philippe Suter. "Ordered Sets in the Calculus of Data Structures". In: *Proceedings of the 19th EACSL annual conference on Computer Science Logic (CSL)*. 2010, pp. 34–48.

[11] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. "Comfusy: A Tool for Complete Functional Synthesis". In: *Proceedings of the 22nd international conference on Computer Aided Verification (CAV)*. 2010, pp. 430–433.

[12] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. "Complete Functional Synthesis". In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 2010, pp. 316–329.

[13] Philippe Suter, Mirco Dotta, and Viktor Kuncak. "Decision Procedures for Algebraic Data Types with Abstractions". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*. 2010, pp. 199–210.

[14] Viktor Kuncak, Ruzica Piskac, Philippe Suter, and Thomas Wies. "Building a Calculus of Data Structures". In: *Proceedings of the 11th international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2010, pp. 26–44.

Miscellaneous.

[15] Etienne Kneuss, Viktor Kuncak, and Philippe Suter. *On Effect Analysis for Programs with Callbacks*. Tech. rep. 180074. École Polytechnique Fédérale de Lausanne, July 2012.

[16] Aurélie Pala and Philippe Suter. "Les «Lieder ohne Worte» et l'esthétique romantique". In: *Schweizer Musikzeitung/Revue Musicale Suisse* (Feb. 2009), pp. 19–20.

[17] Philippe Suter. "Non-Clausal Satisfiability Modulo Theories". MA thesis. École Polytechnique Fédérale de Lausanne, Sept. 2008.

[18] Mirco Dotta, Philippe Suter, and Viktor Kuncak. *On Static Analysis for Expressive Pattern Matching*. Tech. rep. LARA-REPORT-2008-004. École Polytechnique Fédérale de Lausanne, Feb. 2008.

Presentations

- Presented the conference papers [4, 5, 6, 7, 11, 12, 13].
- "Phantm: PHP Analyzer for Type Mismatch", *Open Source Quality seminar, UC Berkeley*, Berkeley, Apr. 2012
- "Combining Synthesis Procedures", "Software Synthesis", Schloß Dagstuhl, Apr. 2012
- "Programming with Decision Procedures", *IBM Research*, Hawthorn, Apr. 2012
- "Sets with Cardinality Constraints in Satisfiability Modulo Theories", *VERIMAG*, Grenoble, Nov. 2011
- "Programming with Z3", *Z3 Special Interest Group Meeting*, Cambridge, Nov. 2011
- "Satisfiability Modulo Computable Functions", "Deduction at Scale", Schloß Ringberg, Mar. 2011
- "Satisfiability Modulo Computable Functions", *Northeastern University*, Boston, Feb. 2011
- "Verification of Purely Functional Scala Programs", *POPL'11 (student session)*, Austin, Jan. 2011
- "Decision Procedures for Algebraic Data Types with Abstractions", *Third Workshop on Formal and Automated Theorem Proving and Applications*, Belgrade, Jan. 2010

Reviewing Activities

Conferences: International Conference on Automated Deduction (*CADE*) 2011, Computer Aided Verification (*CAV*) 2012, International Conference on Compiler Construction (*CC*) 2009, European Symposium on Programming (*ESOP*) 2011 & 2012, European Conference on Object-Oriented Programming (*ECOOP*) 2009, Formal Methods in Computer Aided Design (*FMCAD*) 2011, Frontiers of Combining Systems (*FroCoS*) 2009, International Joint Conference on Automated Reasoning (*IJCAR*) 2010, ACM Conference on Programming Language Design and Implementation (*PLDI*) 2011, ACM Symposium on Principles of Programming Languages (*POPL*) 2011, International Static Analysis Symposium (*SAS*) 2009 & 2011, Verification, Model Checking and Abstract Interpretation (*VMCAI*) 2011 & 2012, Verified Software: Theories, Tools and Experiments (*VSTTE*) 2012. **Journals:** Information and Computation, ACM Transactions on Programming Languages (TOPLAS).

Teaching

Teaching Assistantship

- Compiler Construction (2008, 2009, 2010 & 2011)
- Theoretical Computer Science (2010)
- Software Analysis & Verification (2009)

Supervised Master Theses

- Régis Blanc (*then a PhD student at EPFL*), "Verification of Imperative Programs in Scala", 2012
- Ali Sinan Köksal (*then a PhD student at UC Berkeley*), "Constraint Programming in Scala", 2011
- Etienne Kneuss (*then a PhD student at EPFL*), "Toward Interprocedural Pointer and Effect Analysis for Scala", 2011