CHALMERS
UNIVERSITY OF TECHNOLOGY

Thesis for the Degree of Doctor of Philosophy

# Deductive Program Analysis with First-Order Theorem Provers

Simon Robillard

Department of Computer Science and Engineering
Chalmers University of Technology

Gothenburg, Sweden
2019

Deductive Program Analysis with First-Order Theorem Provers
Simon Robillard
ISBN 978-91-7905-106-8

© 2019 Simon Robillard

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny series nr 4573
ISSN 0346-718X

Technical Report 169D

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed at Reproservice, Chalmers University of Technology
Gothenburg, Sweden, 2019

# Abstract

Software is ubiquitous in nearly all aspects of human life, including safety-critical activities. It is therefore crucial to analyze programs and provide strong guarantees that they perform as expected. Automated theorem provers are increasingly popular tools to assist in this task, as they can be used to automatically discover and prove some semantic properties of programs. This thesis explores new ways to use automated theorem provers for first-order logic in the context of program analysis and verification.

Firstly, we present a first-order logic encoding of the semantics of imperative programs containing loops. This encoding can be used to express both functional and temporal properties of loops, and is particularly suited to program analysis with an automated theorem prover. We employ it to automate functional verification, termination analysis and invariant generation for iterative programs operating over arrays.

Secondly, we describe how to extend theorems provers based on the superposition calculus to reason about datatypes and codatatypes, which are central to many programs. As the first-order theory of datatypes and codatatypes does not have a finite axiomatization, traditional means to perform theory reasoning in superposition-based provers cannot be used. We overcome this by introducing theory extensions as well as augmenting the superposition calculus with new rules.

# 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. These conditions were sometimes challenging, as we were located in different countries during most of my doctoral studies, but ultimately the challenges were met. These years have allowed me to grow into an independent researcher, which is the best that one could wish from a PhD supervisor.

Meanwhile Wolfgang Ahrendt filled his role of co-supervisor perfectly. He provided the guidance that I needed, whenever I needed it, and he allowed me to gather my thoughts during many fruitful discussions.

Wolfgang is also among the many people who make Chalmers the excellent workplace that it is. I dare not list them all, because the list is long and I would likely forget someone. Nevertheless, I am convinced that the research output of the department would not be what it is without its relaxed atmosphere and the institution of *fika*.

Jasmin Blanchette gave me the opportunity to work with him at Vrije Universiteit in Amsterdam. These three months were a very formative time and a pivotal part of my PhD.

While this thesis bears my name, it would not have been possible without the efforts of many others. I thank my co-authors for the discussions, the ideas and for sharing the stress in the hours before a deadline. I also thank Pascal Fontaine and Jeremy Pope for their comments on this thesis, and Evgeny Kotelnikov for his help with the formatting.

I am also indebted to the people who made it possible for me to even arrive to this PhD. This includes my parents, who not only supported me during my early years at the university, but convinced me to spend a few extra years there, not knowing how literally I would take them.

Lastly, I wish to thank Frédéric Loulergue, who gave 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.

This thesis was 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.

# Contents

**1 Introduction 1**

1.1 Deductive Program Analysis . . . . . . . . . . . . . . . . . . 2
  1.1.1 Abstractions of Programs . . . . . . . . . . . . . . . 2
  1.1.2 Program Semantics for Verification . . . . . . . . . 3
  1.1.3 Loop invariants . . . . . . . . . . . . . . . . . . . . 5
  1.1.4 First-Order Logic for Program Verification . . . . 6
1.2 First-Order Theorem Proving . . . . . . . . . . . . . . . . . 7
  1.2.1 Resolution . . . . . . . . . . . . . . . . . . . . . . . 8
  1.2.2 Paramodulation . . . . . . . . . . . . . . . . . . . . 9
  1.2.3 Restricted Calculi . . . . . . . . . . . . . . . . . . . 12
  1.2.4 Redundancy . . . . . . . . . . . . . . . . . . . . . . 13
  1.2.5 Theory Reasoning . . . . . . . . . . . . . . . . . . . 14
  1.2.6 Implementation of a Theorem Prover . . . . . . . . 15
1.3 Structure of the Thesis . . . . . . . . . . . . . . . . . . . . 16
1.4 Perspectives . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**2 Reasoning About Loops Using Vampire in KeY 23**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.2 Input Language . . . . . . . . . . . . . . . . . . . . . . . . . 27
  2.2.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . 27
  2.2.2 Semantics . . . . . . . . . . . . . . . . . . . . . . . 27
2.3 Invariant Generation Using Symbol Elimination . . . . . . 28
  2.3.1 Assertions . . . . . . . . . . . . . . . . . . . . . . . 29
  2.3.2 Extended Expressions . . . . . . . . . . . . . . . . 29
  2.3.3 Loop Analysis and Symbol Elimination . . . . . . 30
2.4 Extracting Loop Properties . . . . . . . . . . . . . . . . . . 30
  2.4.1 Properties of Scalar Variables . . . . . . . . . . . . 31
  2.4.2 Update Properties of Arrays . . . . . . . . . . . . . 31
  2.4.3 Assignments . . . . . . . . . . . . . . . . . . . . . . 33
  2.4.4 Additional Properties . . . . . . . . . . . . . . . . . 33
2.5 Loop Contract and Correctness . . . . . . . . . . . . . . . . 33

2.5.1 Pre-conditions . . . . . . . . . . . . . . . . . . . . . . . 34
2.5.2 Invariant Filtering . . . . . . . . . . . . . . . . . . . . 34
2.5.3 Direct Proof of Correctness . . . . . . . . . . . . . . 35
2.6 Integration with the KeY System . . . . . . . . . . . . . 35
2.6.1 Dynamic Logic . . . . . . . . . . . . . . . . . . . . . . . 36
2.6.2 Symbolic Execution . . . . . . . . . . . . . . . . . . . 36
2.6.3 Integration . . . . . . . . . . . . . . . . . . . . . . . . . 37
2.7 Experimental Results . . . . . . . . . . . . . . . . . . . . . 37
2.7.1 Invariant Generation . . . . . . . . . . . . . . . . . . 39
2.7.2 Invariant Filtering . . . . . . . . . . . . . . . . . . . . 39
2.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

