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

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

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 π is a program. Informally, such a triple can be understood as "if the program state satisfies the assertion P before the execution of π, and π 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 π and an assertion Q, and returning the weakest (i.e., most general) condition that is required to hold before the exectution of π 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 unin-

terpreted 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 θ is a *unifier* of s and t if $s\theta = t\theta$. Moreover, if every unifier of s and t is an instance of θ, then θ is said to be a *most general unifier* (mgu).

The resolution rule is as follows:

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

where θ 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

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

where θ 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 θ 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 θ 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 θ, $s \prec t$ 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 $s \overset{*}{\leftrightarrow}_R t$. By the properties of the simplification order, this is the case only if there exists a term u such that $s \overset{+}{\rightarrow}_R u$ and $t \overset{+}{\rightarrow}_R u$, 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 θ 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 $\boxed{L} \vee \mathcal{C}$ 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 σ is an mgu of L and L', θ 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, θ is an mgu of t and t', $s\theta \not\preceq t\theta$ and $u\theta \not\preceq v[t']\theta$

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

where σ is an mgu of L and L', θ is an mgu of u and u', $s\theta \not\preceq t\theta$ and $t\theta \not\preceq u\theta$

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 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 $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\} \subseteq N$ where $\mathcal{C}_1, \ldots, \mathcal{C}_n \models \mathcal{D}$ 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 θ 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 im-

plementation 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.

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. [14, 47, 91]. 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 [82], the symbol elimination method for generating invariants was introduced. The approach uses first-order theorem proving, in particular the Vampire prover [84]. 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 [54] 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 [14], 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 [59,63], where user-given invariant templates are used, we require no user guidance and infer first-order invariants with arbitrary quantifiers. Contrary to [45], 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 [89] 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 $A, B, C, \ldots$ to denote array variables and lower case letters $a, b, c, \ldots$ 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 that 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 loop are called *extended loop properties*.

[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.

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:

$$\forall i \left(0 \leqslant i < n \implies k^{(i+1)} \approx k^{(i)} + 1\right)$$

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 of 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 [83].

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* (respectively *decreasing*) if, for all possible computations of the loop, in any iteration i such that $0 \leqslant i < n$, it has the property $v^{(i+1)} \geqslant v^{(i)}$ (respectively $v^{(i+1)} \leqslant v^{(i)}$). A variable is said to be *strict* if, for all possible computations of the loop, in any iteration i such that $0 \leqslant i < n$, $v^{(i+1)} \neq v^{(i)}$, i.e., the value of the variable is modified at every iteration. Finally a variable is called *dense* if, for all possible computations of the loop, in any iteration i such that $0 \leqslant i < n, \implies |v^{(i+1)} - v^{(i)}| \leqslant 1$, i.e., its value is increased or decreased by at most one during any iteration.

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:

$$\forall i \left(v^{(i)} \approx v^{(0)} + i \right)$$

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

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

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

$$\forall ij \left(j \geqslant i \implies v^{(j)} \geqslant v^{(i)} \right)$$

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

$$\forall ij \left(j \geqslant i \implies v^{(i)} + j \geqslant v^{(j)} + i \right)$$

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

$$\begin{array}{c}\left(0 \leqslant i < n \wedge B^{(i)}[k^{(i)}] \geqslant C^{(i)}[k^{(i)}] \wedge v \approx B^{(i)}[k^{(i)}] \wedge p \approx k^{(i)}\right) \\ \vee \left(0 \leqslant i < n \wedge \neg B^{(i)}[k^{(i)}] \geqslant C^{(i)}[k^{(i)}] \wedge v \approx C^{(i)}[k^{(i)}] \wedge p \approx k^{(i)}\right)\end{array}$$

   which we denote $upd_A(i, p, v)$

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

$$\begin{array}{c}\left(0 \leqslant i < n \wedge B^{(i)}[k^{(i)}] \geqslant C^{(i)}[k^{(i)}] \wedge p \approx k^{(i)}\right) \\ \vee \left(0 \leqslant i < n \wedge \neg B^{(i)}[k^{(i)}] \geqslant C^{(i)}[k^{(i)}] \wedge p \approx k^{(i)}\right)\end{array}$$

   these are noted $upd_A(i, p)$

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

$$\forall ip \left(\neg upd_A(i, p) \implies B^{(n)}[p] \approx B^{(0)}[p]\right)$$

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

$$\forall ijpv \left(upd_A(i, p, v) \wedge (upd_A(j, p) \implies j \approx i) \implies B^{(n)}[p] \approx v\right)$$

Note that compared to [82], the second property has been modified as it used to read

$$\forall ijpv \left(upd_A(i, p, v) \wedge (upd_A(j, p) \implies j \leqslant i) \implies B^{(n)}[p] \approx v\right)$$

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.

$$\forall i\,(0 \leqslant i < n \wedge B^{(i)}[k^{(i)}] \geqslant C^{(i)}[k^{(i)}] \implies A^{(i+1)}[k^{(i)}] \approx B^{(i)}[k^{(i)}] \wedge k^{(i+1)} \approx k^{(i)} + 1)$$

$$\forall i\,(0 \leqslant i < n \wedge \neg B^{(i)}[k^{(i)}] \geqslant C^{(i)}[k^{(i)}] \implies A^{(i+1)}[k^{(i)}] \approx C^{(i)}[k^{(i)}] \wedge k^{(i+1)} \approx k^{(i)} + 1)$$

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.

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

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. [59, 63])

2.5 Loop Contract and Correctness

Previous works on symbol elimination [54, 68] 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:

$$\forall p \left(0 \leqslant p < l \implies A^{(0)}[p] \neq 0\right)$$

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 selected invariants 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 [54,68] 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 [14] 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 [14] 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 $[p]\varphi$, where p is a program and φ is another DL formula; $[p]\varphi$ 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 $\Gamma \vdash \mathcal{U}[p']\varphi$, 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:

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

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 [15] 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

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

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 [78].

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.

2.7 Experimental Results

We evaluated our tool on 20 challenging array benchmarks taken from academic papers [53, 54] 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

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	

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

8GiB 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 [54] of invariant generation in VAMPIRE. Finally, column N_{filter} reports the 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 [54] of invariant generation in VAMPIRE, our tool brings a significant performance increase: in all examples where the implementation of [54] succeeded to generate invariants, the number of invariants generated by our tool is much less than in [54]. For example, in the case of the program `copy0dd`, the number of invariants generated by our tool has decreased by a factor of 24 when compared to [54]. 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 [54] is missing, the approach of [54] failed to generate quantified loop invariants over arrays. We also note that invariants generated by [54] 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 in 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 bench-

marks for reasoning with quantifiers and theories. We intend to add these examples to the CASC theorem proving competition [127]. 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 [78]. 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 [8].

Chapter 3

Loop Analysis by Quantification over Iterations

Bernhard Gleiss, Laura Kovács and Simon Robillard

Abstract. We present a framework to analyze and verify programs containing loops by using a first-order language of so-called extended expressions. This language can express both functional and temporal properties of loops. We prove soundness and completeness of our framework and use our approach to automate the tasks of partial correctness verification, termination analysis and invariant generation. For doing so, we express the loop semantics as a set of first-order properties over extended expressions and use theorem provers and/or SMT solvers to reason about these properties. Our approach supports full first-order reasoning, including proving program properties with alternation of quantifiers. Our work is implemented in the tool QuIt and successfully evaluated on benchmarks coming from software verification.

Originally published in *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57, pages 381–399. EasyChair, 2018.

3.1 Introduction

One of the major challenges in automating the analysis and verification of programs comes with the presence of loops. Reasoning about such programs requires inferring and proving non-trivial properties that describe the loop behavior. Loop properties can be categorized into two classes: (i) functional properties that describe the loop behavior on program states and summarize, e.g., partial correctness properties of the loop, and (ii) temporal properties that focus on the iterative behavior of the loop, in particular its termination. To analyze loops and reason about their behavior, it is often useful to consider properties that blur the distinction between those two categories, such as safety and liveness properties. While there has been tremendous work on analyzing and verifying program loops, see e.g., [45, 58, 64, 72, 82, 87, 98], traditional means to reason about imperative programs are still poorly equipped to deal with both types of properties in a uniform manner. Complex functional properties are commonly expressed as program assertions featuring quantifiers. For example, the program `reverse` given in Figure 3.1 copies the elements of an array a to an array b, reversing their order. To specify this behavior, we need to use a universally quantified property, e.g., the post-condition:

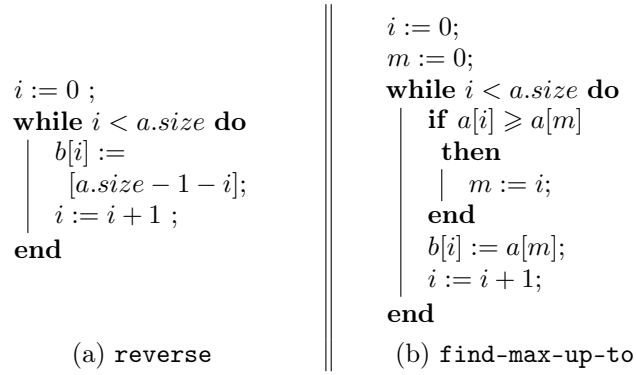
$$\forall j \, (0 \leq j < a.size \implies b[j] \approx a[a.size - 1 - j])$$

In some cases, we even need to use properties with quantifier alternations to give precise program specifications. The program `find-max-up-to` (also given in Figure 3.1) computes, for every position of an array a, the maximum value stored in a up to that position, and stores that value in b. One of the properties that describe its specification is:

$$\forall j \exists k \, (0 \leq j < a.size \implies b[j] \approx a[k])$$

These quantified properties can be verified using, e.g., Hoare logic. On the other hand, temporal properties are best expressed in some form of temporal logic, which usually restricts the use of quantifiers. Furthermore, most verification techniques require program annotations such as invariants and termination measures to be provided by the programmer, which limits their potential for automation.

In this paper, we present a framework to verify properties that combine functional and temporal aspects. The method is based on the first-order language of *extended expressions*, which provides a rich way to express



(a) `reverse` (b) `find-max-up-to`

Figure 3.1. Motivating examples over arrays with first-order properties.

both temporal and functional loop properties, including full quantification over loop iterations and program values. The semantics of a given loop can be encoded as a formula in this language, thus providing an axiomatization for the set of properties that hold for this loop (Section 3.3). Extended expressions are more expressive than program assertions typically used in program analysis and verification: program assertions reason about single program states, whereas extended expressions correspond to properties over sequences of states, i.e., program traces.

By expressing the loop semantics as a set of extended expressions, we reduce various applications of program analysis and verification to problems of first-order logic. In particular, we show that partial correctness and termination properties of loops can naturally be expressed as extended expressions. Similarly, the problem of invariant generation is a special instance of first-order reasoning about extended expressions. Namely, by using consequence finding and symbol elimination over extended expressions in first-order theorem proving we automatically infer first-order loop invariants (Section 3.4).

Analyzing loops in our framework is thus reduced to the problem of reasoning about extended expressions. This problem can be solved by automated reasoning engines, such as first-order theorem provers and SMT solvers. We describe how encoding the loop semantics into extended expressions can be optimized for these tools, in particular by limiting the need to perform inductive reasoning and exploiting reasoning with both theories and quantifiers (Section 3.5).

To illustrate the practical application of our framework, we implemented our work in the tool QuIt, which translates programs into ex-

tended expressions and uses automated reasoning engines to prove these properties (Section 3.6). We evaluate our work on verification problems taken from related works [23, 53] as well as from the array manipulating program category of the software verification benchmark suite SV-Comp [19]. For that, we used QuIt in conjunction with the first-order prover Vampire [84] and the SMT solvers CVC4 [9] and Z3 [50]. We show that, unlike existing methods, our approach supports reasoning about first-order loop properties with arbitrary use of quantifiers. We support quantification over loop iterations and program values, generating and proving loop properties in full first-order theories. By using our framework of extended expression, we are able to prove the safety assertions of each example of Figure 3.1.

Contributions. Extended expressions were first introduced for invariant generation in [82] and later used in [1] (Chapter 2 of this thesis) to prove partial correctness of programs. The work presented in this paper extends this line of work and brings the following contributions:

1. We formalize the semantics of the language of extended expressions;
2. We describe the axiomatization of the theory of extended expressions that hold for a given loop and prove its completeness (up to completeness of the background theory);
3. We show how extended expressions can be used to express and verify functional and temporal properties about programs, in particular partial correctness and termination;
4. We prove the soundness of using symbol elimination for invariant generation, based on extended expressions and consequence finding in first-order theorem proving;
5. We experiment with different background theories, in particular arrays and natural numbers, and compare different provers on these encodings.

3.2 Preliminaries

3.2.1 First-Order Logic

We consider standard many-sorted first-order logic modulo a background theory. We denote the theory with T and its signature with Σ_T. We assume that Σ_T includes sorts, equality $\approx$ over each sort and the interpreted functions and predicates of linear arithmetic $0, 1, +, <$. For example, in

one set of our experiments, T is the combined theory of linear integer arithmetic and arrays. We assume a given domain, and a mapping of the symbols in Σ_T to this domain. Any interpretation that respects that mapping is called a T-interpretation. Semantic consequence under T is denoted $A \models_T B$, i.e., any T-interpretation that satisfies A satisfies B.

We call a closed first-order formula a sentence. The valuation of sentences is defined in the usual way. In particular for the valuation of quantified sentences, we consider extensions of interpretations to variables: given an interpretation $\mathcal{I}$, we denote by $\mathcal{I}[x \leftarrow d]$ the interpretation that extends $\mathcal{I}$ by mapping the variable x to the domain value d. The sentence $\forall x.\, \varphi$ (resp. $\exists x.\, \varphi$) is true in $\mathcal{I}$ if φ is true in $\mathcal{I}[x \leftarrow d]$ for any (resp. some) value d of the appropriate domain.

3.2.2 Program Semantics

Throughout this paper, we assume a given loop $L = (C, \pi)$, where π is a program corresponding to the loop body and C is a Boolean expression representing the loop condition. The finite set of program locations[1] occurring in π and C is denoted by **Loc**.

We do not consider a particular programming language for π. Instead, we only rely on the denotational semantics of π, defined as a transition relation on program states. A state is a mapping from program locations to values of the appropriate sort. The semantics of π is described by the relation $\mathcal{S}_\pi$: for any pair of states (σ, σ'), the pair belongs to $\mathcal{S}_\pi$ if the execution of π in state σ can lead to state σ'. If π is a deterministic program, $\mathcal{S}_\pi$ is a function, but in general we do not assume this property. We require $\mathcal{S}_\pi$ to be total, but this does not limit our framework to loops with terminating bodies. If the loop body π is not guaranteed to terminate, we can use a special state $\sigma_\perp$ to represent non-terminating computations, with the requirement that $(\sigma_\perp, \sigma) \in \mathcal{S}_\pi$ if and only if $\sigma = \sigma_\perp$.

Definition 1 (L-sequence)**.** An L-sequence is an infinite sequence of states $\sigma_0, \sigma_1, \ldots$ such that for any natural number i, $(\sigma_i, \sigma_{i+1}) \in \mathcal{S}_\pi$.

The set of L-sequences corresponds to all possible executions of the loop L. A non-terminating loop execution corresponds to an L-sequence in which the condition C is true in all states σ_i. Terminating loop executions correspond to a prefix $\sigma_0, \ldots, \sigma_k$ of an L-sequence such that the condition C is true in the states σ_0 to σ_{k-1} and false in σ_k. The requirement

[1] We do not use the term "program variables", in order to avoid confusion with variables occurring in formulas.

on $\mathcal{S}_\pi$ to be total is necessary so that L-sequences only include infinite sequences (which is needed to provide a full interpretation of the language of extended expressions, see Section 3.3.1).

In practice, for most languages, $\mathcal{S}_\pi$ can easily be computed when π does not itself contain loops. In the presence of nested loops, it is possible to rely on an over-approximation of the actual semantics relation. If invariants are given for the nested loop, they can be used to improve the accuracy of the approximation. This approach is sound in the sense that, for every possible execution of the loop, there exists an L-sequence. However, in the rest of this paper, we assume that $\mathcal{S}_\pi$ is exact, and for concrete examples, we consider only non-nested loops.

3.2.3 Language of Assertions

The central idea of our work is the use of a language of *extended expressions* in which formulas can express properties of executions of the loop L, i.e., L-sequences. Extended expressions are more expressive than the kind of assertions traditionally used in program verification: program assertions express properties of a single program state, whereas extended expressions describe sequences of states (that is, traces). To establish a correspondence between extended expressions and assertions, we formally define a language for assertions, denoted by $\mathcal{L}_{asrt}$.

The signature of the language $\mathcal{L}_{asrt}$ is $\Sigma_T \cup \Sigma_{asrt}$, where Σ_{asrt} is the set that includes a constant symbol $\mu_l : \tau$ for every location l in **Loc**, where τ is the sort corresponding to the type of the location l.

Definition 2. Given a program state σ, the σ-interpretation is the unique interpretation $\mathcal{I}$ for $\mathcal{L}_{asrt}$ such that:

1. $\mathcal{I}$ is a T-interpretation;

2. $\mathcal{I}(\mu_l) = \sigma(l)$ for each program location $l \in$ **Loc**.

If a sentence F is true in the σ-interpretation, we write $\vDash_\sigma F$. Using Hoare triple notation, we write $\{P\}\,\pi\,\{Q\}$ to denote that, for any state σ, if $\vDash_\sigma P$, then for any state σ' such that $(\sigma, \sigma') \in \mathcal{S}_\pi$, $\vDash_{\sigma'} Q$.

Definition 3. $\mathcal{L}_{asrt}$ is said to be *expressive with respect to* π if for every formula F in $\mathcal{L}_{asrt}$, there exists a sentence $\mathrm{pre}_\pi(F)$ with the following property: for states σ, if F is true in the σ-interpretation (possibly extended to some variables), then, for all states σ', $\mathrm{pre}_\pi(F)$ is true in the (similarly extended) σ'-interpretation if and only if $(\sigma', \sigma) \in \mathcal{S}_\pi$.

Remark 1. The above definition indicates that $\mathcal{L}_{asrt}$ can be used to express the weakest pre-condition of π. It is not possible to prove this property without limiting π to specific programming constructs and specifying the theory T, but in practice many assertion languages are expressive. We note that, as in [101], we could define the expressivity of $\mathcal{L}_{asrt}$ by requiring the existence of a strongest post-condition.

3.3 Extended Expressions

We now define the first-order language of extended expressions. The semantics of a program can be expressed in this language and used further as an axiomatization for the set of valid program properties.

3.3.1 Syntax and Semantics

The language of extended expressions is denoted $\mathcal{L}_{extd}$. Its signature is $\Sigma_T \cup \Sigma_{extd}$, where Σ_{extd} includes, for every location l in **Loc**, a function symbol $\nu_l : \mathbb{N} \to \tau$, where τ is a sort corresponding to the type of the location l. We call these symbols *extended symbols* and use the notation $\nu_l^{(i)}$ to denote the application of an extended symbol ν_l to a term i. The semantics of $\mathcal{L}_{extd}$ is based on the possible executions of the loop L.

Definition 4. Given an infinite sequence of states $\bar{\sigma} = \sigma_0, \sigma_1, \ldots$ (that is not required to have the properties of an L-sequence), the *$\bar{\sigma}$-interpretation* is the unique interpretation $\mathcal{I}$ such that:

1. $\mathcal{I}$ is a T-interpretation;
2. $\mathcal{I}(\nu_l) = f_l$ for each location $l \in$ **Loc**, where f_l is a function such that for any $i \in \mathbb{N}$, $f_l(i) = \sigma_i(l)$.

If, for a sequence $\bar{\sigma}$, a sentence F in $\mathcal{L}_{extd}$ is true under the $\bar{\sigma}$-interpretation, we write $\models_{\bar{\sigma}} F$. If for all L-sequences $\bar{\sigma}$, we have $\models_{\bar{\sigma}} F$, then we say that F is *L-valid*, denoted $\models_L F$. Intuitively, L-valid sentences are the properties that are true for all executions of L.

3.3.2 Relativised Formulas

We now describe how to obtain a formula in $\mathcal{L}_{extd}$ corresponding to an assertion in $\mathcal{L}_{asrt}$.

Definition 5 (Relativised formula)**.** Given a (possibly open) formula F in $\mathcal{L}_{asrt}$ and a term t of sort $\mathbb{N}$, we define the *relativised formula*, denoted

$F^{(t)}$, as the formula obtained by replacing every occurrence of a symbol $\mu_l \in \mathcal{L}_{asrt}$ from F by the term $\nu_l^{(t)}$.

For example given a term i of sort $\mathbb{N}$ and a formula $F = \exists x.\, \mu_l \approx 2 \times x$, the relativised formula $F^{(i)}$ is $\exists x.\, \nu_l^{(i)} \approx 2 \times x$. The relativised formula is in $\mathcal{L}_{extd}$, and the set of variables occurring free in it is exactly the set of variables occurring free in F or in t.

Lemma 1 (Semantics of relativised formula)**.** *Let F be a formula in $\mathcal{L}_{asrt}$, $\bar{\sigma}$ an infinite sequence of states, and m a natural number. Let $\mathcal{I}$ denote the $\bar{\sigma}$-interpretation. The value of $F^{(t)}$ under $\mathcal{I}[t \leftarrow m]$ is identical (for any interpretation of the free variables) to the value of F under the σ_m-interpretation.*

Proof. By induction on the syntactic structure of F. For the base case, it is easy to check that any term in F has the same interpretation as the corresponding term in the relativised formula. □

3.3.3 Axiomatization of Valid Loop Properties

Let us consider the theory of L-valid sentences, i.e., the set of sentences $F \in \mathcal{L}_{extd}$ such that $\vDash_L F$. In order to axiomatize this theory, we need to encode in $\mathcal{L}_{extd}$ the semantics of π, and thus describe L-sequences. Provided that $\mathcal{L}_{asrt}$ is expressive with respect to π, the semantics of the loop (ignoring its condition) can be described by the following axiom:

$$\forall \bar{x_l}\, i\, \left(S^{(i+1)} \Rightarrow \mathrm{pre}_\pi(S)^{(i)}\right) \qquad (\mathsf{Step}_L)$$

where S is the formula $\bigwedge_l \mu_l \approx x_l$, and $\bar{x_l}$ is a set of distinct variables (one for each location $l \in \mathbf{Loc}$).

For example, let us consider the following loop:

```
while a ≠ b do
|  if a > b then
|  |  a := a − b;
|  else
|  |  b := b − a;
|  end
end
```

The set of locations read or modified by the loop is $\{a, b\}$, therefore the formula S is $x \approx \mu_a \wedge y \approx \mu_b$. Using a typical predicate transformer calculus, we can compute the weakest pre-condition $\mathrm{pre}_\pi(S) = (\mu_a >$

$\mu_b \implies x \approx \mu_a - \mu_b \wedge y \approx \mu_b) \wedge (\neg\mu_a > \mu_b \implies x \approx \mu_a \wedge y \approx \mu_b - \mu_a)$. Therefore the axiom Step_L for this particular loop is

$$\begin{aligned}\forall xyi \Big(& x \approx \nu_a^{(i+1)} \wedge y \approx \nu_b^{(i+1)} \implies \\ & \big(\nu_a^{(i)} > \nu_b^{(i)} \implies x \approx \nu_a^{(i)} - \nu_b^{(i)} \wedge y \approx \nu_b^{(i)}\big) \\ & \wedge \big(\neg\nu_a^{(i)} > \nu_b^{(i)} \implies x \approx \nu_a^{(i)} \wedge y \approx \nu_b^{(i)} - \nu_a^{(i)}\big)\Big).\end{aligned}$$

We see that Step_L can equivalently be expressed without using variables for locations, in this case:

$$\begin{aligned}\forall i \Big(& \nu_a^{(i)} > \nu_b^{(i)} \implies \nu_a^{(i+1)} \approx \nu_a^{(i)} - \nu_b^{(i)} \wedge \nu_b^{(i+1)} \approx \nu_b^{(i)}\Big) \\ \wedge \Big(& \neg\nu_a^{(i)} > \nu_b^{(i)} \implies \nu_a^{(i+1)} \approx \nu_a^{(i)} \wedge \nu_b^{(i+1)} \approx \nu_b^{(i)} - \nu_a^{(i)}\Big).\end{aligned}$$

This simplification is desirable in practice as it limits the number of quantifiers. We will however consider the syntactic form presented above in order to keep the presentation simple.

Lemma 2 (Soundness)**.** $\vDash_L \mathsf{Step}_L$.

Proof. Let $\bar{\sigma} = \sigma_0, \sigma_1, \ldots$ be an L-sequence and $\mathcal{I}$ the $\bar{\sigma}$-interpretation, we show that Step_L is true in $\mathcal{I}$.

Let m be a natural number and let $\bar{d}$ be values of the domain corresponding to variables $\bar{x}_l$. Let $\mathcal{I}'$ be the interpretation $\mathcal{I}[i \leftarrow m+1, \bar{x}_l \leftarrow \bar{d}]$. If $S^{(i+1)}$ is false in $\mathcal{I}'$, the formula $S^{(i+1)} \implies \mathrm{pre}_\pi(S)^{(i)}$ is true in $\mathcal{I}'$. Otherwise, by Lemma 1, it must be the case that S is true in the σ_{m+1}-interpretation (extended with the interpretation $\bar{d}$ of the free variables $\bar{x}_l$). Since $(\sigma_m, \sigma_{m+1}) \in \mathcal{S}_\pi$, we have that $\vDash_{\sigma_m} \mathrm{pre}_\pi(S)$, therefore $\mathrm{pre}_\pi(S)^{(i)}$ is true in $\mathcal{I}'$. The formula Step_L is true in $\mathcal{I}$ for any interpretation of its quantified variables. □

Lemma 3 (Completeness)**.** *Let F be a sentence in $\mathcal{L}_{extd}$ such that $\vDash_L F$, then $\mathsf{Step}_L \vDash_T F$.*

Proof. Let $\mathcal{I}$ be a T-interpretation that satisfies Step_L. We define $\bar{\sigma} = \sigma_0, \sigma_1, \ldots$ to be the infinite sequence of states such that for any number i and any program location $l \in \mathbf{Loc}$, $\sigma_i(l) = f_l(i)$, where f_l is $\mathcal{I}(l)$. Clearly $\mathcal{I}$ is the $\bar{\sigma}$-interpretation. Let us show that $\bar{\sigma}$ is a L-sequence.

Let m be a number, and let $\bar{d}$ denote the values of all the program locations $l \in \mathbf{Loc}$ in state σ_{m+1}. Let $\mathcal{I}'$ be the interpretation $\mathcal{I}[i \leftarrow m, \bar{x}_l \leftarrow \bar{d}]$. By choice of $\bar{d}$, $\vDash_{m+1} S$ with the σ_{m+1}-interpretation extended with $[\bar{x}_l \leftarrow \bar{d}]$. Thus by Lemma 1, $S^{(i+1)}$ is true in $\mathcal{I}'$. Since $\mathcal{I}'$ satisfies Step_L, $\mathrm{pre}_\pi(S)^{(i)}$ is in particular true in $\mathcal{I}'$. By Lemma 1, we have $\vDash_{\sigma_m} \mathrm{pre}_\pi(S)$

(with the same extension of the σ_m-interpretation as before). By definition of $\text{pre}_\pi(S)$, this implies $(\sigma_m, \sigma_{m+1}) \in \mathcal{S}_\pi$, therefore $\bar{\sigma}$ satisfies the definition of an L-sequence.

Since $\mathcal{I}$ is a $\bar{\sigma}$-interpretation derived from an L-sequence, it must satisfy F. □

Theorem 1 (L-validity)**.** *For any sentence F in $\mathcal{L}_{extd}$, $\models_L F$ if and only if $\mathsf{Step}_L \models_T F$.*

Proof. One direction of the equivalence is given by Lemma 3. For the other direction, let $\bar{\sigma}$ be an L-sequence and $\mathcal{I}$ the $\bar{\sigma}$-interpretation. By Lemma 2, $\mathcal{I}$ is a model of Step_L. In addition, $\mathcal{I}$ is a T-interpretation, therefore by the assumption it is also a model of F. □

Remark 2. The theory of L-valid sentences is a superset of the theory T. Therefore in order for that theory to be complete, T must be complete as well. Theorem 1 shows that this is indeed a sufficient condition. In that sense, it can be seen as a result of relative completeness.

3.4 Applications of Extended Expressions

We now detail how applications of program analysis and verification can be expressed as problems over extended expressions. In particular, we show that proving partial correctness or termination of programs can be reduced to proving properties of extended expressions. Further, the task of invariant generation can be solved by using consequence finding and symbol elimination in first-order theorem proving over extended expressions.

3.4.1 Verifying Partial Loop Correctness

The partial correctness of the loop L with respect to a pre-condition P and a post-condition Q (both sentences in $\mathcal{L}_{asrt}$) and the loop condition C can be expressed as a sentence in $\mathcal{L}_{extd}$:

$$\forall n \left(P^{(0)} \wedge \forall m \left(m < n \implies C^{(m)} \right) \wedge \neg C^{(n)} \right) \Rightarrow Q^{(n)} \qquad (\mathsf{Correct})$$

Lemma 4 (Partial correctness of the loop)**.** *If $\models_L \mathsf{Correct}$, then for any state σ satisfying P, any terminating execution of the loop L starting in σ leads to a state that satisfies Q.*

Proof. Let $\sigma_0, \ldots, \sigma_k$ be a finite sequence of states corresponding to a terminating execution of L, such that $\sigma_0 = \sigma$. For any two consecutive states σ_i and σ_{i+1} in the sequence, $(\sigma_i, \sigma_{i+1}) \in \mathcal{S}_\pi$. Let $\bar{\sigma}$ be an L-sequence such that $\sigma_0, \ldots, \sigma_k$ is a prefix of $\bar{\sigma}$, and $\mathcal{I}$ the $\bar{\sigma}$-interpretation. By the assumption, $\mathcal{I}$ is a model of Correct. Let $\mathcal{I}'$ be the interpretation $\mathcal{I}[n \leftarrow k]$.

By $\vDash_{\sigma_0} P$ and Lemma 1, we have that $P^{(0)}$ is true in $\mathcal{I}'$. By property of the finite execution of the loop $\sigma_0, \ldots, \sigma_k$, the loop condition C is true in every state of the sequence except the last. Therefore, $\vDash_{\sigma_k} \neg C$ and $\vDash_{\sigma_i} C$ for any number i such that $i < k$. Using Lemma 1 we can check that $\forall m \left(m < n \implies C^{(m)}\right)$ and $\neg C^{(n)}$ are true in $\mathcal{I}'$. Consequently $Q^{(n)}$ is true in $\mathcal{I}'$, and by Lemma 1, $\vDash_{\sigma_k} Q$, that is, the final state of the execution satisfies Q. □

3.4.2 Termination, Safety, Liveness

Similarly, proving the termination of L under a pre-condition P can be reduced to checking the L-validity of the sentence

$$P^{(0)} \Rightarrow \exists n. \neg C^{(n)} \qquad \text{(Termin)}$$

Lemma 5 (Termination of the loop)**.** *If* $\vDash_L$ Termin*, then for any state* σ *satisfying* P*, any execution of the loop* L *starting in* σ *terminates.*

Proof. By contradiction let $\bar{\sigma}$ be an L-sequence corresponding to a non-terminating execution of L, i.e., all its states σ_i verify $\vDash_{\sigma_i} C$, and such that $\sigma_0 = \sigma$. Let $\mathcal{I}$ be the $\bar{\sigma}$-interpretation. Since $\vDash_{\sigma_0} P$, by Lemma 1, $P^{(0)}$ is true in $\mathcal{I}$, and since $\mathcal{I}$ is a model of Termin, it is in particular a model of $\exists n. \neg C^{(n)}$. By Lemma 1 there exists a number k such that $\vDash_{\sigma_k} \neg C$, which contradicts our hypothesis. □

Finally, it is possible to express safety and liveness properties as extended expressions. Given an assertion A, the safety property with respect to A is

$$\forall n. \neg A^{(n)} \qquad \text{(Safe)}$$

and the liveness property

$$\forall m \exists n \left(m < n \wedge A^{(n)}\right) \qquad \text{(Live)}$$

It is easy to check that these formulas correspond to the expected semantic properties of the loop, in a fashion similar to the proofs of lemmas 4 and 5.

3.4.3 Invariant Generation Via Symbol Elimination

Our framework provides a way to verify the correctness of an iterative program without the explicit use of invariants. Nevertheless, it can be useful to obtain loop invariants for the program, e.g., to gain some insight in the behavior of the loop or to verify it using another tool, which typically requires invariants in the form of user annotations. Our framework can be used to derive invariants as logical consequences of the extended expressions.

In program verification, the notion of invariant typically refers to *inductive* invariants, i.e., assertions F such that $\{C \wedge F\}\, \pi\, \{F\}$. In practice, interesting invariants are those that hold at the start of the loop. Therefore in the presence of a pre-condition P, we wish to find invariants F such that $P \vDash_T F$. Given these two requirements, we use the following definition:

Definition 6 (P-invariant)**.** Given a sentence P in $\mathcal{L}_{asrt}$, a sentence F in $\mathcal{L}_{asrt}$ is a P-invariant if for any prefix $\sigma_0, \ldots, \sigma_k$ of an L-sequence such that $\vDash_{\sigma_0} P$ and $\vDash_{\sigma_i} C$ for any number $i < k$, then any state σ of that prefix verifies $\vDash_{\sigma} F$.

Intuitively, if an execution of the loop L starts in a state satisfying P, every state of this execution satisfies F, up to and including the final state if the loop terminates. It is easy to show that an inductive invariant I is a P-invariant for any sentence P such that $P \vDash_T I$. Conversely however, not all P-invariants F are inductive: there may exist a pair of states (σ, σ') that violates $\{C \wedge F\}\, \pi\, \{F\}$, but only if σ is not reachable in any L-sequence that starts with a state satisfying P.

Lemma 6. *Let P and F be sentences in $\mathcal{L}_{asrt}$. Let* Inv *denote the sentence*

$$\forall n \left(P^{(0)} \wedge \forall m \left(m < n \implies C^{(m)} \right) \implies F^{(n)} \right) \qquad (\mathsf{Inv})$$

If $\vDash_L \mathsf{Inv}$, then F is a P-invariant.

Proof. Let $\bar{\sigma}$ be an L-sequence such that $\vDash_{\sigma_0} P$ and k be a number such that $\vDash_{\sigma_i} C$, for any number $i < k$. Let $\mathcal{I}$ be the $\bar{\sigma}$ interpretation and j be a number such that $j \leqslant k$. Since $\mathcal{I}[n \leftarrow j]$ is a model of Inv, it is in particular a model of $F^{(n)}$. Therefore, by Lemma 1, $\vDash_{\sigma_j} F$. □

Lemma 6 provides a way to check that a given sentence is a P-invariant. More interestingly, it also gives the basis of a procedure to generate P-invariants. This procedure was first introduced in [82] but never proven

sound until now. We outline it here again in order to provide a formal argument for its soundness based on Lemma 6.

We have so far only considered reasoning about extended expressions. In order to generate P-invariants, we must now turn our attention to assertions, i.e., formulas in $\mathcal{L}_{asrt}$. Firstly, let us observe that for any formula F in $\mathcal{L}_{asrt}$ and any term t of sort $\mathbb{N}$, the relativised formula $F^{(t)}$ is equivalent to $\bigwedge_l \nu_l^{(t)} \approx \mu_l \implies F$ (regardless of the interpretation of the symbols μ_l). Thus Inv is equivalent to

$$\forall n \left(P^{(0)} \wedge \forall m \left(m < n \implies C^{(m)} \right) \wedge \bigwedge_{l \in \mathbf{Loc}} \nu_l^{(n)} \approx \mu_l \implies F \right)$$

Secondly, if F is a closed formula (in particular, not featuring any occurrence of the variable n), Inv can be rewritten so that the quantifier is moved to the antecedent

$$\exists n \left(P^{(0)} \wedge \forall m \left(m < n \implies C^{(m)} \right) \wedge \bigwedge_{l \in \mathbf{Loc}} \nu_l^{(n)} \approx \mu_l \right) \implies F$$

Let us denote by InvGen the sentence

$$\exists n \left(P^{(0)} \wedge \forall m \left(m < n \Rightarrow C^{(m)} \right) \wedge \bigwedge_{l \in \mathbf{Loc}} \nu_l^{(n)} \approx \mu_l \right) \qquad (\mathsf{InvGen})$$

By Theorem **??**, the condition in Lemma 6 is equivalent to $\mathsf{Step}_L \cup \mathsf{InvGen} \vDash_T F$. In order to generate P-invariants, it is therefore enough to find consequences (under theory T) of $\mathsf{Step}_L \cup \mathsf{InvGen}$. Theorem provers based on saturation provide a natural way to perform consequence finding. The sentences Step_L and InvGen are clausified (in the process, a constant symbol n corresponding to the existentially quantified variable is introduced by Skolemization) and the resulting set of clauses is saturated: new clauses are repeatedly produced by (sound) inferences between clauses of the set, and the conclusion added to the set. Typically this process is used to derive the empty clause as part of a proof by contradiction. However in this case the initial set of clauses has a model, therefore the process will only stop if no new non-redundant clauses can be derived.

Any clause generated during saturation by a sound calculus is a logical consequence of the original set of clauses. To be a P-invariant, it must only contain symbols from $\mathcal{L}_{asrt}$. Unguided consequence finding is unlikely to yield such formulas consistently, so we take advantage of the reduction ordering of the superposition calculus to orient the search. Superposition

is a family of calculii parametrized by a reduction ordering on terms. This ordering is used to restrict the number of possible inferences while preserving the refutational completeness of the calculus. For example the superposition rule

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

which uses an equality literal $t \approx s$ to rewrite a term t' (unifiable with t under a most general unifier θ), is applied only if $s\theta \not\succeq t\theta$ according to the term ordering. By choosing the right term ordering, we can ensure that symbol-eliminating inferences are favored by the prover. In particular, terms featuring extended symbols should be larger than others. One possibility to accomplish this is to choose a Knuth-Bendix term ordering in which extended symbols are given a large weight and a large precedence.

The set of clauses resulting from the clausification of InvGen contains, for each program location l, the unit clause $\nu_l^{(n)} = \mu_l$, therefore any clause featuring a term unifiable with $\nu_l^{(n)}$ may be used in a symbol eliminating inference. Every time a new clause is generated that does not feature an extended symbol, it may be reported as an invariant. Optionally, it may be preferable to report such clauses only if they feature symbols from Σ_{asrt}. Clauses that do not include such symbols are theory tautologies: they are technically invariant but not very useful for program verification.

3.5 Automated Reasoning with Extended Expressions

As shown in Section 3.4, analyzing loops in our framework is reduced to the problem of reasoning about extended expressions, which can be solved by automated reasoning engines, such as first-order theorem provers and SMT solvers. In this section we describe how encodings of the loop semantics into extended expressions can be optimized for these tools.

3.5.1 Avoiding Induction

Theorem **??** provides a powerful way to reduce loop analysis and verification tasks to the problem of proving that a certain extended expression is entailed by Step_L. Unsurprisingly, induction often plays a key role in these proofs. Step_L essentially describes the semantics of one arbitrary iteration of the loop, whereas loop analysis is often concerned with proving properties of arbitrary iterations, for example for all iterations or for

a symbolic iteration in which the loop condition is negated for the first time.

Extending automated theorem provers with inductive reasoning is a very challenging problem, with some preliminary yet still limited results in [46, 74, 136]. In order to avoid inductive reasoning and thus make our framework more friendly to automated provers, we use a number of so-called *trace lemmas* in addition to Step_L. These lemmas correspond to valid properties of the loop.

Definition 7. A *trace lemma* for a given loop is a sentence F such that $\mathsf{Step}_L \models_T F$.

For any set of trace lemmas Lem_L, it is obvious that $\mathsf{Step}_L \cup \mathsf{Lem}_L$ is T-equivalent to Step_L. Hence, Step_L can be replaced by $\mathsf{Step}_L \cup \mathsf{Lem}_L$ in loop analysis, in particular in the applications described in Section 3.4. While this substitution makes no difference on a theoretical level, a careful choice of Lem_L often leads to a dramatic improvement of the performance of automated reasoning tools. The choice of trace lemmas to include in Lem_L depends on the theory T, the class of programs targeted, and the type of properties that one wishes to analyze. Consider for example the extended expression

$$\forall ij \left(i < j \implies \nu_l^{(i)} \leqslant \nu_l^{(j)} \right)$$

Many loops include so-called monotonically increasing locations l for which this property would hold. In addition, the property is likely to be useful in many verification tasks. On the other hand, proving that it is entailed by Step_L requires reasoning by induction and hence automated theorem provers are unlikely to discover it. For these reasons, the property is a good candidate trace lemma: every time we perform a verification task, we will first check that the property holds for each location in the given loop, and if so, add it to the set of trace lemmas that will later be used in conjunction with Step_L.

Proving that a given sentence F is a trace lemma of L is in general as difficult as proving the L-validity of other extended expressions. Therefore, we rely on sound but incomplete methods to derive trace lemmas of L. Currently in our work, the verification of trace lemmas is accomplished by lightweight static analysis techniques. For example, the property described above is added to the set of trace lemmas when all assignments to the location l are increments by a positive constant. A more general method to check whether a sentence is a trace lemma is to reduce it to a minimal condition on one iteration of the loop body π. The property

given in example holds if and only if $\{\mu_l \approx x\}\, \pi\, \{\mu_l \geqslant x\}$ for some variable x. Simple properties such as these can often be verified automatically. This hints at a generic way to describe trace lemmas and use them: (i) an extended expression F (often a property universally quantified over iterations) is first proven to be equivalent to an easily verifiable condition on the loop body in the form of a Hoare triple (ii) any time the condition is verified, F is included in Lem_L.

A complete description of the set of trace lemmas used in our work is given in an earlier publication [80] (reproduced as part of Chapter 2 in this thesis) . In general, our trace lemmas fall under two categories: properties of monotonically increasing or decreasing locations and properties of array updates.

3.5.2 Encoding of Natural Numbers

The theory T must include a sort of natural numbers, as well as the predicates and functions of linear arithmetic that are needed to formulate the axiom Step_L. However, not all automated tools for reasoning in first-order logic support the theory of natural numbers. Therefore we experimented with two encodings of natural numbers. Our first encoding uses integers, where every axiom or goal is modified to ensure that only non-negative integers are considered. The second encoding is based on a term algebra generated by two constructors, a constant *zero* and a unary function *succ*. The predicate $<$ is recursively axiomatized.

Both encodings will be handled differently by provers, and yield different proofs. Linear arithmetic is a staple of reasoning modulo theory, and all SMT solvers include a solver for it. For first-order theorem provers based on saturation, reasoning about linear arithmetic is traditionally accomplished by including a (partial) axiomatization in the set of clauses to saturate. Recently, these provers have taken advantage of SMT solvers to perform theory reasoning on ground clauses [110] as well as non-ground clauses [112]. Regarding term algebras, the SMT solvers Z3 and CVC4 both include theory solvers for the ground theory. The automated theorem prover Vampire allows reasoning with term algebras, based on a conservative extension of the theory complemented by dedicated inference rules [26, 81] (Chapters 4 and 6 in this thesis).

In addition, the two encodings allow the expression of different properties. The integer-based encoding can more easily express relations between integer-valued program locations and iteration numbers. For example, if a location l of the loop satisfies $\{\mu_l \approx x\}\, \pi\, \{\mu_l \approx x + c\}$ for some variable

x and some constant c then the following trace lemma can be used:

$$\forall i \left(0 \leqslant i \implies \nu_l^{(i)} \approx \nu_l^{(0)} + i \times c\right)$$

If natural numbers are encoded as algebraic terms, an equivalent trace lemma cannot be expressed without extending the theory to include a function mapping natural numbers to integers. Instead we can use a weaker property, for example (if $c \neq 0$):

$$\forall ij \left(\nu_l^{(i)} \approx \nu_l^{(j)} \implies i \approx j\right)$$

3.5.3 Representation of Arrays

Unlike scalar program locations, the logical encoding of arrays is not straightforward. In earlier approaches [82], we used the following functional representation of arrays. In $\mathcal{L}_{asrt}$, arrays are represented as functions from the sort of indices to the sort of values. In $\mathcal{L}_{extd}$, arrays are represented by binary functions: the first argument of the function takes an iteration and the second an index, so that $a(i, p)$ denotes the value stored at position p at the ith iteration.

Later experiments suggested that using a dedicated theory of arrays might make it easier to prove properties of programs [32]. In this setting, we have one sort τ for each type of array, equipped with operations store and select that represent writing to and reading from an array, respectively. Array locations are then treated like other locations: in $\mathcal{L}_{asrt}$ they are represented by constants of type τ, and in $\mathcal{L}_{extd}$ they are represented by a function from $\mathbb{N}$ to τ.

3.6 Experiments

3.6.1 Implementation

We implemented our work in the tool QuIt[2]. QuIt consists of 12000 lines of C++ code. Inputs to QuIt are programs written in a guarded command language. In addition to the program itself, the input also includes pre- and post-conditions (P and Q) in the form of first-order logic assertions with unbounded quantification. QuIt converts this program to a first-order problem, according to one of its three modes of operation:

[2] `http://www.cse.chalmers.se/~simrob/downloads/quit.tar.gz`

1. *Verification mode*, to prove partial program correctness (Section 3.4.1). In this setting, the first-order problem produced by QuIt contains the hypothesis Step_L, the trace lemmas and theory axioms, and the goal Correct to be proven.

2. *Termination mode*, to prove program termination (Section 3.4.2). In this case, QuIt generates a similar problem as in its verification mode, but with the goal Termin.

3. *Invariant generation mode*, to generate invariants by symbol elimination (Section 3.4.3). The problem produced contains the hypothesis InvGen, trace lemmas and theory axioms. Extended symbols are marked for symbol elimination. No goal to be proven is provided, since the aim is to produce consequences of properties of extended expressions, rather than to find a proof.

QuIt is partially based on code from [80] that was previously integrated in the first-order theorem prover Vampire. We made QuIt a standalone tool that can interact with various provers, including both SMT solvers and first-order theorem provers. For that, QuIt outputs problems in the TPTP syntax of first-order theorem provers [126], in particular in the TFF input representation of many-sorted first-order logic. Further, QuIt also generates its output in the SMT-LIB input syntax of SMT solvers [11]. As such, problems generated by QuIt in the verification and termination modes can be passed to any tool that supports the TPTP and/or SMT-LIB syntax. Currently, only Vampire is able to perform symbol elimination, which is necessary to handle the invariant generation problems generated by QuIt.

3.6.2 Experimental Results

To evaluate our implementation, we collected benchmarks from the work of [53] and the SV-Comp repository of software verification benchmarks [19]. We converted these examples manually into our input-format. Since our approach establishes program correctness rather than searching for counterexamples, we omitted benchmarks where assertions are violated. We also omitted examples not supported by our framework due to language features such as multiple loops or memory management. Lastly, we removed duplicate problems differing from other examples only in the names of program locations. As a result, our benchmarks include 55 test cases, all featuring arrays.

The assertion language used in these benchmarks does not allow quantification and relies on loops to encode some quantified properties. For

Benchmark	Vampire				CVC4				Z3			
	A+T	A+I	F+T	F+I	A+T	A+I	F+T	F+I	A+T	A+I	F+T	F+I
absolute-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
absolute-prop2	✓	✓	t	✓	t	t	t	t	t	t	t	t
atleast-one-iteration	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
both-or-none	✓	✓	✓	✓	t	t	t	t	t	t	t	t
check-equal-set-flag	t	t	t	t	t	t	t	t	t	t	t	t
copy	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-nonzero-prop1	t	t	t	t	t	t	t	t	t	t	t	t
copy-nonzero-prop2	t	t	t	t	t	t	t	t	t	t	t	t
copy-odd	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-partial	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-positive	t	t	t	t	t	t	t	t	t	t	t	t
copy-two-indices	t	✓	t	✓	t	t	t	t	t	t	t	t
find1-prop1	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
find1-prop2	✓	t	✓	t	t	t	t	t	t	t	t	t
find1-prop3	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find2-prop1	✓	✓	✓	✓	✓	t	✓	t	✓	✓	✓	✓
find2-prop2	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
find2-prop3	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
find-max	t	t	t	t	t	t	t	t	t	t	t	t
find-max-up-to-prop1	t	t	t	t	t	t	t	t	t	t	t	t
find-max-up-to-prop2	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find-max-from-second	t	t	t	t	t	t	t	t	t	t	t	t
find-min	t	t	t	t	t	t	t	t	t	t	t	t
find-min-up-to	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find-sentinel	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
find-two-max-prop1	t	t	t	t	t	t	t	t	t	t	t	t
find-two-max-prop2	t	t	t	t	t	t	t	t	t	t	t	✓
in-place-max	t	t	t	t	t	t	t	t	t	t	t	t
increment-by-one-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
increment-by-one-prop2	✓	✓	t	✓	t	t	t	t	t	t	t	t
indexn-is-arraylength	✓	✓	✓	✓	t	t	t	t	✓	✓	✓	✓
init	✓	✓	✓	✓	t	t	t	t	t	t	t	t
init-conditionally-prop1	t	t	t	t	t	t	t	t	t	t	t	t
init-conditionally-prop2	t	t	t	t	t	t	t	t	t	t	t	t
init-even	t	✓	t	✓	t	t	t	t	t	✓	t	t
init-non-constant	✓	✓	✓	✓	t	t	t	t	t	t	t	t
init-partial	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
init-previous-plus-one	t	t	t	t	t	t	t	t	t	t	t	✓
max-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
max-prop2	✓	✓	✓	✓	t	t	t	t	t	t	t	t
merge-interleave-prop1	t	✓	t	✓	t	t	t	t	t	t	t	t
merge-interleave-prop2	t	t	t	t	t	t	t	t	t	t	t	t
palindrome	t	t	t	t	t	t	t	t	t	t	t	t
partition	t	t	t	t	t	t	t	t	t	t	t	t
partition-init	t	t	t	t	t	t	t	t	t	t	t	t
push-back-prop1	t	✓	t	✓	t	t	t	t	t	✓	t	✓
push-back-prop2	t	✓	t	✓	t	t	t	t	t	t	t	t
reverse	✓	✓	✓	✓	t	t	t	t	t	t	t	t
set-to-one	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
str-cpy	✓	✓	✓	✓	t	t	t	t	t	t	t	t
str-len	✓	✓	✓	✓	t	t	t	t	t	t	t	t
swap-prop1	t	t	t	t	t	t	t	t	t	t	t	t
swap-prop2	t	t	t	t	t	t	t	t	t	t	t	t
vector-addition	✓	✓	✓	✓	t	t	t	t	t	t	t	t
vector-subtraction	✓	✓	✓	✓	t	t	t	t	t	t	t	t
Total	35				1				13			
Unique	24				0				2			

Table 3.1. Results of theorems provers on QuIt-generated partial correctness problems. Success is denoted by ✓ and timeout by t.

example, the program fragment:

$$\texttt{for(int i} = 0;\ \texttt{i} < \texttt{a.length};\ ++\texttt{i)}\ \{\texttt{assert(F(a[i]))}\}$$

is used to encode the universal property

$$\forall i\ (0 \leqslant i < a.length \implies F(a[i]))$$

Using program code to encode first-order properties is however restrictive, as only universally quantified properties over finite domains can be naturally encoded. Since our framework supports unbounded quantification in first-order properties, we used quantified assertions rather than loops to describe the properties to verify.

To overcome the challenges of first-order reasoning with theories, quantifiers and induction (see Section 3.5), in QUIT we used four different encodings of the first-order background theory: (i) theory of arrays and term algebras (denoted by `A+T`), (ii) theory of arrays and linear integer arithmetic (denoted by `A+I`), (iii) first-order theory of term algebras with uninterpreted functions modeling arrays (denoted by `F+T`), and (iv) first-order theory linear of integer arithmetic with uninterpreted functions modeling arrays (denoted by `F+I`). That is, natural numbers were encoded either by term algebra axioms or by a sound, but incomplete axiomatization of linear integer arithmetic. By applying these four encodings to our 55 examples, QUIT produced all together 220 examples for each of its modes. To prove these examples, we interfaced QUIT with three solvers, namely VAMPIRE, CVC4 and Z3. We report on our experiments, which were performed on an Intel Core i5 machine running at 2.9Ghz.

Proving partial correctness. For each choice of background theory encoding, we used QUIT in the verification mode to construct a first-order formalization of partial correctness (which took less than a second for any benchmark) and then ran each of the provers on the resulting problem with a timeout of 60 seconds. Our results are summarized in Table 3.1. The first column of this table names the benchmark name as in SV-Comp. For each solver, we then report on its result on the problem generated by QUIT using one encoding of the background theory: ✓ denotes success (the prover proved the QUIT problem), while t denotes failure due to time-out. Table 3.1 also reports on the total number of problems solved by each prover, as well as on the number of problems that were uniquely solved by only one prover.

Table 3.1 shows that VAMPIRE outperforms both SMT solvers on problems created by QUIT, regardless of the options chosen. This likely

stems from the use of many quantified properties among the trace lemmas. Concerning the background theory and the choice of encoding, it is difficult to identify a winning encoding. Each configuration was able to uniquely solve some problems. This suggests that a portfolio approach might be advisable: the different possible encodings of the problem can all be generated, and proof attempts may be conducted by different provers, possibly in parallel. In order to test the usefulness of trace lemmas, we also ran the partial correctness experiment without including any such lemmas, and instead only including the hypothesis Step_L. Only 3 programs could be proven correct in this setting, demonstrating the crucial role of trace lemmas.

Proving termination. We used QuIt in the termination mode to construct a first-order encoding of program termination (which again took less than a second for any benchmark). We ran each prover on the resulting problem with a timeout of 60 seconds. Some of the 55 examples differ only by their post-condition, which is irrelevant for termination, so our termination benchmarks include 43 different programs. Vampire was able to prove termination of 42 of these programs. The example for which termination could not be proven is `find1`, in which the loop condition depends on the value of a location set in the loop body. Z3 managed to prove termination of 23 programs, whereas CVC4 did not solve any of the termination problems. While our benchmarks do not yield challenging termination problems, they correspond to common programming patterns. We believe that the ability to check their termination automatically, in the same framework used to verify correctness, is of great practical use.

Generating invariants. To generate invariants, we interfaced QuIt only with Vampire since it is currently the only solver able to perform symbol elimination over first-order properties. Since the problem created in invariant generation mode is satisfiable, saturation may never terminate, generating an infinite set of logical consequences. Therefore we ran Vampire with a fixed time limit of 10 seconds on each QuIt problem. Our experiments show that Vampire was able to find invariants for all the problems. Depending on the background theory encoding used in the QuIt problem, Vampire generated between 411 and as many as 11000 clauses within the time limit, each clause representing a loop invariant. The criteria used to evaluate the quality of these invariants depends largely on the application. To verify partial correctness with respect to pre- and post-conditions P and Q, a common task is to use invariants to prove that Q holds after the execution of the loop (typically one would also need to prove that the invariant is true under P; this is guaranteed

by our definition of P-invariants). In order to test the quality of our generated invariants for this application, we used the following procedure: we constructed a new first-order problem containing the generated invariants and the negation of the loop condition as hypotheses, and added the post-condition as the goal to be proven. We then ran VAMPIRE on the resulting problems, by using it in portfolio mode with a time limit of 60 seconds.

For those benchmarks where partial correctness can be proven from the invariants, we can analyze the resulting proof in order to gather some interesting invariants. For example for program `reverse` (Figure 3.1) the following invariant was generated and later used to prove correctness:

$$\forall x\ (x < 0\ \vee\ \neg x < i\ \vee\ b[(a.length - 1) - x] \approx a[x])$$

We were able to prove partial correctness from the generated invariants for 21 programs in total, a subset of the programs for which we were able to establish correctness using only extended expressions. The list of these programs is given in Table 3.2. Programs for which the post-condition could be proven by VAMPIRE from the invariants generated are denoted by ✓.

3.7 Related work

Our definition of the theory of L-valid sentences is reminiscent of modal logics: we consider sentences that are true across a certain class of interpretations (one interpretation of each possible execution of L), akin to the multiple worlds used by Kripke semantics. However in our setting there is no notion of accessibility between those different worlds. This allows the use of first-order quantification, without the difficulties that are inherent in defining semantics for *first-order* modal logic [57]. In addition, automated reasoning for modal logics remains a difficult problem, despite efforts in that direction [94].

Analyzing loops and generating quantified invariants has been addressed by a large number of approaches. One line of research iteratively generates quantifier-free properties that are generalized into universally quantified invariants. The work of [72] generates universally quantified inductive invariants by iteratively inferring and strengthening candidate invariants. The method uses SMT solving and is therefore restricted to first-order theories with a finite model property. SMT-based invariant generation is also performed in [64] and universal invariants with a

Benchmark	Vampire			
	A+T	A+I	F+T	F+I
absolute-prop1	t	t	t	t
absolute-prop2	t	t	t	t
atleast-one-iteration	t	t	t	t
both-or-none	t	t	t	t
check-equal-set-flag	t	t	t	t
copy	✓	✓	✓	✓
copy-nonzero-prop1	t	t	t	t
copy-nonzero-prop2	t	t	t	t
copy-odd	✓	✓	✓	✓
copy-partial	✓	✓	✓	✓
copy-positive	t	t	t	t
copy-two-indices	t	t	t	✓
find1-prop1	t	t	t	t
find1-prop2	t	t	t	t
find1-prop3	✓	✓	✓	✓
find2-prop1	✓	✓	✓	✓
find2-prop2	✓	✓	✓	✓
find2-prop3	✓	✓	✓	✓
find-max	t	t	t	t
find-max-from-second	t	t	t	t
find-max-up-to-prop1	t	t	t	t
find-max-up-to-prop2	t	t	t	t
find-min	t	t	t	t
find-min-up-to	t	t	t	t
find-sentinel	t	t	t	t
find-two-max-prop1	t	t	t	t
find-two-max-prop2	t	t	t	t
in-place-max	t	t	t	t
increment-by-one-prop1	✓	✓	✓	✓
increment-by-one-prop2	t	✓	t	✓
indexn-is-arraylength	t	✓	t	✓
init	✓	✓	✓	✓
init-conditionally-prop1	t	t	t	t
init-conditionally-prop2	t	t	t	t
init-even	t	t	t	t
init-non-constant	✓	✓	t	✓
init-partial	✓	✓	✓	✓
init-previous-plus-one	t	t	t	t
max-prop1	t	t	t	t
max-prop2	t	t	t	t
merge-interleave-prop1	t	✓	t	✓
merge-interleave-prop2	t	t	t	t
palindrome	t	t	t	t
partition	t	t	t	t
partition-init	t	t	t	t
push-back-prop1	t	✓	t	t
push-back-prop2	t	✓	t	✓
reverse	t	t	t	✓
set-to-one	t	t	t	t
str-cpy	✓	✓	✓	✓
str-len	✓	✓	✓	✓
swap-prop1	t	t	t	t
swap-prop2	t	t	t	t
vector-addition	t	✓	t	✓
vector-subtraction	✓	✓	✓	✓
Total	14	20	13	21
Unique	0	0	0	2

Table 3.2. Results of Vampire on proving partial correctness using invariants generated by symbol elimination.

bounded number of universal quantifiers are inferred. In [2], Craig interpolation over bounded loop executions is used to generate candidate ground invariants and terms to be universally quantified in those invariants. Candidate invariants are also used in the formula slicing approach of [73]. In [23], templates of quantified invariants are used to reduce the problem of quantified invariant generation to computing quantifier-free invariants. Template invariants together with SMT-based constraint solving is also used in [89] to generate universal invariants. Unlike these works, we are not limited to universal invariants but can infer first-order loop properties with alternations of quantifiers. First-order resolution has previously been used to derive invariants with alternations of quantifiers in [29]. In this work, the derivation is goal-oriented, whereas our technique does not require a post-condition to be given. The work of [98] relies on Craig interpolation in superposition theorem proving to generate quantified invariants. The approach is however restricted to universal invariants.

Our use of trace lemmas to guide the automation shares some similarity with template-based approaches for invariant generation [39, 63]. Our work, however, does not require any assumptions on the syntactic shape of the target invariants. Instead, assumptions are made about semantic patterns that are often shared across many programs. The invariants are not restricted to the shape of the trace lemmas, and the lemmas are discovered automatically, without user guidance. Moreover, our approach can be used with arbitrary first-order theories, even with theories that have no interpolation property and/or a finite axiomatization.

Another line of work focuses on the design of specialized abstract domains to represent and infer universal properties by abstract interpretation. The fluid updates abstraction of [53] creates pair-wise points-to relations over arrays and solves these constrains using SMT solving. The array segmentation domain of [45] reasons about the contents of an array by dividing it into consecutive subsets of array elements. These methods are very expressive, but limited to their respective abstract domains, and to universal invariants. For example, the abstraction domain used in [45] would not be able to handle the program `reverse` from Figure 3.1. Rather than performing a custom, domain-specific analysis, our work introduces a generic first-order framework for deriving and proving first-order loop properties.

3.8 Conclusion

We described a logical framework for expressing and proving complex properties of loops. Our framework is based on the first-order language of extended expressions and supports full first-order quantification over both program values and iterations. We showed how to use our work to automate various tasks of program analysis and verification, in particular by using our approach in conjunction with automated reasoning techniques in first-order logic. For future work, we plan to extend our programming model by considering various background theories. For example, the theory of term algebras could be used to reason about programs with recursive data structures. Another interesting question is whether our semantics of iterations can be extended to support nested and consecutive loops in a way that remains tractable for automated theorem provers.

Chapter 4

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.

Originally published in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, volume 52, number 1, pages 260-270. ACM, 2017.

4.1 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 [34, 43, 62]; 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:

$$\texttt{x} \neq \texttt{zero} \vee \texttt{y} \neq \texttt{zero} \Rightarrow \texttt{add x y} \neq \texttt{zero}$$

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 [96]. 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 [96] 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 [12,113] 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 [9]. 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 $x \not\approx f(x)$, $x \not\approx f(f(x)), \ldots$ 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 4.4). 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 4.5).

We implemented our new inference system, as well as the subterm relation, in the first-order theorem prover Vampire [84]. 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 [113]. This set of examples were generated using the Isabelle inductive theorem prover [100] and translated by the Sledgehammer system [24]. Further, we also used problems from [38] with many quantifier alternations over term algebras. When compared to state-of-the-art SMT solvers, such as CVC4 and Z3 [50], 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 4.6).

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 4.4).

- 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 4.5).

- 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 4.6).

4.2 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.

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 [120] for details on computing mgus.

4.3 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. [122], 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.

4.3.1 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 $f \neq g$;

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

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.

4.3.2 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 2. *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., [122].

Decidability of $\mathcal{T}_{FT}$ in Theorem 2 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 2 follows from a result in [55]: 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 [133].

Let us make the following important observation. The decidability and other results of Theorem 2 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 [77] one can then prove the following result.

Theorem 3. *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 3 but refer to [77] 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:

$$\forall x.\, x + 0 = x$$

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

$$\forall x.\, x \cdot 0 = 0$$

$$\forall xy.\, s(x) \cdot y = (x \cdot y) + y$$

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 3 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).

4.3.3 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:

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

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

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

Given the other axioms, (A3) and (A3') are logically equivalent, but some authors prefer the presentation based on destructors. Note, however, that the behavior of a destructors p_f^i is unspecified on some terms.

4.3.4 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 $\tau_1 \times \cdots \times \tau_n \to \tau$. 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:

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

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

$$\exists y.\, x \approx leaf(y) \vee \exists y_1\, y_2\, y_3.\, x \approx node(y_1, y_2, y_3)$$

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

$$leaf(x) \approx leaf(y) \implies x \approx y$$

$$node(x_1, x_2, x_3) \approx node(y_1, y_2, y_3) \implies x_1 \approx y_1 \wedge x_2 \approx y_2 \wedge x_3 \approx y_3$$

$$x \not\approx node(x, y_1, y_2)$$

$$x \not\approx node(y_1, y_2, x)$$

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

$$\ldots$$

Figure 4.1. The instantiation of the theory axioms for the signature Σ_{Bin}.

term algebra axioms are only using sorts with constructors. The axioms of this theory of trees, as defined previously, are shown in Figure 4.1.

4.4 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 $n \geq i \geq 1$.

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

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

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 4. *$\mathcal{T}^+_{FT}$ 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 2, 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 $\mathcal{T}^+_{FT}$. □

Note that the full first-order theory of term algebras with the subterm predicate is undecidable [132].

The important difference between $\mathcal{T}_{FT}$ and $\mathcal{T}^+_{FT}$ is that $\mathcal{T}^+_{FT}$ *is finitely axiomatizable*. This fact and Theorem 4 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 2.

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 4, 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. [38], 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 3.6 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:

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

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.

4.5 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.

4.5.1 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 $f(s) \approx s \vee \mathcal{C}$ can be replaced by a simpler, yet equivalent, clause $\mathcal{C}$. Likewise, the clause $f(s) \approx f(t) \vee \mathcal{C}$ is equivalent, by injectivity of the constructors, to the clause $s \approx t \vee \mathcal{C}$. The clause $s \approx t \vee \mathcal{C}$ is also smaller than $f(s) \approx f(t) \vee \mathcal{C}$, 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.

4.5.2 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

$$\frac{f(s) \approx g(t) \vee \mathcal{C}}{\mathcal{C}}\ \mathsf{Dist}^{+}$$

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

$$\frac{f(s) \not\approx g(t) \vee \mathcal{C}}{\emptyset}\ \mathsf{Dist}^{-}$$

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

4.5.3 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

$$\frac{f(s_1 \ldots s_n) \approx f(t_1, \ldots, t_n) \vee \mathcal{C}}{\begin{array}{c} s_1 \approx t_1 \vee \mathcal{C} \\ \ldots \\ s_n \approx t_n \vee \mathcal{C} \end{array}}\ \mathsf{Inj}^{+}$$

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

$$\frac{f(s_1 \ldots s_n) \not\approx f(t_1, \ldots, t_n) \vee \mathcal{C}}{s_1 \not\approx t_1 \vee \ldots \vee s_n \not\approx t_n \vee \mathcal{C}}\ \mathsf{NInj}$$

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

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).

4.5.4 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 $f(t_1, \ldots, t_n)$, 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

$$\frac{s \approx t \vee \mathcal{C}}{\mathcal{C}}\ \mathsf{Acycl}^+$$

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

The deletion rule is

$$\frac{s \not\approx t \vee \mathcal{C}}{\emptyset}\ \mathsf{Acycl}^-$$

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:

$$\frac{t \approx u \vee \mathcal{C}}{s \not\approx u \vee \mathcal{C}}\ \mathsf{Acycl'}$$

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.

4.6 Experimental Results

4.6.1 Implementation

We implemented the subterm relation of Section 4.4 and simplification rules of Section 4.5 in the first-order theorem prover Vampire [84]. 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].

4.6.2 Input Syntax and Tool Usage

In our work, we used an extended SMTLIB syntax [10] 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 4.5. Alternatively, the user can choose not to activate the inference rules in our implementation. The inclusion of the *Sub* predicate and its axioms, as presented in Section 4.4, 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 [113]. 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.

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

4.6.3 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 [38]. 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 4.2.

- Problems about functional programs, generated by the Isabelle interactive theorem prover [100] and translated by the Sledgehammer system [24]. 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 originally appeared in [113] and, to the best of our knowledge, represent the set of all publicly available benchmarks on algebraic data types.

4.6.4 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 [113], 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 4.1. The first column indicates the time required by Vampire using the theory

```
(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 4.2. An instance of the game theory problem from [38], 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 $winning_1(s(z))$.

axioms (A) described in Section 4.4, and the second and third columns give the time needed when the simplification rules (R) are also activated in Vampire (Section 4.5). 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 4.1 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 solvers CVC4 and Z3 are able to disprove the negated conjecture only for $k = 1$ or $k = 2$. SMT solvers can

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 4.1. Time required to prove unsatisfiability of different instances of the game theory problem from [38].

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 [38], the authors are able to solve the problem for k as high as 80, using an implementation of the decision procedure presented in [130]. 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 4.1 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 [49]). Table 4.1 also shows that adding simplification rules as described in Section 4.5 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 4.2. 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 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,

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

Table 4.2. Number of problems solved among the 6282 Isabelle problems translated by SledgeHammer.

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 4.2 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 4.3 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 4.3. 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 4.3. Summarizing, Table 4.2 shows that Vampire outperforms the best existing solvers so far. The experimental results of Tables 4.1-4.2 provide an evidence that our methods for proving properties of algebraic data types outperform methods currently used by SMT solvers.

4.6.5 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;

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

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.

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.

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

Table 4.4. Comparison of proof option values for acyclicity in Vampire.

The results of these experiments are shown in Table 4.4. 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 4.5) instead of theory axioms (Section 4.4) 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.

4.7 Related Work

The problem of reasoning over term algebras first appears in the restricted form of syntactic unification, mentioned in [66]. The algorithm for syntactic unification was later described in [120], and later refined into quasi-linear [13, 70, 97] and linear algorithms [103].

The full-first order theory of term algebras over a finite signature was first studied in [96], where its decidability was proved by quantifier elimination. Other quantifier elimination procedures appeared in [40, 69, 95, 122]. [55] 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 [38]. The theory of finite or infinite trees is also studied in [130] 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 [12] and later extended to a decision procedure for co-algebraic data types in [113]. These decision procedures exploit SMT-style reasoning and are supported by CVC4. Z3 also supports proving properties about algebraic data types [22]. 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.

4.8 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 [113]. 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 [135], 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 [33] and automated reasoning tools for higher-order logic, e.g. Sledgehammer [24].

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.

Chapter 5

An Inference Rule for the Acyclicity Property of Term Algebras

Simon Robillard

Abstract. Term algebras are important structures in many areas of mathematics and computer science. Reasoning about their theories in superposition-based first-order theorem provers is made difficult by the acyclicity property of terms, which is not finitely axiomatizable. We present an inference rule that extends the superposition calculus and allows reasoning about term algebras without axioms to describe the acyclicity property. We detail an indexing technique to efficiently apply this rule in problems containing a large number of clauses. Finally we experimentally evaluate an implementation of this extended calculus in the first-order theorem prover Vampire. The results show that this technique is able to find proofs for difficult problems that existing SMT solvers and first-order theorem provers are unable to solve.

Originally published in *Proceedings of the 4th Vampire Workshop*, volume 53, pages 20-32. EasyChair, 2018.

5.1 Introduction

Term algebras are central to many areas of mathematics and computer science. In logic, they are closely related to the concept of Herbrand structures, and in other areas of mathematics, they can be used to formalize inductively defined structures. They are also useful in the study of programming languages, particularly those that manipulate inductive data types. The ability to reason about term algebras efficiently in an automatic prover is therefore of great importance.

The main difficulty in reasoning about their theory is caused by the *acyclicity property* of terms, which states that a term cannot be equal to one of its own subterms. This property cannot be described by a finite number of axioms in first-order logic, making it troublesome for first-order theorem provers based on superposition. Such provers find refutation proofs by saturating a set of clauses, and theory reasoning is accomplished by explicitly adding the theory axioms to the set of clauses to be saturated. Term algebra reasoning has therefore often been carried out by using dedicated decision procedures [122, 128, 130] which typically cannot be used on problems containing other theories. More recently, support for term algebra reasoning has been added to some SMT solvers [12, 114], but these are usually not as efficient as superposition-based provers on problems that contain heavily quantified formulas.

We previously tackled the problem of reasoning about term algebras in a superposition prover [81] (Chapter 4 of this thesis) by introducing a conservative extension of their theories, in which an additional predicate symbol is used to represent the *subterm* relation over terms. The predicate is defined by additional axioms; in particular it has the property of being irreflexive, which corresponds to the restriction that terms cannot be equal to their own subterms. This technique provides an easy way to perform complete reasoning in the theory of term algebras using any first-order theorem prover. However the subterm relation is transitive, which means that provers may generate a large number of clauses containing the subterm predicate, most of which will not be used in the proof.

In this paper we present an alternative solution to reason about the acyclicity property of term algebras. Instead of relying on axioms, we extend the superposition calculus with a new inference rule. The inferences that result from it are sound in all interpretations that satisfy the acyclicity property. This enables the prover to generate useful new consequences while minimizing the number of generated clauses, thus improving the efficiency of the proof search.

This paper is organized as follows. We define term algebras and their first-order theory in Section 5.2 and recall some notions about superposition in Section 5.3. In Section 5.4, we describe the new inference rule. Technical details allowing the rule to be efficiently implemented in a first-order theorem prover are given in Section 5.5. Lastly in Section 6.7, we evaluate this approach, as implemented in the first order-theorem prover Vampire, and compare it to other tools and techniques.

5.2 Term Algebras

In this section, we define term algebras and their first-order theories, and describe some of their properties. The context of this presentation is unsorted first-order logic, but the results can be extended to a many-sorted logic in a straightforward manner. Equality is part of the logic, the notation $\approx$ stands for the equality predicate in first-order logic, and $\not\approx$ for its negation.

5.2.1 First-Order Theory

Let Σ be a finite collection of function symbols containing at least one constant. We call these symbols *term constructors* and denote them by the letters e, f, g, ... We denote by $\mathcal{T}(\Sigma)$ the set of ground terms constructed from Σ.

The Σ-*term algebra* is the algebraic structure whose carrier set is $\mathcal{T}(\Sigma)$ and in which terms are interpreted as themselves: every constant symbol e is interpreted as the corresponding constant in $\mathcal{T}(\Sigma)$, and every n-ary function symbol f is interpreted as the function from $\mathcal{T}(\Sigma)^n$ to $\mathcal{T}(\Sigma)$ that maps the tuple $(t_1, \ldots, t_n)$ to the element $\mathsf{f}(t_1, \ldots, t_n)$.

Definition 8. We now define the first-order theory $\mathcal{T}_{FT}$ as the set of formulas that are consequences of the following axioms:

$$\forall x \left(\bigvee_{\mathsf{f} \in \Sigma} \exists \overline{y}.\, x \approx \mathsf{f}(\overline{y}) \right) \tag{A1}$$

$$\forall \overline{xy}.\, \mathsf{f}(\overline{x}) \not\approx \mathsf{g}(\overline{y}) \tag{A2}$$

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

$$\forall \overline{xy}\ (\mathsf{f}(\overline{x}) \approx \mathsf{f}(\overline{y}) \implies \overline{x} \approx \overline{y}) \tag{A3}$$

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

$$\forall x.\, x \not\approx t[x] \tag{A4}$$

for every term $t[x] \neq x$ in which x appears.

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 (A2) ensures that terms constructed from different constructors are distinct, while axiom (A3) describes the injectivity of term constructors. (A4) is an axiom schema: the resulting formulas assert that no term can be equal to one of its own subterms, i.e., terms are acyclic. The Σ-term algebra is a model of every formula in (A1)–(A4), and therefore of every formula in $\mathcal{T}_{FT}$.

We say that a formula is a *pure term algebra formula* if its language only contains function symbols from Σ and the equality predicate symbol. Consider for example a term algebra signature consisting of a constant z and a unary symbol s. The sentence

$$\forall xy\ (f(x,\mathsf{z}) \approx x \wedge f(x,\mathsf{s}(y)) \approx \mathsf{s}(f(x,y))) \implies \forall xy.\, f(x,y) \approx f(y,x) \tag{S}$$

is not a pure term algebra formula because it contains the function symbol f, which is not part of the set of term constructors Σ.

The theory $\mathcal{T}_{FT}$ is complete on pure term algebra formulas [95]: for any such formula φ, either $\varphi \in \mathcal{T}_{FT}$ or $\neg\varphi \in \mathcal{T}_{FT}$.

5.2.2 Acyclicity and Induction

The acyclicity property is closely related to the notion of induction. In order to illustrate this, we consider a theory $\mathcal{T}_{FT^{\mathrm{Ind}}}$, in which the acyclicity axiom schema (A4) is replaced by an induction axiom schema.

Definition 9. $\mathcal{T}_{FT^{\mathrm{Ind}}}$ is the set of formulas that are consequences of the axioms (A1)–(A3) and of the formulas that instantiate the following axiom schema:

$$\bigwedge_{\mathsf{f}\in\Sigma} \Big(\forall \bar{x}\, \big(\varphi(x_1)\wedge\cdots\wedge\varphi(x_n) \implies \varphi(\mathsf{f}(x_1,\ldots,x_n))\big)\Big) \implies \forall x.\, \varphi(x) \tag{A5}$$

for every sentence $\varphi(x)$ in the language in which x is the only free variable.

Lemma 7 (Properties of $\mathcal{T}_{FT^{\mathrm{Ind}}}$)**.** *The following properties hold:*

1. *$\mathcal{T}_{FT^{Ind}}$ is consistent;*
2. *for every formula ψ, if $\psi \in \mathcal{T}_{FT}$, then $\psi \in \mathcal{T}_{FT^{Ind}}$;*

3. *for every pure term algebra formula ψ, either $\psi \in \mathcal{T}_{FT^{Ind}}$ or $\neg\psi \in \mathcal{T}_{FT^{Ind}}$;*

4. *for every pure term algebra formula ψ, $\psi \in \mathcal{T}_{FT^{Ind}}$ if and only if $\psi \in \mathcal{T}_{FT}$.*

Proof. (1) holds because $\mathcal{T}(\Sigma)$ is a model of $\mathcal{T}_{FT^{\text{Ind}}}$. (2) holds because every formula of the axiom schema (A4) is a consequence of the axioms of $\mathcal{T}_{FT^{\text{Ind}}}$ (the acyclicity axioms can be proven by induction). (3) is a consequence of the completeness of $\mathcal{T}_{FT}$ and of (2). Finally for (4), one direction of the equivalence if is given by (2). For the other direction, we must show that $\psi \in \mathcal{T}_{FT^{\text{Ind}}}$ implies $\psi \in \mathcal{T}_{FT}$. By contradiction, assume $\psi \in \mathcal{T}_{FT^{\text{Ind}}}$ and $\psi \notin \mathcal{T}_{FT}$. Then by completeness of $\mathcal{T}_{FT}$, we have that $\neg\psi \in \mathcal{T}_{FT}$, and by (2) it follows that $\neg\psi \in \mathcal{T}_{FT^{\text{Ind}}}$, which contradicts the consistency of $\mathcal{T}_{FT^{\text{Ind}}}$. □

Naturally, if we consider languages that include arbitrary predicate and function symbols, $\mathcal{T}_{FT}$ is a strict subset of $\mathcal{T}_{FT^{\text{Ind}}}$. Consider again the sentence (S) given above: intuitively, the term algebra generated by the signature $\Sigma = \{\mathsf{z}, \mathsf{s}\}$ is isomorphic to the algebra of natural numbers, and f can be interpreted as the addition on these numbers. The sentence, which expresses the commutativity of f, belongs to $\mathcal{T}_{FT^{\text{Ind}}}$ but cannot be proven without induction, and therefore does not belong to $\mathcal{T}_{FT}$.

From the previous results we gather that the acyclicity property is strictly weaker than the principle of induction, while at the same time being sufficiently strong to ensure the completeness of $\mathcal{T}_{FT}$ on pure term algebra formulas.

5.3 First-Order Logic and Superposition

We now recall some definitions related to first-order logic and the superposition calculus. A more complete overview of these topics can be found in [84].

First-order theorem provers work by applying inferences to a set of clauses and adding the conclusions to that set. The typical application is to find refutation proofs, by deriving the empty clause from a set that initially contains hypotheses and a negated conjecture. Satisfiability results are also possible if the empty clause is not found and the set is saturated – no new inferences are possible among its clauses. The calculi used by these provers are based on the superposition calculus, which has the property of being refutationally complete: for any unsatisfiable set of clauses, there exists a refutation proof in the calculus.

Terms appearing in clauses may contain variables (denoted $x, y, z, \dots$) which are implicitly universally quantified. To perform inferences between such quantified clauses, rules in the superposition calculus require the detection of unifiable terms and the computation of a *substitution* under which those terms are equal. A substitution is a function from variables to terms. Given a term t and a substitution θ, we denote by $t\theta$ the application of θ to t, in which all occurrences of variables $x_1, \dots, x_n$ in t have been simultaneously replaced by $x_1\theta, \dots, x_n\theta$. Application of a substitution can be extended to literals and clauses. The composition of two substitutions σ and τ is the function that takes every variable x to $(x\sigma)\tau$. It is itself a substitution and is denoted $\sigma\tau$. An *equation* is a pair of terms, denoted $s \stackrel{?}{=} t$, and a substitution θ is a *unifier* of s and t, or a solution of the equation, if $s\theta = t\theta$. Moreover, if for every unifier τ of an equation E, there exists a substitution δ such that $\tau = \sigma\delta$, then σ is a *most general unifier* (mgu) of E. The notions of unifier and mgu can be applied to a finite set of equations, if the substitution is a solution of every equation in the set.

To direct the proof search, first-order theorem provers use a *selection function*, a function that selects a non-empty subset of literals in any non-empty clause. The selection function is used to restrict the number of possible inferences (resolution is performed only on selected literals, for example) while preserving refutational completeness of the system, provided that the function satisfies certain properties. Different selection functions may be used, leading to variations of the calculus. To indicate that a literal is among the selected literals in a clause, we show it over a gray background, e.g., $s \approx t \vee \mathcal{C}$.

The calculus is also parametrized by an ordering on terms. This is another important component to limit possible inferences and ensure the efficiency of the proof search. However the rule described in the following does not use order restrictions, therefore we do not describe the notion further.

5.4 An Inference Rule for Acyclicity

In this section we describe an inference rule that extends the superposition calculus and allows reasoning about the acyclicity property without adding the corresponding axioms to the set of clauses to saturate.

In the remainder of this text, it is important to distinguish symbols belonging to Σ (*term constructors*) from other symbols. This distinction is necessary because the rule must be applicable and sound even for problems

that contain uninterpreted symbols or other theory symbols. Indeed, even clauses resulting from the clausification of a pure term algebra formula may contain non-constructor symbols, in particular those introduced by Skolemization. Constructors are denoted by the letters e, f, g, ... while we use s, t, u, ... to denote arbitrary terms.

Definition 10. We say that a term s *occurs under term constructors* in a term t, if t is of the form $\mathsf{f}(u_1, \ldots, u_n)$ and there exists u_i with $1 \leqslant i \leqslant n$ such that either:

1. $s = u_i$
2. s occurs under term constructors in u_i.

We will use the notation $p[s]_\Sigma$ to denote a term in which s occurs under constructors. For any ground terms s and $p[s]_\Sigma$ and any interpretation $\mathcal{I}$ that satisfies the instances of axiom schema (A4), it must be the case that $\mathcal{I}(p[s]_\Sigma) \neq \mathcal{I}(s)$.

Definition 11. The inference rule Acycl^+, which takes an arbitrary number n of premises, is defined as follows:

$$\frac{t'_1 \approx p[t_2]_\Sigma \vee \mathcal{C}_1 \qquad t'_2 \approx q[t_3]_\Sigma \vee \mathcal{C}_2 \qquad \ldots \qquad t'_n \approx r[t'_1]_\Sigma \vee \mathcal{C}_n}{(\mathcal{C}_1 \vee \mathcal{C}_2 \vee \cdots \vee \mathcal{C}_n)\theta}\ \mathsf{Acycl}^+$$

where θ is an mgu of the set of equations $\{t_1 \stackrel{?}{=} t'_1, \ldots, t_n \stackrel{?}{=} t'_n\}$.

In essence, this rule finds a set of equalities that contradict the acyclicity property under a certain substitution. Since these equalities cannot all be true under the substitution, the conclusion indicates that at least one $\mathcal{C}_i$ must be true. We first illustrate this principle with some examples of concrete applications, then demonstrate the soundness of the inference rule with Lemma 8.

Example 1. In this simple example, the rule has only one premise and the unifier is the empty substitution:

$$\frac{s \approx \mathsf{g}(\mathsf{e}, \mathsf{f}(s)) \vee \mathcal{C}}{\mathcal{C}}$$

Example 2. Here is a more complex example of application of the rule, with three premises:

$$\frac{s \approx \mathsf{f}(t) \vee \mathcal{C}_1 \qquad t \approx \mathsf{g}(u(x), \mathsf{e}) \vee \mathcal{C}_2[x] \qquad u(s) \approx \mathsf{f}(s) \vee \mathcal{C}_3}{\mathcal{C}_1 \vee \mathcal{C}_2[s] \vee \mathcal{C}_3}$$

The substitution $\{x \mapsto s\}$ is used as unifier and applied to the conclusion.

Example 3. Consider the clause

$$s \approx \mathsf{f}(u(s)) \vee \mathcal{C}$$

The rule may not be applied to this clause without additional premises, as s does not occur under term constructors in the right-hand side of the selected literal. Since u is not a term algebra constructor, there exist interpretations that associate $u(s)$ with a term (in the domain of discourse) not featuring s as a subterm, and in which $s \approx \mathsf{f}(u(s))$ holds.

Lemma 8 (Soundness of Acycl^+)**.** *For any interpretation $\mathcal{I}$ that satisfies the instances of axiom schema (A4), if the premises of the rule hold in $\mathcal{I}$, then the conclusion holds in $\mathcal{I}$ as well.*

Proof. Let θ be a unifier of the equations $\{t_1 \stackrel{?}{=} t'_1, \ldots, t_n \stackrel{?}{=} t'_n\}$ (in particular, θ may be an mgu, although this is not required for soundness).

By contradiction, assume that $(\mathcal{C}_1 \vee C_2 \vee \cdots \vee \mathcal{C}_n)\theta$ does not hold in $\mathcal{I}$. Instances of the premises $(t'_1 \approx p[t_2]_\Sigma \vee \mathcal{C}_1)\theta$, $(t'_2 \approx q[t_3]_\Sigma \vee \mathcal{C}_2)\theta$, $\ldots$, $(t'_n \approx r[t_1]_\Sigma \vee \mathcal{C}_n)\theta$ hold in $\mathcal{I}$, therefore it must the case that $(t'_1 \approx p[t_2]_\Sigma)\theta$, $(t'_2 \approx q[t_3]_\Sigma)\theta$, $\ldots$, $(t'_n \approx r[t_1]_\Sigma)\theta$ also hold. Since $t_i\theta = t'_i\theta$ for $1 \leqslant i \leqslant n$, there exists at least one ground term $p'[t_1\theta]_\Sigma$ such that $\mathcal{I}(p'[t_1\theta]_\Sigma) = \mathcal{I}(t_1\theta)$, which contradicts the hypothesis on $\mathcal{I}$. □

In addition to Acycl^+, we can also add a rule to deal with disequalities:

$$\frac{t \not\approx p[t]_\Sigma \vee A}{\emptyset}\ \mathsf{Acycl}^-$$

As denoted by the double line, this is a *simplification rule*, i.e., a rule that deletes its premise after application. Here the rule generates no conclusion, but merely deletes a clause that is always true in the theory. Such rules do not add to the deductive power of the calculus, but by deleting useless clauses they lighten the load of the prover and thus play an important practical role. Their application is also very inexpensive and should be carried out eagerly: every time a clause is generated it may be tested against this rule and discarded when applicable.

5.5 Implementation

There exist different saturation algorithms, as described in [117], but in general a prover will maintain a set of *active clauses* such that all possible

inferences between these clauses have been performed. Every time a new clause $\mathcal{C}$ is selected for inference, all active clauses must be tested to check whether they can participate in an inference with $\mathcal{C}$. In the case of the rule Acycl^+, this test is particularly difficult:

1. An arbitrary number of clauses can participate in the inference. Assuming that the number of active clauses is k, an exhaustive test of the 2^k possible combinations would be very impractical, given that k is often large. In contrast, all other rules of the standard superposition calculus are either unary or binary.

2. The rule requires computing an mgu over a set of equations, rather than over a single equation like other rules of the calculus. While this problem is well-known and can be solved efficiently [97], first-order theorem provers typically avoid solving it directly and instead rely on *term indexing* [125] to retrieve, among a set of indexed terms, all of those that are unifiable with a given term t.

In order to achieve a practical implementation of the rule, it is important to develop an indexing strategy enabling the prover to retrieve sets of clauses to be used as premises. In this section we describe an algorithm that accomplishes this.

5.5.1 Data Structures

The indexing strategy relies on the use of two auxiliary data structures to retrieve terms, literals and clauses that appear among the set of active clauses.

Subterm index. This index must support queries over terms with the following invariant: given a term s, the query $\texttt{subtermClause}(s)$ must return all pairs $(t, \mathcal{C})$ such that t is a term and $\mathcal{C}$ is an active clause of the form $s \approx p[t]_\Sigma \vee \mathcal{C}'$. This index can be implemented straightforwardly as a map, and must be updated every time a clause is added to or removed from the set of active clauses.

Unification index. This index supports a query over terms: given a term s, the query $\texttt{unifiable}(s)$ returns all pairs (t, σ) such that there exists an active clause $s' \approx t \vee \mathcal{D}$ and s' is unifiable with s under an mgu σ. Like the other one, this index must be updated every time the set of active clauses is modified. The efficient implementation of such an index is not trivial: unlike the subterm index, a query upon term s may return results even if s does not appear in any of the active clauses. Efficient retrieval of unifiable terms is central to the implementation of

first-order theorem provers, and any state-of-the-art prover should offer data structures that can be used to implement such an index.

5.5.2 Retrieving Premises

Every time a new clause $\mathcal{C}$ is selected for inference, we call the procedure `performInferences`($\mathcal{C}$) to perform all possible inference between $\mathcal{C}$ and active clauses. For this, the subterm and unification indexes are first updated to include $\mathcal{C}$, then a search is conducted to find the sets of premises.

We give now a procedure to detect all sets of premises among active clauses – more precisely, all minimal sets with respect to subsumption – and perform the corresponding applications of Acycl^+ among the active clauses (Algorithm 1).

```
Procedure performInferences(C_1) is
    for t s.t. C_1 = t_1 ≈ p[t]_Σ ∨ C' do
        enumerate(t_1, t, ε, {C_1})
    end
end

Procedure enumerate(t_1, t, θ, P) is
    for (t', σ) ∈ unifiable(tθ) do
        if t' = t_1 then
            apply Acycl+ to P under σθ
        else
            for (t_i, C_i) ∈ subtermClause(t') do
                if C_i ∉ P then
                    enumerate(t_1, t_i, θσ, P ∪ {C_i})
                end
            end
        end
    end
end
```

Algorithm 1: A procedure to apply Acycl^+ among active clauses

Given two terms t_1 and t and a substitution θ, a call to the procedure `enumerate`(t_1, t, θ, P) applies Acycl^+ to all minimal sets of premises that include P. Every call to `enumerate`(t_1, t, θ, P) verifies the following invariant: the selected literals in P imply an equality between $t_1\theta$ and some term $p[t\theta]_\Sigma$. The parameter P is used to collect the premises. The substitution is computed by composing the mgus of each individual equation,

starting with the empty substitution ϵ. The correctness of this construction is proven in Lemma 9: the side condition of this lemma is satisfied because variables from distinct clauses are always distinct themselves.

Lemma 9 (Correctness of the mgu)**.** *Let θ be an mgu of a set of equations E and σ an mgu of an equation $s\theta \stackrel{?}{=} t$ such that t does not contain variables appearing in E, then:*

1. *$\theta\sigma$ is a unifier of $E' = E \cup \{s \stackrel{?}{=} t\}$*
2. *$\theta\sigma$ is a most general unifier of E'*

Proof. Let us first note that since the variables of t do not appear in E, and θ is an mgu of E, the variables of t do not belong to the domain of θ, from which we have that $t\theta = t$.

For (1), we have that θ is a unifier of E, therefore for any substitution δ, $\theta\delta$ is a unifier of E, so in particular this holds for $\theta\sigma$. In addition, as $t\theta = t$, $\theta\sigma$ is also a unifier of $s\theta \stackrel{?}{=} t$.

For (2), we must show that for any α that is a unifier of E', there exists a substitution δ such that $\theta\sigma\delta = \alpha$. As E is a subset of E', α must also be a unifier of E. As θ is an mgu of E, there exists δ_1 such that $\theta\delta_1 = \alpha$. δ_1 is a unifier of $\{s\theta \stackrel{?}{=} t\}$, or equivalently of $\{s\theta \stackrel{?}{=} t\theta\}$, and consequently $\theta\delta_1$ is a unifier of $\{s\theta \stackrel{?}{=} t\}$. As σ is an mgu for that set, there exists δ_2 such that $\sigma\delta_2 = \delta_1$. The substitution δ_2 satisfies $\theta\sigma\delta_2 = \alpha$, showing that $\theta\sigma$ is most general. □

Lemma 10 (Termination)**.** *The procedure* `performInferences` *terminates.*

Proof. Termination of the procedure follows from the following facts:

1. Queries to the subterm index and the unification index always return a finite number of results, so that each call to `enumerate`(t_1, t, θ, P) only makes a finite number of recursive calls.
2. The condition $\mathcal{C}_i \notin P$ ensures that the depth of the recursion is bounded by the number of active clauses, as an element is added to P on every recursive call.

□

5.6 Experiments

We implemented the new inference rule in the first-order theorem prover Vampire. As described in [81] (Chapter 4 of this thesis), Vampire already

provides support for term algebra reasoning, relying on a conservative extension of the theory to enforce the acyclicity property. Our new rule can be used instead of this mechanism, and we compare the two approaches below.

Among currently available benchmarks, few problems rely on the acyclicity property: many of them are theorems that hold on term algebras as well as on similar structures that do not satisfy the acyclicity property (such as algebras of rational trees, described in [95]). For example in [114], the authors find that only 6 problems (among 4170) are solved exclusively when the acyclicity rule of CVC4 is activated, a result confirmed by experiments with Vampire. Moreover, those 6 problems are very simple, and any prover implementing some form of reasoning about the acyclicity property should be able to solve them. In order to provide a meaningful evaluation of the performance of the different techniques for handling the acyclicity property, we generated 200 new problems[1] of various size and complexity. The problems were constructed by generating DNF formulas in which each disjunct contains literals that imply a cyclic equality. The formulas generated in this manner contain between 1 and 20 disjuncts, each containing between 1 and 20 literals. The signature of the algebra was also varied across the different problems. We separated the problems in two sets: the first set contains 100 problems without universal quantifiers, so that their clausified forms feature only ground terms; the remaining 100 problems include quantifiers, and therefore variables are present after clausification. No other theories were involved, and the only uninterpreted symbols are constants which can be seen as Skolem symbols, so that the problems belong to the decidable fragment of the theory of term algebras.

While these problems do not correspond to real applications of theorem provers, they allowed us to specifically evaluate reasoning about acyclicity. In contrast to other available benchmarks, none of these problems can be solved without some non-trivial way to enforce this property of term algebras.

We compare four solvers/configurations:

1. Vampire$^{\text{Inf}}$ uses the extended calculus described in this paper. Axioms (A1)–(A3) are included in the set of clauses to be saturated (together with the problem hypotheses and negated conjecture) but no axioms describing the acyclicity property are used.

2. Vampire$^{\text{Ext}}$ uses the standard superposition calculus, and instead relies on a conservative extension of the theory of term algebras

[1] These benchmarks are available at `http://www.cse.chalmers.se/~simrob/tools.html`

to reason about the acyclicity property. An additional predicate symbol is added to the signature, which corresponds to the subterm relation over terms. In addition to axioms (A1)–(A3), the initial set of clauses also contains formulas that define the subterm relation.

3. CVC4 is an SMT solver that includes a theory solver for the theory of finite term algebras [114]. This solver constructs equivalence classes for terms present in the problem, checking that none of them correspond to infeasible values (cyclic terms).

4. Z3 [50] is another SMT solver with support for this theory.

Apart from the difference above, VAMPIRE$^{\text{INF}}$ and VAMPIRE$^{\text{EXT}}$ share identical parameters. Notably, simplification rules related to term algebra constructors (as described in [81], Chapter 4 of this thesis) are activated, as well as AVATAR [134]. All 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 24 GiB of memory. The different solvers were run on each problem with a time limit of 60 seconds.

Initial tests with these problems led to some minor optimizations in the implementation of the inference rule. After communication with its developer, the theory solver of CVC4 was also improved on the basis of these benchmarks. The results that we give here take these improvements into account. They are presented in Figure 5.1.

Among the 100 problems containing only ground hypothesis clauses, VAMPIRE$^{\text{EXT}}$ was able to solve 90 of the problems and exceeded the time limit for the remaining 10 problems. The total time required to solve the 90 problems was nearly 800 seconds. VAMPIRE$^{\text{INF}}$ however was able to solve all of the problems, and took less than 14 seconds to do so. Each individual problem was solved in at most 0.6 second. CVC4 was able to solve all the problems within the time limit, the combined time to solve these problems was 45 seconds. Z3 was the most efficient, solving all the problems in less than 2 seconds.

On the problems containing variables, the results are generally similar for the two approaches based on superposition: VAMPIRE$^{\text{EXT}}$ solved 93 of the problems before the time limit, taking nearly 700 seconds to do it, while VAMPIRE$^{\text{INF}}$ solved all of the problems in less than 12 seconds. The performance of SMT solvers is however noticeably different on this set of problems: Z3 solved 76 of the problems fairly quickly (103 seconds) but exceeded the time limit for the remaining 24 problems, while CVC4 could only solve 12 problems, in 14 seconds.

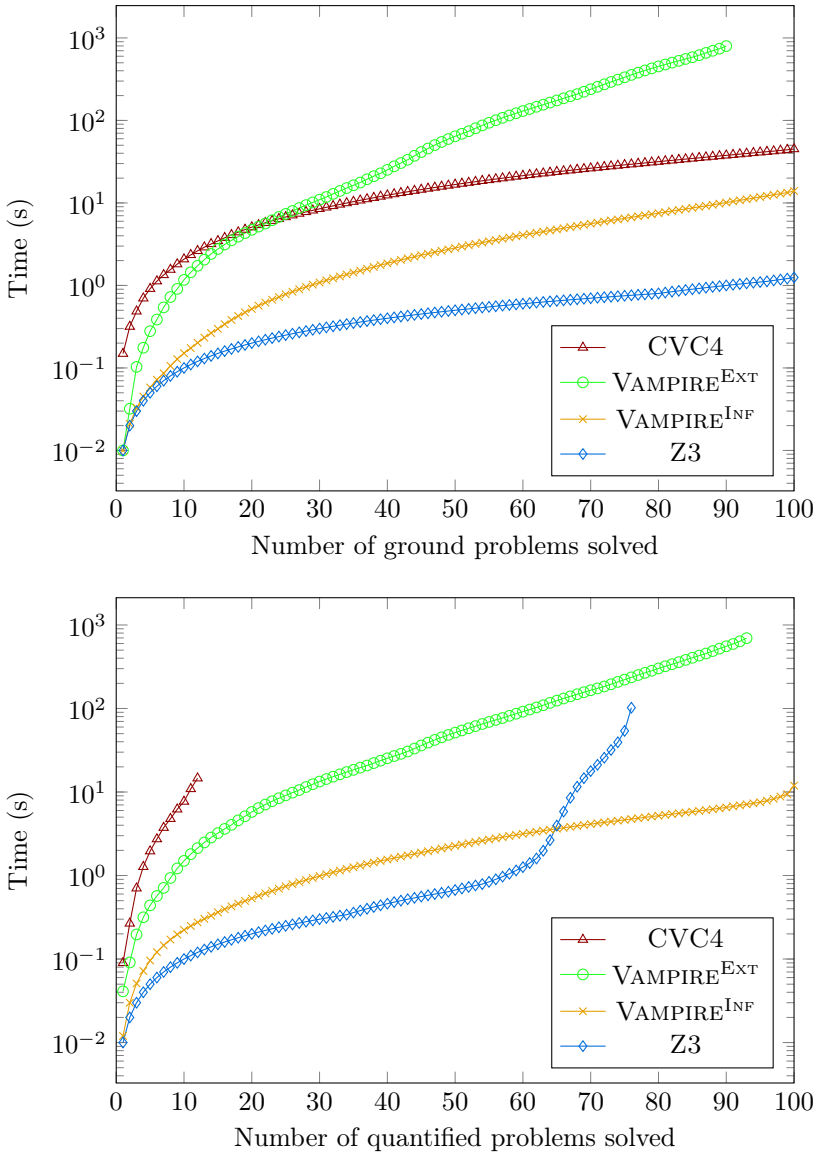


Figure 5.1. Time required to solve a number of problems among both sets. Where no number is given, the solver was unable to solve some of the problems within the 60-second limit imposed on each run.

The new inference rule clearly proves useful for solving difficult problems based on the acyclicity property. In particular, it is the only approach that succeeded in finding proofs for all 200 problems. This approach was also very fast on all benchmarks: in the worst case it took 2.5 seconds to find a proof, while all other problems were solved in less than 1 second.

The presence of non-ground terms in the hypotheses does not affect the performance of VAMPIRE$^{\text{EXT}}$, nor that of VAMPIRE$^{\text{INF}}$, despite the added complexity of finding relevant premises among active non-ground clauses. This is in contrast with the performance of SMT solvers, which is negatively affected by the presence of variables in the clauses.

5.7 Related Work

The idea of replacing axioms by an inference rule is central to paramodulation [138], and consequently to the advent of useful first-order theorem provers. Paramodulation is a rule that replace the axioms of the equality predicate. While finite, the axiomatization of the equality predicate is potentially large, as one additional formula is required for each symbol in the signature. It is theoretically possible to reason about equality without paramodulation, but in practice only very simple problems can be solved that way.

In its form, the rule presented here shares some similarity with the hyper-resolution rule introduced by Robinson [119], as it finds a resolvent among an arbitrary number of clauses. Efficient implementation of hyper-resolution is notoriously difficult, solutions have been proposed by Overbeek [102]. However, hyper-resolution seems to have fallen out of favor among modern provers, perhaps because it yields only small benefits compared to binary resolution.

The acyclicity property is of some importance in Prolog, where it corresponds to the *occur-check* of the unification algorithm. For performance reasons, many Prolog implementations do not enforce this check. This means that such implementations actually solve equations over algebras of infinite trees [37]. This change in semantics has been formalized by van Emdem et al. [131], while Plaisted [106] and Apt [3] establish criteria to determine whether the omission of the occur-check modifies the semantics of a given program.

Decisions procedures based on quantifier elimination have been used to show the completeness of the theory of term algebras, for example by Maher [95], or by Rybiana et al. in the context of a theory extended with queues [122]. More practical decision procedures have been described and

evaluated [128, 130]. Barrett et al. have introduced a theory solver for inductive data types in the SMT solver CVC4 [12] and the SMT solver Z3 uses a comparable theory solver (unpublished work by de Moura). These developments allow reasoning about problems that contain uninterpreted symbols, as well as mixed theories. Reynolds et al. have provided a theory solver that can also reason about co-inductive data types [114], while Bjørner has included a decision procedure for both inductive and co-inductive data types in STeP, the Stanford Temporal Prover [21]. In his PhD thesis, Wand proposes an extension of the superposition calculus with support for inductive data types and inductive reasoning over these types [136].

5.8 Conclusion

We have presented an inference rule aimed at replacing the infinitely many axioms needed to describe the acyclicity property of term algebras. Thanks to the indexing strategy described, an efficient implementation of the rule can be achieved in theorem provers. In comparison to other techniques applicable in saturation-based prover (in particular, the theory extension described in our previous work [81], Chapter 4 of this thesis), this rule generates fewer consequences and does not needlessly expand the search space, leading to better performance. The rule is not proven to be complete with respect to the axioms of acyclicity, but it is empirically shown to outperform other approaches for reasoning about the acyclicity property of term algebras.

Similar approaches could be used to reason about other theories without finite axiomatizations. In particular the theory of infinite trees [37] is a good candidate. This theory provides a first-order semantics for co-inductive data types [114] and shares many similarities with the theory of term algebras. Notably, its uniqueness property – which asserts the existence of unique cyclic elements – is not finitely axiomatizable. A better characterization of that theory and its properties, in particular from the point of view of automated theorem proving, would be helpful for program verification and interactive theorem proving.

Chapter 6

Superposition with Datatypes and Codatatypes

Jasmin Blanchette, Nicolas Peltier and Simon Robillard

Abstract. The absence of a finite axiomatization of the first-order theory of datatypes and codatatypes represents a challenge for automatic theorem provers. We propose two approaches to reason by saturation in this theory: one is a conservative theory extension with a finite number of axioms; the other is an extension of the superposition calculus, in conjunction with axioms. Both techniques are refutationally complete with respect to nonstandard models of datatypes and nonbranching codatatypes. They take into account the acyclicity of datatype values and the existence and uniqueness of cyclic codatatype values. We implemented them in the first-order prover Vampire and compare them experimentally.

Originally published in *9th International Joint Conference on Automated Reasoning (IJCAR 2019)*, volume 10900 of *LNCS*, pages 370-387. Springer, 2018.

6.1 Introduction

The ability to reason about inductive and coinductive datatypes has many applications in program verification, formalization of the metatheory of programming languages, and even formalization of mathematics. Inductive datatypes, or simply *datatypes*, consist of finite values freely generated from constructors. Coinductive datatypes, or *codatatypes*, additionally support infinite values. Non-freely generated (co)datatypes are also useful. All of these variants can be seen as members of a single unifying framework (Section 6.2).

It is well known that the first-order theory of datatypes cannot be finitely axiomatized. Distinctness, injectivity, and exhaustiveness of constructors are easy to axiomatize, but acyclicity is more subtle, and for induction we would need an axiom schema or a second-order axiom. Codatatypes are also problematic: Besides a coinduction principle that is dual to induction, they are characterized by the existence of all possible infinite values, corresponding intuitively to infinite ground terms. Both datatypes and codatatypes represent a challenge for automatic theorem provers.

Superposition [6] is a highly successful calculus for reasoning about first-order clauses and equality. There has been some work on extending superposition with induction [46, 136], including by Kersani and Peltier [75], and on the axiomatization of datatypes, including by Kovács, Robillard, and Voronkov [81] (Chapter 4 of this thesis). In this paper, we propose both axiomatizations and extensions of the superposition calculus to support freely and non-freely generated datatypes as well as codatatypes.

We first focus on a conservative extension of the theory with a finite number of first-order axioms that capture the basic properties of constructors, acyclicity of datatype values, uniqueness of cyclic (ω-regular) codatatype values, and existence of all codatatype cyclic values (Section 6.3). These axioms admit nonstandard models; for example, for the Peano-style natural numbers freely generated by $\mathsf{zero} : \mathit{nat}$ and $\mathsf{suc} : \mathit{nat} \to \mathit{nat}$, we cannot exclude the familiar nonstandard models of arithmetic, in which arbitrarily many copies of $\mathbb{Z}$ may appear besides $\mathbb{N}$. Similarly, the domains interpreting codatatypes are not guaranteed to contain all infinite acyclic values.

The axiomatization of codatatypes up to a suitable notion of nonstandard models constitutes the first theoretical contribution of this paper. Our second, and main, theoretical contribution is an extension of super-

position with inference rules to reason about datatypes and codatatypes (Section 6.4). This is inspired by an acyclicity rule that Robillard presented at the Vampire 2017 workshop [118] (Chapter 5 of this thesis). The main distinguishing feature of our rules is that they are (in combination with a few axioms) refutationally complete and their side conditions have some new order restrictions, helping prune the search space. On the other hand, our approach also requires a relaxation of the side conditions of the superposition rule: For clauses of the form $\mathsf{c}(\bar{s}) \approx t \vee \mathcal{C}$, where c is a constructor and the first literal is maximal and positive, superposition inferences onto t must be performed, as in ordered paramodulation [4]. In addition, we propose, for the first time, calculus extensions to reason about codatatypes.

Both the theory extension and the calculus extension are designed to be refutationally complete with respect to nonstandard models of datatypes and *nonbranching* codatatypes—codatatypes whose constructors have at most one corecursive argument (Section 6.5).

The calculus extension can be integrated into the given clause algorithm that forms the core of a prover's saturation loop (Section 6.6). The inference partners for the acyclicity and uniqueness rules can be located efficiently. We implemented both the axiomatic and the calculus approaches in the first-order prover Vampire [84] and compare them empirically on Isabelle/HOL [100] benchmarks and on crafted benchmarks (Section 6.7).

6.2 Syntax and Semantics

Our setting is a many-sorted first-order logic. We let τ, υ range over simple types (sorts), s, t, u, v range over terms, $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$ range over function symbols, x, y, z range over variables, and $\mathcal{C}, \mathcal{D}, \mathcal{E}$ range over clauses. Literals are atoms of the form $s \approx t$ or $\neg\ s \approx t$, also written $s \not\approx t$. Clauses are finite disjunctions of literals, viewed as multisets. Substitutions are written in postfix notation, with $s\sigma\theta = (s\sigma)\theta$. The notation $\bar{x}$ represents a tuple $(x_1, \ldots, x_m)$, where $m \geqslant 0$, and $[m, n]$ denotes the set $\{m, m+1, \ldots, n\}$, where $m \leqslant n+1$.

A *position p of type τ in t* is a position in t such that $t|_p$ is of type τ. If s, t are terms and P is a set of positions of the same type as s in t, then $t[s]_P$ denotes the term obtained from t by replacing the subterms occurring at a position in P by s: $t[s]_P := s$ if $\varepsilon \in P$; $t[s]_P := t$ if $P = \emptyset$; and $\mathsf{f}(t_1, \ldots, t_n)[s]_P := \mathsf{f}(t_i[s]_{P_i})_{i \in [1,n]}$, with $P_i = \{q \mid i.q \in P\}$ otherwise. Given two positions p and q, we write $p < q$ if p is a proper prefix of q. Let

$\mathcal{C}tr$ be a distinguished finite set of function symbols, called *constructors*. We reserve the letters $\mathsf{c}, \mathsf{d}, \mathsf{e}$ for constructors. A *constructor position* in t is a position q in t such that for every $p < q$, the head symbol of $t|_p$ is a constructor.

Definition 12. The set of *constructor contexts* of profile $\tau \to \upsilon$ is defined inductively as follows: (1) if t is a term of type υ, then t is a constructor context of profile $\tau \to \upsilon$; (2) if $\Gamma_1, \ldots, \Gamma_n$ are constructor contexts of profile $\tau \to \tau_i$ and $\mathsf{c} : \tau_1 \times \cdots \times \tau_n \to \upsilon$ is a constructor, then $\mathsf{c}(\Gamma_1, \ldots, \Gamma_n)$ is a constructor context of profile $\tau \to \upsilon$; (3) the hole $\bullet$ is a constructor context of profile $\upsilon \to \upsilon$.

Every constructor context can be written as $\Gamma[\bullet]_P$, where P is a set of constructor positions of the same type in Γ, denoting the positions of $\bullet$ in Γ. It is *empty* if $\varepsilon \in P$, and *constant* if $P = \emptyset$. We write $\Gamma[\bullet]_p$ as an abbreviation for $\Gamma[\bullet]_{\{p\}}$, and we write $\Gamma[t]_P$ to denote the term obtained by replacing every position of P by the term t in the context $\Gamma[\bullet]_P$. Moreover, we write $\tau \rhd \upsilon$ ("υ depends on τ") if there exists a constructor of profile $\tau_1 \times \cdots \times \tau_n \to \upsilon$, with $\tau = \tau_i$ for some $i \in [1, n]$, and $\tau \sim \upsilon$ if $\tau \rhd^* \upsilon$ and $\upsilon \rhd^* \tau$.

Proposition 1. *Let t be a term and let p be a constructor position in t.* $\mathrm{type}(t|_p) \rhd^* \mathrm{type}(t)$. *Consequently, if $\Gamma[\bullet]_P$ is a nonconstant constructor context of profile $\upsilon \to \tau$, then $\upsilon \rhd^* \tau$.*

Proof. The first result is by an immediate induction on p. Then the second result follows from the fact that $P \neq \emptyset$. □

The axioms and rules in this paper are parameterized by the following sets. Let $\mathcal{T}_{\mathsf{ind}}$ and $\mathcal{T}_{\mathsf{coind}}$ be disjoint sets of types, intended to model datatypes and codatatypes, respectively, and assume that the codomain of every constructor is in $\mathcal{T}_{\mathsf{ind}} \cup \mathcal{T}_{\mathsf{coind}}$. Let $\mathcal{C}tr_{\mathsf{inj}} \subseteq \mathcal{C}tr$ be a set of constructors, denoting injective constructors. Let $\bowtie$ be a binary symmetric and irreflexive relation among constructors; $\mathsf{c} \bowtie \mathsf{d}$ indicates that terms with head symbol c are always distinct from terms with head symbol d. Note that $\bowtie$ is not identical to $\neq$, because the constructors are not necessarily free.

We introduce some properties of interpretations that are intended to capture some of the properties of (co)datatypes. An interpretation $\mathcal{I}$ satisfies

- **Exh** (exhaustiveness) iff, for every type $\tau \in \mathcal{T}_{\mathsf{ind}} \cup \mathcal{T}_{\mathsf{coind}}$,

$$\mathcal{I} \models \bigvee_{i=1}^{m} \exists \bar{x}_i.\ x \approx \mathsf{c}_i(\bar{x}_i),$$

  where x is a variable of type τ, $\{\mathsf{c}_1, \ldots, \mathsf{c}_m\}$ is the set of constructors of codomain τ, and $\bar{x}_i$ is a (possibly empty) vector of pairwise distinct variables of the appropriate length and types;

- **Inf** (infiniteness) iff, for every type $\tau \in \mathcal{T}_{\mathsf{ind}} \cup \mathcal{T}_{\mathsf{coind}}$, the domain of τ is infinite;

- **Acy** (acyclicity, for datatypes) iff, for every type $\tau \in \mathcal{T}_{\mathsf{ind}}$ and for every nonempty constructor context $\Gamma[\bullet]_p$ of profile $\tau \to \tau$, where p is a position, we have

$$\mathcal{I} \models \Gamma[x]_p \not\approx x,$$

  where x is a variable of type τ not occurring in Γ;

- **FP** (existence and uniqueness of fixpoints, for codatatypes) iff, for every type $\tau \in \mathcal{T}_{\mathsf{coind}}$, for every nonempty constructor context $\Gamma[\bullet]_P : \tau \to \tau$,

$$\mathcal{I} \models (\exists x.\, \Gamma[x]_P \approx x) \wedge (\Gamma[x]_P \approx x \wedge \Gamma[y]_P \approx y \implies x \approx y),$$

  where x, y are distinct variables of type τ not occurring in Γ

- **Dst** (distinctness of constructors) iff, for every pair of constructors c, d of the same codomain such that $\mathsf{c} \bowtie \mathsf{d}$,

$$\mathcal{I} \models \mathsf{c}(\bar{x}) \not\approx \mathsf{d}(\bar{y})$$

  where $\bar{x}$ and $\bar{y}$ are disjoint vectors of pairwise distinct variables of the appropriate length and types;

- **Inj** (injectivity) iff, for every n-ary constructor $\mathsf{c} \in \mathcal{C}tr_{\mathsf{inj}}$ and pairwise distinct variables $x_1, \ldots, x_n$, $y_1, \ldots, y_n$ of the appropriate types,

$$\mathcal{I} \models \mathsf{c}(x_1, \ldots, x_n) \approx \mathsf{c}(y_1, \ldots, y_n) \implies \bigwedge_{i=1}^{n} x_i \approx y_i.$$

Most datatypes occurring in practice are recursive, so condition **Inf** is usually satisfied. In particular, it is the case for any nonempty freely generated (co)datatype τ such that $\tau \rhd^+ \tau$. Conditions **Dst** and **Inj** are defined by finite sets of axioms, but not conditions **Acy** and **FP**. In

Section 6.3, we introduce conservative extensions of the considered formula so that conditions **Acy** and **FP** are satisfied. Then in Section 6.4, we replace some of these axioms by inference rules.

We assume that $\tau \not\sim \upsilon$ whenever $\tau \in \mathcal{T}_{\mathsf{ind}}$ and $\upsilon \in \mathcal{T}_{\mathsf{coind}}$. Intuitively, this condition means that a datatype cannot be defined by mutual recursion with a codatatype, which is a very natural restriction [25]. If this condition does not hold, it is easy to see that there is no interpretation that satisfies both **Acy** and **FP**. On the other hand, we may have $\tau \rhd^+ \upsilon$ or $\upsilon \rhd^+ \tau$ with $\tau \in \mathcal{T}_{\mathsf{ind}}$ and $\upsilon \in \mathcal{T}_{\mathsf{coind}}$—a datatype can depend on a codatatype or vice versa. There may also exist types not belonging to $\mathcal{T}_{\mathsf{ind}} \cup \mathcal{T}_{\mathsf{coind}}$, and the types in $\mathcal{T}_{\mathsf{ind}} \cup \mathcal{T}_{\mathsf{coind}}$ may depend on them. Finally, we assume without loss of generality that for each type τ, there exists a ground term t (not necessarily built from constructors) of type τ.

6.3 Axioms

The axioms Exhaust for exhaustiveness, Dist for distinctness, and Inj for injectivity correspond to the formulas used to express the properties **Exh**, **Dst**, and **Inj** in Section 6.2. The other axioms are introduced below.

6.3.1 Acyclicity

For all types τ, υ such that $\tau \sim \upsilon$, we introduce a predicate symbol $\mathsf{sub}_\upsilon^\tau$ on $\tau \times \upsilon$ together with the following axioms, where $\tau \sim \upsilon \sim \upsilon'$ and $\mathsf{c} : \cdots \times \upsilon \times \cdots \to \upsilon'$ is a constructor:

$$\mathsf{Sub}_1\colon\ \mathsf{sub}_\tau^\tau(x, x) \qquad \mathsf{Sub}_2\colon\ \neg\, \mathsf{sub}_\upsilon^\tau(x, y) \vee \mathsf{sub}_{\upsilon'}^\tau(x, \mathsf{c}(\bar{z}, y, \bar{z}'))$$

$$\mathsf{NSub}\colon\quad \neg\, \mathsf{sub}_\tau^{\upsilon'}(\mathsf{c}(\bar{z}, x, \bar{z}'), x) \qquad \text{if } \tau \in \mathcal{T}_{\mathsf{ind}}$$

Let $\mathsf{Sub} = \mathsf{Sub}_1 \wedge \mathsf{Sub}_2$. The least fixpoint model of Sub is the usual subterm relation for constructor terms. The axiom NSub states that no term of a type in $\mathcal{T}_{\mathsf{ind}}$ may occur at a nonempty constructor position in itself.

Proposition 2. *Let $\Gamma[\bullet]_p$ be a constructor context of profile $\upsilon \to \tau$, where p is a position of type υ in Γ and $\tau \sim \upsilon$. Then $\mathsf{Sub} \models \mathsf{sub}_\tau^\upsilon(x, \Gamma[x]_p)$, where x is a variable of type υ.*

Proof. By an immediate induction on Γ. □

Definition 13. An interpretation $\mathcal{I}$ is sub-*minimal* if, for all $\tau \sim \upsilon$, $\mathsf{sub}^{\tau}_{\upsilon}(x, y)$ is equivalent to

$$\bigvee\{\exists \bar{z}.\ y \approx \Gamma[x]_p \mid \Gamma[\bullet]_p \text{ is a constructor context of profile } \tau \to \upsilon\}$$

where $\bar{z}$ denotes the vector of variables in Γ that are distinct from x, y.

Proposition 3. *Any* sub*-minimal interpretation satisfies* Sub.

Proof. Let $\mathcal{I}$ be a sub-minimal interpretation. By letting $p = \varepsilon$ in Definition 13, we deduce that $\mathcal{I} \models \mathsf{sub}(x, x)$. Furthermore, for any valuation η such that $\mathcal{I}, \eta \models \mathsf{sub}(x, y)$, we have $\mathcal{I}, \eta \models \exists \bar{z}''.\ y \approx \Gamma[x]_p$, for some constructor context $\Gamma[\bullet]_p$; thus $\mathcal{I}, \eta' \models y \approx \Gamma[x]_p$, for some extension η' of η (we assume by renaming that $\bar{z}''$ is disjoint from the vectors of variables $\bar{z}$ and $\bar{z}'$ occurring in Sub_2). Consequently $\mathcal{I}, \eta' \models \mathsf{c}(\bar{z}, y, \bar{z}') \approx \mathsf{c}(\bar{z}, \Gamma[x]_p, \bar{z}')$, hence $\mathcal{I}, \eta' \models \mathsf{c}(\bar{z}, y, \bar{z}') \approx \mathsf{c}(\bar{z}, \Gamma, \bar{z}')[x]_{i.p}$, with $i = |\bar{z}| + 1$, and thus $\mathcal{I}, \eta \models \exists z''.\ \mathsf{c}(\bar{z}, y, \bar{z}') \approx \mathsf{c}(\bar{z}, \Gamma, \bar{z}')[x]_{i.p}$. It is clear that $\mathsf{c}(\bar{z}, \Gamma, \bar{z}')[\bullet]_{i.p}$ is a constructor context, hence by Definition 13, we have $\mathcal{I}, \eta \models \mathsf{sub}(x, \mathsf{c}(\bar{z}, y, \bar{z}'))$. Consequently, $\mathcal{I} \models \mathsf{Sub}$. □

6.3.2 Contexts and Fixpoints

For every pair of types $\tau, \upsilon \in \mathcal{T}_{\mathsf{coind}}$ with $\tau \sim \upsilon$, we introduce a type $\boxed{\tau}_{\upsilon}$ to denote contexts $\Gamma[\bullet]_P$ of profile $\upsilon \to \tau$.

Let hole_{υ} (for every $\upsilon \in \mathcal{T}_{\mathsf{coind}}$) be a constant of type $\boxed{\upsilon}_{\upsilon}$, denoting an empty context. All constructors $\mathsf{c} : \tau_1 \times \cdots \times \tau_n \to \tau$ and types υ such that $\exists i\, \upsilon \rhd^* \tau_i$ are associated with new n-ary constructors $\boxed{\mathsf{c}}_{\upsilon} : \upsilon_1 \times \cdots \times \upsilon_n \to \boxed{\tau}_{\upsilon}$, where for every $i \in [1, n]$, $\upsilon_i = \boxed{\tau_i}_{\upsilon}$ if $\upsilon \rhd^* \tau_i$ and $\upsilon_i = \tau_i$ otherwise. Let $\mathsf{app}^{\tau}_{\upsilon} : \boxed{\tau}_{\upsilon} \times \upsilon \to \tau$, $\mathsf{cyc}_{\upsilon} : \boxed{\upsilon}_{\upsilon} \to \upsilon$, and $\mathsf{cst}^{\tau}_{\upsilon} : \tau \to \boxed{\tau}_{\upsilon}$ be new function symbols. As usual, type indices are often omitted for readability. Intuitively, if y denotes the context $\Gamma[\bullet]_P$, then $\mathsf{app}(y, x)$ denotes the term $\Gamma[x]_P$, $\mathsf{cyc}(y)$ denotes the fixpoint of $\Gamma[\bullet]_P$, and $\mathsf{cst}^{\tau}_{\upsilon}$ denotes a constant context (i.e., a context $\Gamma[\bullet]_P$ with $P = \emptyset$).

We consider the following axioms, where $\upsilon \in \mathcal{T}_{\mathsf{coind}}$ and x, y, x_i, z_i are

pairwise distinct variables of the appropriate types:

$$\mathsf{App}_1\colon\ \mathsf{app}^{\tau}_{\upsilon}(\mathsf{cst}^{\tau}_{\upsilon}(x), y) \approx x \qquad \mathsf{App}_2\colon\ \mathsf{app}^{\upsilon}_{\upsilon}(\mathsf{hole}_{\upsilon}, y) \approx y$$

$$\mathsf{App}_3\colon\ \mathsf{app}^{\tau}_{\upsilon}(\boxed{\mathsf{c}}_{\upsilon}(x_1, \ldots, x_n), y) \approx \mathsf{c}(t_1, \ldots, t_n)$$
if $\mathsf{c} : \tau_1 \times \cdots \times \tau_n \to \tau$ is a constructor and $\exists i\, \upsilon \rhd^* \tau_i$
with $t_i = \mathsf{app}^{\tau_i}_{\upsilon}(x_i, y)$ if $\upsilon \rhd^* \tau_i$ and $t_i = x_i$ otherwise

$$\mathsf{Uniq}\colon\ x \approx \mathsf{hole}_{\upsilon} \vee y \not\approx \mathsf{app}^{\upsilon}_{\upsilon}(x, y) \vee z \not\approx \mathsf{app}^{\upsilon}_{\upsilon}(x, z) \vee y \approx z$$

$$\mathsf{Cycl}\colon\ \mathsf{cyc}_{\upsilon}(x) \approx \mathsf{app}^{\upsilon}_{\upsilon}(x, \mathsf{cyc}_{\upsilon}(x))$$

$$\mathsf{Hole}_1\colon\ \mathsf{hole}_{\upsilon} \not\approx \mathsf{cst}^{\upsilon}_{\upsilon}(x) \qquad \mathsf{Hole}_2\colon\ \mathsf{hole}_{\upsilon} \not\approx \boxed{\mathsf{c}}_{\upsilon}(x_1, \ldots, x_n) \quad \text{if } \mathsf{c} : \cdots \to \upsilon$$

Let $\mathsf{App} = \mathsf{App}_1 \wedge \mathsf{App}_2 \wedge \mathsf{App}_3$ and $\mathsf{Hole} = \mathsf{Hole}_1 \wedge \mathsf{Hole}_2$.

Example 4. Let $\mathsf{c} : \tau_0 \times \upsilon \to \tau$ be a constructor, with $\upsilon \rhd^* \tau_0$. Then the profile of $\boxed{\mathsf{c}}_{\upsilon}$ is $\boxed{\tau_0}_{\upsilon} \times \boxed{\upsilon}_{\upsilon} \to \boxed{\tau}_{\upsilon}$. The term $t := \boxed{\mathsf{c}}_{\upsilon}(\mathsf{cst}^{\tau_0}_{\upsilon}(x), \mathsf{hole}_{\upsilon})$ encodes the constructor context $\mathsf{c}(x, \bullet)$. If $\mathsf{a} : \upsilon$, then

$$\begin{aligned}\mathsf{app}^{\tau}_{\upsilon}(t, \mathsf{a}) &=_{\mathsf{App}} \mathsf{c}(\mathsf{app}^{\tau_0}_{\upsilon}(\mathsf{cst}^{\tau_0}_{\upsilon}(x), \mathsf{a}), \mathsf{app}^{\upsilon}_{\upsilon}(\mathsf{hole}_{\upsilon}, \mathsf{a})) \\ &=_{\mathsf{App}} \mathsf{c}(x, \mathsf{app}^{\upsilon}_{\upsilon}(\mathsf{hole}_{\upsilon}, \mathsf{a})) \\ &=_{\mathsf{App}} \mathsf{c}(x, \mathsf{a})\end{aligned}$$

where $=_{\mathsf{App}}$ denotes equality modulo App (i.e., $s =_{\mathsf{App}} t \iff \mathsf{App} \models s \approx t$).

By contrast, if $\upsilon \not\rhd^* \tau_0$, the profile of $\boxed{\mathsf{c}}_{\upsilon}$ is $\tau_0 \times \boxed{\upsilon}_{\upsilon} \to \boxed{\tau}_{\upsilon}$, and the above context is encoded by $t' := \boxed{\mathsf{c}}_{\upsilon}(x, \mathsf{hole}_{\upsilon})$, with $\mathsf{app}^{\tau}_{\upsilon}(t', \mathsf{a}) =_{\mathsf{App}} \mathsf{c}(x, a)$. The difference between the two cases is that if $\upsilon \not\rhd^* \tau_0$, then all the contexts of profile $\upsilon \to \tau_0$ are constant. Thus they may be replaced by terms of type τ_0. There is no need to encode them using the function cst.

Remark 3. The axiom Uniq can be replaced by

$$\mathsf{Uniq}'\colon\ x \approx \mathsf{hole}_{\upsilon} \vee y \not\approx \mathsf{app}^{\upsilon}_{\upsilon}(x, y) \vee y \approx \mathsf{cyc}_{\upsilon}(x)$$

for $\upsilon \in \mathcal{T}_{\mathsf{coind}}$. Indeed, it is clear that $\mathsf{Uniq} \wedge \mathsf{Cycl} \iff \mathsf{Uniq}' \wedge \mathsf{Cycl}$.

Proposition 4. *Let t be a ground term of type $\boxed{\tau}_{\upsilon}$, with $\tau, \upsilon \in \mathcal{T}_{\mathsf{coind}}$ and $\tau \sim \upsilon$. There exists a ground constructor context $\Gamma[\bullet]_P$ of profile $\upsilon \to \tau$ such that $\mathsf{App} \models \mathsf{app}^{\tau}_{\upsilon}(t, x) \approx \Gamma[x]_P$, for all variables $x : \upsilon$. Furthermore, if $t \neq \mathsf{hole}_{\tau}$, then $\varepsilon \notin P$.*

Proof. The proof is by induction on t. If $t = \mathsf{hole}_{\tau}$, then $\mathsf{App}_2 \models \mathsf{app}(t, x) \approx x$. Let $\Gamma[\bullet]_P = \bullet$ with $P = \{\varepsilon\}$. By definition, $\Gamma[x]_P = x$

hence $\mathsf{App} \models \mathsf{app}(t, x) \approx \Gamma[x]_P$. If $t = \mathsf{cst}^\tau_\upsilon(s)$, then $\mathsf{App}_1 \models \mathsf{app}^\tau_\upsilon(t, x) \approx s$. Let $\Gamma = s$ and $P = \emptyset$. By definition, $\Gamma[x]_P = s$, hence $\mathsf{App} \models \mathsf{app}(t, x) \approx \Gamma[x]_P$. Now assume that $t = \boxed{\mathsf{c}}(t_1, \ldots, t_n)$. We have $\mathsf{App}_3 \models \mathsf{app}(t, x) \approx \mathsf{c}(t'_1, \ldots, t'_n)$ with $t'_i = \mathsf{app}(t_i, x)$ if $\upsilon \rhd^* \tau_i$ and $t'_i = t_i$ otherwise. Let $i \in [1, n]$ such that $\upsilon \rhd^* \tau_i$. Then we have necessarily $\tau_i \in \mathcal{T}_{\mathsf{coind}}$, hence by the induction hypothesis, there exists a ground constructor context $\Gamma_i[\bullet]_{P_i}$ such that $\mathsf{App} \models \mathsf{app}(t_i, x) \approx \Gamma_i[x]_{P_i}$. If $\upsilon \not\rhd^* \tau_i$, then we let $\Gamma_i = t_i$ and $P_i = \emptyset$. Let $\Gamma = \mathsf{c}(\Gamma_1, \ldots, \Gamma_n)$ and $P = \{i.q \mid i \in [1, n], q \in P_i\}$. It is clear that $\Gamma[x]_P \approx \mathsf{c}(\Gamma_i[x]_{P_i})_{i \in [1,n]}$, thus $\mathsf{App} \models \mathsf{app}(t, x) \approx \Gamma[x]_P$. It is easy to check that the second part of the proposition is satisfied in every case. □

Proposition 5. *Let $\Gamma[\bullet]_P$ be a constructor context of profile $\upsilon \to \tau$. If $\tau, \upsilon \in \mathcal{T}_{\mathsf{coind}}$ and $\tau \sim \upsilon$, there exists a term $u : \boxed{\tau}_\upsilon$ such that $\mathsf{App} \models \mathsf{app}(u, x) \approx \Gamma[x]_P$, for all variables $x : \upsilon$. Furthermore, if $\varepsilon \notin P$, the head symbol of u is either cst or a symbol $\boxed{\mathsf{c}}$.*

Proof. The proof is by induction on Γ. If $\varepsilon \in P$, then $\Gamma[x]_P = x$ and $\tau = \upsilon$. Let $u = \mathsf{hole}_\tau$. By definition, we have $\mathsf{App}_2 \models \mathsf{app}(u, x) \approx x$. If $P = \emptyset$, then $\Gamma[x]_P = \Gamma$ and Γ is a term of type τ. Let $u = \mathsf{cst}^\tau_\upsilon(\Gamma)$. We have $\mathsf{App}_1 \models \mathsf{app}^\tau_\upsilon(u, x) \approx \Gamma$. Otherwise, P contains a nonempty position, hence Γ must be of the form $\mathsf{c}(\Gamma_1, \ldots, \Gamma_n)$ for some constructor $\mathsf{c} : \tau_1 \times \cdots \times \tau_n \to \tau$. Furthermore, there exists $i \in [1, n]$ such that $\upsilon \rhd^* \tau_i$. Let $P_i = \{q \mid i.q \in P\}$, for $i \in [1, n]$. Let $i \in [1, n]$ such that $\tau_i \rhd^* \upsilon$. We have $\tau_i \in \mathcal{T}_{\mathsf{coind}}$, hence, by the induction hypothesis, there exists a term u_i such that $\mathsf{App} \models \mathsf{app}(u_i, x) \approx \Gamma_i[x]_{P_i}$. If $\tau_i \not\rhd^* \upsilon$, then all the constructor contexts of profile $\upsilon \to \tau_i$ are constant, thus Γ_i is a term, and we let $u_i = \Gamma_i$. Let $u = \boxed{\mathsf{c}}(u_1, \ldots, u_n)$. By App_3, we deduce $\mathsf{App} \models \mathsf{app}(\boxed{\mathsf{c}}(u_1, \ldots, u_n), x) \approx \mathsf{c}(u_1, \ldots, u_n)$, where $u_i = \Gamma_i[x]_{P_i}$ if $\tau_i \sim \upsilon$ and $u_i = \Gamma_i$ otherwise. If P contains a position of the form $i.q$, then necessarily $\tau_i \sim \upsilon$, thus we have $\Gamma[x]_P = \mathsf{c}(u_1, \ldots, u_n)$. Hence $\mathsf{App} \models \mathsf{app}(u, x) \approx \Gamma[x]_P$. It is clear that the second part of the proposition holds in every case. □

6.3.3 Soundness and Completeness

We prove that the above axioms indeed capture all the intended properties.

Lemma 11 (Soundness of the Axioms)**.** *If interpretation $\mathcal{I}$ satisfies* **Acy** *and* **FP**, *there exists a* sub*-minimal extension $\mathcal{J}$ of $\mathcal{I}$ validating* Sub, NSub, App, Uniq, Cycl, *and* Hole.

Proof. To simplify notations, we assume that $\mathcal{I}$ is a term model (on an extended signature, with an infinite set of new constant symbols denoting elements of the domain). We define the extension $\mathcal{J}$ of $\mathcal{I}$ and check that it fulfills all the desired properties:

- The interpretation of sub is defined in such a way that $\mathcal{J}$ is sub-minimal. By Proposition 3, $\mathcal{J} \models \mathsf{Sub}$. If $\mathcal{J} \not\models \mathsf{NSub}$, then $\mathcal{J} \models \exists \bar{z}, x, \bar{z}'.\ \mathsf{sub}(\mathsf{c}(\bar{z}, x, \bar{z}'), x)$. By definition of the interpretation of sub, this entails that $\mathcal{I} \models \exists \bar{z}, x, \bar{z}', \bar{y}.\ x \approx \Gamma[\mathsf{c}(\bar{z}, x, \bar{z}')]_p$, for some constructor context Γ and position p, where $\bar{y}$ denotes the vector of variables in Γ. This contradicts the fact that $\mathcal{I}$ satisfies condition **Acy**.

- The domain of $\boxed{\tau}_\upsilon$ is the set of ground terms of type $\boxed{\tau}_\upsilon$ defined on the signature (modulo equality on elements of $\mathcal{I}$), with $\mathsf{f}^{\mathcal{J}}(\bar{t}) = \mathsf{f}(\bar{t})$, for any function symbol f of codomain $\boxed{\tau}_\upsilon$ and for any vector of ground terms $\bar{t}$. This set is not empty since it contains $\mathsf{cst}^\tau_\upsilon(t)$ for every ground term $t : \tau$ (and all types are inhabited). Furthermore, $\mathcal{J} \models \mathsf{Hole}$, since two ground terms with distinct heads are necessarily (syntactically) distinct.

- We define the interpretation of $\mathsf{app}(u, v)$ by induction on u, using the equations in App as rewrite rules, from the left to the right, replacing the constructors c in the right-hand side by their interpretation in $\mathcal{I}$. It is clear that app is unambiguously and completely defined, and by definition $\mathcal{J} \models \mathsf{App}$. We check that $\mathcal{J} \models \mathsf{Uniq}$. Let u be a ground term of type $\boxed{\tau}_\upsilon$ distinct from hole_υ. By Proposition 4, there exists a ground constructor context $\Gamma[\bullet]_P$ such that $\mathsf{App} \models \mathsf{app}(u, x) \approx \Gamma[x]_P$, with $\varepsilon \notin P$. Thus $\mathcal{J} \models \mathsf{app}(u, y) \approx y \wedge \mathsf{app}(u, z) \approx z \implies \Gamma[y]_P \approx y \wedge \Gamma[z]_P \approx z$, where y, z are variables some type $\tau \in \mathcal{T}_{\mathsf{coind}}$. Since $\mathcal{I}$ satisfies condition **FP**, this entails that $\mathcal{J} \models \mathsf{app}(u, y) \approx y \wedge \mathsf{app}(u, z) \approx z \implies y \approx z$. Thus $\mathcal{J} \models \mathsf{Uniq}$.

- Let u be a ground term of type $\boxed{\tau}_\upsilon$ with $\tau \in \mathcal{T}_{\mathsf{coind}}$. By Proposition 4, there exists a ground constructor context $\Gamma[\bullet]_P$ such that $\mathsf{App} \models \mathsf{app}(u, v) \approx \Gamma[v]_P$. We define the interpretation of $\mathsf{cyc}_\tau(u)$ as the unique fixpoint of $\Gamma[\bullet]_P$. By definition, we have $\mathcal{J} \models \Gamma[\mathsf{cyc}_\tau(u)]_P \approx \mathsf{cyc}_\tau(u)$, therefore $\mathcal{J} \models \mathsf{app}(u, \mathsf{cyc}_\tau(u)) \approx \mathsf{cyc}_\tau(u)$. Hence $\mathcal{J} \models \mathsf{Cycl}$.

□

Lemma 12 (Completeness of the Axioms)**.** *Any model of the set of axioms* $\{\mathsf{Sub}, \mathsf{NSub}, \mathsf{App}, \mathsf{Uniq}, \mathsf{Cycl}, \mathsf{Hole}\}$ *fulfills* **Acy** *and* **FP**.

Proof. Let $\mathcal{I}$ be a model of $\{\mathsf{Sub}, \mathsf{NSub}, \mathsf{App}, \mathsf{Uniq}, \mathsf{Cycl}, \mathsf{Hole}\}$.

Acyclicity: If $\mathcal{I}$ does not satisfy condition **Acy**, there exists a nonempty constructor context $\Gamma[\bullet]_p$ of type $\tau \in \mathcal{T}_{\mathsf{ind}}$ such that $\mathcal{I} \models \exists x, \bar{z}.\ x \approx \Gamma[x]_p$, where $\bar{z}$ denotes the vector of variables in Γ. Since $p \neq \varepsilon$, p is of the form $q.i$, for some number i, and the subcontext at position q in $\Gamma[\bullet]_P$ is of the form $\mathsf{c}(\bar{u}, \bullet, \bar{v})$, for some constructor c. By definition $\Gamma[\mathsf{c}(\bar{u}, x, \bar{v})]_q = \Gamma[x]_p$; thus $\mathcal{I} \models \exists x, \bar{z}.\ x \approx \Gamma[\mathsf{c}(\bar{u}, x, \bar{v})]_q$. By Proposition 2, we deduce that $\mathcal{I} \models \mathsf{sub}(\mathsf{c}(\bar{u}, x, \bar{v}), x)$, yielding a contradiction with the axiom NSub.

Let $\Gamma[\bullet]_P$ be a nonempty constructor context of profile $\tau \to \tau$. By Proposition 5, there exists a term $t' : \boxed{\tau}_\tau$ such that $\mathsf{App} \models \mathsf{app}(t', x) \approx \Gamma[x]_P$, and we have $\mathcal{I} \models \mathsf{app}(t', x) \approx \Gamma[x]_P$ $(*)$. Note that, by Hole, $\mathcal{I} \models t' \not\approx \mathsf{hole}_\tau$, since the head symbol of t' is cst or $\boxed{\mathsf{c}}$.

- **Existence of fixpoint:** By $(*)$, $\mathcal{I} \models \mathsf{app}(t', \mathsf{cyc}(t')) \approx \Gamma[\mathsf{cyc}(t')]_P$. By Cycl, we deduce that $\mathcal{I} \models \mathsf{cyc}(t') \approx \Gamma[\mathsf{cyc}(t')]_P$; thus $\mathcal{I} \models \exists x.\ \Gamma[x]_P = x$.
- **Uniqueness:** By $(*)$, $\mathcal{I} \models \Gamma[x]_P \approx x \wedge \Gamma[y]_P \approx y \implies \mathsf{app}(t', x) \approx x \wedge \mathsf{app}(t', y) \approx y$. By Uniq, we deduce that $\mathcal{I} \models \Gamma[x]_P \approx x \wedge \Gamma[y]_P \approx y \implies x \approx y$.

□

Lemma 13 (Completeness of the Theory)**.** *Let $\mathcal{T}$ be the theory of free constructors, as defined by the properties* **Exh***,* **Inf***,* **Acy***,* **FP***,* **Dst***, and* **Inj***, with $\mathcal{C}tr_{\mathsf{inj}} = \mathcal{C}tr$ and $\mathsf{c} \bowtie \mathsf{d}$ for all distinct constructors c and d. If S is a first-order sentence in which the only symbols occurring are constructors and equality $(\approx)$, then either $\mathcal{T} \models S$ or $\mathcal{T} \models \neg S$.*

Comon and Lescanne [41] provide a decision procedure for equational formulas over finite and infinite trees, which correspond respectively to freely generated datatypes and codatatypes. It is based on a collection of equivalence-preserving transformation rules for eliminating quantifiers and normalizing the formulas. The set of formulas $T = \{\mathsf{Dist}, \mathsf{Inj}, \mathsf{Exhaust}, \mathsf{Sub}, \mathsf{NSub}, \mathsf{App}, \mathsf{Uniq}, \mathsf{Cycl}, \mathsf{Hole}\}$ forms the axiomatization of a conservative extension of the theory of (co)datatypes. We can thus derive a decision procedure for testing satisfiability of first-order sentences S containing only constructors symbols and the equality predicate in the above theory. By interleaving the steps of two fair saturation procedures of the superposition calculus, the first over $S \cup T$ and the second over $\neg S \cup T$, one of the two attempts is guaranteed to derive a refutation in finite time.

6.4 Inference Rules

As an alternative to the above axiomatization, we propose an extension of the superposition calculus [6] with dedicated rules. Unless otherwise noted, the usual conventions of superposition apply. The standard notion of redundancy is used, with respect to the theory of equality. The notation $[\neg]\, s \approx t$ indicates that the literal is selected by a well-behaved selection function. We let $[\neg]\, s \approx t$ stand for either $s \approx t$ or $s \not\approx t$.

6.4.1 Superposition

We denote by $\mathcal{SP}$ the usual rules of the superposition calculus, called Sup, EqRes, and EqFact below, with a slight relaxation of the application conditions of Sup: Superposition inside the nonmaximal term of an equation is allowed if the head symbol of the maximal term is a constructor. This ensures that in the rewrite system built from saturated clause sets for defining a model, the right-hand side of every rule is irreducible if the head of the left-hand side is a constructor. This property is crucial for the completeness results.

Thus, our superposition rule is as follows:

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

where $\sigma = mgu\{t \stackrel{?}{=} t'\}$, t' is not a variable, and $s\sigma \not\preceq t\sigma$; moreover, $u\sigma \not\preceq v[t']\sigma$ if $[\neg]$ is $\neg$ or if the head symbol of t is not a constructor.

The equality resolution rule is as usual:

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

where $\sigma = mgu\{s \stackrel{?}{=} s'\}$.

Similarly for the equality factoring rule:

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

where $\sigma = mgu\{u \stackrel{?}{=} u'\}$, $t\sigma \not\preceq u$, and $t\sigma \not\preceq s\sigma$.

6.4.2 Infiniteness

The next rule captures infiniteness of (co)datatypes:

$$\frac{\left(\bigvee_{i=1}^{n} x \approx t_i\right) \vee \mathcal{C}}{\mathcal{C}}\ \mathsf{Inf}$$

if x is a variable of a type $\tau \in \mathcal{T}_{\mathsf{ind}} \cup \mathcal{T}_{\mathsf{coind}}$ and does not occur in $\mathcal{C}$ or $t_1, \ldots, t_n$.

Lemma 14 (Soundness of Inf)**.** *Let N be a clause set, and let $\mathcal{I}$ be a model of N satisfying* **Inf**. *If $\mathcal{C}$ is derived from N by* Inf*, then $\mathcal{I} \models \mathcal{C}$.*

Proof. Since $\mathcal{I}$ satisfies **Inf**, the domain of τ is infinite. Therefore, for every valuation η, $\mathcal{I}, \eta \not\models \forall x \bigvee_{i=1}^{n} x \approx t_i$ (since x does not occur in $t_1, \ldots, t_n$), hence $\mathcal{I}, \eta \models \mathcal{C}$ (since x does not occur in $\mathcal{C}$). □

6.4.3 Distinctness

The distinctness property of constructors takes the form of a unary and a binary rule:

$$\frac{\mathsf{c}(\bar{s}) \approx t \vee \mathcal{C}}{\mathcal{C}\sigma}\ \mathsf{Dist}_1$$

if $\sigma = mgu\{t \stackrel{?}{=} \mathsf{d}(\bar{x})\}$, where $\mathsf{c} \bowtie \mathsf{d}$ and $\bar{x}$ is a vector of fresh pairwise distinct variables; and

$$\frac{\mathsf{d}(\bar{t}) \approx u' \vee \mathcal{D} \qquad \mathsf{c}(\bar{s}) \approx u \vee \mathcal{C}}{(\mathcal{C} \vee \mathcal{D})\sigma}\ \mathsf{Dist}_2$$

if $\mathsf{c} \bowtie \mathsf{d}$, $\sigma = mgu\{u \stackrel{?}{=} u'\}$, $\mathsf{c}(\bar{s})\sigma \not\trianglelefteq u\sigma$, and $\mathsf{d}(\bar{t})\sigma \not\trianglelefteq u'\sigma$.

Lemma 15 (Soundness of Dist_1 and Dist_2)**.** *Let N be a clause set, and let $\mathcal{I}$ be a model of N satisfying* **Dst**. *If a clause $\mathcal{C}$ is derived from N by* Dist_1 *or* Dist_2*, then $\mathcal{I} \models \mathcal{C}$.*

Proof. It is easy to check that the conclusion can be derived by superposition and equality resolution from the premises and the axiom Dist. □

6.4.4 Distinctness

The distinctness property of constructors takes the form of a unary and a binary rule:

$$\frac{\mathsf{c}(\bar{s}) \approx t \vee \mathcal{C}}{\mathcal{C}\sigma}\ \mathsf{Dist}_1$$

if $\sigma = mgu\{t \stackrel{?}{=} \mathsf{d}(\bar{x})\}$, where $\mathsf{c} \bowtie \mathsf{d}$ and $\bar{x}$ is a vector of fresh pairwise distinct variables; and

$$\frac{\mathsf{d}(\bar{t}) \approx u' \vee \mathcal{D} \qquad \mathsf{c}(\bar{s}) \approx u \vee \mathcal{C}}{(\mathcal{C} \vee \mathcal{D})\sigma}\ \mathsf{Dist}_2$$

if $\mathsf{c} \bowtie \mathsf{d}$, $\sigma = mgu\{u \stackrel{?}{=} u'\}$, $\mathsf{c}(\bar{s})\sigma \not\trianglerighteq u\sigma$, and $\mathsf{d}(\bar{t})\sigma \not\trianglerighteq u'\sigma$.

Lemma 16 (Soundness of Dist_1 and Dist_2)**.** *Let N be a clause set, and let $\mathcal{I}$ be a model of N satisfying* **Dst***. If a clause $\mathcal{C}$ is derived from N by Dist_1 or Dist_2, then $\mathcal{I} \models \mathcal{C}$.*

Proof. It is easy to check that the conclusion can be derived by superposition and equality resolution from the premises and the axiom Dist. □

6.4.5 Distinctness

The distinctness property of constructors takes the form of a unary and a binary rule:

$$\frac{\mathsf{c}(\bar{s}) \approx t \vee \mathcal{C}}{\mathcal{C}\sigma}\ \mathsf{Dist}_1$$

if $\sigma = mgu\{t \stackrel{?}{=} \mathsf{d}(\bar{x})\}$, where $\mathsf{c} \bowtie \mathsf{d}$ and $\bar{x}$ is a vector of fresh pairwise distinct variables; and

$$\frac{\mathsf{d}(\bar{t}) \approx u' \vee \mathcal{D} \qquad \mathsf{c}(\bar{s}) \approx u \vee \mathcal{C}}{(\mathcal{C} \vee \mathcal{D})\sigma}\ \mathsf{Dist}_2$$

if $\mathsf{c} \bowtie \mathsf{d}$, $\sigma = mgu\{u \stackrel{?}{=} u'\}$, $\mathsf{c}(\bar{s})\sigma \not\trianglerighteq u\sigma$, and $\mathsf{d}(\bar{t})\sigma \not\trianglerighteq u'\sigma$.

Lemma 17 (Soundness of Dist_1 and Dist_2)**.** *Let N be a clause set, and let $\mathcal{I}$ be a model of N satisfying* **Dst***. If a clause $\mathcal{C}$ is derived from N by Dist_1 or Dist_2, then $\mathcal{I} \models \mathcal{C}$.*

Proof. It is easy to check that the conclusion can be derived by superposition and equality resolution from the premises and the axiom Dist. □

Remark 4. If t is not a variable, the premise of Dist_1 is redundant after the rule is applied. Unifying t with $\mathsf{c}(\bar{x})$ can be useful when t is a variable. For example, from the clause $\mathsf{c}(x) \approx x$, we can derive □ by unifying x with $\mathsf{d}(\bar{y})$, where $\mathsf{d} \bowtie \mathsf{c}$.

6.4.6 Injectivity

The injectivity property of constructors is also captured by two rules:

$$\frac{\mathsf{c}(s_1, \ldots, s_m) \approx t \vee \mathcal{C}}{(s_i \approx x_i \vee \mathcal{C})\sigma}\ \mathsf{Inj}_1$$

if $i \in [1, m]$, $\mathsf{c} \in \mathcal{C}tr_{\mathsf{inj}}$, $\sigma = mgu\{t \stackrel{?}{=} \mathsf{c}(x_1, \ldots, x_m)\}$, and $x_1, \ldots, x_m$ are fresh pairwise distinct variables; and

$$\frac{\mathsf{c}(s_1, \ldots, s_m) \approx u' \vee \mathcal{D} \qquad \mathsf{c}(t_1, \ldots, t_m) \approx u \vee \mathcal{C}}{(s_i \approx t_i \vee \mathcal{C} \vee \mathcal{D})\sigma}\ \mathsf{Inj}_2$$

if $i \in [1, m]$, $\mathsf{c} \in \mathcal{C}tr_{\mathsf{inj}}$, $\sigma = mgu\{u \stackrel{?}{=} u'\}$, $u\sigma \not\approx \mathsf{c}(\bar{s})\sigma$, and $u'\sigma \not\approx \mathsf{c}(\bar{t})\sigma$.

Lemma 18 (Soundness of Inj_1 and Inj_2)**.** *Let N be a clause set, and let $\mathcal{I}$ be a model of N satisfying* **Inj**. *If a clause $\mathcal{C}$ is derived from N by Inj_1 or Inj_2, then $\mathcal{I} \models \mathcal{C}$.*

Proof. It is easy to check that the conclusion can be derived by superposition and equality resolution from the premises and the axiom Inj. □

Remark 5. If Inj_1 is applied on every argument $i \in [1, m]$ and t is not a variable, the premise becomes redundant and can be removed. Unifying t with the term $\mathsf{c}(x_1, \ldots, x_m)$ is useful when t is a variable. For example, given the clause $\mathsf{c}(x, \mathsf{a}) \approx x$, we can derive $\mathsf{a} \approx x_2$ by Inj_1, from which □ can be derived by Inf.

6.4.7 Acyclicity

The acyclicity rule attempts to detect constraints that would force a datatype value to be cyclic. The simplest example is a clause of the form $\Gamma[s] \approx s$, where Γ is a nonempty constructor context. More generally, the clauses

$$s_1 \approx \Gamma_1[s_2] \qquad s_2 \approx \Gamma_2[s_3] \qquad \cdots \qquad s_{n-1} \approx \Gamma_{n-1}[s_n] \qquad s_n \approx \Gamma_n[s_1]$$

entail a constraint $s_1 \approx \Gamma_1[\Gamma_2[\cdots[\Gamma_{n-1}[\Gamma_n[s_1]]]\cdots]]$. Moreover, the rule must support variables and nonunit clauses, and it should be finitely branching if we want to incorporate it in saturation-based provers—i.e., the set of clauses derivable from a given finite set of premises by a single rule should be finite. Finally, clauses of the form $\Gamma[x] \approx s \vee \mathcal{C}$, where x occurs in $\mathcal{C}$, are problematic, because there are infinitely many instantiations of x that can result in a cyclic constraint: s, $\mathsf{c}(s)$, $\mathsf{c}(\mathsf{c}(s))$, etc. To cope with all these subtleties, we first need to develop a considerable theoretical apparatus before we can even state the rule.

Definition 14. A *chain* built on a nonempty sequence of (variable-disjoint) clauses $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ under condition $\mathcal{D}$ is a sequence $(t_1, \ldots, t_{n+1})$ of terms satisfying the following conditions:

1. for every $i \in [1,n]$, $\mathcal{C}_i$ is of the form $s_i \approx \Gamma_i[s'_{i+1}]_{p_i} \vee \mathcal{C}'_i$, where p_i is a nonempty constructor position in Γ_i;

2. there exists a substitution σ such that either (a) σ is an mgu of $E = \{s'_i \stackrel{?}{=} s_i \mid i \in [2,n]\}$ or (b) σ is an mgu of $\{s'_{n+1} \stackrel{?}{=} s_1\} \cup E$;

3. $t_i = s_i\sigma$ for $i \in [1,n]$ and $t_{n+1} = s'_{n+1}\sigma$;

4. $\mathcal{D} = \bigvee_{i=1}^{n} \mathcal{C}'_i\sigma$;

5. $\mathrm{type}(t_1) \sim \cdots \sim \mathrm{type}(t_{n+1})$;

6. $(\Gamma_i[s'_{i+1}]_{p_i} \approx s_i)\sigma$ is strictly maximal in $\mathcal{C}_i\sigma$, and no literal is selected in $\mathcal{C}_i\sigma$;

7. $s_i\sigma \not\succ \Gamma_i[s'_{i+1}]_{p_1}\sigma$, for $i \in [1,n]$;

8. for every $i \in [2,n]$, s'_i is not a variable.

The expression $\Gamma_1[\cdots[\Gamma_n[\bullet]_{p_n}]\cdots]_{p_1}\sigma$ is the chain's constructor context, σ is its mgu, and $p_1.\cdots.p_n$ is its constructor position. If $t_1 = t_{n+1}$, the sequence is called a *cycle*. A chain is *direct* if $t_i \neq t_j$ for all $i,j \in [1,n+1]$ with $i \neq j$ and $\{i,j\} \neq \{1,n+1\}$, and *variable-ended* if s'_{n+1} is a variable.

Remark 6. Conditions 5 to 8 are optional. They help prune the search space.

Remark 7. It is tempting to replace the condition $s_i\sigma \not\succ \Gamma_i[s'_{i+1}]_{p_1}\sigma$ for $i > 1$ by the stronger condition $\Gamma_i[s'_{i+1}]_{p_1}\sigma \succ s_i\sigma$, because if the latter condition does not hold, the Superposition rule applies into s'_i generating a clause $(\Gamma_{i-1}[\Gamma_i[s'_{i+1}]_{p_i}]_{p_{i-1}} \approx s_{i-1} \vee \mathcal{C}'_{i-1} \vee \mathcal{C}'_i)\theta$, where σ is an instance of θ, and we could construct a smaller chain with the same first and last terms without using $\mathcal{C}_i$. But this is not compatible with the redundancy criteria. For example, given $\{\mathsf{f}(x,1) \approx \mathsf{c}(\mathsf{g}(x)), \mathsf{g}(x) \approx \mathsf{c}(\mathsf{f}(x,x)), \mathsf{f}(0,1) \approx \mathsf{c}(\mathsf{c}(\mathsf{f}(0,0)))\}$ with $\mathsf{g}(0) \succ \mathsf{c}(\mathsf{f}(0,0))$, $\mathsf{c}(\mathsf{f}(1,1)) \succ \mathsf{g}(1)$, no chain from $\mathsf{f}(1,1)$ to $\mathsf{f}(1,1)$ could be derived (with the strengthened condition), because $\mathsf{f}(x,1) \approx \mathsf{c}(\mathsf{c}(\mathsf{f}(x,x)))$ is redundant (assuming we have $x \approx 0 \vee x \approx 1$). Indeed, $\mathsf{f}(0,1) \approx \mathsf{c}(\mathsf{c}(\mathsf{f}(0,0)))$ occurs in the set, and $\mathsf{f}(1,1) \approx \mathsf{c}(\mathsf{c}(\mathsf{f}(1,1)))$ can easily be derived from smaller instances by substitutivity.

We state some basic properties of chains:

Proposition 6. *Let $(t_1,\ldots,t_{n+1})$ be a chain, and let $i,j \in [1,n+1]$, with $i \leq j$. With the notations of Definition 14, we have*

$$\mathcal{C}_1,\ldots,\mathcal{C}_n \models \mathcal{D} \vee (t_i \approx \Gamma_i[\cdots[\Gamma_{j-1}[t_j]_{p_{j-1}}]\cdots]_{p_i})\sigma$$

In particular, if $t_{n+1} = t_1$, *then*

$$\mathcal{C}_1, \ldots, \mathcal{C}_n \models \mathcal{D} \vee (t_j \approx \Gamma_j[\cdots[\Gamma_n[\Gamma_1[\cdots[\Gamma_{i-1}[t_i]_{p_{i-1}}]\cdots]_{p_1}]_{p_n}]\cdots]_{p_j})\sigma$$

Proof. The proof follows immediately from the definition, by transitivity and substitutivity of $\approx$. □

Proposition 7. *Let* $\bar{t} = (t_1, \ldots, t_n, t_1)$ *be a cycle. For any number* k, *the sequence* $\bar{s} = (t_{1+k}, \ldots, t_{n+k}, t_{1+k})$, *with* $t_i := t_{i-n}$ *if* $i > n$, *is a cycle.*

Proof. Let $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ be the sequence of clauses forming $\bar{t}$. It is easy to check that $\bar{s}$ is a cycle formed by $(\mathcal{C}_{1+k}, \ldots, \mathcal{C}_{n+k})$ with $\mathcal{C}_i := \mathcal{C}_{i-n}$ if $i > n$, as conditions of Definition 14 are invariant by circular permutation if $t_1 = t_{n+1}$. □

Definition 15. A chain $(t_1, \ldots, t_{n+1})$ built on a sequence $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ is an *extension* of an acyclic chain $(s_1, \ldots, s_{m+1})$ if $n \geq m$, the latter chain is built on $(\mathcal{C}_1, \ldots, \mathcal{C}_m)$, and the same (oriented) literals and positions are considered in each clause $\mathcal{C}_i$ in both chains.

Proposition 8. *Let* $\bar{s}$ *be a chain of constructor context* $\Gamma[\bullet]_p$ *and of mgu* σ, *and let* $\bar{t}$ *be an extension of* $\bar{s}$. *Then the mgu of* $\bar{t}$ *is of the form* $\sigma\theta$, $\bar{t}$ *is of the form* $(\bar{s}\theta, \bar{u})$, *and the constructor context of* $\bar{t}$ *is of the form* $\Gamma\theta[\Delta[\bullet]_q]_p$.

Proof. It is clear that the unification problem associated with $\bar{s}$ (as defined in Definition 14, condition 2) is included into that of $\bar{t}$. Hence the mgu of $\bar{t}$ is an instance of that of $\bar{s}$. Then the proof follows immediately from the definitions. The case where $\bar{u}$ is empty occurs when $\bar{s} = \bar{t}$, or when $m = n$ and $\bar{t}$ is a cycle (i.e., $\bar{t}$ is built on the same clauses as $\bar{s}$, but it satisfies condition 2(b) of Definition 14 instead of 2(a)). □

Since chains can be arbitrarily long, we need to impose some additional conditions to prune them and ensure that the rules are finitely branching. Let **Keep** be a property of chains that fulfills the following requirements:

(i) if a chain $\bar{t}$ does not satisfy **Keep**, no extension of $\bar{t}$ satisfies **Keep**;

(ii) for every finite clause set N, the set of chains built on a sequence of renamings of clauses in N and satisfying **Keep** is finite;

(iii) for every cycle $(t_1, \ldots, t_n, t_1)$, there exists a chain $(s_1, \ldots, s_m)$ with $m \leqslant n$ that satisfies **Keep** and such that for some k, the cycle $(t_{1+k}, \ldots, t_{n+k}, t_{1+k})$ (with $t_i := t_{i-n}$ if $i > n$) is an extension of $(s_1, \ldots, s_m)$.

For example, **Keep** can be defined as the set of chains built on clauses $\mathcal{C}_i$ that are pairwise distinct modulo renaming and such that $\mathcal{C}_1$ is the most recently processed clause. This is the definition we use in our description of the extended saturation loop (Section 6.6) and in the implementation in Vampire.

Remark 8. Condition (i) is essential in practice, to ensure that the chains can be incrementally constructed in an efficient way, because it ensures that the construction can be stopped when a prefix not satisfying **Keep** is obtained. Condition (ii) is not used in the following, but it ensures that the rule is finitely branching. Condition (iii) is essential for completeness.

Definition 16. A chain of length n is *eligible* if it is variable-ended and $n = 1$, or if it is not variable-ended, it satisfies **Keep**, and either it is a cycle or there exists an extension of length $n + 1$ that does not satisfy **Keep**.

Remark 9. The conditions on eligible chains are the strongest ones preserving completeness, but they are not necessary for soundness. They may thus freely be relaxed if this yields a more efficient procedure.

The acyclicity rule follows:

$$\frac{\mathcal{C}_1 \quad \cdots \quad \mathcal{C}_n}{\mathcal{D} \vee \mathcal{E}}\ \mathsf{Acycl}$$

if there exists a direct, eligible chain $(t_1, \ldots, t_{n+1})$ built on $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ under condition $\mathcal{D}$ and either $t_1 = t_{n+1}$ and $\mathcal{E} = \emptyset$ or $t_1 \neq t_{n+1}$ and $\mathcal{E} = \neg\, \mathsf{sub}(t_1, t_{n+1})$

Intuitively, the existence of the chain guarantees (if $\mathcal{D}$ is false) that there exists a nonempty constructor context $\Gamma[\bullet]_p$ such that $t_1 \approx \Gamma[t_{n+1}]_p$ holds. If $t_1 = t_{n+1}$, this contradicts acyclicity. Otherwise, we deduce that t_1 cannot occur at a constructor position inside the constructor term corresponding to t_{n+1}; hence $\mathsf{sub}(t_1, t_{n+1})$ is false.

Since the interpretation of sub is not completely axiomatized, the Acycl rule is not sound in general, in the sense that the derived clauses are not logical consequences of the premises, even if the considered interpretation satisfies condition **Acy**. However, it is sound if one restricts oneself to interpretations that are sub-minimal. By Lemma 11, we know that this restriction does not involve any loss of generality, because a sub-minimal model exists for every satisfiable clause set (with no occurrences of sub).

Lemma 19 (Soundness of Acycl)**.** *Let N be a clause set, and let $\mathcal{I}$ be a* sub*-minimal model of N satisfying* **Acy***. If $\mathcal{C}$ is derived from N by* Acycl*, then $\mathcal{I} \models \mathcal{C}$.*

Proof. Let η be a valuation such that $\mathcal{I}, \eta \not\models \mathcal{D} \vee \mathcal{E}$ (with the notations of the rule). By Proposition 6, we have $\mathcal{I}, \eta \models t_1 \approx \Gamma[t_{n+1}]_p$ for some nonempty constructor context. If $t_1 = t_{n+1}$, this yields an immediate contradiction with the hypothesis that $\mathcal{I}$ satisfies **Acy**. Otherwise, we must have $\mathcal{E} = \neg\,\mathsf{sub}(t_1, t_{n+1})$, hence $\mathcal{I}, \eta \models \mathsf{sub}(t_1, t_{n+1})$. Because $\mathcal{I}$ is sub-minimal, this entails that there exists a constructor context $\Delta[\bullet]_q$ such that $\mathcal{I}, \eta \models \exists \bar{z}.\ t_{n+1} \approx \Delta[t_1]_q$; thus $\mathcal{I}, \eta \models \exists \bar{z}.\ t_{n+1} \approx \Delta[\Gamma[t_{n+1}]_p]_q$. Again, this contradicts the hypothesis that $\mathcal{I}$ satisfies **Acy**. □

6.4.8 Uniqueness of Fixpoints

The uniqueness rule also depends on the notion of chain:

$$\frac{\mathcal{C}_1 \quad \cdots \quad \mathcal{C}_n}{\mathcal{D} \vee \left(\bigvee_{p \in P} u|_p \not\approx \mathsf{app}(s_p, t_1)\right) \vee u' \not\approx z \vee z \approx t_1}\ \mathsf{Uniq}$$

if there exists an eligible chain $(t_1, \ldots, t_{n+1})$ of constructor context $\Gamma[\bullet]_q$ built on $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ under condition $\mathcal{D}$ and the following requirements are met:

1. $u = \Gamma[t_{n+1}]_q$;
2. P is the set of prefix-minimal positions p of some type $\tau \sim \mathrm{type}(t_1)$ in u with $p \not< q$;
3. for every $p \in P$, s_p is a fresh variable of type $\boxed{\upsilon}_\tau$, where υ, τ are the types of $u|_p$ and t_1, respectively;
4. u' is obtained from u by replacing all terms at a position $p \in P$ by $\mathsf{app}(s_p, z)$.

Intuitively, the existence of the chain ensures (if $\mathcal{D}$ is false) that $t_1 \approx \Gamma[t_{n+1}]_q$. If $t_1 = t_{n+1}$, we could derive $y \not\approx \Gamma[y]_q \vee y \approx t_1$ by uniqueness. However, this would not be sufficient for completeness. First, t_1 may be distinct from t_{n+1}, but we may have $t_{n+1} = \Delta[t_1]_Q$, for some constructor context Δ, in which case we should derive $y \not\approx \Gamma[\Delta[y]_Q]_q \vee y \approx t_1$ instead. Second, t_1 may also occur at other positions in Γ (not $<$-comparable with q). To capture all these cases using a finitely branching rule (i.e., without having to "guess" constructor contexts), we introduce new variables s_p whose purpose is to denote the context Γ_p such that $\Gamma_p[t_1] = u|_p$. (If t_1 does not occur inside $u|_p$, then Γ_p is constant.)

Example 5. From the clause $\mathsf{a} \approx \mathsf{c}(\mathsf{b}, x)$, using the chain (a, x), with the constructor context $\mathsf{c}(\mathsf{b}, \bullet)$, we derive

$$\mathsf{b} \not\approx \mathsf{app}(x_1, \mathsf{a}) \vee x \not\approx \mathsf{app}(x_2, \mathsf{a}) \vee z \not\approx \mathsf{c}(\mathsf{app}(x_1, z), \mathsf{app}(x_2, z)) \vee z \approx \mathsf{a}$$

Then $u = \mathsf{c}(\mathsf{b}, x)$ and $P = \{1, 2\}$.

From the clauses $\mathsf{a} \approx \mathsf{c}(\mathsf{b}, \mathsf{a})$ and $\mathsf{b} \approx \mathsf{d}(\mathsf{a}, \mathsf{a})$, using the chain $(\mathsf{a}, \mathsf{b}, \mathsf{a})$, with the constructor context $\mathsf{c}(\mathsf{d}(\mathsf{a}, \bullet), \mathsf{a})$, we derive

$$\begin{aligned}&\mathsf{a} \not\approx \mathsf{app}(x_{1.1}, \mathsf{a}) \vee \mathsf{a} \not\approx \mathsf{app}(x_{1.2}, \mathsf{a}) \vee \mathsf{a} \not\approx \mathsf{app}(x_2, \mathsf{a})\\ &\vee z \not\approx \mathsf{c}(\mathsf{c}(\mathsf{app}(x_{1.1}, z), \mathsf{app}(x_{1.2}, z)), \mathsf{app}(x_2, z)) \vee z \approx \mathsf{a}\end{aligned}$$

In this case, $u = \mathsf{c}(\mathsf{d}(\mathsf{a}, \mathsf{a}), x)$ and $P = \{1.1, 1.2, 2\}$.

Lemma 20 (Soundness of Uniq)**.** *Let N be a clause set, and let $\mathcal{I}$ be a model of $N \cup \{\mathsf{App}, \mathsf{Hole}\}$ satisfying* **FP**. *If $\mathcal{C}$ is derived from N by* Uniq*, then $\mathcal{I} \models \mathcal{C}$.*

Proof. Note that since $\mathcal{I}$ satisfies **FP**, necessarily $\mathcal{I} \models \mathsf{Uniq}$. Let η be a valuation. By Proposition 6, we have $\mathcal{I}, \eta \models \mathcal{D} \vee t_1 \approx u$. Let $\boxed{u}$ be the term obtained from u by replacing each constructor c by $\boxed{\mathsf{c}}_\tau$ and each term at a position $p \in P$ by s_p. It is straightforward to verify (by induction on u) that $\mathsf{App} \models \mathsf{app}(\boxed{u}, z) \approx u'$ and that

$$\mathsf{App} \models \big(\bigwedge\nolimits_{p \in P} u|_p \approx \mathsf{app}(s_p, t_1)\big) \implies u \approx \mathsf{app}(\boxed{u}, t_1)$$

Consequently, $\mathcal{I}, \eta \models \mathcal{D} \vee \bigvee_{p \in P} u|_p \not\approx \mathsf{app}(s_p, t_1) \vee t_1 \approx \mathsf{app}(\boxed{u}, t_1)$. Furthermore, we have $\mathsf{Hole} \models \boxed{u} \not\approx \mathsf{hole}$, because by definition of chains $\Gamma[\bullet]_P$ is not empty. Since

$$\mathsf{Uniq} \models \boxed{u} \approx \mathsf{hole} \vee t_1 \not\approx \mathsf{app}(\boxed{u}, t_1) \vee z \not\approx \mathsf{app}(\boxed{u}, z) \vee z \approx t_1$$

we deduce that

$$\mathcal{I}, \eta \models \mathcal{D} \vee \bigvee\nolimits_{p \in P} u|_p \not\approx \mathsf{app}(s_p, t_1) \vee u' \not\approx z \vee z \approx t_1$$

□

We also introduce the following optional simplification rule:

$$\frac{\Gamma_n[\cdots[\Gamma_1[s']_P]\cdots]_P \approx s \vee \mathcal{C}}{(\Gamma_1[s]_P \approx s \vee \mathcal{C})\sigma}\ \mathsf{Compr}$$

where s and s' are terms of the same type $\tau \in \mathcal{T}_{\mathsf{coind}}$ and P is a nonempty set of constructor positions in Γ_i, for $i \in [1, n]$, such that $\varepsilon \notin P$, and $\sigma = mgu\{s \stackrel{?}{=} s', \Gamma_1 \stackrel{?}{=} \cdots \stackrel{?}{=} \Gamma_n\}$.

Proposition 9 (Soundness of Compr)**.** *Let N be a clause set, and let $\mathcal{I}$ be a model of N satisfying* **FP**. *If $\mathcal{D}$ is derived from N by* Compr*, then $\mathcal{I} \models \mathcal{D}$.*

Proof. Let $\mathcal{D} = (\Gamma_1[s]_P \approx s \vee \mathcal{C})\sigma$, and let η be a valuation such that $\mathcal{I}, \eta \not\models \mathcal{C}\sigma$. Since $\mathcal{I} \models N$, we have $\mathcal{I}, \eta \models (\Gamma_n[\cdots[\Gamma_1[s']_P]\cdots]_P \approx s)\sigma$, i.e., $\mathcal{I}, \eta \models (\Gamma_n[\cdots[\Gamma_1[s]_P]\cdots]_P \approx s)\sigma$. Then, since $\mathcal{I}$ satisfies **FP** (existence of fixpoints), $\mathcal{I}, \eta \models \exists x.\ x \approx (\Gamma_1\sigma)[x]_P$; thus there exists an extension η' of η such that $\mathcal{I}, \eta' \models x \approx (\Gamma_1\sigma)[x]_P$ for some fresh variable x. Then, since $\Gamma_1\sigma = \cdots = \Gamma_n\sigma$, we have $\mathcal{I}, \eta' \models x \approx \Gamma_n[\cdots[\Gamma_1[x]_P]\cdots]_P$, and since $\mathcal{I}$ satisfies **FP** (uniqueness) we deduce that $\mathcal{I}, \eta' \models s\sigma \approx x$ thus $\mathcal{I}, \eta \models (\Gamma_1[s]_P \approx s)\sigma$. □

6.5 Refutational Completeness

We establish the refutational completeness of the calculus presented in Section 6.4. This result ensures that the axioms for distinctness, injectivity, and acyclicity (NSub) may be omitted. The axiom Uniq may also be omitted in some cases, formally defined below. The axiom Sub is still needed since it is used in the completeness proof for Acycl.

If $N \not\ni \square$ is a clause set saturated under $\mathcal{SP}$, then R_N denotes the set of rewrite rules constructed as usual from N and $\to_{R_N}$ denotes the (one-step) reduction relation. We refer to the literature [6, 99] for details about the construction of R_N. The notation $\mathcal{M}_N$ denotes the model of N defined by the congruence $\overset{*}{\leftrightarrow}_{R_N}$ on ground terms.

We first establish some results about the form of the rules in R_N.

Proposition 10. *Let N be a clause set saturated under $\mathcal{SP}$ and* Inf*. Let $u \approx v \vee \mathcal{C} \in N$, and let θ be a substitution such that $u\theta \succ v\theta$, $(u \approx v)\theta \succ \mathcal{C}\theta$, and $\mathcal{M}_N \not\models \mathcal{C}\theta$. If $\mathrm{type}(u) \in \mathcal{T}_{\mathsf{ind}} \cup \mathcal{T}_{\mathsf{coind}}$, then u is not a variable.*

Proof. If u is a variable, then due to the order conditions u cannot occur in the scope of a function symbol in $\mathcal{C}$ or v, or in a negative literal of $\mathcal{C}$, hence it occurs only at root positions in equations. Consequently, $u \approx v \vee \mathcal{C}$ is of the form $\bigvee_{i=1}^{n} u \approx t_i \vee \mathcal{C}'$, where u does not occur in $\mathcal{C}'$ or $t_1, \ldots, t_n$ (with $v \in \{t_1, \ldots, t_n\}$). Then Inf applies and derives $\mathcal{C}'$. Since N is saturated under Inf, we deduce that $\mathcal{M}_N \models \mathcal{C}' \models \mathcal{C}\theta$, which contradicts the hypotheses. □

Corollary 4.1. Let N be a clause set saturated under $\mathcal{SP}$ and Inf. For every rule $\mathsf{c}(\bar{t}) \to_{R_N} s$ in R_N, where c is a constructor, s is R_N-irreducible.

Proof. Assume that s is reducible in R_N. This means that there exist a subterm u at position p of s and a rule $u \to_{R_N} v$ in R_N. Consequently, there exist two clauses $\mathcal{C} = t' \approx s' \vee \mathcal{C}'$ and $\mathcal{D} = u' \approx v' \vee \mathcal{D}'$, and two

substitutions σ and θ such that $t'\theta = \mathsf{c}(\bar{t})$, $u'\sigma = u$, $s'\theta = s$, $v'\sigma = v$, and $\mathcal{M}_N \not\models \mathcal{C}'\theta \vee \mathcal{D}'\sigma$. By definition of R_N, $x\theta$ is R_N-irreducible, for every $x \in \mathrm{dom}(\theta)$, consequently p is a non-variable position in s'. By Proposition 10, t' is not a variable, hence the head symbol of t' is c. Therefore, superposition into s' is allowed in the relaxed calculus, and the inference yields a clause $(t' \approx s'[v']_p \vee \mathcal{C}' \vee \mathcal{D}')\eta$, where $\eta = mgu\{s'|_p \stackrel{?}{=} u'\}$. Since $\theta\sigma$ is an instance of η, we deduce that there exist ground clauses $\mathcal{E}_1, \ldots, \mathcal{E}_n$ that are instances of clauses in N such that $\mathsf{c}(\bar{t}) \approx s[v]_p \vee \mathcal{C}'\theta \vee \mathcal{D}'\sigma \succeq \mathcal{E}_i$ for $i \in [1, n]$ and $\mathcal{E}_1, \ldots, \mathcal{E}_n \models \mathsf{c}(\bar{t}) \approx s[v]_p \vee \mathcal{C}'\theta \vee \mathcal{D}'\sigma$. Let R'_N be the rules added before $\mathsf{c}(\bar{t}) \to_{R_N} s$ in R_N and let $\mathcal{M}'$ be the corresponding interpretation. By construction of the model, $\mathsf{c}(\bar{t})$ is R'_N-irreducible and $\mathcal{M}' \not\models \mathcal{C}'\theta \vee \mathcal{D}'\sigma$, moreover, since $\mathcal{C}\theta \succ \mathcal{E}_i$, we have $\mathcal{M}' \models \mathcal{E}_1, \ldots, \mathcal{E}_n$ thus $\mathcal{M}' \models \mathsf{c}(\bar{t}) \approx s[v]_p$, a contradiction. □

Lemma 21 (Infiniteness)**.** *Let N be a clause set saturated under $\mathcal{SP}$ and* Inf*. If $\square \notin N$, then $\mathcal{M}_N$ satisfies* **Inf***.*

Proof. Let $\tau \in \mathcal{T}_{\mathsf{ind}} \cup \mathcal{T}_{\mathsf{coind}}$. We assume, without loss of generality, that the model is constructed over a signature that contains at least two (non-constructor) symbols $\mathsf{a} : \tau$ and $\mathsf{f} : \tau \to \tau$, not occurring in N. By Proposition 10, all the rules in R_N are of the form $\mathsf{g}(\bar{t}) \to_{R_N} s$, where g occurs in N. Thus $\mathsf{f}^n(\mathsf{a})$ is R_N-irreducible for every natural number n, and the domain of τ is infinite. □

Lemma 22 (Distinctness)**.** *Let N be a clause set saturated under $\mathcal{SP}$,* Dist_1*,* Dist_2*, and* Inf*. For all ground terms $a = \mathsf{c}(\bar{a})$ and $b = \mathsf{d}(\bar{b})$ such that $\mathsf{c} \bowtie \mathsf{d}$, we have $a \not\stackrel{*}{\leftrightarrow}_{R_N} b$.*

Proof. Assume $a \stackrel{*}{\leftrightarrow}_{R_N} b$. Without loss of generality, we can assume that $a \succ b$ and that $\bar{a}$ and $\bar{b}$ are R_N-irreducible. The proof proceeds by case analysis over the reducibility of a and b.

- a is irreducible. Since $a \succ b$, it cannot be the case that $a \stackrel{*}{\leftrightarrow}_{R_N} b$.
- b is irreducible and $a \stackrel{+}{\to}_{R_N} b$. Then R_N contains a rule $a \to_{R_N} a'$, with $a' \to^*_{R_N} b$. By Corollary 4.1, we know that a' is R_N-irreducible, therefore we must have $a' = b$. There exist a clause $\mathcal{C} = u \approx v \vee \mathcal{C}'$ and a substitution θ, with $\mathcal{M}_N \not\models \mathcal{C}'\theta$, $u\theta = a$, and $v\theta = b$. By Proposition 10, u cannot be a variable, hence its head symbol is c. Consequently, there is an inference Dist_1 taking $\mathcal{C}$ for premise, and deriving a clause $\mathcal{C}'\sigma$, with $\sigma = mgu\{v \stackrel{?}{=} \mathsf{d}(\bar{x})\}$. It is clear that θ is an instance of σ (more exactly of the restriction of σ to the variables of $\mathcal{C}$). Thus $\mathcal{M}_N \models \mathcal{C}'\sigma \models \mathcal{C}'\theta$, leading to a contradiction.

- There exists a term c such that $a \xrightarrow{+}_{R_N} c$ and $b \xrightarrow{+}_{R_N} c$. Then R_N must contain two rules of the form $a \to a'$ and $b \to b'$, corresponding to two clauses $\mathcal{C}$ and $\mathcal{D}$ of the form $u \approx u' \vee \mathcal{C}'$ and $v \approx v' \vee \mathcal{D}'$ with $\mathcal{M}_N \not\models \mathcal{C}\theta \vee \mathcal{C}'\theta'$, $a' \xrightarrow{*}_{R_N} c$, $b' \xrightarrow{*}_{R_N} c$, $u\theta = a$, $v\theta' = b$, $u'\theta = a'$, and $v'\theta' = b'$. By Proposition 10, u and v cannot be variables, hence there head symbols must be c and d respectively. By Corollary 4.1, a' and b' are R_N-irreducible, and therefore $a' = b' = c$. The rule Dist_2 can be applied to clauses $\mathcal{C}$ and $\mathcal{D}$, yielding a clause $(\mathcal{C} \vee \mathcal{C}')\sigma$, where $\sigma = mgu\{u' \stackrel{?}{=} v'\}$. Then $\theta\theta'$ is an instance of σ; thus $\mathcal{M}_N \models (\mathcal{C} \vee \mathcal{C}')\sigma \models \mathcal{C}\theta \vee \mathcal{C}'\theta'$, a contradiction.

□

Lemma 23 (Injectivity)**.** *Let N be a clause set saturated under $\mathcal{SP}$, Inf, Inj_1, and Inj_2. For all ground terms $a = \mathsf{c}(a_1, \ldots, a_n)$ and $b = \mathsf{c}(b_1, \ldots, b_n)$ with $\mathsf{c} \in \mathcal{C}tr_{\mathsf{inj}}$ and such that $a_i \not\xrightarrow{*}_{R_N} b_i$ for some $i \in [1, n]$, we have $a \not\xrightarrow{*}_{R_N} b$.*

Proof. Assume $a \xleftrightarrow{*}_{R_N} b$. Without loss of generality, we can assume that $a \succ b$ and that $a_1, \ldots, a_n, b_1, \ldots, b_n$ are R_N-irreducible. The proof is similar to that of Lemma 22.

- a is irreducible. Since $a \succ b$, it cannot be the case that $a \xleftrightarrow{*}_{R_N} b$.
- b is irreducible and $a \xrightarrow{+}_{R_N} b$. Then R_N contains a rule $a \to_{R_N} a'$, with $a' \to^*_{R_N} b$. By Corollary 4.1, we know that a' is irreducible, therefore we must have $a' = b$. There exists a clause $\mathcal{C} = u \approx v \vee \mathcal{C}'$, with $\mathcal{M}_N \not\models \mathcal{C}'\theta$, $u\theta = a$, and $v\theta = b$. By Proposition 10, u cannot be a variable; thus u is of the form $\mathsf{c}(u_1, \ldots, u_n)$. Then there is an inference Dist_1 taking $\mathcal{C}$ for premise, and deriving a clause $(u_i \approx x_i \vee \mathcal{C}')\sigma$, with σ is an mgu of v and $\mathsf{d}(x_1, \ldots, x_n)$. The substitution $\{x_i \mapsto b_i\}\theta$ is an instance of σ; thus $\mathcal{M}_N \models (u_i \approx x_i \vee \mathcal{C}')\sigma \models (u_i \approx x_i \vee \mathcal{C}')\{x_i \mapsto b_i\}\theta = a_i \approx b_i \vee \mathcal{C}'\theta$, leading to a contradiction.
- There exists a term c such that $a \xrightarrow{+}_{R_N} c$ and $b \xrightarrow{+}_{R_N} c$. Then R_N must contain two rules of the form $a \to a'$ and $b \to b'$, corresponding to two clauses $\mathcal{C}$ and $\mathcal{D}$ of the form $u \approx u' \vee \mathcal{C}'$ and $v \approx v' \vee \mathcal{D}'$ with $\mathcal{M}_N \not\models \mathcal{C}\theta \vee \mathcal{C}'\theta'$, $a' \xrightarrow{*}_{R_N} c$, $b' \xrightarrow{*}_{R_N} c$, $u\theta = a$, $v\theta' = b$, $u'\theta = a'$, and $v'\theta' = b'$. By Proposition 10, u and v cannot be variables, hence they must be of the form $\mathsf{c}(u_1, \ldots, u_n)$ and $\mathsf{c}(v_1, \ldots, v_n)$ respectively. By Corollary 4.1, a' and b' are R_N-irreducible, and therefore $a' = b' = c$. The rule Dist_2 can be applied to clauses $\mathcal{C}$ and $\mathcal{D}$, yielding a clause of the form $(u_i \approx v_i \vee \mathcal{C} \vee \mathcal{C}')\sigma$, where $\theta\theta'$ is an instance of σ. Thus $\mathcal{M}_N \models (u_i \approx v_i \vee \mathcal{C} \vee \mathcal{C}')\sigma \models a_i \approx b_i \vee \mathcal{C}\theta \vee \mathcal{C}'\theta'$, a contradiction.

□

The completeness proof for acyclicity requires further definitions and results.

Definition 17. Let $\mathcal{I}$ be an interpretation and t be a term. A constructor context $\Gamma[\bullet]_p$ is a *minimal cyclicity witness* for t and $\mathcal{I}$ if it is of the same type as t, p is a position of the same type as t in Γ, $\mathcal{I} \models t \approx \Gamma[t]_p$, and $|q| \geq |p|$ for every position $q \neq \varepsilon$ and constructor context $\Delta[\bullet]_q$ such that $\mathcal{I} \models t \approx \Delta[t]_q$.

Proposition 11. *Let $(t_1, \ldots, t_n, t_1)$ be a cycle of constructor context $\Gamma[\bullet]_p$ for a clause set N under condition $\mathcal{D}$. If $\mathcal{I} \models N \cup \{\neg\, \mathcal{D}\sigma\}$, and $\Gamma[\bullet]_p$ is a minimal cyclicity witness for $t_1\sigma$ and $\mathcal{I}$, then $(t_1, \ldots, t_n, t_1)$ is direct.*

Proof. Assume that $(t_1, \ldots, t_n, t_1)$ is not direct. By definition, there exist $i, j \in [1, n]$ such that $i < j$ and $t_i = t_j$. By Proposition 6, since $\mathcal{I} \not\models \mathcal{D}\sigma$ we have $\mathcal{I} \models (t_1 \approx \Gamma[t_1]_{p_1.\cdots.p_n})\sigma$ for some positions $p_1, \ldots, p_n$, with $p = p_1.\cdots.p_n$. Furthermore, again by Proposition 6, we also have $\mathcal{I} \models (t_1 \approx \Gamma_i[t_i]_{p_1\cdots.p_{i-1}})\sigma$ and $\mathcal{I} \models (t_j \approx \Gamma_j[t_n]_{p_j\cdots.p_{n-1}})\sigma$, for some constructor contexts Γ_i and Γ_j.

Since $t_i = t_j$, we deduce: $\mathcal{I} \models (t_1 \approx \Gamma_i[\Gamma_j[t_1]_{p_j.\cdots.p_n}]_{p_1.\cdots.p_{i-1}})\sigma$, which entails that $\Gamma\sigma$ is not a minimal cyclicity witness for $t_1\sigma$, because, since $i < j$, necessarily $0 < |p_1.\cdots.p_{i-1}.p_j.\cdots.p_n| < |p_1.\cdots.p_n|$. □

Lemma 24. *Let $t : \tau$ and $s : \upsilon$ be ground terms with $\tau \sim \upsilon$. Let $\Gamma[\bullet]_p$ be a ground constructor context of type τ, where p is a position of type υ in Γ. Let N be a clause set saturated under $\mathcal{SP}$ and* Inf. *Assume that t, s, and $\Gamma|_{p'}$ are R_N-irreducible, for every position $p' \not\leqslant p$. If $\mathcal{M}_N \models \Gamma[s]_p \approx t$, then R_N contains n rules $\Gamma_i[a_{i+1}]_{p_i} \to_{R_N} a_i$, for $i \in [1, n]$, with $\Gamma[s]_p = \Gamma_0[\Gamma_1[\cdots[\Gamma_n[a_{n+1}]_{p_n}]\cdots]_{p_1}]_{p_0}$, $p_0.p_1.\cdots.p_n = p$, $a_{n+1} = s$, and $t = \Gamma_0[a_1]_{p_0}$.*

Proof. Let $t' = \Gamma[s]_p$. Since $\mathcal{M}_N \models t' \approx t$ and t is R_N-irreducible, we have $t' \to_{R_N}^k t$, for some natural number $k \geq 0$. The proof is by induction on k. It is immediate if $k = 0$, since in this case $t = t'$, by letting $n = 0$, $\Gamma_0 = \Gamma$ and $p_0 = p$ (with $a_1 = s$). Otherwise, since $\Gamma|_{p'}$ is R_N-irreducible, for $p' \not\leqslant p$, the first rule in the $\to_{R_N}$-derivation from t' to t must be applied at some position $p' \leqslant p$. We denote by p_n the position such that $p = p'.p_n$. Thus there exists a rule $\Delta[s]_{p_n} \to_{R_N} b$ such that $t'|_{p'} = \Delta[s]_{p_n}$, and $t'[b]_{p'} \to_{R_N}^{k-1} t$. Since Γ is a constructor context, the head symbol of Δ is a constructor, and by Corollary 4.1, b must be R_N-irreducible. By the induction hypothesis, since $t'[b]_{p'} \to_{R_N}^{k-1} t$, there exist rules $\Gamma_i[a_{i+1}]_{p_i} \to_{R_N} a_i$,

for $i \in [1, n-1]$, with $t'[b]_{p'} = \Gamma_0[\cdots[\Gamma_{n-1}[a_n]_{p_{n-1}}]\cdots]_{p_0}$, $p_0.\cdots.p_{n-1} = p'$, $a_n = b$ and $t = \Gamma_0[a_1]_{p_0}$. By letting $\Gamma_n = \Delta$ and $a_{n+1} = s$, we get $t' = \Gamma_0[\cdots[\Gamma_{n-1}[\Gamma_n[a_{n+1}]_{p_n}]_{p_{n-1}}]\cdots]_{p_0}$ with $p_0.\cdots.p_{n-1}.p_n = p$. □

Lemma 25 (Acyclicity)**.** *If* $\mathsf{Sub} \subseteq N$ *and* $N \not\ni \square$ *is saturated under* $\mathcal{SP}$, Acycl, *and* Inf, *then* $\mathcal{M}_N$ *satisfies condition* **Acy**.

Proof. Assume that there exist a ground term t of some type $\tau \in \mathcal{T}_{\mathsf{ind}}$ and a ground constructor context $\Gamma[\bullet]_p$ such that $\mathcal{M}_N \models t \approx \Gamma[t]_p$, with $p \neq \varepsilon$. W.l.o.g., we may assume that t is a minimal term (with respect to $\succ$) of some type in $\mathcal{T}_{\mathsf{ind}}$ such that a context Γ satisfying the condition above exists, that Γ is a minimal cyclicity witness for t and $\mathcal{M}_N$ and that $\Gamma|_{p'}$ is R_N-irreducible for every $p' \not\leqslant p$. Since t is minimal, t is R_N-irreducible. Let $t' = \Gamma[t]_p$.

By Lemma 24, R_N contains n rules $\Gamma_i[a_{i+1}]_{p_i} \to_{R_N} a_i$, for $i \in [1, n]$, such that $t' = \Gamma_0[\cdots[\Gamma_n[a_{n+1}]_{p_n}]\cdots]_{p_0}$, $p_0.\cdots.p_n = p$, $a_{n+1} = t$ and $t = \Gamma_0[a_1]_{p_0}$. If $p_0 \neq \varepsilon$, then $t \succ a_1$, and

$$\mathcal{M}_N \models a_1 \approx \Gamma_1[\cdots[\Gamma_n[\Gamma_0[a_1]_{p_0}]]_{p_n}]\cdots]_{p_0}$$

which contradicts the minimality of t, because $\mathrm{type}(a_1) \sim \mathrm{type}(t) \in \mathcal{T}_{\mathsf{ind}}$. Thus we have $p_0 = \varepsilon$ and Γ_0 is empty.

By definition of R_N, there exist n clauses $\mathcal{C}_i = u_i \approx s_i \vee \mathcal{C}'_i$ and substitutions θ_i (for $i \in [1, n]$) such that, for every $i \in [1, n]$, $u_i\theta_i \succ s_i\theta_i$, $(u_i \approx s_i)\theta_i \succ \mathcal{C}'_i\theta_i$, $u_i\theta_i = \Gamma_i[a_{i+1}]_{p_i}$, $s_i\theta_i = a_i$ and $\mathcal{M}_N \not\models \mathcal{C}'_i\theta_i$. Let q_i be the maximal prefix of p_i that is a position in u_i. We distinguish two cases.

- If there exists $i \in [1, n]$ such that $u_i|_{q_i}$ is a variable x, then $\mathcal{C}_i$ must be of the form $u_i[x]_{q_i} \approx s_i \vee \mathcal{C}'_i$. By Proposition 10, u_i is not a variable, hence $q_i \neq \varepsilon$. Consequently, the Acycl rule applies on $\mathcal{C}_i$ (using a trivial chain (s_i, x) of length 1 that is eligible since it ends with a variable) and derives the clause: $\mathcal{C}'_i \vee \neg\,\mathsf{sub}(s_i, x)$. Since $\mathcal{M}_N$ is saturated under Acycl, this entails that $\mathcal{M}_N \models \mathcal{C}'_i \vee \neg\,\mathsf{sub}(s_i, x)$, hence $\mathcal{M}_N \not\models \mathsf{sub}(s_i, x)\theta_i$. Let q'_i be the position such that $q_i.q'_i = p_i$. We have $x = u_i|_{q_i}$. Thus:

  $$\begin{aligned} x\theta_i &\overset{*}{\leftrightarrow}_{R_N} \Gamma_i|_{q_i}[a_{i+1}]_{q'_i} \\ &\overset{*}{\leftrightarrow}_{R_N} \Gamma_i|_{q_i}[\Gamma_{i+1}[\cdots[\Gamma_n[a_{n+1}]_{p_n}]\cdots]_{p_{i+1}}]_{q'_i} \\ &\overset{*}{\leftrightarrow}_{R_N} \Gamma_i|_{q_i}[\Gamma_{i+1}[\cdots[\Gamma_n[t]_{p_n}]\cdots]_{p_{i+1}}]_{q'_i} \\ &\overset{*}{\leftrightarrow}_{R_N} \Gamma_i|_{q_i}[\Gamma_{i+1}[\cdots[\Gamma_{i-1}[\Gamma_1[\cdots[\Gamma_{i-1}[a_i]_{p_{i-1}}]\cdots]_{p_1}]_{p_{n-1}}]\cdots]_{p_{i+1}}]_{q'_i} \end{aligned}$$

  By Proposition 2, this entails that $\mathcal{M}_N \models \mathsf{sub}(a_i, x\theta_i)$, that is, $\mathcal{M}_N \models \mathsf{sub}(s_i, x)\theta_i$, a contradiction.

- Otherwise, we must have $p_i = q_i$, for all $i \in [1, n]$. Let $s'_{i+1} = u_i|_{p_i}$, for $i \in [1, n]$. The substitution $\theta_1 \dots \theta_n$ is a solution of $s'_{n+1} \stackrel{?}{=} s_1 \wedge \bigwedge_{i=2}^{n} s'_i \stackrel{?}{=} s_i$, hence this problem admits an mgu σ, with $\theta_1 \dots \theta_n = \sigma\eta$. Let $(t_1, \dots, t_{n+1}) = (s_1, \dots, s_{n+1})\sigma$. It is easy to check that all the conditions of Definition 14 are satisfied, hence $(t_1, \dots, t_{n+1})$ is a cycle. By definition of **Keep**, there exists k such that $(t_{1+k}, \dots, t_{n+k}, t_{1+k})$ (with $t_i := t_{i-n}$ if $i > n$) is a cycle, and there exists a chain $(b_1, \dots, b_{m+1})$ satisfying **Keep** such that $(t_{1+k}, \dots, t_{n+k}, t_{1+k})$ is an extension of $(b_1, \dots, b_{m+1})$. We assume that $(b_1, \dots, b_{m+1})$ is the longest chain with this property. By definition, this entails that $(b_1, \dots, b_{m+1})$ is eligible (indeed, either $m = n$ and the chain is a cycle, or $(b_1, \dots, b_{m+1}) \neq (t_{1+k}, \dots, t_{n+k}, t_{1+k})$ and there exists an extension of length $m + 2$ of $(b_1, \dots, b_{m+1})$ that does not satisfied **Keep**). Furthermore, since Γ is a minimal cyclicity witness for t and $\mathcal{M}_N$, $(t_1, \dots, t_{n+1})$ is direct by Proposition 11, hence $(b_1, \dots, b_{m+1})$ is also direct. Consequently, the Acycl rule applies on the clauses $\mathcal{C}_{1+k}, \dots, \mathcal{C}_{m+k}$, yielding a clause of the form $\mathcal{C}'_{1+k}\theta \vee \dots \vee \mathcal{C}'_{m+k}\theta \vee \mathcal{E}$, where the subclause $\mathcal{E}$ is either $\Box$ or $\neg\, \mathsf{sub}(b_{m+1}, b_1)$. By Proposition 8, there exists σ' such that $\sigma = \theta\sigma'$ and $t_{i+k} = b_i\sigma'$ (for $i = 1, \dots, m$). We have $\mathcal{C}'_i\theta\sigma'\eta = \mathcal{C}'_i\sigma\eta = \mathcal{C}_i\theta_i$, hence $\mathcal{M}_N \not\models \mathcal{C}'_{1+k}\theta\sigma'\eta \vee \dots \vee \mathcal{C}'_{m+k}\theta\sigma'\eta$. Since N is saturated under Acycl, $\mathcal{M}_N \models \mathcal{C}'_{1+k}\theta \vee \dots \vee \mathcal{C}'_{m+k}\theta \vee \mathcal{E}$, hence we deduce that $\mathcal{M}_N \models \mathcal{E}\sigma'\eta$. Thus $\mathcal{M}_N \not\models \mathsf{sub}(b_{m+1}, b_1)\sigma'\eta$. By Proposition 6, there exist some constructor context Δ and position $r \neq \varepsilon$ such that $\mathcal{M}_N \models (t_{m+1+k} \approx \Delta[t_{1+k}]_r)\eta$, i.e., $\mathcal{M}_N \models (b_{m+1} \approx \Delta[b_1]_r)\sigma'\eta$. This contradicts the fact that $\mathcal{M}_N \not\models \mathsf{sub}(b_{m+1}, b_1)\sigma'\eta$.

$\Box$

Remark 10. The Inf rule is needed for completeness. For example, it is clear that the clause $x \approx \mathsf{a} \vee x \approx \mathsf{b}$ contradicts acyclicity, but no contradiction can be derived without using Inf. The relaxation of the application conditions of Sup is also essential. Consider the clause set $N = \{\mathsf{a}_1 \approx \mathsf{c}(\mathsf{a}_2), \mathsf{a}_2 \approx \mathsf{a}_3, \mathsf{a}_3 \approx \mathsf{c}(\mathsf{a}_1)\}$, with $\mathsf{c}(\dots) \succ \mathsf{a}_{i+1} \succ \mathsf{a}_i$. It is clear that N is saturated without the relaxation, and N contradicts acyclicity, since $N \models \mathsf{a}_1 \approx \mathsf{c}(\mathsf{c}(\mathsf{a}_1))$. With the relaxation, Sup derives the clause $\mathsf{a}_2 \approx \mathsf{c}(\mathsf{a}_1)$; then Acycl exploits the cycle $(\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_1)$ to derive $\Box$.

For the Uniq rule, we provide a restricted completeness result, under the assumption that the considered constructor context contains at most one occurrence of $\bullet$.

Lemma 26 (Uniqueness of Fixpoints)**.** *If* $\mathsf{App} \subseteq N$ *and* $N \not\ni \Box$ *is saturated under* $\mathcal{SP}$*,* Uniq*, and* Inf*, then* $\mathcal{M}_N \models x \approx \Gamma[x]_r \wedge y \approx \Gamma[y]_r \Rightarrow$

$x \approx y$ for every constructor context of the form $\Gamma[\bullet]_r$ of type $\tau \in \mathcal{T}_{\mathsf{coind}}$, where r is a nonempty position of type τ in Γ.

Proof. The proof is similar to that of Lemma 25. Let t be a ground term such that there exist a nonempty constructor context $\Gamma[\bullet]_r$ and a ground term s of type τ with $\mathcal{M}_N \models t \approx \Gamma[t]_r \wedge s \approx \Gamma[s]_r \wedge t \not\approx s$. W.l.o.g., we assume that t is a minimal (with respect to $\succ$) term such that Γ and s exist, and that $\Gamma|_p$ is irreducible, for every $p \not\leqslant r$. By Lemma 24, since $\mathcal{M}_N \models t \approx \Gamma[t]_r$, R_N contains n rules $\Gamma_i[a_{i+1}]_{p_i} \rightarrow_{R_N} a_i$, for $i \in [1, n]$, with $\Gamma[t]_r = \Gamma_0[\cdots[\Gamma_n[a_{n+1}]_{p_n}]\cdots]_{p_0}$, $p_0.\cdots.p_n = r$, $a_{n+1} = t$ and $t = \Gamma_0[a_1]_{p_0}$.

Assume first that $p_0 \neq \varepsilon$. We have:

$$\mathcal{M}_N \models a_1 \approx \Gamma_1[\cdots[\Gamma_n[\Gamma_0[a_1]_{p_0}]\ldots]_{p_n}]\cdots]_{p_0}$$

Moreover, from $\mathcal{M}_N \models s \approx \Gamma[s]_r$, we deduce:

$$\mathcal{M}_N \models \Gamma_1[\cdots[\Gamma_n[s]_{p_n}]\cdots]_{p_1} \approx \Gamma_1[\cdots[\Gamma_n[\Gamma[s]_r]_{p_n}]\cdots]_{p_1}$$

i.e.:

$$\mathcal{M}_N \models \Gamma_1[\cdots[\Gamma_n[s]_{p_n}]\cdots]_{p_1} \approx \Gamma_1[\cdots[\Gamma_n[\Gamma_0[\cdots[\Gamma_n[s]_{p_n}]\cdots]_{p_0}]_{p_n}]\cdots]_{p_1}$$

thus by letting $s' = \Gamma_1[\cdots[\Gamma_n[s]_{p_n}]\cdots]_{p_1}$, we get

$$\mathcal{M}_N \models s' \approx \Gamma_1[\cdots[\Gamma_{n-1}[\Gamma_0[s']_{p_0}]_{p_{n-1}}]\cdots]_{p_0}$$

By minimality of t, since $t \succ a_1$ we deduce that $\mathcal{M}_N \models a_1 \approx s'$, hence that $\mathcal{M}_N \models \Gamma_0[a_1]_{p_0} \approx \Gamma_0[s']_{p_0}$, i.e., $\mathcal{M}_N \models t \approx s$, which contradicts our assumption.

Thus $p_0 = \varepsilon$, and Γ_0 is empty. By definition of R_N, there exist clauses $\mathcal{C}_i = u_i \approx s_i \vee \mathcal{C}'_i$ and substitutions θ_i (for $i \in [1, n]$) such that, for every $i \in [1, n]$, $u_i\theta_i \succ s_i\theta_i$, $(u_i \approx s_i)\theta_i \succ \mathcal{C}'_i\theta_i$, $u_i\theta_i = \Gamma_i[a_{i+1}]_{p_i}$, $s_i\theta_i = a_i$ and $\mathcal{M}_N \not\models \mathcal{C}'_i\theta_i$. Let q_i be the maximal prefix of p_i that is a position in u_i. We distinguish two cases.

- If there exists $i \in [1, n]$ such that $u_i|_{q_i}$ is a variable y, then $\mathcal{C}_i$ must be of the form $u_i[y]_{q_i} \approx s_i \vee \mathcal{C}'_i$. We assume, w.l.o.g., that i is the minimal number having this property. By Proposition 10, u_i is not a variable, hence $q_i \neq \varepsilon$. Consequently, the Uniq rule applies on $\mathcal{C}_i$ and derives the clause $\mathcal{C}'_i \vee \mathcal{D}$, with

$$\mathcal{D} = \big(\bigvee_{p \in P} u_i|_p \not\approx \mathsf{app}(s_p, s_i)\big) \vee u' \not\approx z \vee z \approx s_i$$

where P is the set of prefix-minimal positions p of some type $\tau \sim \text{type}(s_i)$ in $u_i[y]_{q_i}$ such that $p \not< q_i$. Since $\mathcal{M}_N$ is saturated under Acycl, this entails that $\mathcal{M}_N \models \mathcal{C}'_i \vee \mathcal{D}$, hence $\mathcal{M}_N \models \mathcal{D}\theta_i$. Γ can be written on the form

$$\Gamma = \Delta[\Delta'[\bullet]_{p_i.\cdots.p_n}]_{p_1.\cdots.p_{i-1}}$$

Every position $p \in P$ is a position in Γ_i, since $u_i\theta_i = \Gamma_i[a_{i+1}]_{p_i}$. Thus for every $p \in P$, $p_1.\cdots.p_{i-1}.p$ is a position in Γ. Moreover, by Proposition 4, there exists a term s'_p such that $\mathsf{App} \models \forall x.\ \mathsf{app}(s'_p, x) \approx \Gamma|_{p_1.\cdots.p_{i-1}.p}[\Delta[x]_{p_1.\cdots.p_{i-1}}]_{Q_p}$, where Q_p denotes the set of positions of $\bullet$ in $\Gamma|_{p_1.\cdots.p_{i-1}.p}$ (this set is either empty, if $p \neq q_i$, or a singleton $\{q'_i.p_{i+1}.\cdots.p_n\}$ with $q_i.q'_i = p_i$ if $p = q_i$). Let η be a substitution mapping each variable s_p to s'_p and mapping z to $\Delta'[s]_{p_i.\cdots.p_n}$. By definition of η, since $\mathcal{M}_N \models \mathsf{App}$, we have:

$$\mathcal{M}_N \models (\mathsf{app}(s_p, s_i) \approx \Gamma|_{p_1.\cdots.p_{i-1}.p}[\Delta[s_i]_{p_1.\cdots.p_{i-1}}]_{Q_p})\eta$$

Furthermore, $\mathcal{M}_N \models s_i\theta_i \approx \Delta'[t]_{p_i.\cdots.p_{n-1}}$ hence $\mathcal{M}_N \models (\mathsf{app}(s_p, s_i) \approx \Gamma|_{p_1.\cdots.p_{i-1}.p}[\Delta[\Delta'[t]_{p_i.\cdots.p_n}]_{p_1.\cdots.p_{i-1}}]_{Q_p})\theta_i\eta$ Thus we have $\mathcal{M}_N \models (\mathsf{app}(s_p, s_i) \approx \Gamma|_{p_1.\cdots.p_{i-1}.p}[t]_{Q_p})\theta_i\eta$ which entails that $\mathcal{M}_N \models (\mathsf{app}(s_p, s_i) \approx \Gamma[t]_r|_{p_1.\cdots.p_{i-1}.p})\theta_i\eta$. Therefore $\mathcal{M}_N \models (u_i|_p \approx \mathsf{app}(s_p, s_i))\theta_i\eta$ for every $p \in P$. Similarly, if $p \neq q_i$, then $\mathsf{App} \models (\mathsf{app}(s_p, z) \approx u_i|_p)\theta_i\eta$, thus $\mathcal{M}_N \models (u' \approx u_i[\mathsf{app}(s_{q_i}, z)]_{q_i})\theta_i\eta$, hence:

$$\mathcal{M}_N \models (u' \approx u_i[\Gamma|_{p_1.\cdots.p_{i-1}.q_i}[\Delta[\Delta'[s]_{p_i.\cdots.p_n}]_{p_1.\cdots.p_{i-1}}]_{Q_{q_i}}]_{q_i})\theta_i\eta$$

and therefore $\mathcal{M}_N \models (u' \approx u_i[\Gamma|_{p_1.\cdots.p_{i-1}.q_i}[s]_{Q_{q_i}}]_{q_i})\theta_i\eta$; thus $\mathcal{M}_N \models (u' \approx z)\theta_i\eta$. Because $\mathcal{M}_N \models \mathcal{D}\theta_i$, we deduce that $\mathcal{M}_N \models (z \approx s_i)\theta_i\eta$, and therefore $\mathcal{M}_N \models (\Delta[z]_{p_1.\cdots.p_{i-1}} \approx \Delta[s_i]_{p_1.\cdots.p_{i-1}})\theta_i\eta$. Consequently, $\mathcal{M}_N \models s \approx t$.

- Otherwise, we must have $p_i = q_i$, for $i \in [1, n]$. Let $s'_{i+1} = u_i|_{p_i}$, for $i \in [1, n]$. The substitution $\theta_1 \ldots \theta_n$ is a solution of $s'_{n+1} \stackrel{?}{=} s_1 \wedge \bigwedge_{i=2}^{n} s'_i \stackrel{?}{=} s_i$, hence this problem admits an mgu σ, with $\theta_1 \ldots \theta_n = \sigma\eta$. It is easy to check that the sequence $(t_1, \ldots, t_{n+1}) = (s_1, \ldots, s_{n+1})\sigma$ is a cycle; thus there exists a k such that the sequence $(t_{1+k}, \ldots, t_{n+k}, t_{1+k})$ (with $t_i := t_{i-n}$ if $i > n$) is a cycle, and there exists a chain $(b_1, \ldots, b_{m+1})$ satisfying **Keep** such that $(t_{1+k}, \ldots, t_{n+k}, t_{1+k})$ is an extension of $(b_1, \ldots, b_{m+1})$. We assume that $(b_1, \ldots, b_{m+1})$ is the longest chain having this property, which entails that $(b_1, \ldots, b_{m+1})$ is eligible. The Uniq rule applies on the clauses $\mathcal{C}_{1+k}, \ldots, \mathcal{C}_{m+k}$, yielding a clause of the form $\mathcal{C}'_{1+k}\theta \vee \cdots \vee$

$\mathcal{C}'_{m+k}\theta \vee \mathcal{E}$, where $\mathcal{E} = \left(\bigvee_{p\in P} u|_p \not\approx \mathsf{app}(s_p, b_1)\right) \vee u' \not\approx z \vee z \approx b_1$ and P is the set of prefix-minimal positions p of some type $\tau \sim \mathrm{type}(s_{1+k})$ in the constructor context $u = \Gamma_{1+k}[\ldots[\Gamma_{m+k}[\bullet]_{p_{m+k}}\ldots]_{p_{1+k}}]$ such that $p \not< p_{1+k}.\cdots.p_{m+k}$.

By Proposition 8, there exists a substitution σ' such that $\sigma = \theta\sigma'$ and $t_{i+k} = b_i\sigma'$ for $i \in [1, m+1]$. We have $\mathcal{C}'_i\theta\sigma'\eta = \mathcal{C}'_i\sigma\eta = \mathcal{C}'_i\theta_i$, hence $\mathcal{M}_N \not\models \mathcal{C}'_{1+k}\theta\sigma'\eta \vee \cdots \vee \mathcal{C}'_{m+k}\theta\sigma'\eta$. Since N is saturated under Uniq, $\mathcal{M}_N \models \mathcal{C}'_{1+k}\theta \vee \cdots \vee \mathcal{C}'_{m+k}\theta \vee \mathcal{E}$, hence we deduce that $\mathcal{M}_N \models \mathcal{E}\sigma'\eta$.

It is clear that there exist constructor contexts Δ and $\Delta'[\bullet]_{r'}$ such that:

$$\Gamma[\Gamma[\bullet]_r]_r|_{p_1.\cdots.p_{1+k}} = \Delta[\Delta'[\bullet]_{r'}]_{p_{2+k}.\cdots.p_{n+k}}$$

By Proposition 4, for every position $p \in P$, there exists a term s'_p such that $\mathsf{App} \models \forall x.\, \mathsf{app}(s'_p, x) \approx \Gamma|_{p_1.\cdots.p_k.p}[\Delta[x]_{p_{2+k}.\cdots.p_{n+k}}]_{Q_p}$, where Q_p denotes the set of positions of $\bullet$ in $\Gamma|_{p_1.\cdots.p_k.p}$ (as in the previous case, this set contains at most one element). Let γ be a substitution mapping each variable s_p to s'_p and mapping z to $\Delta'[s]_{r'}$. As in the previous case, it is easy to check that $\mathcal{M}_N \models (u|_p \approx \mathsf{app}(s_p, b_1))\sigma'\eta\gamma$ for every $p \in P$, and $\mathcal{M}_N \models (u' \approx z)\sigma'\eta\gamma$; thus we have $\mathcal{M}_N \models (z \approx b_1)\sigma'\eta\gamma$, whence $\mathcal{M}_N \models \Delta'[s]_{r'}\sigma'\eta \approx b_1\sigma'\eta$, thus $\mathcal{M}_N \models \Gamma[\Delta[\Delta'[s]_{r'}\sigma'\eta]_{p_{2+k}.\cdots.p_{n+k}}]_{p_1.\cdots.p_{1+k}} \approx \Gamma[\Delta[b_1]_{p_{2+k}.\cdots.p_{n+k}}]_{p_1.\cdots.p_{1+k}}\sigma'\eta$ and therefore $\mathcal{M}_N \models s \approx t$.

□

Definition 18. A signature is *coinductively nonbranching* if for every constructor $\mathsf{c} : \tau_1 \times \cdots \times \tau_n \to \tau$ such that $\tau \in \mathcal{T}_{\mathsf{coind}}$, there exists at most one $i \in [1, n]$ such that $\tau_i \sim \tau$.

For example, the signature is coinductively nonbranching for infinite streams and possibly infinite lists, but not for infinite binary trees.

Proposition 12. *Let t be a term of type $\tau \in \mathcal{T}_{\mathsf{coind}}$ and p_1, p_2 be two positions in t of types τ_1 and τ_2, respectively. If the signature is coinductively nonbranching, and if $\tau_1 \sim \tau_2 \sim \tau$, then $p_1 \leqslant p_2$ or $p_2 \leqslant p_1$.*

Proof. Let p be the greatest common prefix of p_1 and p_2. If $p_1 \not\leqslant p_2$ and $p_2 \not\leqslant p_1$, then necessarily $p \neq p_1, p_2$; thus $p_i = p.j_i.p'_i$, where j_1 and j_2 are distinct integers. Then $t|_p$ is of the form $\mathsf{c}(t_1, \ldots, t_n)$, where c is a constructor, $j_1, j_2 \in [1, n]$, $\tau_i \rhd^* \mathrm{type}(t_{j_i})$ and $\mathrm{type}(t_{j_i}) \rhd^* \tau$ thus $\mathrm{type}(t_{j_1}) \sim \mathrm{type}(t_{j_2}) \sim \mathrm{type}(t|_p)$, with contradicts the fact that the signature is coinductively nonbranching. □

Corollary 4.2 (Fixpoints). Assume that the signature is coinductively nonbranching. If $\mathsf{Cycl} \cup \mathsf{App} \subseteq N$ and $N \not\ni \Box$ is saturated under $\mathcal{SP}$, Uniq, and Inf, then $\mathcal{M}_N$ satisfies condition **FP**.

Proof. Let $\Gamma[\bullet]_P$ be a nonempty constructor context of type $\tau \in \mathcal{T}_{\mathsf{coind}}$, where P is a set of positions of type τ in Γ. By the hypothesis of the corollary, we may assume, by Proposition 12, that P is a singleton $\{r\}$. Since $\mathcal{M}_N \models \mathsf{Cycl}$, we have $\mathcal{M}_N \models \exists x.\ x \approx \Gamma[x]_r$. By Lemma 26, $\mathcal{M}_N \models x \approx \Gamma[x]_r \wedge y \approx \Gamma[y]_r \Rightarrow x \approx y$. $\Box$

Example 6. Corollary 4.2 does not hold for arbitrary signatures. The clause set $\{\mathsf{a} \approx \mathsf{c}(\mathsf{d}(\mathsf{a},\mathsf{b})),\ \mathsf{b} \approx \mathsf{e}(\mathsf{d}(\mathsf{a},\mathsf{b})),\ \mathsf{a}' \approx \mathsf{c}(\mathsf{d}(\mathsf{a}',\mathsf{b}')),\ \mathsf{b}' \approx \mathsf{e}(\mathsf{d}(\mathsf{a}',\mathsf{b}')),\ \mathsf{d}(\mathsf{a},\mathsf{b}) \not\approx \mathsf{d}(\mathsf{a}',\mathsf{b}')\}$ contradicts **FP**, because $\mathsf{d}(\mathsf{a},\mathsf{b})$ and $\mathsf{d}(\mathsf{a}',\mathsf{b}')$ are both solutions of $x \approx \mathsf{d}(\mathsf{c}(x),\mathsf{e}(x))$. However, the Uniq rule applies only with constructor contexts of head symbol c (if the chain starts with a or a') or e (if the chain starts with b or b').

Observe that in the proof of Lemma 26, the variables $p \in P$ (with the notations of the Uniq rule), are always instantiated with a term $\mathsf{cst}(u|_p)$, except when $p = q$. Thus the result holds for this particular instantiation of the rule, and all terms $\mathsf{app}(s_p, x)$ with $p \neq q$ may be replaced by $u|_p$ in this case. However, being able to instantiate s_p by terms different from $\mathsf{cst}(u|_p)$ is useful when contexts with several holes are considered, although the rule is not complete in this case. Note also that if the signature is coinductively nonbranching, then necessarily $P = \{q\}$, by Proposition 12.

6.6 Saturation Procedure

The inference rules of the calculus presented in Section 6.4 are all finitely branching, provided that the eligibility criterion is applied for the Acycl and Uniq rules. As a result, saturation of a clause set can be carried out using standard saturation procedures. These generally work by maintaining a set of *passive clauses* that initially contains all the clauses to saturate and a set of *active clauses* that is initially empty. The algorithm heuristically chooses a passive clause that becomes the *given clause*, moves it to the active clauses, and performs all possible inferences between it and the active clauses. Conclusions are added to the set of passive clauses, and the procedure is iterated until $\Box$ is derived, or until the set of passive clauses is empty, in which case the set of active clauses is saturated.

To improve search, it is useful to distinguish between *simplifying rules* and *generating rules*. In simplifying rules, at least one of the premises

is redundant with respect to the conclusion. The Inf rule is simplifying, as well as the Dist_1 and Inj_1 rules when the term t is not a variable, and the Acycl rule when there is only one premise and $t_1 = t_n$. Since they allow the replacement of a clause by another, simplifying rules should be applied immediately after the generation of a new clause. For the same reason, they should be applied to all literals (rather than only to maximal literals) and without any order condition.

In addition to the calculus, we propose the following simplifying rules to eliminate theory tautologies:

$$\frac{\mathsf{c}(\bar{s}) \not\approx \mathsf{d}(\bar{t}) \vee \mathcal{C}}{\emptyset}\ \mathsf{Dist}^-$$

$$\frac{s \not\approx \Gamma[s] \vee \mathcal{C}}{\emptyset}\ \mathsf{Acycl}^-$$

where $\mathsf{c} \bowtie \mathsf{d}$, $\Gamma[\bullet]$ is a nonempty constructor context, and $\mathrm{type}(s) \in \mathcal{T}_{\mathsf{ind}}$.

Moreover, the following rule applies injectivity of constructors $\mathsf{c} \in \mathcal{C}tr_{\mathsf{inj}}$ to simplify literals:

$$\frac{\mathsf{c}(s_1, \ldots, s_n) \not\approx \mathsf{c}(t_1, \ldots, t_n) \vee \mathcal{C}}{\left(\bigvee_{i=1}^{n} s_i \not\approx t_i\right) \vee \mathcal{C}}\ \mathsf{Inj}^-$$

The soundness of Inj^- follows from c's being a function symbol, but since it is also injective, the premise is redundant with respect to the theory. We conjecture that the addition of these simplification rules preserves refutational completeness.

If all constructors are free (i.e., $\mathcal{C}tr_{\mathsf{inj}} = \mathcal{C}tr$ and $\mathsf{c} \bowtie \mathsf{d}$ holds for all distinct constructors c and d), by applying the above rules eagerly, we also guarantee that in any literal $[\neg]s \approx t$ in an active clause, at most one of s or t has a constructor for head symbol, as (dis)equalities between constructor terms will have been simplified directly after clause generation. This invariant enables a few optimizations in the implementation of the generating rules, notably during the detection of chains.

The relaxation of the application conditions of the Sup rule increases the number of clauses it must generate and hence may be detrimental to the search. We can reduce the incidence of this scenario by choosing a term order that considers constructors as smaller than non-constructors. For path orders, we can choose a symbol precedence $\succ$ such that $\mathsf{f} \succ \mathsf{c}$ for all non-constructor symbols f and constructors c.

To implement the Acycl and Uniq rules, we must be able to efficiently detect eligible chains among the set of active clauses. Testing all subsets

of the active clauses is impractical, and the detection of a chain requires the computation of an mgu over a set of equations, instead of a single equation. We present a procedure that takes the given clause $\mathcal{C}_1$ as input and applies the two rules to all subsets of clauses containing $\mathcal{C}_1$ and upon which an eligible chain can be built. There are three cases in which the rules must be applied: when the chain is a cycle, when it is variable-ended and has length 1, and when there exists an extension of the chain that violates **Keep**. The procedure relies on a data structure that provides a $\mathsf{nextLinks}(s')$ operation, where s' is a term. For each literal $s \approx t$ in an active clause $\mathcal{C}$ such that s is unifiable with s' under an mgu σ and $s\sigma \not\succeq t\sigma$, the operation returns the tuple $(\mathcal{C}, \sigma, T)$, where T is the set of terms under nonempty constructor positions in t. This operation can be implemented using term indexing techniques already found in state-of-the-art provers.

The procedure $\mathsf{considerGiven}(\mathcal{C}_1)$ applies the rule Acycl or Uniq to all subsets of actives clauses that contain the given clause $\mathcal{C}_1$ and form an eligible chain:

Procedure $\mathsf{considerGiven}(\mathcal{C}_1)$ **is**
  for s'_2 **such that** $\mathcal{C}_1 = s_1 \approx \Gamma[s'_2] \vee \mathcal{D}_1$ **do**
    $\mathsf{extendChain}(s_1, s'_2, \{\}, \{\mathcal{C}_1\})$

Procedure $\mathsf{extendChain}(s_1, s'_i, \theta, Ch)$ **is**
  if $s_1\theta = s'_i\theta$ **then**
    apply rule Acycl or Uniq to chain Ch under mgu θ
  else if s'_i is a variable **then**
    if $|Ch| = 1$ **then**
      apply rule Acycl or Uniq to chain Ch under mgu θ
  else if exists $(\mathcal{C}_i, \sigma, T) \in \mathsf{nextLinks}(s'_i\theta)$ **such that** $\mathcal{C}_i \in Ch$ **then**
    apply rule Acycl or Uniq to chain Ch under mgu θ
  else
    for $(\mathcal{C}_i, \sigma, T) \in \mathsf{nextLinks}(s'_i\theta)$ **do**
      for $s'_{i+1} \in T$ **do**
        $\mathsf{extendChain}(s_1, s'_{i+1}, \sigma\theta, Ch \uplus \{\mathcal{C}_i\})$

6.7 Evaluation

We implemented the calculus presented above in the first-order theorem prover Vampire [84]. Our source code is publicly available.[1] The new rules are added to the existing calculus, which includes other sound

[1] http://github.com/vprover/vampire/releases/tag/ijcar2018-data

rules and a sophisticated redundancy elimination mechanism. Vampire can process input files in SMT-LIB [11] format and recognizes the command `declare-datatypes`, as well as the nonstandard command `declare-codatatypes`. These commands trigger the addition of relevant axioms or the activation of inference rules, according to user-specified options. This implementation is an extension of previous work done in Vampire [81] (Chapter 4 of this thesis). The behavior of this older implementation can be replicated by enabling only the simplification rules of the calculus and adding the axioms Dist, Inj, Exhaust, Sub, and NSub to the initial clause set.

We evaluated the implementation on 4170 problems that were used previously by Reynolds and Blanchette [114] to evaluate CVC4. These were generated by translating Isabelle problems to SMT-LIB using the Sledgehammer bridge [104]. We also used synthetic problems that exercise the properties of cyclic values. Both benchmark sets and detailed results are available online.[2]

All the experiments in this section were carried out on a cluster on which each node is equipped with two quad-core Intel processors running at 2.4 GHz, with 24 GiB of memory. A 60 s time limit per problem was enforced. We used a single basic saturation strategy relying on the DISCOUNT saturation algorithm. The calculus was parameterized by a Knuth–Bendix term order, unless otherwise noted. This simple approach provides a homogeneous basis on which to compare the performance of the different procedures. It typically solves fewer problems than the portfolio approach commonly used with Vampire and other provers, in which several different strategies are tried in short time slices.

We first compare the performance of three configurations of the prover on the Isabelle problems. The first configuration corresponds to the axiomatic approach presented in Section 6.3: the axioms Dist, Inj, Exhaust, Sub, NSub, App, Uniq, Cycl, and Hole are added to the set of clauses to saturate, and only standard inferences rules are used by the prover. Superposition need not rewrite the nonmaximal side of an equation.

The second configuration implements part of the calculus presented in Section 6.4. Only the axioms Exhaust, Sub, NSub, App, Uniq, Cycl, and Hole are added to the clauses, and the rules Dist_1, Dist_2, Inj_1, and Inj_2 are used during the search, in addition to the simplification rules described in Section 6.6. The side conditions of Sup are also relaxed. The rules Acycl and Uniq are not used; instead, reasoning on the properties of cyclic terms is based on axioms.

[2] `http://matryoshka.gforge.inria.fr/pubs/supdata_data.tar.gz`

The third configuration uses all the rules described in Section 6.4. Only the axioms Sub and App are added, on which the Acycl and Uniq rules depend, and the axioms Cycl and Exhaust. This configuration is the only one which does not ensure refutational completeness, since Uniq is incomplete with respect to the uniqueness of fixpoints for branching codatatypes.

The first two configurations both solved 1114 problems and the third one solved 1113 problems; 1116 problems are solved by at least one configuration. These homogeneous results do not reveal significant differences between the approaches. To assess the role of the acyclicity property of datatypes and the properties of codatatype fixpoints in the benchmarks, we also tested a system that did not include any axioms and rules related to these properties. With such an incomplete system, we found that 12 problems could not be solved. This is roughly in line with the results of Reynolds and Blanchette using CVC4 on the same problems [114]. No new problems were solved by this configuration, suggesting that reasoning about properties of cyclic terms does not lead to worse performance even when these properties are not needed for refutation.

We also tested variants of the last two configurations in which the calculus was parameterized by a lexicographic path order, to assess whether this term order could improve the performance when used with the relaxed superposition rule. These configurations solved a total of 1104 problems, including 5 new problems. This shows that using a different term order allows the exploration of different parts of the search space, but the choice of order does not seem to impact the performance of the relaxed superposition rule.

Since properties of cyclic values are seldom used in the Isabelle benchmarks, we crafted a set of (refutable) problems to assess the performance of the rules Acycl and Uniq. For a term s and a nonconstant context $\Gamma[\bullet]$, let $\mathsf{exchain}(s, \Gamma[\bullet])$ denote any sentence

$$\exists s_2, \ldots s_n \forall t_1, \ldots t_m.\ s \approx \Gamma_1[s_2] \wedge \cdots \wedge s_n \approx \Gamma_n[s],$$

where $t_1, \ldots, t_m$ all occur in Γ and such that $\Gamma_1[\ldots[\Gamma_n[\bullet]]\ldots] = \Gamma[\bullet]$. The formula $\exists s.\, \mathsf{exchain}(s, \Gamma[\bullet])$, where $\mathrm{type}(s) \in \mathcal{T}_{\mathsf{ind}}$, forms an acyclicity problem. The set of acyclicity problems used in our experiments is denoted AC. If $m = 0$, the clausified form of this problem is ground (ACG). The formula $\exists s_1, s_2.\, \mathsf{exchain}(s_1, \Gamma[\bullet]) \wedge \mathsf{exchain}(s_2, \Gamma[\bullet]) \wedge s_1 \not\approx s_2$, where $\mathrm{type}(s_1) \in \mathcal{T}_{\mathsf{coind}}$, forms a uniqueness problem (U). Note that in such a problem, the two chains may not be formed upon the same equalities, although they build the same constructor context. Similarly, if $m =$

0, we obtain a ground uniqueness problem (UG). Finally, the sentence $\forall s.\ \neg\,\mathsf{exchain}(s, \Gamma[\bullet])$, for $\mathrm{type}(s) \in \mathcal{T}_{\mathsf{coind}}$, forms an existence problem (EX).

We generated 100 instances of each type of problem. The number of problems solved by Vampire (V) on these problems are presented in the following table, along with the results obtained using CVC4's [9] and Z3's [50] native support for datatypes and, in CVC4's case, for codatatypes:

	AC			ACG			U		UG		EX	
	V	CVC4	Z3	V	CVC4	Z3	V	CVC4	V	CVC4	V	CVC4
Axioms	65	–	–	**100**	–	–	**14**	–	10	–	**40**	–
Calculus	82	**100**	59	**100**	**100**	**100**	**14**	12	13	**100**	35	0

The number of problems solved shows that the Acycl rule performs better than the axioms for acyclicity problems with variables. Only one of these problems could be solved by the axiomatic approach and not by the Acycl rule. Both approaches managed to solve all of the ground acyclicity problems. Z3 solved all of the ground problems, performing slightly less well on those featuring universal quantifiers. CVC4 was able to solve all of the acyclicity problems, including those with universal quantifiers, a notable improvement over previous results obtained on similar problems (cf. Section).

On uniqueness problems, the Uniq rule solved a superset of the ground problems solved by the axiomatic approach, whereas on nonground problems each approach uniquely solved 3 problems, for a total of 17 problems solved. Again, CVC4 performed remarkably well on ground problems, while the presence of variables in the problem led to a marked degradation of its performance. Finally, for existence problems, the refutation relies mostly on the Cycl axiom, which is included in the clause set in both Vampire configurations. Yet, the purely axiomatic approach was able to solve 6 problems that could not be solved when the Uniq rule was activated, indicating that the rule might lead the search in a suboptimal direction. The theory solver in CVC4 does not take into account the existence of fixpoints for codatatypes, which is a nonground property. Consequently, none of the existence problems were solved by CVC4.

From the results, it appears that the calculus supersedes the axiomatic approach for problems with datatypes. For codatatypes, both approaches solve different problems, suggesting that they should both be included in a strategy portfolio. However, the conceptual simplicity and easy implementation of the axiomatic approach may outweigh these differences

in performance.

6.8 Related Work

The potential of (co)datatypes for automated reasoning has been studied mostly in the context of satisfiability modulo theories (SMT). Datatypes are parts of the SMT-LIB 2.6 standard [11]. They were implemented in CVC3 by Barrett et al. [12], in Z3 [50] by de Moura, and in CVC4 by Reynolds and Blanchette [114]. The CVC4 work also includes a decision procedure for the ground theory of codatatypes. Moreover, CVC4 supports automatic structural induction [115] and dedicated reasoning support for selectors.

Structural induction has also been added to superposition by Kersani and Peltier [75], Cruanes [46], and Wand [136]. In unpublished work, Wand implemented incomplete inference rules for datatypes, including acyclicity, in his superposition prover Pirate. Robillard's earlier Acycl rule [118] (Chapter 5 of this thesis) has inspired our Acycl rule, but it suffered from many forms of incompleteness. For example, given the unsatisfiable clause set

$$\{a \approx \mathsf{c}(x) \vee p(x), \neg p(\mathsf{c}(a))\},$$

the old Acycl rule derived only $p(a)$ before reaching saturation. Another issue concerned cycles built from multiple copies of the same premise. Consider the unsatisfiable clause set

$$\{a \approx \mathsf{c}(f(zero)),\, f(x) \approx \mathsf{c}(f(suc(x))),\, f(suc(suc(zero))) \approx \mathsf{c}(a)\}.$$

The new rule can build a cycle of length 5 by using the second clause twice, with $x \mapsto zero$ and $x \mapsto suc(zero)$, whereas the old rule never reused clauses, to achieve finite branching. Our solution to achieve finite branching involves using the sub predicate, pushing the burden of enumerating possibly infinitely many cycles onto the core superposition calculus.

In the context of program verification, Bjørner [21] introduced a decision procedure for (co)datatypes in STeP, the Stanford Temporal Prover. The program verification tool Dafny provides both a syntax for defining (co)datatypes and some support for automatic (co)induction proofs [92]. Other verification tools such as Leon [129] and RADA [105] also include (semi-)decision procedures for datatypes. We refer to Barrett et al. [12] and Reynolds and Blanchette [114] for further discussions of related work.

6.9 Conclusion

We presented two approaches to reason about datatypes and codatatypes in first-order logic: an axiomatization and an extension of the superposition calculus. We established completeness results about both. We also showed how to integrate the new inference rules in a saturation prover's main loop and implemented them in the Vampire prover. The empirical results look promising, although it is not clear from our benchmarks how often the most difficult properties—acyclicity for datatypes, existence and uniqueness of fixpoints for codatatypes—are useful in practice.

This work is part of a wider research program that aims at bridging the gap between automatic theorem provers and their applications to program verification and interactive theorem proving. In future work, we want to reconstruct the new proof rules in Isabelle, to make it possible to enable datatype reasoning in Sledgehammer. We also believe that further tuning and evaluations could help improve the calculus and the heuristics.

Bibliography

[1] Wolfgang Ahrendt, Laura Kovács, and Simon Robillard. Reasoning about loops using Vampire in KeY. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 434–443. Springer, 2015.
— One citation on page 47

[2] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *International Conference on Computer Aided Verification*, pages 679–685. Springer, 2012.
— One citation on page 67

[3] Krzysztof R. Apt and Alessandro Pellegrini. Why the occur-check is not a problem. In *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 69–86. Springer, 1992.
— One citation on page 110

[4] Leo Bachmair, Nachum Dershowitz, and Jieh Hsiang. Orderings for equational proofs. In *The Symposium on Logic in Computer Science*, pages 346–357. IEEE Computer Society, 1986.
— One citation on page 116

[5] Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In *International Conference on Automated Deduction*, volume 449 of *LNCS*, pages 427–441. Springer, 1990.
— One citation on page 13

[6] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
— 4 citations on pages 13, 115, 125, and 134

[7] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. Extending SMT solvers to higher-order logic. In *International Conference on Automated Deduction*, LNCS. Springer, 2019. To appear.
— One citation on page 20

[8] 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 41

[9] 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.
— 3 citations on pages 47, 72, and 148

[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.smt-lib.org`, 2016.
— One citation on page 85

[11] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.7. Technical report, University of Iowa, 2017. `www.smt-lib.org`.
— 3 citations on pages 61, 146, and 149

[12] 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.
— 5 citations on pages 72, 92, 97, 111, and 149

[13] Lewis Denver Baxter. *The Complexity of Unification*. PhD thesis, University of Waterloo Waterloo, Ontario, 1976.
— One citation on page 91

[14] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.
— 4 citations on pages 25, 26, 35, and 36

[15] 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 36

[16] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with lambdas. In *International Conference on Automated Deduction*, LNCS. Springer, 2019. To appear.
— One citation on page 20

[17] Christoph Benzmüller, Nik Sultana, Lawrence C Paulson, and Frank Theiß. The higher-order prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404, 2015.
— One citation on page 6

[18] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A concrete memory model for CompCert. In *International Conference on Interactive Theorem Proving*, pages 67–83. Springer, 2015.
— One citation on page 3

[19] Dirk Beyer. Software verification with validation of results. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–349. Springer, 2017.
— 2 citations on pages 47 and 61

[20] Ahmed Bhayat and Giles Reger. Higher-order reasoning Vampire style. In *Automated Reasoning Workshop*, pages 19–20, 2018.
— One citation on page 20

[21] Nikolaj Bjørner. *Integrating Decision Procedures for Temporal Verification*. PhD thesis, Stanford University, 1999.
— 2 citations on pages 111 and 149

[22] Nikolaj Bjørner, 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 92

[23] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. On solving universally quantified Horn clauses. In *Static Analysis*, pages 105–125. Springer, 2013.
— 2 citations on pages 47 and 67

[24] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending Sledgehammer with SMT solvers. *International Conference on Automated Deduction*, 6803:116–130, 2013.
— 4 citations on pages 20, 73, 86, and 93

[25] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 93–110. Springer, 2014.
— One citation on page 119

[26] Jasmin Christian Blanchette, Nicolas Peltier, and Simon Robillard. Superposition with datatypes and codatatypes. In *Automated Reasoning*, pages 370–387. Springer, 2018.
— One citation on page 59

[27] Daniel Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4(4):412–430, 1975.
— One citation on page 12

[28] Chad E Brown. Satallax: An automatic higher-order prover. In *Automated Reasoning*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.
— One citation on page 6

[29] Ritu Chadha and David A Plaisted. On the mechanical derivation of loop invariants. *Journal of Symbolic Computation*, 15(5-6):705–744, 1993.
— One citation on page 67

[30] YuTing Chen and Carlo A Furia. Triggerless happy – intermediate verification with a first-order prover. In *International Conference on integrated Formal Methods*, volume 10510 of *LNCS*, pages 295–311. Springer, 2017.
— One citation on page 20

[31] YuTing Chen and Carlo A Furia. Robustness testing of intermediate verifiers. In *International Symposium on Automated Technology for Verification and Analysis*, volume 11138 of *LNCS*, pages 91–108. Springer, 2018.
— One citation on page 20

[32] YuTing Chen, Laura Kovács, and Simon Robillard. Theory-specific reasoning about loops with arrays using Vampire. In *Proceedings of the 3rd Vampire Workshop*, EPiC Series in Computing, pages 16–32. EasyChair, 2017.
— One citation on page 60

[33] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.
— One citation on page 93

[34] Keith L Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Springer, 1978.
— One citation on page 71

[35] Edmund Melson Clarke Jr. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM (JACM)*, 26(1):129–147, 1979.
— One citation on page 4

[36] David R. Cok, David Déharbe, and Tjark Weber. The 2014 SMT competition. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:207–242, 2014.
— One citation on page 16

[37] Alain Colmerauer. Prolog and infinite trees. *Logic Programming*, 16:231–251, 1982.
— 2 citations on pages 110 and 111

[38] Alain Colmerauer and Dao Thi-Bich-Hanh. Expressiveness of full first order constraints in the algebra of finite or infinite trees. In *Principles and Practice of Constraint Programming*, volume 1894 of *LNCS*, pages 172–186. Springer, 2000.
— 6 citations on pages 73, 81, 86, 87, 88, and 91

[39] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. Linear invariant generation using non-linear constraint solving. In *International Conference on Computer Aided Verification*, pages 420–432. Springer, 2003.
— 2 citations on pages 6 and 67

[40] Hubert Comon. *Unification et disunification: Théorie et applications*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 1988.
— One citation on page 91

[41] Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3-4):371–425, 1989.
— One citation on page 124

[42] Stephen A Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.
— One citation on page 4

[43] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.
— One citation on page 71

[44] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
— One citation on page 1

[45] 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.
— 4 citations on pages 6, 26, 45, and 67

[46] Simon Cruanes. Superposition with structural induction. In *FroCoS 2017*, volume 10483 of *LNCS*, pages 172–188. Springer, 2017.
— 4 citations on pages 16, 58, 115, and 149

[47] 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 25

[48] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.
— One citation on page 7

[49] Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In *International Conference on Automated Deduction*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
— 2 citations on pages 20 and 88

[50] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis*

of Systems, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
— 5 citations on pages 47, 73, 108, 148, and 149

[51] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
— One citation on page 5

[52] Edsger W Dijkstra. A discipline of programming. *Prentice-Hall Series in Automatic Computation*, 1976.
— One citation on page 1

[53] 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.
— 4 citations on pages 37, 47, 61, and 67

[54] 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 25, 33, 35, 37, and 39

[55] 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 77 and 91

[56] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In *Programming Languages and Systems*, pages 125–128. Springer, 2013.
— One citation on page 20

[57] Melvin Fitting and Richard L Mendelsohn. *First-order modal logic*, volume 277 of *Synthese Library*. Springer Science & Business Media, 1998.
— One citation on page 65

[58] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. *ACM SIGPLAN Notices*, 37(1):191–202, 2002.
— One citation on page 45

[59] 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 26 and 33

[60] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *International Conference on Automated Deduction*, pages 167–182. Springer, 2007.
— One citation on page 20

[61] Yeting Ge and Leonardo De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *International Conference on Computer Aided Verification*, pages 306–320. