Thesis for the Degree of Licentiate of Engineering

Analysis of Iterative or Recursive Programs Using a First-Order Theorem Prover

Simon Robillard

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Gothenburg, Sweden
2016

Analysis of Iterative or Recursive Programs
Using a First-Order Theorem Prover

Technical Report 158L
ISSN 1652-876X
Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY AND
UNIVERSITY OF GOTHENBURG
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed at Reproservice, Chalmers University of Technology
Gothenburg, Sweden, 2016

Abstract

Static analysis of program semantics can be used to provide strong guarantees about the correctness of software systems. In this thesis, we explore ways to perform automated program analysis and verification using a first-order theorem prover.

First we present an extension to the *symbol elimination* technique for automatic generation of loop invariants. This extension introduces a new input format intended to act as an intermediate verification language, facilitating the analysis of programs written in a variety of languages. It also integrates program annotations (pre- and post-conditions), so that symbol elimination can be used not only to generate invariants, but also to prove the correctness of programs independently of other tools.

We then present ways to perform complete reasoning in the theory of term algebras in a first-order theorem prover. As term algebras provide a concrete semantics for values of algebraic data types, this extension enables one to reason about programs manipulating such data types, in particular in functional languages.

Both works were implemented using the first-order theorem prover Vampire; these implementations are presented along with experiments on a number of verification problems.

Contents

Acknowledgements

I would like to thank my supervisor, Laura Kovács, for trusting me with a PhD position, and for providing the conditions needed to accomplish the work presented here. Wolfgang Ahrendt also deserves praise for his involvement as co-supervisor.

Thanks to the many people who make Chalmers such an amazing working place. Among them, I particularly want to mention Evgeny Kotelnikov, for his company during many work-related trips, for our discussions and for his help in assembling the present document.

I owe a lot of gratitude to my parents, who not only supported me during my early years at the university, but were in fact the ones who convinced me to spend a few years there. They didn't know at the time how long I would stay!

I wish to particularly thank Frédéric Loulergue, for giving me the opportunity to discover the academic world during my undergraduate studies. Without his trust and support during those years, I would undoubtedly not be where I am today.

Finally, the work in this thesis has been supported by the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014 TheProSE , the Swedish VR grant GenPro D0497701 and the Austrian research project FWF S11409-N23.

Chapter 1

Introduction

With computers being used in many varied applications, there is a strong need to ensure that software systems perform tasks as expected, without faults or unintended behaviors. A common approach relies on testing the system: by running the program from a pre-determined configuration, it is easy to check that the result produced conforms to the expectation. But this simple process is limited: no matter how much time and effort are spent, testing can only cover a finite number of situations, whereas even moderately complex programs can run in an infinite number of different ways. For critical applications, where trust is paramount, it becomes necessary to go beyond testing, and to instead formally prove that a program is correct.

Instead of dealing directly with the executable program, it becomes useful to work on an abstract representation, which describes the meaning of the program in precise mathematical terms: this representation is called the program *semantics*. One can analyze the semantics and its properties using mathematical reasoning. It is possible to describe the intended behavior of the program as a mathematical formula called the program *specification*, and show that the program semantics satisfies this formula, in a process called *program verification*.

Proving properties of program is not very different from the way mathematicians prove theorems about numbers, spaces or other abstract objects. This vision of programming as a mathematical activity, known generally as *formal methods*, was pioneered in the 1970's by computer scientists Edgar Dijsktra and Tony Hoare, among others. Their ideas have paved the way for many different program analysis techniques, suitable for different styles of programs and properties. Despite a strong interest from the scientific community, formal methods have historically been faced with an important practical obstacle. Using the tools of formal methods to prove the correctness of a program is often a long and tedious process, as proofs must cover every case of the program during every step of computation. Additionally, few people possess the mathematical knowledge required to apply these methods. For a long time, it seemed that the complexity of formal methods outweighed their benefits, even for those applications that needed strong correctness guarantees.

In the past two decades however, formal methods have progressively gained in practicality, thanks in large part to the progress accomplished in the fields of model checking, abstract interpretation and automated theorem proving. The latter is an area of research that aims at creating tools that can reason about mathematical statements, proving or disproving them without assistance from a human operator. Such tools are ideally suited to the task of program verification, as they excel at solving large problems made of many relatively simple steps.

In this thesis, we describe new ways to use an automated theorem prover to ease program analysis and program verification. We particularly focus on using a first-order theorem prover to analyze programs that work by iterating, i.e. repeating an operation a certain number of times.

1.1 Analysis of Iterative or Recursive Programs

The concept of iteration is central to programming: many useful operations can only be accomplished by repeating a sequence of computational steps until a certain state is reached. Any sufficiently expressive (i.e. Turing-complete) programming language or computation model includes a construct to perform iterating computations. In imperative languages, this is commonly accomplished with loops, while in functional languages and other paradigms, recursive functions are used instead. The concept of recursion is even more deeply embedded in functional languages, since most of them also define the objects of their computations through the use of recursive data types.

Since the concept of iteration lends such power to programming languages, it should come as no surprise that it is also one of the major sources of difficulty for program analysis. Firstly, the possibility of iterating means that some programs may never reach a state where the iterating computation ends; such a program will not terminate. While the study of termination is an interesting and complex problem for program verification, we will generally leave it aside in this thesis. Instead we focus on the question of partial correctness: in the cases where a program does terminate, what properties does it verify? Even this restricted goal remains very complex. It usually requires techniques related to mathematical induction: for example in many cases it is necessary to use *invariants*, properties of the program that remain true during the iteration process. This usage of invariant is very similar in nature to the way mathematicians commonly use induction hypotheses in proofs.

In the following section, we describe more precisely the techniques traditionally used to verify imperative programs with loops and functional programs with recursive equations.

1.1.1 Use of Invariants in Program Analysis

One the pivotal points for program verification was the introduction of Hoare logic [31] to express the relation between program semantics and specification, thus allowing one to formally express the correctness of programs. The central feature of Hoare logic are triples of the form

$$\{P\}\ \pi\ \{Q\}$$

where P and Q are logical assertions about program states, and π is a program. The triple can be understood as "if the program state satisfies the condition P before the execution of π, it satisfies the condition Q after." By using the appropriate P and Q, it is possible to describe what a program is supposed to do. For example an (incomplete) specification of the binary search procedure could be the following: "if the input list is sorted and contains the element searched (pre-condition), the procedure returns true (post-condition)".

The original presentation of Hoare logic also includes a calculus to prove whether a triple is valid, thus guaranteeing that the program satisfies its specification. Unfortunately most of the inference rules in this calculus require the use of intermediate lemmas, which makes them poorly suited to automated proof search. This problem was partially solved by [22], which provided a calculus based on "predicate transformers" to prove the correctness of programs. The idea of predicate transformers is that given a program π and a post-condition Q, we can compute the weakest (i.e. most general) pre-condition, noted $wp(\pi, Q)$, that must be true at the beginning of the program. What is then left to do to prove program correctness is to show that the actual pre-condition given in the specification is at least as strong as the weakest pre-condition:

$$P \Rightarrow wp(\pi, Q)$$

For the most part, $wp(\pi, Q)$ can be computed automatically by composing the effects of individual statements of the program. However, to compute the weakest pre-condition of a program with a loop, we need to provide an invariant: a property of the loop that is true before the loop begin and is preserved by any iteration of the loop. There are of course many different logical properties that are invariants for a given loop, but for the purpose of program verification, we specifically need one that is strong enough to verify the post-condition.

More generally, invariants are needed for almost all techniques that rely on abstract reasoning about unrestricted program loops (e.g. symbolic execution, code optimization or design by contract). Furthermore, invariants can help in understanding and designing complex algorithms. Yet finding correct and relevant invariants is one of the most difficult steps of verifying a program.

A lot of research has therefore focused on the goal of generating invariants automatically: by analyzing the loop of the body statically (without running the program), it is possible to infer some properties that are preserved iteration after iteration. In Chapter 2, we present an invariant generation technique based on the inference system of a first-order theorem prover.

1.1.2 Analyzing Functional Programs

The techniques employed to verify the correctness of functional programs vary significantly from those mentioned so far. Unlike imperative programs, describing the semantics of a functional program in mathematical terms is quite natural: programs written in functional languages directly correspond to functions, like those commonly used by mathematicians. Since those programs are described by means of oriented equations, it is easy to encode the equations in a logic with equality and universal quantifiers in order to obtain a mathematical representation of the program. The function thus defined is called the denotational semantics of the program, and can easily be embedded in a logical formula to represent the correctness of the program. In this context, there is no need to develop a specialized logic for the purpose of program verification, first-order logic can directly be used. For example, given the denotational semantics of a program f, a predicate P over its input set (the pre-condition) and a predicate Q over its output set (the post-condition), the following formula essentially has the same meaning as a triple in Hoare logic:

$$\forall x, P(x) \Rightarrow Q(f(x))$$

One can then prove that this formula holds, and the program is therefore correct w.r.t. its specification. For that, it is possible to use existing reasoning tools, including automated theorem provers. However to prove the correctness of the program, it may be necessary to reason about properties of the objects manipulated by the program, such as values assumed by the terms x and $f(x)$ in the formula above. This is the motivation behind the work presented in Chapter 3.

1.2 Automated Theorem Proving

The idea of a procedure to decide whether a given mathematical statement is a theorem or not has certainly been a long-standing dream for many mathematicians. In 1928, David Hilbert gave a precise definition of the problem and made it a part of his program to reform the foundation of mathematics: his goal was to find a decision procedure able to prove or disprove any statement in first-order logic. A year later, such an algorithm was discovered for statements about Presburger arithmetic – a limited form of arithmetic which cannot express properties such as divisibility or primeness of numbers.

In spite of this promising start, Hilbert's goal was shortly after proven unattainable by Gödel's incompleteness theorems, as well as the works of Church and Turing on undecidability. More precisely, first-order logic was shown to be only semi-decidable: it is possible to find a procedure that will, for a given first-order formula F, eventually halt and report if F is valid, but may run indefinitely in the case where F is invalid.

While this theoretical limit shows that automated reasoning is a very difficult problem, much effort has been spent to develop techniques to reason efficiently about many problems in first-order logic. The tools that have come out of this research have found applications in computer-verified mathematics as well as in software and hardware verification.

1.2.1 First-order Theorem Provers

One of the first major steps towards the efficient automation of reasoning in first-order logic was the development of the resolution calculus [50]. This calculus includes only a single rule, which relies on the principle of term unification to perform inferences in a way that avoids the combinatorial explosion caused by other methods to reason about quantified formulas. It is refutationally complete, meaning that for any unsatisfiable formula formula, there exists a proof that the formula implies $\perp$ (the truth constant “false”) in the calculus. Since the validity of a formula is equivalent to the unsatisfiability of its negation, theorem provers use this property to find proofs by contradiction: proving that a formula F is valid can be accomplished by systematically searching for a proof that $\neg F$ implies $\bot$.

The resolution calculus was later extended with the paramodulation rule [58] which performs inferences based on the equality predicate. Indeed most problems rely on the notion of equality, but the axiomatization of the equality predicate is both inconvenient (an axiom must be added for every function and predicate symbol in the problem signature) and detrimental to the efficiency of the proof search. By replacing this axiomatization with the paramodulation rule, these problems are avoided. A later improvement was the use of rewriting orderings to order equalities [35, 1]. Ordered paramodulation, also called superposition, produces fewer consequences than standard paramodulation, thus improving efficiency, while preserving the completeness of the system.

The calculi used by modern first-order theorem provers still rely mainly on resolution and superposition. For this reason, they are also often designated superposition theorem provers.

1.2.2 Theory Reasoning

Many mathematical problems are defined in the context of a theory: some symbols of the problem are given a specific meaning. Consider for example the application of program verification: such problems are likely to include terms that are intended to represent numbers, arrays, data structures, etc.

For first-order logic, a natural way to reason in the presence of theories is to add the theory axioms to the assumptions of the problem. This is the approach generally taken in first-order theorem provers, but it may be difficult to implement if the theory has no finite axiomatization. An alternative approach may be to extend the calculus with dedicated rules.

1.2.3 Other Logics and Theorem Provers

While this thesis focuses solely on first-order theorem provers, it is interesting to assess the capabilities of different types of automated provers. The main defining feature of a theorem prover is the logic in which it is able to reason: this affects both the encoding of the problem and the efficiency of the reasoning. For example, propositional logic offers the benefit of decidability, and SAT solvers (e.g. Lingeling [9], MiniSat [25]) can often solve the problem of deciding the satisfiability of a propositional formula very efficiently despite the fact that it is NP-hard in general. However, describing problems in propositional logic may require non-trivial encodings or be simply infeasible. On the other end of the spectrum, higher-order logic or dependent type theory can be used to describe problems using complex mathematical concepts, but reasoning in these logics is complex, making efficient automation difficult to attain.

In this regard, first-order logic offers an interesting compromise: it can describe most problems in a natural way, but is still semi-decidable, and there are efficient tools to reason about it. This makes it an interesting choice of logic for many applications, among which program analysis.

Both SMT solvers (such as Z3 [21] or CVC4 [3]) and first-order theorem provers (Spass [57], E [52], Vampire [40]) can perform automated reasoning in first-order logic, but the way they reason about it differs substantially. SMT solvers combine the approach of SAT solvers for Boolean reasoning and specialized procedures called theory solvers for domain-specific reasoning. As a result, they generally perform well on problems involving theory reasoning, while first-order provers are often more successful where many quantifiers are involved in the problem description. The challenge for both categories of provers is to solve problems that include both quantifiers are theory symbols.

1.3 Contributions of the thesis

This thesis presents contributions to the field of program analysis and verification that rely on the use of a first-order theorem prover.

Both of these contributions come with implementations integrated in the first-order theorem prover Vampire [40]. These tools can be used on their own, or included in a modular verification architecture, to combine the benefits of interactive tools with the convenience of a fully automatic reasoning engine.

Paper 1: Reasoning About Loops Using Vampire in KeY

The symbol elimination method is a novel way to generate invariants, which relies on the consequence finding mechanism provided by first-order theorem provers. It was originally introduced in [38]. In the paper reproduced in this thesis, we present new extensions of the symbol elimination technique:

- A new input format: a guarded command language meant to be used as an intermediate verification language to describe loops in a variety of programming languages
- The ability to specify pre- and post-conditions of the loops to be verified: these can be used to produce stronger invariants, to filter the most relevant invariants among those generated, or even to perform the proof of correctness of the loop directly within the tool, rather than using an external tool
- The integration of our invariant generation tool in the KeY verification framework for the Java programming language, which demonstrates how the guarded command language can be used to describe programs in mainstream languages
- Refinements in the static analysis phase of the symbol elimination process that the quality of invariants generated

Statement of contribution. This paper is co-authored with Laura Kovács and Wolfgang Ahrendt. Simon Robillard is the main author.

It was originally published in the peer-reviewed *20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 20)* and presented in Suva, Fiji. It is reproduced here in an extended version which includes material published in *Proceedings of the 1st and 2nd Vampire workshops.*

Paper 2: Coming to Terms with Quantified Reasoning

Most functional programming languages manipulate data defined with the use of algebraic data types. Term algebras provide a concrete semantics for such data types. The ability to reason efficiently about such algebras is therefore crucial to analyze functional programs and verify their correctness. In the second paper reproduced in this thesis, we present ways to reason about term algebras in a first-order theorem prover. The contributions of this paper include:

- A conservative extension of the theory of term algebras based on a finite number of axioms (whereas the theory itself is not finitely axiomatizable)
- Inference rules dealing specifically with term algebra symbols, which improve the efficiency of reasoning about problems with term algebras

- The implementation of the above in the first-order theorem prover Vampire, making it the first superposition theorem prover able to perform complete reasoning in the theory of term algebras

Statement of contribution. This paper is co-authored with Andrei Voronkov and Laura Kovács. Simon Robillard is the main author.

It has been accepted for publication in the peer-reviewed *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)* and will be presented in Paris, France.

Chapter 2

Reasoning About Loops Using Vampire in KeY

Wolfgang Ahrendt, Laura Kovács and Simon Robillard

Abstract. We describe symbol elimination and consequence finding in the first-order theorem prover Vampire for automatic generation of quantified invariants, possibly with quantifier alternations, of loops with arrays. Unlike the previous implementation of symbol elimination in Vampire, our work is not limited to a specific programming language but provides a generic framework by relying on a simple guarded command representation of the input loop. We also improve the loop analysis part in Vampire by generating loop properties more easily handled by the saturation engine of Vampire. Our experiments show that, with our changes, the number of generated invariants is decreased, in some cases, by a factor of 20. We also provide a framework to use our approach to invariant generation in conjunction with pre- and post-conditions of program loops. We use the program specification to find relevant invariants as well as to verify the partial correctness of the loop. As a case study, we demonstrate how symbol elimination in Vampire can be used as an interface for realistic imperative languages, by integrating our tool in the KeY verification system, thus allowing reasoning about loops in Java programs in a fully automated way, without any user guidance.

Originally published in *20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 9450 of *LNCS*, pages 434–443. Springer, 2015.

2.1 Introduction

Reasoning about the (partial) correctness of programs with loops requires loop invariants. Typically, loop invariants are provided by the user as annotations to the program, see e.g. [7, 18, 42]. Providing such annotations requires a considerable amount of work by highly qualified personnel and often makes program analysis prohibitively expensive. Therefore, automation of invariant generation is invaluable in making program analysis scale to large, realistic examples.

In [38], the symbol elimination method for generating invariants was introduced. The approach uses first-order theorem proving, in particular the Vampire prover [40]. Symbol elimination allows the generation of quantified invariants, possibly with quantifier alternations, for programs with unbounded data structures, such as arrays. While experiments of invariant generation in Vampire show that symbol elimination generates non-trivial invariants, the initial implementation [24] of program analysis and invariant generation in Vampire has various disadvantages: it can only be used with programs written in C, the number of generated invariants is too large, and generating relevant invariants did not take into account the program specification. Moreover, the process of invariant generation was not integrated, nor evaluated in a verification framework, making it hard to assess the quality and practical impact of invariant generation by symbol elimination. In this paper we address these limitations, as follows.

We provide a new and fully automated tool for invariant generation, by using symbol elimination in Vampire. To this end, we re-implemented program analysis and invariant generation in Vampire. Our implementation is fully compatible with the most recent development changes in Vampire. It is designed to be independent of any particular programming language: inputs to our tool are program loops written in a simple guarded command language. Details on the guarded language representation used by our work are given in Section 2.2, whereas symbol elimination in Vampire is described in Section 2.3.

Our work is compatible with recent developments in Vampire. In order to take advantage of these changes, the program analysis phase of symbol elimination – during which some lightweight static analysis techniques are used as a first step to symbol elimination – has been modified and improved. We propose new ways for extending quantified loop properties describing valid loop properties, by simplifying the properties over array updates and next state relations. These improvements result in properties that are more easily handled by the inference engine of Vampire; they are detailed in Section 2.4. We also extended symbol elimination by taking into consideration also the loop specification (contract), which may optionally be given by the user in the form of pre- and post- conditions. If available, pre-conditions are used to derive more precise invariants, and post-conditions can be used to select the subset of invariants relevant to the verification task. We also turn symbol elimination into an automatic (incomplete) way to directly prove the correctness of the loop w.r.t. to a contract. Our work provides an alternative to Hoare-style verification of loops and avoids the need for explicitly stated invariants. Generating relevant invariants and proving partial correctness of loops using symbol elimination are presented in Section 2.5.

Reasoning about real programming languages poses several challenges, e.g. using machine integers instead of mathematical ones or reasoning about out-of-bound array accesses. In order to showcase the relevance of our implementation in real applications, we integrated our approach to loop reasoning in Vampire into the KeY verification system [7], thus allowing automatic reasoning about loops in Java programs, as demonstrated in Section 2.6. We experimentally evaluate invariant generation in Vampire on realistic examples, the results are given in Section 2.7.

The main advantage of our tool comes with its full automation for generating invariants, possibly with quantifier alternations. Unlike [29, 27], where user-given invariant templates are used, we require no user guidance and infer first-order invariants with arbitrary quantifiers. Contrary to [17], we do not use specialized abstract domains, but use saturation theorem proving to generate quantified invariants. Theorem proving, in the form of SMT solving, is also used in [41] to automatically compute loop invariants, however only with universal quantifiers.

In order to achieve the above improvements and extensions to symbol elimination, we completely re-implemented symbol elimination in Vampire. Our work provides a new and fully automated tool for invariant generation and proving partial correctness of loops. Our implementation required 3000 lines of C++ code, is fully compatible with the recent version of Vampire (version 3.0), and is available at `www.cse.chalmers.se/~simrob`. The integration of Vampire with KeY required about 1000 lines of Java code.

2.2 Input Language

2.2.1 Syntax

Inputs to our approach are loops with nested conditionals, written in a simple guarded command language. Loops may contain scalar variables and arrays ranging over (unbounded) integers. In what follows, we use upper case letters $A, B, C, \dots$ to denote array variables and lower case letters $a, b, c, \dots$ for scalars. We use standard arithmetical function symbols $+, -, \cdot, \div$ and predicate symbols $\leq, \geq$. We write $A[p]$ to mean (an access to) the array element at position p in the array A.

We describe loops by a *loop condition* and an ordered collection of *guarded statements*; the loop condition is a quantifier-free Boolean formula over program variables. A guarded statement is a pair of a *guard* (also a Boolean formula) and a collection of assignments. In our setting, a guarded statement cannot contain two assignments to the same scalar variable v. If two array assignments $A[i] := e$ and $A[j] := e'$ occur in a guarded statement, the condition $i \neq j$ is added to the guard. These two restrictions ensure that each location is modified at most once by a given guarded statement.

In addition to the loop itself, pre- and post-conditions can also be specified, using the keywords `requires` and `ensures`, respectively. Pre- and post-conditions are Boolean formulas over program variables, possibly with quantifiers.

Figure 2.1 gives an example of a loop using the syntax supported by our work.

2.2.2 Semantics

We define the semantics of the guarded command language by the notion of *program states* mapping scalar variables to values of the correct type and arrays to functions. Note that arrays bounds are not dealt with in the semantics: in a given state, an array storing values of type τ is treated as a total function of type $\mathbb{Z} \to \tau$. Array bounds checking may easily be encoded with the help of guards if required. Evaluation of program expressions in a given state is done in the standard way.

In our setting, there is exactly one program state for each loop iteration. The symbol n is used to denote the upper bound on the number of loop iterations, so that for any loop iteration i we have $0 \leq i < n$. We write σ_0 and σ_n to respectively speak about the initial and final state of the loop. If the loop condition is valid in a given program state σ_i, the first guarded

```
requires (k == 0);
ensures forall int p, (0 <= p & p < n) ==>
                        (A[p] >= B[p]
                        & A[p] >= C[p]
                        & (A[p] == B[p] | A[p] == C[p]));
while (k < n) do
  :: B[k] >= C[k] -> A[k] = B[k]; k = k + 1;
  :: true         -> A[k] = C[k]; k = k + 1;
od
```

Figure 2.1. Example of an input to our work. This example loop is composed of two guarded statements; it computes the maximum of elements in arrays B and C at every position and writes it in the corresponding position in the array A. The program specification is given by the pre- (`requires`) and post-conditions (`ensures`).

statement whose guard is valid is executed: its assignments are applied simultaneously to σ_i, yielding the state σ_{i+1}. For example, executing the guarded statement

```
true -> x = 0; y = x;
```

in a state where $x = 1$ holds, yields a state in which $y = 1$ and not $y = 0$.

If the loop condition is not valid, or if none of the guards hold, the loop is terminated: σ_i becomes the final state of the loop σ_n.

Note that while these semantics are deterministic, our method for invariant generation could be adapted to work with non-deterministic semantics with only minor changes.

2.3 Invariant Generation Using Symbol elimination

The symbol elimination method aims at producing invariants for a given loop, i.e. first-order formulas in a language of assertions $\mathcal{L}_{asrt}$ that hold at arbitrary iterations of the loop. The central idea of symbol elimination is to use formulas expressed in a language of extended expressions $\mathcal{L}_{extd}$ during intermediate steps of the procedure. This language can express richer properties of the loop than is possible with $\mathcal{L}_{asrt}$: while any formula using symbols in $\mathcal{L}_{asrt}$ has a semantic equivalent in $\mathcal{L}_{extd}$, the converse is not true. During the procedure, we first deploy static analysis techniques to extract properties of the loop expressed in $\mathcal{L}_{extd}$. In a second phase, we use saturation theorem proving to discover consequences of those properties that can be expressed using only symbols from $\mathcal{L}_{asrt}$. Such properties are loop invariants.

In this section, we define $\mathcal{L}_{asrt}$ and $\mathcal{L}_{extd}$, then describe the symbol elimination procedure to generate loop invariants. The definitions assume a given loop, in particular they depend on the set of program variables used within that loop.

2.3.1 Assertions

We define $\mathcal{L}_{asrt}$, the language of assertions, as follows. For each scalar variable `v` of type τ in the loop, $\mathcal{L}_{asrt}$ includes two symbols $v : \tau$ and $v_{init} : \tau$. For each array A storing values of type τ, $\mathcal{L}_{asrt}$ includes a function symbol of type $\mathbb{Z} \to \tau$. Interpretation of a formula using symbols in $\mathcal{L}_{asrt}$ depends on a given program state σ. The symbol v is interpreted as the value of the program variable `v` in that state, while v_{init} is interpreted as the value of that variable at the start of the loop.

An invariant is a formula that uses symbols from $\mathcal{L}_{asrt}$ and is valid for any state σ_i. The pre- and post-conditions of the loops are formulas in $\mathcal{L}_{asrt}$ that are required to hold at the initial state σ_0 and the final state σ_n, respectively.

2.3.2 Extended Expressions

Unlike $\mathcal{L}_{asrt}$, symbols in $\mathcal{L}_{extd}$ do not depend on a particular program state for interpretation. Formulas using such symbols can express properties of the loop at arbitrary states, such as the relation between two successive program states.

For every variable `v` of type τ, $\mathcal{L}_{extd}$ includes a function of type $\mathbb{Z} \to \tau$[1]. For convenience, applications of these functions are noted $v^{(i)}$; they are interpreted as the value of `v` in the state σ_i. For each array A, $\mathcal{L}_{extd}$ includes a function of type $\mathbb{Z} \times \mathbb{Z} \to \tau$. Similarly, we use the notation $A^{(i)}[p]$ to represent the value stored at position p after the ith iteration. We call $v^{(i)}$ and $A^{(i)}[p]$ *extended expressions.* Note for any program expression E, we can build a term (or predicate, in the case of Boolean program expressions) by systematically replacing each variable by its extended expression. We may simply abbreviate such construction $E^{(i)}$.

$\mathcal{L}_{extd}$ also includes the symbol n which denotes the upper bound on the number of loop iterations. Formulas in $\mathcal{L}_{extd}$ that are valid for a given

[1]The type $\mathbb{N} \to \tau$ would perhaps be more accurate, but in practice it is more efficient to add predicates enforcing the non-negativity where needed.

loop are called *extended loop properties.*

The following semantic equivalences relate $\mathcal{L}_{asrt}$ and $\mathcal{L}_{extd}$

$$\begin{array}{rcl} v^{(0)} & \equiv & v_{init} \\ v^{(n)} & \equiv & v \\ A^{(0)}[p] & \equiv & A_{init}[p] \\ A^{(n)}[p] & \equiv & A[p] \end{array}$$

2.3.3 Loop Analysis and Symbol Elimination

In the first step of our invariant generation procedure, we perform simple static analysis to generate extended loop properties. For example, analyzing the program in Figure 2.1 would lead to generating the following property:

$$(\forall i)(0 \leq i < n \implies k^{(i+1)} = k^{(i)} + 1)$$

This property, which describes the assignment to the variable `k` at each iteration, is added to the list of extended properties as an assumption. A comprehensive description of the analysis performed by our tool and the resulting properties is given in Section 2.4. Note that this phase is quite flexible, and additional properties (user knowledge, invariants generated by other tools...) could potentially be added to the list of extended properties.

While the properties extracted during that phase are valid at arbitrary loop iterations, they are not yet invariants as they use symbols extended expressions, symbols that are not in $\mathcal{L}_{asrt}$. The next step in our invariant generation process is to eliminate symbols that are not in $\mathcal{L}_{asrt}$. This is done by generating formulas that only use symbols from $\mathcal{L}_{asrt}$ and are logical consequences of the properties in $\mathcal{L}_{extd}$. To this end we use the prover to perform symbol elimination and generate invariants in $\mathcal{L}_{asrt}$. For more details on symbol elimination we refer to [39].

2.4 Extracting Loop Properties

In this section, we list the properties extracted from the loop during the first phase of invariant generation. It is important to note that there is no definitive way to chose which properties must be extracted from the loop, as long as those properties are indeed consequences of the loop semantics. The strength and the formulation of the properties play a great role in the quality of the invariants produced.

2.4.1 Properties of Scalar Variables

Program variables that are never updated by the loop body are treated as constant symbols during the analysis. For variables that are updated, simple static analysis techniques are used to characterize the behavior of those updates.

Let us call a scalar variable v *increasing* if, for all possible computations of the loop, it has the property

$$(\forall i)(0 \leq i < n \implies v^{(i+1)} \geq v^{(i)})$$

Similarly, we call v *decreasing* if

$$(\forall i)(0 \leq i < n \implies v^{(i+1)} \leq v^{(i)})$$

A variable is said to be *strict* if it is modified at every iteration, i.e.

$$(\forall i)(0 \leq i < n \implies v^{(i+1)} \neq v^{(i)})$$

Finally a variable is called *dense* if its value is increased or decreased by at most one during any iteration

$$(\forall i)(0 \leq i < n \implies |v^{(i+1)} - v^{(i)}| \leq 1)$$

Having detected those properties of the variables, the following properties are added to the list of extended properties:

1. If v is increasing, strict and dense, we add the property:

$$(\forall i)(v^{(i)} = v^{(0)} + i)$$

2. If v is increasing and strict, but not dense, we add the property:

$$(\forall i)(\forall j)(j > i \implies v^{(j)} > v^{(i)})$$

3. If v is increasing but not strict, we add the property:

$$(\forall i)(\forall j)(j \geq i \implies v^{(j)} \geq v^{(i)})$$

4. If v is increasing and dense, but not strict, we add the property:

$$(\forall i)(\forall j)(j \geq i \implies v^{(i)} + j \geq v^{(j)} + i)$$

Similar properties, with the required modifications, are generated for decreasing variables.

2.4.2 Update Properties of Arrays

In order to describe the behavior of arrays, for each array we analyze the guarded statements to collect:

1. the conditions under which the array is updated at position p by the value v during iteration i. Let us consider the example in Figure 2.1, for the array A (the only one to be updated), these conditions are

$$\begin{array}{ll} & (0 \leq i < n \ \wedge \ B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \ \wedge \ v = B^{(i)}[k^{(i)}] \ \wedge \ p = k^{(i)}) \\ \vee & (0 \leq i < n \ \wedge \ \neg B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \ \wedge \ v = C^{(i)}[k^{(i)}] \ \wedge \ p = k^{(i)}) \end{array}$$

   which we denote $\mathit{upd}_A(i,p,v)$

2. the conditions under which the array is updated at position p during iteration i, by any value. For the same example, they are

$$\begin{array}{ll} & (0 \leq i < n \ \wedge \ B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \ \wedge \ p = k^{(i)}) \\ \vee & (0 \leq i < n \ \wedge \ \neg B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \ \wedge \ p = k^{(i)}) \end{array}$$

   these are noted $\mathit{upd}_A(i,p)$

After this analysis we can express the following properties of the array:

1. if the array is never updated at a position p, the value at this position remains constant

$$(\forall i\, p) \left(\neg \mathit{upd}_A(i,p) \implies B^{(n)}[p] = B^{(0)}[p]\right)$$

2. if the array is updated only once at a position p, the value associated with this update is the final value

$$(\forall i\, j\, p\, v) \left(\mathit{upd}_A(i,p,v) \wedge (\mathit{upd}_A(j,p) \implies j = i) \implies B^{(n)}[p] = v\right)$$

Note that compared to [38], the second property has been modified as it used to read

$$(\forall i\, j\, p\, v) \left(\mathit{upd}_A(i,p,v) \wedge (\mathit{upd}_A(j,p) \implies j \leq i) \implies B^{(n)}[p] = v\right)$$

While less general, the new property is more easily handled by the prover, since equality is a built-in predicate of the superposition calculus used by Vampire.

In previous implementations, predicate symbols corresponding to upd_A were used in both properties, and assumptions giving the predicate definitions were also added. Those predicate symbols were then eliminated. The new tool replaces every occurrence of the predicate symbol directly by its definition, thus increasing efficiency and the quality of invariants produced.

2.4.3 Assignments

The relation between two consecutive states, and in particular the effects of assignments on states, can be described by extended expressions.

For the program in Figure 2.1, the following two properties (one for each guarded statement) are extracted and added to the extended properties.

$$\begin{aligned}
(\forall i)(0 \leq i < n \wedge B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \implies\ & A^{(i+1)}[k^{(i)}] = B^{(i)}[k^{(i)}] \\
& \wedge k^{(i+1)} = k^{(i)} + 1 \\
(\forall i)(0 \leq i < n \wedge \neg B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \implies\ & A^{(i+1)}[k^{(i)}] = C^{(i)}[k^{(i)}] \\
& \wedge k^{(i+1)} = k^{(i)} + 1
\end{aligned}$$

2.4.4 Additional Properties

Finally the property indicating that the loop condition and one guard must hold at any given iteration is added to the assumptions.

$$(\forall i)(0 \leq i < n \implies \bigvee_j G_j^{(i)} \wedge C^{(i)})$$

In the original description of the symbol elimination method, arithmetic function and predicate symbols were introduced as needed and given an axiomatization. This is no longer necessary, as we use the default symbols now provided by Vampire. At the moment, any arithmetic reasoning in Vampire is still based on axiomatic theories, but symbol elimination would directly benefit from any further development concerning arithmetic reasoning in Vampire.

As noted before, the list of extended properties is not definitive. This makes our method flexible, as *ad-hoc* properties can potentially be added to the assumptions, whether it be user knowledge or properties gathered by other invariant generation techniques (e.g. [27, 29])

2.5 Loop Contract and Correctness

Previous works on symbol elimination [32, 24] report every property discovered during symbol elimination. This often results in hundreds of clauses being reported to the user in a few seconds, many of which are consequences of each other. To address this issue, a post-processing step was added during which some redundant clauses were eliminated. However minimizing a set of first-order clauses is an undecidable problem. Even if a minimal set of clauses is obtained, previous works on symbol elimination do not take into account a verification contract (specification) for analyzing and verifying loops. Therefore there is no realistic way to assess the quality of generated invariants in the process of verification. We also note that symbol elimination generates invariants that hold at any iteration of the loop, but may not be inductive. Using non-inductive invariants makes software verification harder.

By enabling the user to specify a post-condition of the loop, and using it to select relevant invariants within the set produced by symbol elimination, we address those issues. Unlike previous works, our work enables the user to specify optional pre- and post-conditions for the loop under analysis, using the keywords `requires` and `ensures`, respectively. They are expressions in $\mathcal{L}_{asrt}$ (quantified Boolean formulas over program variables).

2.5.1 Pre-conditions

Recall that any expression in $\mathcal{L}_{asrt}$ can be translated to an expression $\mathcal{L}_{extd}$. Pre-conditions given by the user as expressions in $\mathcal{L}_{asrt}$ are simply translated to $\mathcal{L}_{extd}$ and added to the extended properties. For example this precondition

```
requires forall int p, 0 <= p & p < l ==> A[p] != 0
```

results in the following property being added to the extended properties:

$$(\forall i)(0 \leq p < l \implies A^{(0)}[p] \neq 0)$$

Such additional information enables symbol elimination to derive stronger invariants.

2.5.2 Invariant filtering

Given a loop condition C, a post-condition P and a set of invariants $I_1, \ldots, I_k$ produced by symbol elimination, we attempt to prove P under the assumptions $I_1 \wedge \cdots \wedge I_k \wedge \neg C$. If the refutation proof succeeds, we can select the subset of invariants that were effectively used: they are among the leaves of the proof tree.

This filtering process is carried out in parallel of symbol elimination. One instance S_{gen} of the saturation algorithm is ran to generate invariants, possibly with a time limit. Another instance S_{filter} is started on a different thread, it initially tries to prove P assuming only $\neg C$. Each time a new invariant is discovered by S_{gen}, it is added to the list of assumptions in S_{filter}, and the proof attempt is restarted. This way, the process can stop as soon as the set of discovered invariants is strong enough to imply the post-condition. If the time limit of S_{gen} is reached however, the whole process is aborted.

This filtering mechanism also provides a good heuristic to select an inductive invariant. While this is not always true, our experiments (Section 2.7) show that the set of invariant selected is usually inductive.

2.5.3 Direct Proof of Correctness

During invariant filtering, we use invariants, which are consequences of the extended properties, to prove the post-condition. In any case where this succeeds, the post-condition is also a consequence of the extended properties.

As an alternative to invariant filtering, our tool offers the option of omitting the symbol elimination stage and proving the post-condition from the extended properties themselves. In this setting, no invariants are used or reported. This provides an alternative to classic Hoare-style verification of loops which, while incomplete, is fully automatic.

Finding a direct proof of correctness of the loop is faster than performing invariant filtering (see Section 2.7) and should succeed for every program where invariant filtering succeeds. In some cases, due to the fact that extended properties are stronger than the invariants they imply, a direct proof may even succeed where invariant filtering does not.

2.6 Integration with the KeY System

While previous implementations of symbol elimination [32, 24] used a syntax similar to the C programming language, only a subset of C programs could be analyzed. Many aspects of the semantics of C were not taken into account.

By using a guarded command language, our implementation clarifies the semantics of the input language. It is consequently easier to use the guarded command language as an representation of the semantics of a program given in another language. In our experiments, we demonstrated this possibility by using the KeY verification system [7] to translate Java programs with loops into our guarded command language.

In this section we describe the integration of our invariant generation method to the KeY verification system. We discuss the modularity afforded by our tool and its applicability to realistic examples.

2.6.1 Dynamic Logic

KeY [7] is a deductive verifier for functional correctness properties of Java source code. It uses dynamic logic (DL), a modal logic for reasoning about programs. DL extends first-order logic with the modality $[p]\phi$, where p is a program and ϕ is another DL formula; $[p]\phi$ is true in a state from which running the program p, in case of termination, results in a state where ϕ is true.

2.6.2 Symbolic Execution

KeY uses symbolic execution. For that, DL is extended by "explicit substitutions", called updates. During the symbolic execution of a program p, the effects of p are *gradually*, from the front, turned into updates, and applied to each other. After some proof steps, an intermediate proof node may look like $\Gamma \vdash \mathcal{U}[p']\phi$, where a certain prefix of p has turned into update $\mathcal{U}$, representing the effects so far, while a "remaining" program p' is yet to be executed. Note that most proofs branch over case distinctions, usually triggered by Boolean expressions in the source code. The semantics of the $\Gamma \vdash \mathcal{U} \ldots$ part of a sequent is in many ways close to those of a guarded assignment in Vampire's programming model. Γ can be understood in the same way as Vampire's guards, while updates and Vampire's assignments share the same semantics of simultaneous application. We therefore use symbolic execution to perform the translation of Java programs to Vampire's guarded command language, as follows. Given a program p containing a loop, we apply symbolic execution to all instructions preceding the loop, leading to a sequent:

$$\Gamma \vdash \mathcal{U}[\texttt{while (}se\texttt{) \{ }b\texttt{ \}; }p']\phi$$

where *se* is a *side effect-free* Java expression[2]. As a step towards employing Vampire, we launch a separate KeY proof at this point, starting from the sequent: $\Gamma, se' \vdash \mathcal{U}\mathcal{V}[b]\psi$. Here, se' is the result of applying $\mathcal{U}$ to se, $\mathcal{V}$ is an anonymizing update [8] meant to remove information on variables modified by the loop body b, and ψ is an uninterpreted predicate. This side proof is not meant to prove anything, but only to carry out symbolic execution of *any* iteration (hence $\mathcal{V}$) of the loop body b. Since ψ is uninterpreted, the side proof started with this sequent cannot be completed; however, assuming that they do not themselves contain an unannotated loop, instructions of b can be symbolically executed. We are then left with a proof tree containing one or more open nodes, all of which have the form: $\Gamma' \vdash \{v_1 := e_1; \ldots; v_k := e_k\}[\,]\psi$. Each of these nodes corresponds to a possible path of symbolic execution, which is transformed into a guarded assignment:

```
Gamma' -> v1 = e1; ...  ; vk = ek;
```

Currently this translation is not complete: if a nested loop is present within the loop body b, its translation requires it to be annotated with an invariant. Other language features, such as exception throwing and catching, abrupt termination and heap-related properties, among others, are not supported. Many of those aspects can be easily and efficiently encoded by introducing additional Boolean variables in the program, however at the time of writing, Boolean variables are not supported by our tool. This support should be added soon, thanks to the recent introduction of a first-class Boolean sort in Vampire [37].

2.6.3 Integration

If the user is satisfied with delegating the proof of correctness of the loop to Vampire, when the Vampire proof succeeds, it is possible to simply complete the main KeY proof by applying a dedicated axiomatic rule. If more transparency is desired, it is of course possible to import the invariants produced by Vampire (with or without invariant filtering) into KeY and use these invariants in the KeY inference rule normally used with user-annotated invariants. KeY will however need to prove that the invariants generated by Vampire are indeed invariants.

[2]More complex Boolean expressions are transformed away by KeY rules.

2.7 Experimental Results

We evaluated our tool on 20 challenging array benchmarks taken from academic papers [23, 24] and the C standard library. Our benchmarks are listed in Table 2.1. The program `absolute` computes the absolute value of every element in an array, whereas `copy`, `copyOdd` and `copyPositive` copy (some) elements of an array to another. The example `find` searches for the position of a certain value in an array, returning -1 if the value is absent. The program `findMax` locates the maximum in an unsorted array. The examples `init`, `initEven`, and `initPartial` initialize (some) array elements with a constant, whereas `initNonConstant` sets the value of array elements to a value depending on array positions. `inPlaceMax` replaces every negative value in an array by 0, and `max` computes the maximum of two arrays at every position. `mergeInterleave` interleaves the content of two arrays, whereas `partition` copies negative and non-negative values from a source array into two different destination arrays. `reverse` copies an array in reverse order, and `swap` exchanges the content of two arrays. Finally, `strcpy` and `strlen` are taken from the standard C library. Each benchmark contains a loop together with its specification. Our benchmarks are available at the URL of our tool.

We carried out two sets of experiments: (i) invariant generation, by using a guarded command representation of the benchmarks as inputs to our tool; (ii) loop analysis of realistic Java programs, by specifying the examples as Java methods with JML contracts as inputs to our tool and using our integration of invariant generation in KeY. All experiments were performed on a computer with a 2.1 GHz quad-core processor and 8GB of RAM.

Table 2.1 summarizes our results. The second column indicates whether the benchmark loops contain conditionals. Column Δ_{direct} shows the time required to prove the partial correctness of the benchmarks, by proving the loop specification from the extended properties generated by program analysis in Vampire. On the other hand, column Δ_{filter} gives the time needed by our tool to generate the relevant invariants from which the loop post-condition can be proved. The time results are given in seconds. Where no time is given, a correctness proof/filtering of relevant invariants was not successful. Column N_5 shows the number of all invariants generated by our tool with a time limit of 5 seconds (before filtering of relevant invariants). The figure listed in parentheses gives the number of invariants produced by a previous implementation [24] of invariant generation in Vampire. Finally, column N_{filter} reports the

Table 2.1. Experimental results on loop reasoning using Vampire.

Name	Cond.	Δ_{direct}	Δ_{filter}	N_5	N_{filter}
absolute	yes	0.271	2.358	19	3
copy	no	0.043	2.194	9 (37)	1
copyOdd	no	0.122	2.090	9 (214)	1
copyPartial	no	0.042	3.145	9	1
copyPositive	yes			9	
find	yes			123	
findMax	yes			3	
init	no	0.035	2.059	9 (35)	1
initEven	no			10	
initNonConstant	no	0.114	2.054	9 (104)	1
initPartial	no	0.042	3.129	9	1
inPlaceMax	yes			39	
max	yes	0.696	3.535	20	2
mergeInterleave	no			20	
partition	yes			164 (647)	
partitionInit	yes			98 (169)	
reverse	no	0.038		9 (42)	
strcpy	no	0.036	2.126	9	1
strlen	no	0.018	2.023	2 (26)	1
swap	no			26	

number of invariants selected as relevant invariants; the conjunction of these invariants is the relevant invariant from which the loop specification can be derived.

2.7.1 Invariant Generation

Note that for all examples, our tool successfully generated quantified loop invariants. Moreover, when compared to the previous implementation [24] of invariant generation in Vampire, our tool brings a significant performance increase: in all examples where the implementation of [24] succeeded to generate invariants, the number of invariants generated by our tool is much less than in [24]. For example, in the case of the program `copyOdd`, the number of invariants generated by our tool has decreased by a factor of 24 when compared to [24]. This increase in performance is due to our improved program analysis for generating extended loop properties. For the examples where the number of invariants generated by [24] is missing, the approach of [24] failed to generate quantified loop invariants over arrays. We also note that invariants generated by [24] are logical consequences of the invariants generated by our tool.

2.7.2 Invariant Filtering

When evaluating our tool for proving correctness of the examples, we succeeded for 11 examples out of 19, as shown in column Δ_{direct} of Table 2.1. For these 11 examples, the partial correctness of the loop was proved by Vampire by using the extended loop properties generated by our tool. Further, for 10 out of these 11 examples, our tool successfully selected the relevant invariants from which the loop specification could be proved. For the example `reverse` the relevant invariants could not be selected within a 5 seconds time, even though the partial correctness of the loop was established using the extended properties of the loop. The reason why the relevant invariants were not generated lies in the translation of the Java method into our guarded command representation: due to the limited representation of heap-related properties, the post-condition given to Vampire is weaker than the original proof obligation in KeY. This causes the invariant relevance filter to miss properties required to carry out the proof within KeY, even though the relevant invariants were generated by our tool.

When analyzing the 9 examples for which our tool failed to generate relevant invariants and to prove partial correctness, we noted that these examples involve non-trivial arithmetic and array reasoning. We believe that improving reasoning with full first-order theories in Vampire would allow us to select the relevant invariants from those generated by our tool.

2.8 Conclusion

We provide a new and fully automated tool for invariant generation, by re-implementing and improving program analysis and symbol elimination in Vampire. One of these improvements is the dedicated parser for the guarded command language, which can now be used a simple way to describe the semantics of a loop. We also introduce a number of simplifications during the generation of extended properties of loops, leading to an increased quality in the invariants produced. We allow the possibility of specifying a verification contract for the loop being analyzed, and we add a filtering stage to output only invariants that are relevant to the partial correctness of the loop w.r.t. to that contract. We also extend symbol elimination to directly prove partial correctness of loops, without the need for explicitly stating invariants. We experimentally evaluated our tool on a number of examples. We integrated our tool with the KeY verification system, allowing automatic reasoning about realistic Java programs using first-order proving. We experimentally evaluated our tool on a number of examples coming from KeY.

For future work, we intend to improve theory reasoning in Vampire; this should benefit program analysis as well as more traditional applications of the theorem prover. The analysis of programs that we perform generates first-order problems, which we believe are challenging benchmarks for reasoning with quantifiers and theories. We intend to add these examples to the CASC theorem proving competition [53]. We are also interested in analyzing more complex programs and support the translation of the full semantics of a programming language such as Java into our program analysis framework. For doing so, new features and extensions of the TPTP language supported by first-order theorem provers are needed, for example the use of a first class Boolean sort as described in [37]. Finally, in order to target a greater number of programming languages, it would be useful to provide a front-end to an intermediate verification language, e.g. Boogie [2].

CHAPTER 3

Coming to Terms with Quantified Reasoning

Laura Kovács, Simon Robillard and Andrei Voronkov

Abstract. The theory of finite term algebras provides a natural framework to describe the semantics of functional languages. The ability to efficiently reason about term algebras is essential to automate program analysis and verification for functional or imperative programs over algebraic data types such as lists and trees. However, as the theory of finite term algebras is not finitely axiomatizable, reasoning about quantified properties over term algebras is challenging.

In this paper we address full first-order reasoning about properties of programs manipulating term algebras, and describe two approaches for doing so by using first-order theorem proving. Our first method is a conservative extension of the theory of term algebras using a finite number of statements, while our second method relies on extending the superposition calculus of first-order theorem provers with additional inference rules.

We implemented our work in the first-order theorem prover Vampire and evaluated it on a large number of algebraic data type benchmarks, as well as game theory constraints. Our experimental results show that our methods are able to find proofs for many hard problems previously unsolved by state-of-the-art methods. We also show that Vampire implementing our methods outperforms existing SMT solvers able to deal with algebraic data types.

To appear in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM, 2017.

3.1 Introduction

Applications of program analysis and verification often require generating and proving properties about algebraic data types, such as lists and trees. These data types (sometimes also called recursive or inductive data types) are special cases of term algebras, and hence reasoning about such program properties requires proving in the first-order theory of term algebras. Term algebras are of particular importance for many areas of computer science, in particular program analysis. Terms may be used to formalize the semantics of programming languages [28, 13, 16]; they can also themselves be the object of computation. The latter is especially obvious in the case of functional programming languages, where algebraic data structures are manipulated. Consider for example the following declaration, in the functional language ML:

```
datatype nat = zero | succ of nat;
```

Although the functional programmer calls this a data type declaration, the logician really sees the declaration of an (initial) algebra whose signature is composed of two symbols: the constant *zero* and the unary function *succ*. The elements of this data type/algebra are all ground (variable-free) terms over this signature, and programs manipulating terms of this type can be declared by means of recursive equations. For example, one can define a program computing the addition of two natural numbers by the following two equations:

```
add zero x = x
add (succ x) y = succ (add x y)
```

Verifying the correctness of programs manipulating this data type usually amounts to proving the satisfiability of a (possibly quantified) formula in the theory of this term algebra. In the case of the program defined above, a simple property that one might want to check is that adding a non-zero natural number to another results in a number that is also different from zero:

```
x ≠ zero ∨ y ≠ zero ⇒ add x y ≠ zero
```

Note that depending on the semantics of the programming language, there may exist cyclic terms such as the one satisfying the equation $x \approx succ(x)$, or even infinite terms, but in a strictly evaluated language, only finite non-cyclic terms lead to terminating programs. Since program verification is in general concerned with program safety and termination, it is desirable to consider in particular the theory of finite term algebras, denoted by $\mathcal{T}_{FT}$ in the sequel.

The full first-order fragment of $\mathcal{T}_{FT}$ is known to be decidable [44]. One may hence hope to easily automate the process of reasoning about properties of programs manipulating algebraic data types, such as lists and trees, corresponding to term algebras. However, properties of such programs are not confined strictly to $\mathcal{T}_{FT}$ for the following reasons: program properties typically include arbitrary function and predicate symbols used in the program, and they may also involve other theories, for example the theory of integer/real arithmetic. Decidability of $\mathcal{T}_{FT}$ is however restricted to formulas that only contain term algebra symbols, that is, uninterpreted functions, predicates and other theory symbols cannot be used. If this is not the case, non-linear arithmetic could trivially be encoded in $\mathcal{T}_{FT}$, implying thus the undecidability of $\mathcal{T}_{FT}$. Due to the decidability requirements of $\mathcal{T}_{FT}$ on the one hand, and the logical structure of general program properties over term algebras on the other hand, decision procedures based on [44] for reasoning about programs manipulating algebraic data cannot be simply used. For the purpose of proving program properties with symbols from $\mathcal{T}_{FT}$, one needs more sophisticated reasoning procedures in extensions of $\mathcal{T}_{FT}$.

For this purpose, the works of [5, 49] introduced decision procedures for various fragments of the theory of term algebras; these techniques are implemented as satisfiability modulo theory (SMT) procedures, in particular in the CVC4 SMT solver [3]. However, these results target mostly reasoning in quantifier-free fragments of term algebras. To address this challenge and provide efficient reasoning techniques with both quantifiers and term algebra symbols, in this paper we propose to use first-order theorem provers. We describe various extensions of the superposition calculus used by first-order theorem provers and adapt the saturation algorithm of theorem provers used for proof search.

Theory-specific reasoning in saturation-based theorem provers is typically conducted by including the theory axioms in the set of input formulas to be saturated. Unfortunately a complete axiomatization of the theory of term algebras requires an infinite number of sentences: the *acyclicity rule*, which ensures that a model does not include cyclic terms, is described by an infinite number of inequalities $x \not\simeq f(x)$, $x \not\simeq f(f(x)), \ldots$ This property of term algebras prevents us from performing theory reasoning in saturation-based proving in the usual way.

As a first attempt to remedy this state of affairs, in this paper we present a conservative extension of the theory of term algebras that uses a finite number of sentences (Section 3.5). This extension relies on the addition of a predicate to describe the "proper subterm" relation between terms. This approach is complete and can easily be used in any first-order theorem prover without any modification.

Unfortunately, the subterm relation is transitive, so that the number of predicates produced by saturation quickly becomes a burden for any prover. To improve the efficiency of the reasoning, we offer an alternative solution: extending the inference system of the saturation theorem prover with additional rules to treat equalities between terms (Section 3.6).

We implemented our new inference system, as well as the subterm relation, in the first-order theorem prover Vampire [40]. We tested our implementation on two sets of benchmarks. We used 4170 problems describing properties of functional programs manipulating algebraic data types; these problems were taken from [49]. This set of examples were generated using the Isabelle inductive theorem prover [47] and translated by the Sledgehammer system [11]. Further, we also used problems from [14] with many quantifier alternations over term algebras. When compared to state-of-the-art SMT solvers, such as CVC4 and Z3 [21], our experimental results give practical evidence of the efficiency and logical strength of our work: many hard problems that could not be solved before by any existing technique can now be solved by our work (see Section 3.7).
Contributions. The main contributions of our paper are summarized below.

- We extend the theory $\mathcal{T}_{FT}$ of finite term algebras with a subterm relation denoting proper subterm relations between terms. We call this extension $\mathcal{T}_{FT}^{+}$ and prove that $\mathcal{T}_{FT}^{+}$ is a conservative extension of $\mathcal{T}_{FT}$. When compared to $\mathcal{T}_{FT}$, the advantage of $\mathcal{T}_{FT}^{+}$ is that it is finitely axiomatizable and hence can be used by any first-order theorem prover. Moreover, one can combine $\mathcal{T}_{FT}^{+}$ with other theories, going even to undecidable fragments of the combined theory of term algebras and other theories. As an important consequence of this conservative extension, our work yields a superposition-based decision procedure for term algebras (Section 3.5).

- We show how to optimize superposition-based first-order reasoning using new, term algebra specific, simplification rules, and an incomplete, but simple, replacement for a troublesome acyclicity axiom. Our new inference system provides an alternative and efficient approach to axiomatic reasoning about term algebras in first-order theorem proving and can be used with combinations of theories (Section 3.6).

- We implement our work in the first-order theorem prover Vampire. Our works turns Vampire into the first first-order theorem prover able to reason about term algebras, and therefore about algebraic data types. Our experiments show that our implementation outperforms state-of-the-art SMT solvers able to reason with algebraic data types. For example, Vampire solved 50 SMTLIB problems that could not be solved by any other solver before (Section 3.7).

3.2 Preliminaries

We consider standard first-order predicate logic with equality. The equality symbol is denoted by $\approx$. We allow all standard boolean connectives and quantifiers in the language. We assume that the language contains the logical constants $\top$ for always true and $\bot$ for always false formulas.

Throughout this paper, we denote terms by r, s, u, t, variables by x, y, z, constants by a, b, c, d, function symbols by f, g and predicate symbols by p, q, all possibly with indices. We consider equality $\approx$ as part of the language, that is, equality is not a symbol. For simplicity, we write $s \not\approx t$ for the formula $\neg(s \approx t)$.

An *atom* is an equality or a formula of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and $t_1, \ldots, t_n$ are terms. A *literal* is an atom A or its negation $\neg A$. Literals that are atoms are called *positive*, while literals of the form $\neg A$ are *negative*. A *clause* is a disjunction of literals $L_1 \vee \ldots \vee L_n$, where $n \geq 0$. When $n = 0$, we will speak of the empty clause, denoted by $\Box$. The empty clause is always false.

We denote atoms by A, literals by L, clauses by C, D, and formulas by F, G, possibly with indices.

Let F be a formula with free variables $\bar{x}$, then $\forall F$ (respectively, $\exists F$) denotes the formula $(\forall \bar{x})F$ (respectively, $(\exists \bar{x})F$). A formula is called *closed*, or a *sentence*, if it has no free variables. A formula or a term is called *ground* if it has no occurrences of variables.

A *signature* is any finite set of symbols. The *signature of a formula* F is the set of all symbols occurring in this formula. For example, the signature of $(\forall x)b \approx g(x)$ is $\{g, b\}$. When we speak about a *theory*, we either mean a set of all logical consequences of a set of formulas (called *axioms* of this theory), or a set of all formulas valid on a class of first-order structures. Specifically, we are interested in the theories of term algebras, in which case we use the second meaning. When we discuss a theory, we call symbols occurring in the signature of the theory *interpreted*, and all other symbols *uninterpreted*.

By an *expression* E we mean a term, atom, literal, or clause. A *substitution* θ is a finite mapping from variables to terms. An *application* of this substitution to an expression (e.g. a term or a clause) E, denoted by $E\theta$, is the expression obtained from E by the simultaneous replacement of each variable x in it, such that $\theta(x)$ is defined, by $\theta(x)$. We write $E[s]$ to mean an expression E with a particular occurrence of a term s. A *unifier* of two expressions E_1 and E_2 is a substitution θ such that $E_1\theta = E_2\theta$. It is known that if two expressions have a unifier, then they have a so-called *most general unifier (mgu)* – see [50] for details on computing mgus.

3.3 Superposition and Proof Search

We now recall some terminology related to inference systems and first-order theorem proving. Inference systems are used in the theory of superposition [46] implemented by several leading automated first-order theorem provers, including Vampire [40] and E [52]. The material of this section is based on [40], adapted to our setting.

3.3.1 The Superposition Inference System

First-order theorem provers perform inferences on clauses. An *inference rule* is an n-ary relation on formulas, where $n \geq 0$. The elements of such a relation are called *inferences* and usually written as:

$$\frac{C_1 \quad \dots \quad C_n}{C} \ .$$

The clauses $C_1, \dots, C_n$ are called the *premises* of this inference, whereas the clause C is the *conclusion* of the inference. An *inference system* $\mathbb{I}$ is a set of inference rules. An *axiom* of an inference system is any conclusion of an inference with 0 premises. Any inferences with 0 premises and a conclusion C will be written without the bar line, simply as C.

Modern first-order theorem provers use and implement the *superposition inference system*, which is parametrized by a *simplification ordering* $\preceq$ on terms and a *selection function* on clauses.

An ordering $\preceq$ on terms is called a *simplification ordering* if it satisfies the following conditions:

1. $\preceq$ is *well-founded*: there exists no infinite sequence of terms $t_0, t_1, \dots$ such that $t_0 \preceq t_1 \preceq \dots$;

- Resolution

$$\frac{\underline{A} \vee C_1 \qquad \underline{\neg A'} \vee C_2}{(C_1 \vee C_2)\sigma} \qquad\qquad \frac{\underline{s \not\approx s'} \vee C}{C\theta}$$

where $\sigma = mgu(A, A')$, $\theta = mgu(s, s')$ and A is not an equality predicate

- Superposition

$$\frac{\underline{l \approx r} \vee C_1 \qquad \underline{L[l']} \vee C_2}{(C_1 \vee L[r] \vee C_2)\theta}$$

$$\frac{\underline{l \approx r} \vee C_1 \qquad \underline{t[l'] \approx t'} \vee C_2}{(C_1 \vee t[r] \approx t' \vee C_2)\theta} \qquad \frac{\underline{l \approx r} \vee C_1 \qquad \underline{t[l'] \not\approx t'} \vee C_2}{(C_1 \vee t[r] \not\approx t' \vee C_2)\theta}$$

where l' not a variable, L is not an equality, $\theta = mgu(l, l')$, $l\theta \not\preceq r\theta$ and $t[l']\theta \not\preceq t'\theta$

- Factoring

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\sigma} \qquad\qquad \frac{s \approx t \vee \underline{s' \approx t'} \vee C}{(s \approx t \vee t \not\approx t' \vee C)\theta}$$

where $\sigma = mgu(A, A')$, $\theta = mgu(s, s')$, $s\theta \not\preceq t\theta$ and $t\theta \not\preceq t'\theta$

Figure 3.1. The superposition calculus $\mathcal{S}$.

2. $\preceq$ is *stable under substitution*: if $s \preceq t$ then $s\theta \prec t\theta$, for every term s, t and substitution θ;

3. $\preceq$ is *monotonic*: if $s \preceq t$ then $l[s] \preceq l[t]$ for all terms l, s, t;

4. $\preceq$ has the *subterm property*: if s is a proper subterm of t, then $s \preceq t$.

Given two terms $s \preceq t$, we say that s is smaller than t and t is larger/greater than s wrt $\preceq$. This ordering $\preceq$ can be extended to literals and clauses.

A *selection function* selects in every non-empty clause a non-empty subset of literals. In the following, we underline literals to indicate that they are selected in a clause; that is we write $\underline{L} \vee C$ to denote that the literal L is selected. A selection function is said to be *well-behaved* if in a given clause it selects either a negative literal or all the maximal literals wrt the simplification ordering $\preceq$.

We now fix a simplification ordering $\preceq$ and a well-behaved selection function and define the *superposition inference system.* This inference system, denoted by $\mathcal{S}$, consists of the inference rules of Figure 3.1. The inference system $\mathcal{S}$ is a sound and refutationally complete inference system for first-order logic with equality. By refutational completeness we mean that if a set S of formulas is unsatisfiable, then $\square$ is derivable from S in $\mathcal{S}$.

3.3.2 Proof Search by Saturation

Superposition theorem provers implement proof-search algorithms in $\mathcal{S}$ using so-called *saturation algorithms*, as follows. Given a set S of formulas, superposition-based theorem provers try to saturate S with respect to $\mathcal{S}$, that is build a set of formulas that contains S and is closed under inferences in $\mathcal{S}$. At every step, a saturation algorithm selects an inference of $\mathcal{S}$, applies this inference to S, and adds conclusions of the inferences to the set S. If at some moment the empty clause $\square$ is obtained, by soundness of $\mathcal{S}$, we can conclude that the input set of clauses is unsatisfiable. To ensure that a saturation algorithm preserves completeness of $\mathcal{S}$, the inference selection strategy must be fair: every possible inference must be selected at some step of the algorithm. A saturation algorithm with a fair inference selection strategy is called a *fair saturation algorithm.*

A naive implementation of fair saturation algorithms based on $\mathcal{S}$ will not yield however an efficient theorem prover. This is because at every step of the saturation algorithm, the number of clauses in the set S of clauses, representing the proof-search space, grows. Therefore, for the efficiency of organizing proof search, one needs to use the notion of *redundancy*, which allows to delete so-called redundant clauses during saturation from the search space. A clause $C \in S$ is *redundant* in S if it is a logical consequence of those clauses in S that are strictly smaller than C w.r.t. the simplification ordering $\preceq$. In a nutshell, saturation algorithms using redundancy not only generate but also delete clauses from the set S of clauses. Deletion of redundant clauses is desirable since every deletion reduces the search space. If a newly generated clause C' during one step of saturation makes some clauses in S redundant, adding C' to the search space will remove other (more complex) clauses from S. This observation is exploited by first-order theorem provers in the process of prioritizing inferences during inference selection, giving rise to so-called *simplifying* and *generating* inferences. Simplifying inferences make one or more clauses in the search space redundant and thus delete clauses from the search space. That is, an inference

$$\frac{C_1 \quad \dots \quad C_n}{C} \; .$$

is called *simplifying* if at least one of the premises C_i becomes redundant (and deleted) after the addition of the conclusion C to the search space. Inferences that are not simplifying are *generating*: instead of simplifying clauses in the search space, they generate and add a new clause to the search space. Efficient saturation algorithms exploit simplifying and generating inferences, as follows: from time to time provers try to search for simplifying inferences at the expense of delaying generating inferences.

3.4 The Theory of Finite Term Algebras

A definition of the first-order theory of term algebras over a finite signature can be found in e.g. [51], along with an axiomatization of this theory and a proof of its completeness. In this section we overview this theory and known results about it.

3.4.1 Definition

Let Σ be a finite set of function symbols containing at least one constant. Denote by $\mathcal{T}(\Sigma)$ the set of all ground terms built from the symbols in Σ.

The Σ-*term algebra* is the algebraic structure whose carrier set is $\mathcal{T}(\Sigma)$ and defined in such a way that every ground term is interpreted by itself (we leave details to the reader). We will sometimes consider extensions of term algebras by additional symbols. Elements of Σ will be called *term constructors* (or simply just *constructors*), to distinguish them from other function symbols. The Σ-term algebra will also be denoted by $\mathcal{T}(\Sigma)$.

Consider the following set of formulas.

$$\bigvee_{f \in \Sigma} \exists \overline{y}\ (x \approx f(\overline{y})) \tag{A1}$$

$$f(\overline{x}) \not\approx g(\overline{y}) \tag{A2}$$

for every $f, g \in \Sigma$ such that $f \neq g$;

$$f(\overline{x}) \approx f(\overline{y}) \to \overline{x} \approx \overline{y} \tag{A3}$$

for every $f \in \Sigma$ of arity ≥ 1;

$$t \not\approx x \tag{A4}$$

for every non-variable term t in which x appears.

Some of these formulas contain free variables, we assume that they are implicitly universally quantified.

Axiom (A1), sometimes called the domain closure axiom, asserts that every element in Σ is obtained by applying a term constructor to other elements.

Axiom (A3) describes the injectivity of term constructors, while axiom (A2) expresses the fact that terms constructed from different constructors are distinct. Throughout this paper, we refer to (A2) as the distinctness axiom and to (A3) as the injectivity axiom.

The axiom schema (A4), called the acyclicity axiom, asserts that no term is equal to its proper subterm, or in other words that there exist no cyclic terms.

In the following sections we will also discuss theories in which there are non-constructor function symbols. Note that when we deal with such theories, the acyclicity axioms are used only when all symbols in t are constructors.

3.4.2 Known Results

We denote by $\mathcal{T}_{FT}$ the theory axiomatized by (A1)–(A4), that is, the set of logical consequences of all formulas in (A1)–(A4). Note that the Σ-term algebra is a model of all formulas (A1)–(A4), and therefore also a model of $\mathcal{T}_{FT}$.

Theorem 1. *The following results hold.*

1. *$\mathcal{T}_{FT}$ is complete. That is, for every sentence F in the language of $\mathcal{T}(\Sigma)$, either $F \in \mathcal{T}_{FT}$ or $(\neg F) \in \mathcal{T}_{FT}$.*
2. *$\mathcal{T}_{FT}$ is decidable.*
3. *If Σ contains at least one symbol of arity > 1, then the first-order theory of $\mathcal{T}_{FT}$ is non-elementary.*

Completeness of $\mathcal{T}_{FT}$ is proved in a number of papers - a detailed proof can be found in, e.g., [51].

Decidability of $\mathcal{T}_{FT}$ in Theorem 1 is implied by the completeness of $\mathcal{T}_{FT}$ and by the fact that $\mathcal{T}_{FT}$ has a recursive axiomatization. More precisely, completeness gives the following (slightly unusual) decision procedure: given a sentence F, run any complete first-order theorem proving procedure (e.g., a complete superposition theorem prover) simultaneously and separately on F and $\neg F$. We can get around the problem that the axiomatisation is infinite but throwing in axioms, one after one, while running the proof search — indeed, by the compactness property of first-order logic, if a formula G is implied by an infinite set of formulas, it is also implied by a finite subset of this set. One of contributions of this paper is showing how to avoid dealing with infinite axiomatizations.

Further, the non-elementary property of $\mathcal{T}_{FT}$ in Theorem 1 follows from a result in [26]: every theory in which one can express a pairing function has a hereditarily non-elementary first-order theory.

Note that the completeness of $\mathcal{T}_{FT}$ implies that $\mathcal{T}_{FT}$ is exactly the set of all formulas true in the Σ-term algebra. First-order theories of term algebras are closely related to non-recursive logic programs, for related complexity results, also including the case with only unary functions, see [55].

Let us make the following important observation. The decidability and other results of Theorem 1 do not hold when uninterpreted functions or predicates are added to $\mathcal{T}_{FT}$. If we add to the Σ-term algebra uninterpreted symbols, one can for example use these symbols to provide recursive definitions of addition and multiplication, thus encoding first-order Peano arithmetic. Using the same reasoning as in [36] one can then prove the following result.

Theorem 2. *The first-order theory of Σ-algebras with uninterpreted symbols is Π^1_1-complete, when Σ contains at least one non-constant.*

We will not give a full proof of Theorem 2 but refer to [36] for details. Here, we only show how to encode non-linear arithmetic in $\mathcal{T}_{FT}$ using

Σ-term algebra uninterpreted symbol, which is relatively straightforward. Assume, without loss of generality, that Σ contains a constant 0 and a unary function symbol s (successor). Then all ground terms, and hence all term algebra elements are of the form $s^n(0)$, where $n \geq 0$. We will identify any such term $s^n(0)$ with the non-negative integer n.

Add two uninterpreted functions $+$ and $\cdot$ and consider the set A of formulas defined as follows:

$$\forall x\ (x + 0 = x)$$

$$\forall x \forall y\ (s(x) + y = s(x + y))$$

$$\forall x\ (x \cdot 0 = 0)$$

$$\forall x \forall y\ (s(x) \cdot y = (x \cdot y) + y)$$

It is not hard to argue that in any extension of the Σ-algebra satisfying A, the functions $+$ and $\cdot$ are interpreted as the addition and multiplication on non-negative integers. Let now G be any sentence using only $+, \cdot, s, 0$. Then we have that $A \implies G$ is valid in the Σ-algebra if and only if G is a true formula of arithmetic.

Note that Theorem 2 refers to the theory of algebras, i.e. the set of formulas valid on Σ-algebra. In view of this theorem, with uninterpreted symbols of arity ≥ 1 in the signature, this includes more formulas than the set of formulas derivable from (A1)–(A4).

3.4.3 Other Formalizations

Instead of using existential quantifiers in (A1), one can also use axioms based on destructors (or projection functions) of the algebra. For all function symbols f of arity $n > 0$ and all $i = 1, \ldots, n$, introduce a function p_f^i. The *destructor axioms* using these functions are:

$$x \approx f(p_f^1(x), \ldots, p_f^n(x)). \qquad \text{(A1')}$$

The axiom (A3) can be replaced by the following axioms, which can be considered as a definition of destructors:

$$p_f^i(f(x_1, \ldots, x_i, \ldots, x_n)) \approx x_i \qquad \text{(A3')}$$

Given the other axioms, (A3) and (A3') are logically equivalent, but some

$$\exists y\big(x \approx \mathit{leaf}(y)\big) \vee \exists y_1, y_2, y_3\big(x \approx \mathit{node}(y_1, y_2, y_3)\big)$$

$$\mathit{node}(x_1, x_2, x_3) \not\approx \mathit{leaf}(y_1)$$

$$\mathit{leaf}(x) \approx \mathit{leaf}(y) \rightarrow x \approx y$$

$$\mathit{node}(x_1, x_2, x_3) \approx \mathit{node}(y_1, y_2, y_3) \rightarrow x_1 \approx y_1 \wedge x_2 \approx y_2 \wedge x_3 \approx y_3$$

$$x \not\approx \mathit{node}(x, y_1, y_2)$$

$$x \not\approx \mathit{node}(y_1, y_2, x)$$

$$x \not\approx \mathit{node}(\mathit{node}(x, y_1, y_2), y_3, y_4)$$

$$\dots$$

Figure 3.2. The instantiation of the theory axioms for the signature Σ_{Bin}.

authors prefer the presentation based on destructors. Note, however, that the behavior of a destructors p_f^i is unspecified on some terms.

3.4.4 Extension to Many-Sorted Logic

In practice, it can be useful to consider multiple sorts, especially for problems taken from functional programming. In this setting, each term algebra constructor has a type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$. The requirement that there is at least one constant should then be replaced by the requirement that for every sort, there exists a ground term of this sort.

We can also consider similar theories, which mix constructor and non-constructor sorts. That is, some sorts contain constructors and some do not.

Consider an example with the following term algebra signature:

$$\Sigma_{Bin} = \{\mathit{leaf} : \tau \rightarrow \mathit{Bin}, \mathit{node} : \mathit{Bin} \times \tau \times \mathit{Bin} \rightarrow \mathit{Bin}\}$$

This signature defines an algebra of binary trees, where every node and leaf is decorated by an element of a (non-constructor) sort τ. In this case term algebra axioms are only using sorts with constructors. The axioms of this theory of trees, as defined previously, are shown in Figure 3.2.

3.5 A Conservative Extension of the Theory of Term Algebras

In this paper we aim to prove theorems in first-order theories containing constructor-defined types. While in general the theory is Π_1^1-complete, we still want to have a method that behaves well in practice. Our method will be based on extending the superposition calculus by axioms and/or rules for dealing with term algebra constructor symbols.

One of the criteria of behaving well in practice is to have a method that is complete for pure term algebra formulas, that is, without uninterpreted functions. The immediate idea would be to use the axiomatization of term algebras consisting of (A1)–(A4), however this does not work since there is an infinite number of acyclicity axioms.

In this section we show how to overcome this problem by using an extension of term algebras by a binary relation *Sub*, denoting the proper subterm relation. Let us further denote by $\mathcal{T}_{FT}^{+}$ the set of formulas which contains (A1)–(A3), but replaces the acyclicity axiom (A4) by the following axioms (B1)–(B3):

$$Sub(x_i, f(x_1, \ldots, x_i, \ldots, x_n)), \tag{B1}$$

for every $f \in \Sigma$ of arity $n \geq 1$ and every i such that $n \geq i \geq 1$.

$$Sub(x, y) \wedge Sub(y, z) \rightarrow Sub(x, z) \tag{B2}$$

$$\neg Sub(x, x) \tag{B3}$$

Intuitively, the predicate $Sub(s, t)$ holds iff s is a proper subterm of t. Axiom (B1) ensures that this relation holds for terms s appearing directly under a term algebra constructor in t , while (B2) describes the transitivity of the subterm relation and ensures that the relation also holds if s is more deeply nested in t. Axiom (B3) asserts that no term may be equal to its own proper subterm.

We now observe the following properties of (B1)–(B3).

Theorem 3. *$\mathcal{T}_{FT}^{+}$ is a conservative extension of $\mathcal{T}_{FT}$, that is:*

1. *Every theorem in $\mathcal{T}_{FT}$ is a theorem in $\mathcal{T}_{FT}^{+}$;*

2. *Every theorem in $\mathcal{T}_{FT}^{+}$ that uses only symbols from the language of $\mathcal{T}_{FT}$ (i.e. not using the predicate Sub) is also a theorem of $\mathcal{T}_{FT}$.*

Proof. For (1), it is enough to prove that every instance of the acyclicity axiom (A4) of $\mathcal{T}_{FT}$ is implied by axioms of $\mathcal{T}_{FT}^{+}$. To this end, note that for every term t and its proper subterm s, (B1)–(B2) imply $Sub(s,t)$, so every instance of the acyclicity axiom (A4) is implied by (B1)–(B3).

To prove part (2), first note that $\mathcal{T}_{FT}^{+}$ is consistent (sound). This follows from the fact that it has a model, which extends the Σ-term algebra by interpreting Sub as the subterm relation. Now assume, by contradiction, that there is a sentence F not using Sub such that $F \in \mathcal{T}_{FT}^{+}$ and $F \notin \mathcal{T}_{FT}$. By the completeness result of Theorem 1, we then have $\neg F \in \mathcal{T}_{FT}$, which by part (1) implies $\neg F \in \mathcal{T}_{FT}^{+}$. We have both $F \in \mathcal{T}_{FT}^{+}$ and $\neg F \in \mathcal{T}_{FT}^{+}$, which contradicts the consistency of $\mathcal{T}_{FT}^{+}$. $\square$

Note that the full first-order theory of term algebras with the subterm predicate is undecidable [54].

The important difference between $\mathcal{T}_{FT}$ and $\mathcal{T}_{FT}^{+}$ is that $\mathcal{T}_{FT}^{+}$ *is finitely axiomatizable.* This fact and Theorem 3 can be directly used to design *superposition-based proof procedures* for $\mathcal{T}_{FT}$, as follows. Given a term algebra sentence F, we can search for a superposition proof of F from the axioms of $\mathcal{T}_{FT}^{+}$. Such a proof exists if and only if F holds in the Σ-term algebra. This proof procedure can even be turned into a *superposition-based decision procedure for $\mathcal{T}_{FT}$*, which is based on attempting to prove F and $\neg F$ in parallel, until one of them is proved, which is guaranteed by the completeness of $\mathcal{T}_{FT}$ from Theorem 1.

It is interesting that, while proving a formula F with quantifier alternations in this way, first-order theorem provers will first skolemize F, introducing uninterpreted functions. While the first-order theory of term algebras with arbitrary uninterpreted functions is incomplete, our results guarantee *completeness on formulas with uninterpreted functions obtained by skolemization.* This is so because skolemization preserves validity and hence, using Theorem 3, we conclude completeness on skolemized formulas with uninterpreted functions.

While it is hard to expect that proving term algebra formulas by superposition will result in a better decision procedure compared to those described in the literature, see e.g. [14], our approach has the advantage that it can be combined with other theories and can be used for proving formulas in undecidable fragments of the full first-order theory of term algebras. Given a formula containing both constructors, uninterpreted symbols and possible theory symbols, we can attempt to prove this formula by adding the axioms of $\mathcal{T}_{FT}^{+}$ and then use a superposition theorem prover. The results of this section show that this method is strong enough to prove all (pure) term algebra theorems. Our experimental results described in Section 3.7 give an evidence that it is also efficient in practice.

The conservative extension $\mathcal{T}_{FT}^{+}$ presented above thus allows one to encode problems in the theory of term algebras and reason about them using any tool for automated reasoning in first-order logic. However the transitive nature of the predicate *Sub* can impact the performance of provers negatively. Note that the transitivity axiom can also be replaced by axioms of the form:

$$Sub(x, x_i) \rightarrow Sub(x, f(x_1, \ldots, x_i, \ldots, x_n)).$$

Using these new axioms will result in fewer inferences during proof search and a slower growth of the subterm relation, which are important parameters for the provers' performance.

3.6 An Extended Calculus

In this section we describe an alternative way to use superposition theorem provers to reason about term algebras. Instead of including theory axioms in the initial set of clauses, we extend the calculus with inferences rules. This is similar to the way paramodulation is used to replace the axiomatization of equality, apart from the fact that we cannot obtain a calculus that is complete.

3.6.1 A naive calculus

In this section we will consider alternatives and improvements to axiomatizing term algebras. The idea is to add simplification rules specific to term algebras and replace the troublesome acyclicity axiom by special purpose inference rules.

The superposition calculus uses term and clause orderings to orient equalities, restrict the number of possible inferences, and simplification. The general rule is that a clause in the search space can be deleted if it is implied by strictly smaller clauses in the search space.

One obvious idea is to add several simplification rules, corresponding to applications of resolution and/or superposition to term algebra axioms. For example, a clause $f(s) \approx s \vee C$ can be replaced by a simpler, yet equivalent, clause C. Likewise, the clause $f(s) \approx f(t) \vee C$ is equivalent, by injectivity of the constructors, to the clause $s \approx t \vee S$. The clause $s \approx t \vee S$ is also smaller than $f(s) \approx f(t) \vee C$, so it can replace this clause.

Let us start with examples showing that replacing axioms by rules can result in incompleteness even in very simple cases.

Take for example two ground unit clauses $f(a) \approx b$ and $g(a) \approx b$, where all symbols apart from b are constructors. This set of clauses is unsatisfiable in the theory of term algebras. However, if we replace the axiom $f(x) \not\approx g(y)$ by a simplification rule, there are no inferences that can be done between these clauses (assuming we are using the standard Knuth-Bendix ordering).

Another example showing that the acyclicity axiom can be hard to drop or replace is the set of two ground unit clauses $f(a) \approx b$ and $f(b) \approx a$, where f is a constructor. This set of clauses is also unsatisfiable in the theory of term algebras, since it implies $f(f(b)) = b$. Similar to the previous example, there is no superposition inference between these two clauses.

3.6.2 The Distinctness Rule

We implemented an extra simplification and a deletion rule. Such rules will be denoted using a double line, meaning that the clauses in the premise are replaced by the clauses in the conclusion.

The simplification rule is

$$\frac{f(s) \approx g(t) \vee A}{\overline{A}}\ \text{Dist-S}^{+},$$

where f and g are different constructors. Essentially, it removes from the clause a literal false in the theory of term algebras.

The deletion rule is

$$\frac{f(s) \not\approx g(t) \vee A}{\overline{\emptyset}}\ \text{Dist-S}^{-},$$

where f and g are different constructors. It deletes a theory tautology.

3.6.3 The Injectivity Rule

There is a simplification rule based on the injectivity axiom (A3). Suppose that f is a constructor of arity $n > 0$. Then we can use the simplification rule

$$\frac{f(s_1 \ldots s_n) \approx f(t_1, \ldots, t_n) \vee C}{\overline{\begin{array}{c} s_1 \approx t_1 \vee C \\ \cdots \\ s_n \approx t_n \vee C \end{array}}}\ .$$

One can also note that under some additional restrictions the following inference

$$\frac{f(s_1 \dots s_n) \not\approx f(t_1, \dots, t_n) \lor C}{s_1 \not\approx t_1 \lor \dots \lor s_n \not\approx t_n \lor C}$$

can be considered as a simplification rule too. The restriction is the clause ordering condition $\{s_1 \not\approx t_1 \lor \dots \lor s_n \not\approx t_n\} \prec C$.

Note that in both rules the premise is logically equivalent to the conjunction of the formulas in the conclusion in the theory of term algebras and all formulas in the conclusion are smaller than the formula in the premise (subject to the ordering condition for the second rule).

3.6.4 The Acyclicity Rule

Similar to the distinctness axiom and rules, we can introduce a simplification and a deletion rule based on the acyclicity axiom. First, we introduce a notion of a *constructor subterm* as the smallest transitive relation that each of the terms t_i is a constructor subterm of $f(t_1, \dots, t_n)$, where f is a constructor and $n \geq i \geq 1$. For example, if f is a binary constructor, and g is not a constructor, then all constructor subterms of the term $f(f(x, a), g(y))$ are $f(x, a)$, x, a and $g(y)$. Its subterm y is not a constructor subterm. One can easily show that any inequality $s \not\approx t$, where s is a constructor subterm of t is false in any extension of term algebras.

The simplification rule for acyclicity is

$$\frac{s \approx t \lor A}{A},$$

where s is a constructor subterm of t. It deletes from a clause its literal false in all term algebras.

The deletion rule is

$$\frac{s \not\approx t \lor A}{\emptyset},$$

where s is a constructor subterm of t. It deletes a theory tautology.

Further, if we wish to get rid of the subterm relation *Sub*, we can use various rules to treat special cases of acyclicity. If we do this, we will lose completeness even for pure term algebra formulas, but such a replacement can deal with some formulas more efficiently, while still covering a sufficiently large set of problems.

One example of such a special acyclicity rule is the following:

$$\frac{t \approx u \vee A}{s \not\approx u \vee A}$$

where s is a constructor subterm of t. Note that this rule is not a simplification rule, so we do not delete the premise after applying this rule.

3.7 Experimental Results

3.7.1 Implementation

We implemented the subterm relation of Section 3.5 and simplification rules of Section 3.6 in the first-order theorem prover Vampire [40]. Note that Vampire behaves well on theory problems with quantifiers both at the SMT and first-order theorem proving competitions, winning respectively 5 divisions in the SMT-COMP 2016 competition of SMT solvers[1] and the quantified theory division of the CASC 2016 competition of first-order provers [2]. With our implementation, Vampire becomes the first superposition theorem prover able to prove properties of term algebras. Moreover, our experiments described later show that Vampire outperforms state-of-the-art SMT solvers, such as CVC4 and Z3, on existing benchmarks.

Our implementation required altogether about 2,500 lines of C++ code. The new version of Vampire, together with our benchmark suite, is available for download[3].

3.7.2 Input Syntax and Tool Usage

In our work, we used an extended SMTLIB syntax [4] to describe term constructors. Although not yet part of the official SMTLIB standard, this syntax is already supported by the SMT solvers Z3 and CVC4, and its standardization is under consideration.

Our input syntax uses `declare-datatypes` for declaring an abstract data type corresponding to a term algebra sort. This declaration simultaneously adds the term algebra symbols and the *Sub* predicate to the problem signature, adds the distinctness, injectivity, domain closure and subterm axioms to the input set of formulas, and activates the additional inferences rules from Section 3.6. Alternatively, the user can choose not to activate the inference rules in our implementation. The inclusion

[1]`http://smtcomp.sourceforge.net/2016/`
[2]`http://www.cs.miami.edu/~tptp/CASC/J8/`
[3]`http://www.cse.chalmers.se/~simrob/tools.html`

of the *Sub* predicate and its axioms, as presented in Section 3.5, can also be deactivated.

Note that the SMTLIB syntax also provides the not yet standardized command `declare-codatatypes` to declare types of potentially cyclic or infinite data structures. The theory underlying the semantics of such types is almost identical to that of finite term algebras, except that the acyclicity axiom is replaced by a uniqueness rule that asserts that observationally equal terms are indeed equal [49]. Therefore our calculus *minus the acyclicity axioms/rules* is an incomplete but sound inference system for that theory, and users can declare co-algebraic data types in their problems as well. Like acyclicity, the uniqueness principle of co-algebras is not finitely axiomatizable.

3.7.3 Benchmarks

We evaluated our implementation on two sets of problems. These problems included all publicly available benchmarks, as mentioned below.

- A (parametrized) game theory problem originally described in [14]. This problem relies on the term algebra of natural numbers to describe winning and losing positions of a game. It is possible to encode, for a given positive integer k, a predicate $winning_k$ over positions, such that $winning_k(p)$ holds iff there exists a winning strategy from the position p in k or fewer moves. The satisfiability of the resulting first-order formula can be checked by term algebra decision procedures, since it does not use symbols other than those of the term algebra, but it includes $2k$ alternating universal and existential quantifiers. This heavy use of quantifiers makes it an interesting and challenging problem for provers. An example of this problem encoded in the SMTLIB syntax is given in Figure 3.3.

- Problems about functional programs, generated by the Isabelle interactive theorem prover [47] and translated by the Sledgehammer system [11]. The resulting SMTLIB problems include algebraic and co-algebraic data types as well as arbitrary types and function symbols, and also some quantified formulas. Some of these problems are taken from the Isabelle distribution (Distro) and the Archive of Formal Proofs (AFP), others from a theory about Bird and Stern–Brocot trees by Peter Gammie and Andreas Lochbihler (G&L). They are representative of the kind of problems corresponding to program analysis and verification goals. This set of problems

```
(declare-datatypes ()
  ((Nat (z) (s (pred Nat)))))

(assert
  (not
    (exists
      ((w1 Nat))
      (and
        (or
          (= (s z) (s w1))
          (= (s z) (s (s w1)))
        )
        (forall
          ((l0 Nat))
          (=>
            (or
              (= w1 (s l0))
              (= w1 (s (s l0)))
            )
            false))))))

(check-sat)
```

Figure 3.3. An instance of the game theory problem from [14], encoded in SMTLIB syntax. The first command declares a term algebra with a constant `z` and a unary function `s`; note that the projection function `pred` must also be named. The assertion (starting with `assert`) is a formula corresponding to the negation of the predicate $\mathit{winning}_1(s(z))$.

originally appeared in [49] and, to the best of our knowledge, represent the set of all publicly available benchmarks on algebraic data types.

3.7.4 Evaluation

Our experiments were carried out on a cluster on which each node is equipped with two quad core Intel processors running at 2.4 GHz and 24GiB of memory. To compare our work to other state-of-the-art systems, we include the results of running the SMT solvers Z3 and CVC4 on the Isabelle problems, as previously reported in [49], and also add the results of running these two solvers on the game theory problem.

Game theory problems. The times required to solve the game theory problem for different values of the parameter k are shown in Table 3.1. The first column indicates the time required by Vampire using the theory axioms (A) described in Section 3.5, and the second and third columns give the time needed when the simplification rules (R) are also activated in Vampire (Section 3.6). For this particular problem, the acyclicity rule plays no role in the proof, but in order to assess its impact on performance, the third column shows the times needed to solve the problem when the subterm relation axioms (S) are also included in the input. The fourth and fifth columns of Table 3.1 respectively indicate the times needed by CVC4 and Z3 for solving the corresponding problem. Where no value is given, the prover was unable to solve the problem. Despite belonging to a decidable class, this problem is quite challenging for theorem provers and SMT solvers, which is easily explained by the presence of a formula with many quantifier alternations. The SMT solver CVC4 is able to disprove the negated conjecture only for $k = 1$, and Z3 can disprove it only for $k = 1$ or $k = 2$. SMT solvers can also consider the (non-negated) conjecture and try to satisfy it, but this does not produce better results. In comparison, our implementation in Vampire can solve the problem for $k = 6$, that is for formulas with 12 alternated existential and universal quantifiers, in 8.19 seconds. In [14], the authors are able to solve the problem for k as high as 80, using an implementation of the decision procedure presented in [19]. However such a decision procedure would not be able to reason in the presence of uninterpreted symbols, and therefore its usage is much more restricted. The results of Table 3.1 confirm that first-order provers can be better suited than SMT solvers for reasoning about formulas with many quantifiers, despite the various strategies used for quantifier reasoning in SMT solvers (for example, by using E-matching [20]). Table 3.1 also shows that adding simplification rules as described in Section 3.6 improves the behavior of the theorem prover.

Isabelle problems about functional programs. Our results on evaluating Vampire on the Isabelle problems are shown in Table 3.2. The problems were translated by Sledgehammer by selecting some lemmas possibly relevant to a given proof goal in Isabelle and translating them to SMTLIB along with the negation of the goal. While the intent of this translation is to produce unsatisfiable first-order problems, this is not the case for all of the problems tested here. A few problems are satisfiable and it is likely that many are unprovable, for example because the lemmas selected by Sledgehammer are not sufficiently strong to prove the goal. The set of problems originally included 4170 problems, of which 2869

k	Vampire (A)	Vampire (A+R)	Vampire (A+R+S)	CVC4	Z3
1	0.01	0.01	0.01	0.01	0.01
2	0.01	0.01	0.01	0.01	0.01
3	4.98	0.18	0.66	–	–
4	2.21	0.32	0.63	–	–
5	35.16	11.17	15.40	–	–
6	31.57	8.19	11.33	–	–
7	–	–	–	–	–

Table 3.1. Time required to prove unsatisfiability of different instances of the game theory problem from [14].

include at least one algebraic data type and 2825 include at least one co-algebraic data type, some problems containing both. In the presence of co-algebraic data types, CVC4 has a special decision procedure which replaces the acyclicity rule by a uniqueness rule. In our implementation, Vampire simply does not add the acyclicity axiom, but the remaining axioms are added as they hold for co-algebraic data types as well. Unlike CVC4, Z3 does not support reasoning about co-algebraic data types.

In order to test the efficiency of our acyclicity techniques on more examples, we considered problems containing co-algebraic data types: by replacing them with algebraic data types with similar constructors, we obtained different problems where the acyclicity principle applies. Note that not all co-algebraic data type definitions correspond to a well-founded definition for an algebraic data type: after leaving these out, we obtained 2112 new problems.

Table 3.2 summarizes our results on this set of benchmarks, using a single best strategy in Vampire. For each solver, we also show the number of problems solved uniquely only by that solver.

We also ran Vampire with a combination of strategies with a total time limit of 120 seconds. Table 3.3 shows the total number of solved problems, with details on whether the problems contain only algebraic data types, co-algebraic data types, or both. Overall, Vampire is able to solve 1785 problems, that is 4,2% more that CVC4 and 7,3% more than Z3, which is a significant improvement. 50 problems are uniquely solved by Vampire, as listed in column six Table 3.3. When compared to Vampire, only 4 problems were proved by CVC4 alone, while Z3 cannot prove any problem that was not proved by Vampire – see columns seven and eight of Table 3.3. Summarizing, Table 3.2 shows that Vampire outperforms the best existing solvers so far. The experimental results of Tables 3.1-3.2

Prover	Solved	Unique
Z3	1665	5
CVC4	1711	12
Vampire (Best strategy)	1720	31

Table 3.2. Number of problems solved among the 6282 Isabelle problems translated by SledgeHammer.

	Total	Solved			Unique		
		Vampire	CVC4	Z3	Vampire	CVC4	Z3
Data types	3457	999	956	947	23	0	0
Co-data types	1301	430	415	382	16	2	0
Both	1524	356	341	334	11	2	0
Union	6282	1785	1712	1663	50	4	0

Table 3.3. Distribution of solved problems according to the data types they feature

provide an evidence that our methods for proving properties of algebraic data types outperform methods currently used by SMT solvers.

3.7.5 Comparison of Option Values

We were also interested in comparing how various proof option values affect the performance of a theorem prover. For the purpose of this research, the options that we considered are:

1. the Boolean value selecting whether term algebra rules are used;
2. the value selecting how acyclicity is treated (axioms, rules, or none, that is, no acyclicity axioms or rules).

Making such a comparison is hard, since there is no obvious methodology for doing so, especially considering that Vampire has 64 options commonly used in experiments. The majority of these options are Boolean, some are finitely-valued, some integer-valued and some range over other infinite domains. The method we used was based on the following ideas. Suppose we want to compare values for an option π. Then:

1. we use a set of problems obtained by discarding problems that are too easy or currently unsolvable;
2. we repeatedly select a random problem P in this set, a random strategy S and run P on variants of S obtained by choosing all possible values for π using the same time limit.

	off	axioms	rules
Total solved	2030	9086	9602
Solved by only this value	50	70	566

Table 3.4. Comparison of proof option values for acyclicity in Vampire.

We discovered that the results for the term algebra rules are inconclusive (turning them on or off makes little effect on the results) and will present the results for the acyclicity option.

Our selected set of problems consisted of 262 term algebra problems. We made 90,000 runs for each value (off, theory axioms, and the acyclicity rules), that is, 270,000 tests all together, with the time limit of 30 seconds. While interpreting the results, it is worth mentioning the following.

1. When neither acyclicity rules nor acyclicity axioms are used, problems that require acyclicity reasoning become unsolvable. On the other hand, for other problems, this setting results in a smaller search space.

2. When the acyclicity rules are used, the resulting calculus is incomplete even for pure term algebra problems, but the subterm relation is not used, which generally means that fewer clauses should be generated.

The results of these experiments are shown in Table 3.4. We show the total number of successful runs (out of 90,000) and the number of runs where only one value for this option solved the problem. Probably the most interesting observation is that using acyclicity simplification rules (Section 3.6) instead of theory axioms (Section 3.5) results in many more problems solved. This gives us an evidence that the axiomatization based on the subterm relation results in much larger search spaces. This also means that the value resulting in an incomplete strategy in this case generally behaves better.

One should also note the 50 problems solved only when turning acyclicity off. This means that even the light-weight rule-based treatment of acyclicity sometimes results in a large overhead. Moreover, out of these 50 problems 10 were solved in less than 1 second.

3.8 Related Work

The problem of reasoning over term algebras first appears in the restricted form of syntactic unification, mentioned in [30]. The algorithm for syntactic unification was later described in [50], and later refined into quasi-linear [6, 34, 45] and linear algorithms [48].

The full-first order theory of term algebras over a finite signature was first studied in [44], where its decidability was proved by quantifier elimination. Other quantifier elimination procedures appeared in [15, 43, 33, 51]. [26] proved a result implying that the first-order theory of term algebras is non-elementary. There is a large body of research on decidability of various extensions of term algebras, which we do not describe here.

In this paper we do not prove decidability of new theories. However, we present a new superposition-based decision procedure for first-order theories of term algebras using a finitely axiomatizable theory.

Probably the first implementation of a decision procedure for term algebras is described in [14]. The theory of finite or infinite trees is also studied in [19] and a practical decision procedure is given based on rewriting.

Due to recent applications of program analysis, there is now a growing interest in the automated reasoning community for practical implementation of term algebras and their combinations with other theories. A decision procedure for algebraic data types is given in [5] and later extended to a decision procedure for co-algebraic data types in [49]. These decision procedures exploit SMT-style reasoning and are supported by CVC4. Z3 also supports proving properties about algebraic data types [10]. Unlike these techniques, our work targets the full first-order theory of term algebras, with arbitrary use of quantifiers. Our proof search procedure is based on the superposition calculus and allows one to prove properties with both theories and quantifiers.

3.9 Conclusion

We presented two different ways to reason in the presence of the theory of finite term algebras with a superposition-based first-order theorem prover. Our first approach is based on a finitely axiomatizable conservative extension of the theory and can be implemented in any first-order theorem prover. The second technique extends the first with the addition of extra inference and simplification rules having two aims:

1. simplifying more clauses;

2. replacing expensive subterm-based reasoning about acyclicity by

light-weight inference rules (though incomplete even without uninterpreted functions).

While not as efficient as specialized decision procedures for this theory, both our techniques allow us to reason about problems that includes the theory of finite terms algebras and other predicate or function symbols. We evaluated our work on game theory constraints and properties of functional program manipulating algebraic data types.

The next natural development would be to extend our approach to the theories of rational (finite but possibly cyclic) and infinite term algebras. The notion of co-algebras is also closely related to possibly infinite terms, with the addition of a uniqueness principle for cyclic terms. A decision procedure for this theory was included in the SMT solver CVC4 to decide problems involving co-algebraic data types [49]. Co-algebras are also best suited to express the semantics of processes and structures involving a notion of state. Unlike term algebras, co-algebras have been studied almost exclusively from the point of view of category theory, rather than that of first-order logic, so that many theoretical and practical applications remain to be explored there.

An even more interesting avenue to exploit is inductive reasoning about algebraic data types in first-order theorem proving, also based on extensions of the superposition calculus.

The work presented here should be a useful development for the verification of functional programs. For example it would benefit the tool HALO [56], which expresses the denotational semantics of Haskell programs in first-order logic, before using automated theorem provers to verify some of their properties. Our work not only makes the translation easier but also modifies the prover to make it more efficient on the generated problems. This also applies to other tools that already use first-order theorem provers to discharge their proof obligations, such as inductive theorem provers, e.g. HipSpec [12] and automated reasoning tools for higher-order logic, e.g. Sledgehammer [11].

More generally, our work makes an important step towards closing the gap between SMT solvers and first-order theorem provers. The former are traditionally used for problems involving theories, while the latter are better at dealing with quantifiers. Problems that include both quantifiers and theories are very common in practical applications and represent a big challenge due to their intrinsic complexity, both in theory and in practice. Our results show that first-order theorem provers can perform efficient reasoning in the presence of theories, solving many problems previously unsolvable by other tools.

Bibliography

[1] Leo Bachmair, Nachum Dershowitz, and David A Plaisted. Completion without failure. *Resolution of equations in algebraic structures*, 2:1–30, 1989.
— One citation on page 6

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
— One citation on page 29

[3] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer Aided Verification*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
— 2 citations on pages 7 and 34

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`, 2016.
— One citation on page 50

[5] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.
— 2 citations on pages 34 and 57

[6] Lewis Denver Baxter. *The Complexity of Unification.* PhD thesis, University of Waterloo Waterloo, Ontario, 1976.
— One citation on page 57

[7] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of Object-Oriented Software: The KeY Approach.* Springer, 2007.
— 3 citations on pages 13, 14, and 24

[8] Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt. An improved rule for while loops in deductive program verification. In *Formal Methods and Software Engineering*, volume 3785 of *LNCS*, pages 315–329. Springer, 2005.
— One citation on page 25

[9] Armin Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2016. *Proceedings of SAT Competition 2016 – Solver and Benchmark Descriptions*, B-2016-1:44–45, 2016.
— One citation on page 6

[10] Nikolaj Bjorner, Ken McMillan, and Andrey Rybalchenko. Higher-order program verification as satisfiability modulo theories with algebraic data-types. *arXiv preprint arXiv:1306.5264*, 2013.
— One citation on page 57

[11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending Sledgehammer with SMT solvers. *Automated Deduction–Cade-23*, 6803:116–130, 2013.
— 3 citations on pages 35, 51, and 58

[12] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *Automated Deduction–CADE-24*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.
— One citation on page 58

[13] Keith L Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Springer, 1978.
— One citation on page 33

[14] Alain Colmerauer et al. Expressiveness of full first order constraints in the algebra of finite or infinite trees. In *Principles and Practice of Constraint Programming–CP 2000*, volume 1894 of *LNCS*, pages 172–186. Springer, 2000.
— 7 citations on pages 35, 46, 51, 52, 53, 54, and 57

[15] Hubert Comon. *Unification et disunification: Théorie et applications*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 1988.
— One citation on page 57

[16] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.
— One citation on page 33

[17] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Symposium on Principles of Programming Languages*, pages 105–118. ACM, 2011.
— One citation on page 14

[18] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A program analysis perspective. In *Software Engineering and Formal Methods*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.
— One citation on page 13

[19] Thi-Bich-Hanh Dao. *Résolution de Contraintes du Premier Ordre dans la Théorie des Arbres Finis ou Infinis*. PhD thesis, Université Aix-Marseille 2, 2000.
— 2 citations on pages 53 and 57

[20] Leonardo De Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In *Automated Deduction–CADE-21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
— One citation on page 53

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
— 2 citations on pages 7 and 35

[22] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
— One citation on page 3

[23] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *Programming Languages and Systems*, volume 6012 of *LNCS*, pages 246–266. Springer, 2010.
— One citation on page 26

[24] Ioan Dragan and Laura Kovács. Lingva: Generating and proving program properties using symbol elimination. In *Perspectives of System Informatics*, volume 8974 of *LNCS*, pages 67–75. Springer, 2014.
— 5 citations on pages 13, 22, 23, 26, and 27

[25] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
— One citation on page 6

[26] Jeanne Ferrante and Charles W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics.* Springer, 1979.
— 2 citations on pages 42 and 57

[27] Juan Pablo Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. DynaMate: Dynamically inferring loop invariants for automatic full functional verification. In *Hardware and Software: Verification and Testing*, volume 8855 of *LNCS*, pages 48–53. Springer, 2014.
— 2 citations on pages 14 and 22

[28] Joseph A Goguen, James W Thatcher, Eric G Wagner, and Jesse B Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM (JACM)*, 24(1):68–95, 1977.
— One citation on page 33

[29] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *Computer Aided Verification*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.
— 2 citations on pages 14 and 22

[30] Jacques Herbrand. *Recherches sur la théorie de la démonstration.* PhD thesis, Université de Paris, 1930.
— One citation on page 56

[31] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
— One citation on page 3

[32] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Invariant generation in Vampire. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 60–64. Springer, 2011.
— 2 citations on pages 22 and 23

[33] Wilfrid Hodges. *Model Theory.* Cambridge University Press, 1993.
— One citation on page 57

[34] Gérard Huet. *Résolution d'équations dans des langages d'ordre 1, 2...* PhD thesis, Université Paris VII, 1976.
— One citation on page 57

[35] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.
— One citation on page 6

[36] Konstantin Korovin and Andrei Voronkov. Integrating linear arithmetic into superposition calculus. In *Computer Science Logic*, volume 4646 of *LNCS*, pages 223–237. Springer, 2007.
— One citation on page 42

[37] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A first class boolean sort in first-order theorem proving and TPTP. In *Intelligent Computer Mathematics*, volume 9150 of *LNCS*, pages 71–86. Springer, 2015.
— 2 citations on pages 25 and 29

[38] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, volume 8044 of *LNCS*, pages 470–485. Springer, 2009.
— 3 citations on pages 7, 13, and 20

[39] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In *Automated Deduction–CADE-22*, volume 5663 of *LNCS*, pages 199–213. Springer, 2009.
— One citation on page 18

[40] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.
— 5 citations on pages 7, 13, 35, 37, and 50

[41] Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. SMT-based array invariant generation. In *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *LNCS*, pages 169–188. Springer, 2013.
— One citation on page 14

[42] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence,*

and Reasoning, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
— One citation on page 13

[43] Michael J Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 348–357. IEEE, 1988.
— One citation on page 57

[44] Anatoly Ivanovich Mal'cev. Axiomatizable classes of locally free algebras of certain types. *Sibirsk. Mat. Ž.*, 3:729–743, 1962.
— 2 citations on pages 34 and 57

[45] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.
— One citation on page 57

[46] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.
— One citation on page 37

[47] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.
— 2 citations on pages 35 and 51

[48] Michael S Paterson and Mark N Wegman. Linear unification. In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pages 181–186. ACM, 1976.
— One citation on page 57

[49] Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co)datatypes in SMT solvers. In *Automated Deduction–CADE-25*, volume 9195 of *LNCS*, pages 197–213. Springer, 2015.
— 6 citations on pages 34, 35, 51, 52, 57, and 58

[50] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.
— 3 citations on pages 5, 37, and 57

[51] Tatiana Rybina and Andrei Voronkov. A decision procedure for term algebras with queues. *ACM Transactions on Computational Logic*,

2(2):155–181, 2001.
— 3 citations on pages 40, 42, and 57

[52] Stephan Schulz. E – a brainiac theorem prover. *AI Communications*, 15(2-3):111–126, 2002.
— 2 citations on pages 7 and 37

[53] Geoff Sutcliffe and Christian Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.
— One citation on page 29

[54] KN Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM (JACM)*, 34(2):492–510, 1987.
— One citation on page 46

[55] Sergei Vorobyov and Andrei Voronkov. Complexity of nonrecursive logic programs with complex values. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 244–253. ACM, 1998.
— One citation on page 42

[56] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. HALO: Haskell to logic through denotational semantics. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 48, pages 431–442. ACM, 2013.
— One citation on page 58

[57] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *Automated Deduction–CADE-22*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.
— One citation on page 7

[58] Lawrence Wos and George Robinson. Paramodulation and set of support. In *Symposium on Automatic Demonstration*, pages 276–310. Springer, 1970.
— One citation on page 6