**3 Loop Analysis by Quantification over Iterations 43**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . 47
3.2.1 First-Order Logic . . . . . . . . . . . . . . . . . . . . . 47
3.2.2 Program Semantics . . . . . . . . . . . . . . . . . . . . 48
3.2.3 Language of Assertions . . . . . . . . . . . . . . . . . 49
3.3 Extended Expressions . . . . . . . . . . . . . . . . . . . . 50
3.3.1 Syntax and Semantics . . . . . . . . . . . . . . . . . . 50
3.3.2 Relativised Formulas . . . . . . . . . . . . . . . . . . . 50
3.3.3 Axiomatization of Valid Loop Properties . . . . . . 51
3.4 Applications of Extended Expressions . . . . . . . . . . . 53
3.4.1 Verifying Partial Loop Correctness . . . . . . . . . 53
3.4.2 Termination, Safety, Liveness . . . . . . . . . . . . . 54
3.4.3 Invariant Generation Via Symbol Elimination . . . 55
3.5 Automated Reasoning with Extended Expressions . . . . 57
3.5.1 Avoiding Induction . . . . . . . . . . . . . . . . . . . . 57
3.5.2 Encoding of Natural Numbers . . . . . . . . . . . . . 59
3.5.3 Representation of Arrays . . . . . . . . . . . . . . . . 60
3.6 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . 60
3.6.1 Implementation . . . . . . . . . . . . . . . . . . . . . . 60
3.6.2 Experimental Results . . . . . . . . . . . . . . . . . . 61
3.7 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . 65
3.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

**4 Coming to Terms with Quantified Reasoning 69**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 71
4.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.3 The Theory of Finite Term Algebras . . . . . . . . . . . 75
4.3.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . . . 75

4.3.2 Known Results . . . . . . . . . . . . . . . . . . . . 76
4.3.3 Other Formalizations . . . . . . . . . . . . . . . . . . 78
4.3.4 Extension to Many-Sorted Logic . . . . . . . . . . 78
4.4 A Conservative Extension of the Theory of Term Algebras 79
4.5 An Extended Calculus . . . . . . . . . . . . . . . . . . . . 82
4.5.1 A Naive Calculus . . . . . . . . . . . . . . . . . . . . 82
4.5.2 The Distinctness Rule . . . . . . . . . . . . . . . . . 83
4.5.3 The Injectivity Rule . . . . . . . . . . . . . . . . . . 83
4.5.4 The Acyclicity Rule . . . . . . . . . . . . . . . . . . 84
4.6 Experimental Results . . . . . . . . . . . . . . . . . . . . . 84
4.6.1 Implementation . . . . . . . . . . . . . . . . . . . . . 84
4.6.2 Input Syntax and Tool Usage . . . . . . . . . . . . 85
4.6.3 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . 86
4.6.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . 86
4.6.5 Comparison of Option Values . . . . . . . . . . . . 89
4.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . 91
4.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

**5 An Inference Rule for the Acyclicity Property of Term Algebras 95**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.2 Term Algebras . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.2.1 First-Order Theory . . . . . . . . . . . . . . . . . . . 98
5.2.2 Acyclicity and Induction . . . . . . . . . . . . . . . 99
5.3 First-Order Logic and Superposition . . . . . . . . . . . . 100
5.4 An Inference Rule for Acyclicity . . . . . . . . . . . . . . 101
5.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . 103
5.5.1 Data Structures . . . . . . . . . . . . . . . . . . . . . 104
5.5.2 Retrieving Premises . . . . . . . . . . . . . . . . . . 105
5.6 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 111

**6 Superposition with Datatypes and Codatatypes 113**
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 115
6.2 Syntax and Semantics . . . . . . . . . . . . . . . . . . . . 116
6.3 Axioms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.3.1 Acyclicity . . . . . . . . . . . . . . . . . . . . . . . . 119
6.3.2 Contexts and Fixpoints . . . . . . . . . . . . . . . . 120
6.3.3 Soundness and Completeness . . . . . . . . . . . . 122
6.4 Inference Rules . . . . . . . . . . . . . . . . . . . . . . . . . 125

- 6.4.1 Superposition . . . . . . . . . . . . . . . . . . . . . . . . . 125
- 6.4.2 Infiniteness . . . . . . . . . . . . . . . . . . . . . . . . . . 125
- 6.4.3 Distinctness . . . . . . . . . . . . . . . . . . . . . . . . . 126
- 6.4.4 Distinctness . . . . . . . . . . . . . . . . . . . . . . . . . 126
- 6.4.5 Distinctness . . . . . . . . . . . . . . . . . . . . . . . . . 127
- 6.4.6 Injectivity . . . . . . . . . . . . . . . . . . . . . . . . . . 127
- 6.4.7 Acyclicity . . . . . . . . . . . . . . . . . . . . . . . . . . 128
- 6.4.8 Uniqueness of Fixpoints . . . . . . . . . . . . . . . . . . . 132

6.5 Refutational Completeness . . . . . . . . . . . . . . . . . . . . . 134
6.6 Saturation Procedure . . . . . . . . . . . . . . . . . . . . . . . . 143
6.7 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
6.8 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
6.9 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150

**Bibliography** **151**

## CHAPTER 1

# Introduction

Computer systems are now used in a vast array of human activities, from mundane tasks to safety-critical functions. In many of those applications, software faults can have important negative consequences, either financial or human. Just as computer systems have become more prevalent, they have also become more complex. Avoiding faults in software is more necessary than ever, but also more difficult.

Traditionally, the goal of detecting and avoiding software faults has been accomplished with systematic testing. By running a program from a pre-determined configuration, it is relatively easy to check that the result produced conforms to the intent of the developer. However this approach suffers from a major limitation, famously described by Dijkstra: "Program testing can be used to show the presence of bugs, but never to show their absence." Furthermore, testing can only cover a finite number of situations, whereas even moderately complex programs can run in an infinite number of different ways. In order to achieve a higher degree of safety, it is necessary to go beyond testing, and to instead prove that a program is correct, by use of *formal methods* based on the theoretical foundations of programming.

Formal methods have been studied for more than half a century [44,52, 67,88,124]. They offer various mathematical representations of programs and means to prove some properties of these representations. By using abstract reasoning, formal methods can ensure that a program behaves as expected over an infinite domain of input values, and thus help ensure a degree of software quality that is not achievable with mere testing. Despite the higher assurance granted by formal methods, they have, for a long time, only been used on small examples. This is in part because these methods are often hard to adopt, and remain the prerogative of experts. Besides the technical difficulty, the sheer amount of work needed to prove the correctness of large programs can be overwhelming.

In order to overcome these obstacles, we need assistance in the form of tools to automate (parts of) the proving process. *Automated theorem* *proving* – the use of automatic methods to carry out mathematical reasoning and prove (or disprove) logical statements – has a history that precedes the advent of computers. Today, it remains an active field of research that takes advantage of increased hardware capabilities as well as theoretical and algorithmic developments to push the boundaries of what can be proven by computers. Automated theorem provers can be used to reason about all sorts of mathematical questions, and they are particularly well suited to problems of program verification, which often require proving a very large number of relatively simple logical assertions. Proving properties of programs is a challenging task that cannot be fully automated. Nevertheless, automated systems are able to find simple proofs without human guidance, and to provide some assistance in more complex problems. In the context of program verification, such tools can help reduce the work required to prove correctness, and make formal methods more viable.

Increasing the success rate of automated theorem provers for program verification requires a concerted effort between the *users* of provers and their *developers*. The former must make sure that the semantic representations of programs are suitable for theorem provers and exploit their strengths. The latter have to provide features to reduce the burden of encoding these representations, and leverage domain knowledge to increase the performance of provers. This thesis explores both of these avenues of research.

## 1.1 Deductive Program Analysis

### 1.1.1 Abstractions of Programs

A natural way to give a precise description of a programming language is to describe, from a computational point of view, how its different syntactic constructs operate. This style of description is called *operational semantics* [71, 107].

The nature of the description varies greatly with that of the language. For example, a functional language will typically be characterized by the rewrite rules that govern the evaluation of expressions. For an imperative programming language, we may instead define the effects of its commands on some idealized memory model. In its simplest form, this memory model will be a mathematical structure mapping program locations to values. Then an assignment can be defined as an operation that takes a program state (including the mapping) and returns an updated program state with a modified mapping. A precise description of a real programming language will of course require a more complex model [18].

The advantages of operational semantics are based on a close correspondence to the implementation of the language. Programmers will find the style quite natural and informative, while language developers can use an operational description of the semantics to implement an interpreter with minimal effort. This style of semantics is also needed in the development of verified compilers [86, 93].

The low level of abstraction of operational semantics means that the mathematical representation of a program includes a number of details that may not be relevant to the task of program verification, which is often less concerned with *how* a program computes than *what*. *Denotational semantics* takes a higher-level view of a program, describing it as a mathematical object, for example a partial function (or more generally a relation) mapping input to output. This style of semantics can be very useful to give a description of language features that cannot be fully described using a computational representation, such as non-determinism or concurrency. It is also useful to compare programs in different languages, as the abstraction is independent of the syntax of the program.

Denotational semantics abstracts many of the computational details of the programming language, but this often requires more advanced mathematical concepts that those used for operational semantics. For the representation of loops and recursive functions, denotational semantics makes use of fixed-point constructions. This representation is not only non-computational, but it is also a complex mathematical notion about which it is difficult to reason automatically.

### 1.1.2 Program Semantics for Verification

A third way to abstract programs is *axiomatic semantics*. Here, it is not the program itself that is represented, but rather its effect on logical assertions about the program states. Axiomatic semantics is particularly suited to program verification, where assertions are used to specify the intended behavior of a program. The most famous example of axiomatic semantics is Hoare logic [67], whose central syntactic feature is the Hoare triple

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

where $P$ and $Q$ are logical assertions about program states, and $\pi$ is a program. Informally, such a triple can be understood as “if the program state satisfies the assertion $P$ before the execution of $\pi$, and $\pi$ terminates, then the state satisfies the assertion $Q$." For each program construct, an axiomatic rule describes how the construct relates to assertions. For example, program composition is described by a rule that takes for premises triples about two programs, and combines them to infer a triple about their composition:

$$\frac{\{P\}\,\pi_1\,\{Q\} \qquad \{Q\}\,\pi_2\,\{R\}}{\{P\}\,\pi_1;\pi_2\,\{R\}}$$

Atomic commands correspond to rules without premises. For example assignments can be axiomatized as:

$$\{P[x \leftarrow e]\}\,x := e\,\{P\}$$

where $P[x \leftarrow e]$ denotes the substitution of all occurrences of the variable $x$ by the expression $e$ in the formula $P$. For complex program constructs, it may not be easy to convince oneself that such axioms provide a correct description. For this reason, it is common to use operational semantics as a basis to justify the soundness of each rule [42].

Besides the rules describing program constructs, the rule of consequence allows the generalization of triples according to the notion of consequence in the assertion language:

$$\frac{P \implies P' \qquad \{P'\}\,\pi\,\{Q'\} \qquad Q' \implies Q}{\{P\}\,\pi\,\{Q\}}$$

The rules above form a calculus that can be used to prove that a Hoare triple is valid, thus guaranteeing that the program satisfies a given specification. Most of the inference rules in this calculus require the use of intermediate lemmas. For example, in the rule for composition above, the assertion $Q$ is present in the premises but not the conclusion. Consequently, the completeness of the calculus depends on

(i) the axioms themselves;

(ii) the existence of a complete deductive system for assertions introduced by the consequence rule;

(iii) the ability of the assertion language to express the required intermediate lemmas.

Cook [42] defined a suitable notion of relative completeness that isolates those requirements, and proved that under assumptions (ii) and (iii), a complete system could be obtained for a Turing-complete language. However this is not always the case, and many naturally occurring language constructs prevent the existence of such a system [35].

The use of intermediate lemmas in the rules makes the calculus poorly suited to automated proof search. This problem was partially solved by Dijkstra [51], who provided a calculus based on *predicate transformers* to prove the correctness of programs. A predicate transformer for a given program is a function on assertions. For example we can make use of a predicate transformer taking a program $\pi$ and an assertion $Q$, and returning the weakest (i.e., most general) condition that is required to hold before the exectution of $\pi$ for $Q$ to be true after that execution. To prove program correctness, it remains only to show that the actual pre-condition given in the specification is at least as strong as the condition returned by the predicate transformer:

$$P \implies \mathrm{pre}(\pi, Q)$$

Equivalently, it is possible to use predicate transformers based on the strongest post-condition of a program [101].

### 1.1.3 Loop invariants

The predicate transformer calculus avoids the issue of intermediate lemmas for most program constructs. For example for the composition of programs, the weakest pre-condition can be computed in two steps:

$$\mathrm{pre}(\pi_1 ; \pi_2\, ,\, Q) = \mathrm{pre}(\pi_1, \mathrm{pre}(\pi_2, Q))$$

However the weakest pre-condition is generally not computable in the presence of loops. The solution adopted by the predicate transformer calculus is to use a specific kind of intermediate lemma, a *loop invariant*. An invariant for a given loop is an assertion whose truth is preserved by any execution of the loop body. Evidently, if such an assertion is true in the state where a loop execution starts, it also holds when the loop terminates. This justifies the definition of the Hoare rule for loops:

$$\frac{\{I \wedge C\}\, \pi\, \{I\}}{\{I\}\, \text{while } C \text{ do } \pi\, \{I\}}$$

The definition of a predicate transformer for loops relies on those loops being annotated with an invariant, which must be provided by the program developer. Coming up with an arbitrary invariant is not difficult: the always true and always false formulas $\top$ and $\bot$ fit the definition, for any program. The challenge is to find an invariant that is strong enough to imply the post-condition to verify, while also being implied by the pre-condition. In that sense, the predicate transformer calculus requires a step of “invention” to prove the correctness of programs with loops, similar to the act of finding an inductive hypothesis to perform a proof by induction. In program analysis as in inductive theorem proving, the necessity to come up with new formulas during the proving process hinders automation.

Given the undecidability results for properties of Turing-complete languages, finding a fundamental obstacle to automation is not surprising. Nevertheless, it is possible to use automated methods to generate some invariants, and increase the degree to which program verification can be automated. Some of those methods are in some sense complete, but impose strong restrictions on the nature of programs and invariants that are targeted. For example there exist methods to generate only polynomial invariants [79] or that require user-provided templates to impose syntactic constraints on the invariants [39, 63] generated. Other techniques are heuristic in nature and instead attempt to generate useful invariants on a best-effort basis. They are very useful for commonly used but mathematically complex programs such as loops iterating over arrays [45, 82].

### 1.1.4 First-Order Logic for Program Verification

In addition to a representation of the program semantics, an appropriate mathematical language must be chosen. Even axiomatic semantics, where logical statements are at the center of the abstraction, is formulated in a way that is largely independent of the language used for assertions. The choice of a logical language is largely a balancing act between expressivity and ease of reasoning, especially in the context of automation. For example, propositional logic is decidable, and there exists efficient tools to reason about its problems. However it lacks the ability to describe infinite domains, a requirement for many tasks of program verification. First-order logic arguably offers the right level of expressivity to reason about most programs, thanks to the ability to quantify over the values manipulated by those programs. This type of quantification is often a necessity to express meaningful properties of programs. This added expressivity comes at a cost: first-order logic is not decidable, but merely semi-decidable.

Higher-order logics provide even more expressivity, but in general automated tools to reason about them [17, 28] do not perform as well as their first-order counterpart, especially on large problems. Even for programming languages that feature higher-order functions, first-order logic is often sufficient to express most interesting properties. One level of universal quantification over functions can be simulated by using uninterpreted function symbols. Deeper higher-order reasoning (e.g., proving the existence of a function) is rarely needed.

Other approaches use logics that are especially tailored to describe properties of programs. For example, the language of separation logic [116] includes operators specifically used to describe properties of memory. Similarly, logics with modalities can be used to describe temporal properties of programs [108] or to embed program fragments in logical statements [65]. Matching logic [121] offers a way to reason directly about the operational semantics of programs. Automating reasoning in these logics usually necessitates the development of new techniques and tools. In contrast, deductive reasoning in first-order logic is a well studied topic, that can be carried out by efficient tools.

## 1.2 First-Order Theorem Proving

In order to best employ automated theorem provers for program analysis, it is necessary to understand how they operate. In this thesis, we focus on saturation based theorem provers, which work by refutation: checking the validity of a conjecture, or its entailment by axioms, is reduced to checking the unsatisfiability of a sentence.

Early methods for refutation in first-order logic [48, 109] work by enumerating the ground instances of a (Skolemized) sentence until an inconsistent instance is found. Checking the consistency of a ground instance is a problem of propositional logic, and therefore decidable. That technique is a direct application of Herbrand's theorem, which guarantees that the process will terminate if the sentence is unsatisfiable. Since the enumeration depends on the signature of the problem rather than the sentence itself, the search for a refutation is undirected, and therefore very inefficient.

A better approach is to use the structure of the problem to find a refutation. This is the goal of saturation, which works on a clausal representation of the sentence. Inferences are performed among the set of clauses, the conclusion added to the set and the process iterated until (a) the empty clause is derived, yielding a refutation or (b) no more inferences can be performed, i.e., the set is *saturated*. If the calculus used is refutationally complete, and if a fair strategy is used (so that no inference can be delayed indefinitely), saturation of an unsatisfiable set of clauses will eventually terminate in (a). Termination in (b) indicates that the set of clauses is satisfiable, but because first-order logic is semi-decidable, saturation does not always terminate on satisfiable sets.

### 1.2.1 Resolution

A refutationally complete calculus was proposed by Robinson [119], who leveraged term unification to extend the principle of propositional resolution to first-order logic.

In order to present the calculus, let us fix some definitions. An *atom* is 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 a positive or negative occurrence of an atom, and a *clause* is a finite disjunction of literals, viewed as a multiset. Terms occurring in clauses may feature variables (denoted $x, y, z \ldots$) that are interpreted as universally quantified. A substitution is a function from variables to terms. Application of substitutions to variables (and by extension, to terms, literals and clauses) is denoted in postfix notation. A substitution $\theta$ is a *unifier* of $s$ and $t$ if $s\theta = t\theta$. Moreover, if every unifier of $s$ and $t$ is an instance of $\theta$, then $\theta$ is said to be a *most general unifier* (mgu).

The resolution rule is as follows:

$$\frac{L \vee \mathcal{C} \qquad \neg L' \vee \mathcal{D}}{(\mathcal{C} \vee \mathcal{D})\theta}\ \textsf{Res}$$

where $\theta$ is an mgu of the literals $L$ and $L'$. Resolution is a generalization of the principle of *modus ponens.* It finds contradicting parts of two clauses and combines the remaining literals to form a new clause, the *resolvent.* Like the enumeration method, first-order resolution instantiates the clauses, by means of a unifier. A crucial difference is that the instantiation is partial: the use of an mgu ensures that the clauses are instantiated in the most general way required to obtain a contradiction, and no further. This can be seen as combining the two steps of the enumeration method (generating instances, and testing them for inconsistency) in a single operation.

In addition to the resolution rule, the calculus also includes the factoring rule to remove unifiable literals occurring in the same clause

$$\frac{L \vee L' \vee \mathcal{C}}{(L \vee \mathcal{C})\theta}\ \textsf{Fact}$$

where $\theta$ is an mgu of $L$ and $L'$.

To prove the refutational completeness of the calculus, we first focus on the case where all clauses are ground. The proof is obtained by the contrapositive: given a saturated set of ground clauses $N$ that does not contain the empty clause, the set is proven satisfiable by the construction of a Herbrand model. We assume a total order $\prec$ on literals and obtain, by the multiset extension, an order on clauses. The construction of the model starts with an empty Herbrand interpretation – where all atoms are false – which is then iteratively enriched by considering the clauses in the order defined above. If a clause is not satisfied by the interpretation, its maximal literal must occur positively, so the clause has the form $L \vee \mathcal{C}$ and $\mathcal{C}$ is not satisfied by the interpretation. We can add $L$ to the Herbrand model without falsifying any of the clauses considered before. This can be proven by contradiction: if such a clause were falsified, it would have the form $\neg L \vee \mathcal{D}$, with $\mathcal{D}$ not satisfied by the interpretation. The two clauses form the premises of a resolution inference, and since the set $N$ is closed under resolution, it must contain the conclusion $\mathcal{C} \vee \mathcal{D}$. Furthermore, the conclusion is smaller that the premises, so it must be satisfied by the model constructed so far, a contradiction.

Having proven the completeness of the calculus on ground clauses, the proof can be extended to also cover the case of clauses containing variables. This is accomplished by an argument of *lifting*, which is essentially an application of Herbrand's theorem in the context of clausal formulas. Given that a Herbrand interpretation is a model of a set of clauses if and only if it is a model of all of its ground instances, we can prove the satisfiability of a set of clauses by constructing a Herbrand model of its ground instances, as we have already demonstrated. Perhaps surprisingly, most of the effort required to prove the completeness of first-order resolution is spent on the ground (i.e., propositional) case.

For the model construction to be correct, the set of ground instances must be saturated. To ensure this, we must be able to lift every inference: if an inference can be performed between ground instances of some clauses, it must be an instance of an inference that is also possible between those first-order clauses. This condition holds for all resolution and factoring inferences, so the saturation of the set of first-order clauses implies the saturation of the set of its ground instances.

### 1.2.2 Paramodulation

In many problems of first-order logic, the equality predicate (which we denote $\approx$) plays an central role. The equality predicate can be finitely axiomatized, so that first-order equality problems can be dealt with using resolution, by including the axioms in the set of clauses to saturate. This is the standard approach to perform theory reasoning in first-order theorem provers.

This presents many drawbacks. Firstly, the axiomatization of equality, although finite, requires a large number of sentences. Along with the three properties of equivalence

$$\forall x.\, x \approx x$$

$$\forall xy\, (x \approx y \implies y \approx x)$$

$$\forall xyz\, (x \approx y \wedge y \approx z \implies x \approx z)$$

the axioms must also describe the monotonicity of equality under functions and predicates. For every $n$-ary function symbol $f$ we have

$$\forall \bar{x}\bar{y}\ (x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n \Rightarrow f(x_1, \ldots, x_n) \approx f(y_1, \ldots, y_n))$$

and likewise, for every predicate symbol $P$

$$\forall \bar{x}\bar{y}\ (x_1 \approx y_1 \wedge \cdots \wedge x_m \approx y_m \wedge P(x_1, \ldots, x_m) \implies P(y_1, \ldots, y_m))$$

Thus, the axiomatization requires a number of sentences linear in the size of the problem signature. More importantly, the properties of equality mean that positive occurrences of the equality predicate are extremely prolific: they can be used to infer a very large number of clauses, few of which will eventually be used in the refutation proof. For this reason, the idea of using a finite axiomatization of equality together with a resolution based prover is very impractical: for all but the simplest problems, the search space quickly becomes too large for proofs to be found.

A possible improvement is to treat $\approx$ as part of the logical language (rather than the problem signature) and use dedicated rules to capture its properties. Reflexivity is captured by the equality resolution rule

$$\frac{s \not\approx s' \vee \mathcal{C}}{\mathcal{C}\theta}\ \mathsf{EqRes}$$

where $\theta$ is an mgu of $s$ and $s'$.

The key component of the calculus that captures the remaining properties is the paramodulation rule [138]:

$$\frac{t \approx s \vee \mathcal{C} \qquad [\neg]v[t'] \approx u \vee \mathcal{D}}{([\neg]v[s] \approx u \vee \mathcal{C} \vee \mathcal{D})\theta}\ \mathsf{Sup}$$

where $t'$ is not a variable and $\theta$ is an mgu of $t$ and $t'$. By $[\neg]$ we denote the fact that the rule may be applied to either positive or negative literals, the literal in the conclusion having the same polarity as the one in the right premise.

The completeness of the paramodulation calculus can be proven in the same fashion as for resolution. Since we assign a specific interpretation to the equality predicate, standard Herbrand interpretations are impractical. Instead, we now use a term rewriting system $R$. We assume a *simplification order* $\prec$ on terms, i.e., an order that:

(i) is *compatible with term operations*: for any terms $s, t, u$ and any term position $p$, $s \prec t$ implies $u[s]_p \prec u[t]_p$;

(ii) is *closed under substitution*: for any terms $s$ and $t$ and any substitution $\theta$, $s \prec t$ implies $s\theta \prec t\theta$;

(iii) has the *subterm property*: for any terms $s$ and $t$, if $s$ is a proper subterm of $t$ then $s \prec t$.

These properties ensure that the order is well founded. Furthermore, $\prec$ must be total on ground terms. Each equality atom added to the system can then be oriented and interpreted as a rewrite rule. A ground literal $s \approx t$ is true in this interpretation if and only if the pair $(s, t)$ belongs to the rewrite relation corresponding to $R$, denoted $s \overset{*}{\longleftrightarrow}_R t$. By the properties of the simplification order, this is the case only if there exists a term $u$ such that $s \overset{+}{\longrightarrow}_R u$ and $t \overset{+}{\longrightarrow}_R u$, that is, all ground terms that are equal in the model defined by $R$ have a unique normal form.

For the ground case, the proof of completeness follows the same pattern as for the resolution calculus. One complication is that it is more difficult to guarantee that the addition of a rewrite rule does not falsify previously considered clauses. The equality factoring rule helps ensure that invariant:

$$\frac{u \approx t \vee u' \approx s \vee \mathcal{C}}{(u \approx t \vee t \not\approx s \vee \mathcal{C})\theta}\ \mathsf{EqFact}$$

where $\theta$ is an mgu of $u$ and $u'$.

Transposing the proof of completeness to non-ground clauses poses a challenge, as some instances of the paramodulation rule cannot be lifted. For example consider the clauses $s \approx t$ and $P(x) \vee Q(x)$. Among ground instances of these two clauses, inferences can be performed, resulting for example in the conclusion $P(t) \vee Q(s)$. However, since paramodulation cannot occur at variable positions, no first-order inference can be performed between the two clauses.

Because the grounding of an inference does not include all the inferences that are possible between the groundings of its premises, the saturation of a set of first-order clauses does not directly imply that the set of its ground instances is also saturated. To prove completeness, we must observe that the conclusion of a ground instance of a paramodulation inference that cannot be lifted is implied by the (smaller) conclusion of other instances, and thus not needed to construct the model.

The omission of paramodulation at variable positions is critical for the efficiency of the calculus, as the rule would otherwise be very prolific. In fact, if the signature of the problem contains at least one function symbol, we also need to consider paramodulation *under* variable positions to ensure that the rule can be lifted: for example, the clause $P(f(f(t))) \vee Q(f(f(s)))$ would also be the conclusion of a ground instance of paramodulation. Lifting is often a significant challenge in the design of a calculus for first-order logic. The original presentation of paramodulation [138] relied on paramodulation at variable position, together with additional axioms (one per function symbol) to obtain completeness, before the rule could be proven complete without those [27].

### 1.2.3 Restricted Calculi

An important consideration in techniques for automated theorem proving is the reduction of the size of the search space, in order for proofs to be found within reasonable time limits. One way to achieve this is to limit the number of inferences that can be performed between clauses of a given set. Such restrictions will limit the expressivity of the calculus, in the sense that short proofs that could be expressed in the non-restricted calculus will potentially be lost. On the other hand, the reduced number of possible inferences limits the number of "guesses" that must be made in order to derive the empty clause. In the context of automated theorem proving, the second point vastly outweighs the first.

*Ordered resolution* is a refutationally complete restriction of resolution. It makes use of the following observation: to prove the completeness of resolution, we used a total order on literals, and a positive literal was used in the construction of the Herbrand model only if it occurred maximally in a clause. Therefore, resolution inferences need to be performed only when the positive literal resolved upon is maximal in its clause. Other inferences may be dropped from the calculus without compromising its refutational completeness. In order to implement this restriction, the calculus is parameterized by an order on literals and a *selection function* that must return a non-empty set of literals in any non-empty clause [85]. If the selection function is *well-behaved*, that is, it always returns a negative literal or all the maximal literals in a clause, then it is possible to restrict inferences to selected literals without losing completeness. The notation $\boxed{L} \vee C$ indicates that the literal $L$ is selected in the clause.

$$\frac{L \vee \mathcal{C} \qquad \neg L' \vee \mathcal{D}}{(\mathcal{C} \vee \mathcal{D})\sigma}\ \mathsf{Res} \qquad\qquad \frac{s \not\approx s' \vee \mathcal{C}}{\mathcal{C}\theta}\ \mathsf{EqRes}$$

*where $\sigma$ is an mgu of $L$ and $L'$, $\theta$ is an mgu of $s$ and $s'$, and $L$ is not an equality literal*

$$\frac{t \approx s \vee \mathcal{C} \qquad L[t'] \vee \mathcal{D}}{(L[s] \vee \mathcal{C} \vee \mathcal{D})\theta}\ \mathsf{Sup}^P$$

$$\frac{t \approx s \vee \mathcal{C} \qquad v[t'] \approx u \vee \mathcal{D}}{(v[s] \approx u \vee \mathcal{C} \vee \mathcal{D})\theta}\ \mathsf{Sup}^+ \qquad \frac{t \approx s \vee \mathcal{C} \qquad v[t'] \not\approx u \vee \mathcal{D}}{(v[s] \not\approx u \vee \mathcal{C} \vee \mathcal{D})\theta}\ \mathsf{Sup}^-$$

*where $t'$ is not a variable, $L$ is not an equality literal, $\theta$ is an mgu of $t$ and $t'$, $s\theta \not\succeq t\theta$ and $u\theta \not\succeq v[t']\theta$*

$$\frac{L \vee L' \vee \mathcal{C}}{(L \vee \mathcal{C})\sigma}\ \mathsf{Fact} \qquad\qquad \frac{u \approx t \vee u' \approx s \vee \mathcal{C}}{(u \approx t \vee t \not\approx s \vee \mathcal{C})\theta}\ \mathsf{EqFact}$$

*where $\sigma$ is an mgu of $L$ and $L'$, $\theta$ is an mgu of $u$ and $u'$, $s\theta \not\succeq t\theta$ and $t\theta \not\succeq u\theta$*

Figure 1.1. The superposition calculus $\mathcal{SP}$ .

The same restriction can be applied to paramodulation, but we can also go further and break the symmetry of equality, in a manner similar to procedures used to solve equational problems [76]. In the proof of completeness of paramodulation, we assumed a simplification order that is total on ground terms, so that ground equalities could be oriented and treated as rewrite rules. This leads to the observation that paramodulation needs to be performed (on ground clauses) only if $s \prec t$ and $u \prec v[s]$. The generalization of the simplification order to the first-order is necessarily an under-approximation, and cannot be total. So for non-ground clauses, the restrictions are relaxed to $s\theta \not\succeq t\theta$ and $u\theta \not\succeq v[s']\theta$. This restriction of paramodulation gives us *superposition* [5, 6].

With this last refinement, we can now give a full picture of the superposition calculus, in Figure 1.1. In some presentations, equality is the only predicate, and the rules Res, Fact and $\mathsf{Sup}^P$ are omitted. This logic is as expressive: non-equality predicates can be encoded as functions. Together with a constant $\bot$, (dis)equality literals can encode the truth of those predicates.

### 1.2.4 Redundancy

Having restricted the search space at the level of literals and terms, we finally turn our attention to clauses themselves. If we can remove clauses from a set $N$ without affecting its (in)consistency, then doing so will not affect the refutational completeness of the saturation process (although some care must be taken not to affect fairness). For this reason, the calculi used in theorem provers are paired with a notion of redundancy that describes which clauses may be removed from a set. A general criteria for redundancy is the following: a clause $\mathcal{D}$ is redundant in a set $N$ if there exist $\{\mathcal{C}_1, \dots, \mathcal{C}_n\} \subseteq N$ where $\mathcal{C}_1, \dots, \mathcal{C}_n \models \mathcal{D}$ and $\mathcal{C}_i \prec \mathcal{D}$ for $1 \leqslant i \leqslant n$. Intuitively, redundant clauses are those that will not be used in the construction of the model.

This criteria is based on the notion of entailment, which is not decidable, so in practice provers have to settle for an (easily computable) under-approximation of that criteria. For example, tautologies are always redundant. Another example is *subsumption*: a clause $\mathcal{C}$ subsumes a clause $\mathcal{D}$ if there exists $\mathcal{C}'$ and $\theta$ such that $\mathcal{D} = \mathcal{C}\theta \vee \mathcal{C}'$.

### 1.2.5 Theory Reasoning

In many applications of first-order theorem proving, we need to consider specific mathematical structures. For example in the context of program verification, integers, arrays, bit vectors or recursive data structures are common and we need ways to reason efficiently about them.

The most direct way to perform theory reasoning in a saturation theorem prover is to add theory axioms to the set of clauses to saturate. In some cases, proof search may be dominated by inferences between theory axioms and their consequences, neglecting the conjecture. The set of support strategy [139] can be used to restrict inferences among axioms and ensure a goal-directed search [111]. This approach is conceptually simple, does not require any modification to the solver itself, and can be used with any theory. Even if the theory is not axiomatizable in first-order logic, a partial axiomatization may be included. This is an easy, if very incomplete, solution to reason about some problems in non-axiomatizable theories.

For theories that are axiomatizable in first-order logic only with an infinite set of axioms, refutationally complete reasoning is more difficult to reach. It is theoretically possible to modify the saturation algorithm to interleave the enumeration of axioms and the inference of new clauses, in a manner that preserves the fairness of the saturation. However, this would require extensive modification to the prover. An easier solution, when applicable, is to provide a conservative extension of the theory $\mathcal{T}$. By using symbols outside of the language of $\mathcal{T}$, it is sometimes possible to provide a finite axiomatization $A$ of a theory $\mathcal{T}^+$ such that all theorems of $\mathcal{T}^+$ in the language of $\mathcal{T}$ are also theorems of $\mathcal{T}$. It is then possible to use that finite axiomatization in the saturation process: for any sentence $F$ in the language of $\mathcal{T}$, the unsatisfiability of $A \wedge \neg F$ implies that $F \in \mathcal{T}$.

Even in cases where a theory has a finite axiomatization, saturation of the axioms can be very inefficient. We have already given the example of the theory of equality, which suggests that the use of dedicated inference rules to replace axioms and perform theory reasoning can lead to improved performance. Generally, the soundness of these rules (w.r.t. to the intended interpretations) is easy to prove, but completeness is a different matter. Proving completeness requires the construction of a model for saturated sets of clauses. In the case of theory reasoning, additional properties must be checked to ensure that the structure that is built is not only an interpretation of the clauses, but also of the theory.

More recently, there has been work on combining SMT solvers and first-order theorem provers to reason about quantified theory problems. The AVATAR architecture can be used to combine the first-order reasoning power of superposition with SMT solvers to reason on ground clauses [110]. SMT solvers can also be used to reason about non-ground clauses, by helping instantiate them [112].

### 1.2.6 Implementation of a Theorem Prover

Beyond the theoretical foundations presented so far, the success of automated theorem proving requires the implementation of efficient provers. Heuristics play an important role in this task. The superposition calculus presented here can be parameterized in many different ways (simplification order on terms, selection functions), and is often extended by ad-hoc rules.

The design of a saturation algorithm is also crucial. Typically, saturation algorithms follow the *given clause* method. Clauses are partitioned in two sets: *passive* (initially containing all the clauses) and *active clauses* (initially empty), with the invariant that the set of active clauses remains saturated. A passive clause is selected to become the given clause, all possible inferences between it and active clauses are performed and their conclusions added to the passive clauses. Lastly the given clause becomes active, and the process can be repeated until refutation is found or the set of passive clauses becomes empty. The saturation algorithm must also perform redundancy elimination and clause simplification: this can be done *forward* – using the active clauses to simplify the given clause – or *backward*, in which case the given clause is used to simplify active clauses, that then need to be put back among passive clauses. The implementation of those features is often the defining trait of a saturation algorithm. In some cases, it can even be useful to consider incomplete saturation strategies, sacrificing theoretical completeness for practical efficiency [117].

On top of those choices, the developer of a theorem prover faces many engineering challenges. Even with the effort to eliminate redundant clauses, it is not uncommon for provers to handle millions of clauses at a given time. In these conditions, queries for unifiable terms or subsumed clauses cannot be answered by testing all the candidates iteratively. Instead, indexing data structures are used to retrieve terms and clauses as efficiently as possible [125]. Term order (under substitutions) also needs to be checked efficiently.

Multiple first-order theorem provers based on the superposition calculus are available and under active development today, including E [123], Spass [137], Vampire [84] and Zipperposition [46]. The CASC [127] and SMT [36] competitions offer an opportunity to observe the latest developments in the field of automated theorem proving.

## 1.3 Structure of the Thesis

This thesis describes contributions to the field of program analysis and verification. It focuses on the use of first-order theorem provers to perform those tasks, and considers the issue from the point of view of the users of theorem provers as well as that of their developers. Accordingly, the thesis is organized along two main axes of research:

- We describe a novel way to encode the semantics of imperative programs containing loops. This encoding is particularly suited to conducting program analysis and verification using a first-order theorem prover. This work is described in Chapters 2 and 3.
- We describe ways to reason about the theories of datatypes and codatatypes using a saturation-based theorem prover. These theories are particularly useful in program analysis, since many programming languages use these types as the main representation of data. This work is described in Chapters 4, 5 and 6.

### Paper 1: Reasoning About Loops Using Vampire in KeY

The symbol elimination method is a novel way to generate invariants that relies on the consequence finding mechanism provided by first-order theorem provers. It was originally introduced in [82]. 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: Loop Analysis by Quantification over Iterations

This paper formalizes the encoding of the semantics of loop programs that was originally introduced for symbol elimination in [82] and further extended in the previous paper. It also describes new applications of this encoding. Contributions include:

- the formalization of the semantics of the language of extended expressions used for symbol elimination;
- an axiomatization of the theory of extended expressions that hold for a given loop, and a proof of its completeness (up to completeness of the background theory);

- the use of extended expressions to express and verify functional and temporal properties about programs, in particular partial correctness and termination;
- a proof of the soundness of the symbol elimination method for invariant generation;
- experiments with different background theories, in particular arrays and natural numbers, and the comparison of various provers on these encodings.

**Statement of contribution.** This paper is co-authored with Bernhard Gleiss and Laura Kovács. Simon Robillard is the main author.

It was originally published in the peer-reviewed *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 22)* and presented in Awassa, Ethiopia.

## Paper 3: Coming to Terms with Quantified Reasoning

Many 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 these algebras is therefore crucial to analyze functional programs and verify their correctness. In the 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, improving the efficiency of reasoning about problems with term algebras;
- the implementation of the above in the first-order theorem prover VAMPIRE.

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

It was originally published in the peer-reviewed *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)* and was presented in Paris, France.

## Paper 4: An Inference Rule for the Acyclicity Property of Term Algebras

Acyclicity is the property of term algebras that prevents their finite axiomatization. Instead of relying on a conservative extension of the theory to encode the property, this paper proposes an inference rule aimed at capturing it. The paper contributes:

- the description of a rule to capture the acyclicity property of term algebras, and a proof of it soundness;
- details of an efficient implementation of the rule, based on term indexing techniques;
- experimental evidence that the rule outperforms the conservative extension on hard term algebra problems.

**Statement of contribution.** Simon Robillard is the sole author of this paper.

It was originally published in the *Proceedings of the 4th Vampire Workshop* and presented in Gothenburg, Sweden.

## Paper 5: Superposition of Datatypes and Codatatypes

This paper applies the ideas of a conservative extension of a theory and an extended superposition calculus to co-algebraic data types. The main difference between this theory and that of algebraic data types (term algebras) is that the acyclicity property is replaced by the existence of unique fixpoints: cyclic terms exists, and observably similar cyclic terms are equal. The paper also refines the idea of using a calculus to replace some axioms of algebraic data types. The contributions of this paper are the following:

- a conservative extension of the theory of co-algebraic data types based on a finite number of axioms;
- a modification of the acyclicity rule described in the previous paper that makes it complete, in the presence of some axioms;
- a similar approach for the uniqueness of co-algebraic data type fixpoints;
- rules replacing the axioms of distinctness and injectivity common to both algebraic and co-algebraic data types, while preserving completeness;

- proofs of completeness and soundness of the resulting (modular) calculus;
- the implementation of the above in the first-order theorem prover VAMPIRE.

**Statement of contribution.** This paper was co-authored with Jasmin Blanchette and Nicolas Peltier. Simon Robillard was the instigator of the paper. The proof of completeness of the calculus is due to Nicolas Peltier.

It was originally published in the peer-reviewed *9th International Joint Conference On Automated Reasoning* and presented in Oxford, United Kingdom. It is reproduced here in an extended version previously published as a technical report.

## 1.4 Perspectives

A recent trend in the world of automated theorem proving is the convergence of two opposite approaches: model construction (SMT solving) and refutation (saturation-based proving). Historically, the former has been the preferred way to deal with problems featuring theory reasoning, while the latter was able to handle full first-order quantification. In practical applications, problems commonly include both theories and quantifiers. For this reason, researchers are now trying to bridge the gap in both directions. State-of-the-art SMT solvers are equipped with means to deal with quantification [49, 60, 61], while saturation-based provers are extended to reason about various theories, an effort to which this thesis contributes. The combination of the two approaches in a single prover [110, 112] is also a promising venue of research. Another ongoing development is the extension of these proving techniques to higher-order logic, for SMT solvers [7] as well as saturation-based provers [16, 20].

Program verification is one of the domains that have benefited the most from the advances in automated theorem proving. In order to go further, we likely need to improve the interface between program verification tools and general-purpose reasoning engines. Intermediate verification languages [56, 90] can already be used for this purpose, including with saturation-based provers [30], but the lack of robustness remains an issue [31]. Furthermore, in the context of program verification, theorem provers are typically used as trusted black boxes. In order to maximize reliability, it would be preferable to perform some proof reconstruction, an approach already adopted when interfacing automated and interactive theorem provers [24].

Automated theorem proving has been a subject of interest since the early days of computer science. It is a fundamentally challenging task, but thanks to innovative techniques and increased hardware capabilities, automated tools can now tackle some non-trivial problems in various domains of application. In turn, these applications provide the research community with motivating examples, raise new problematics, and drive the development of improved tools. This synergy will hopefully continue and help push the boundaries of the field.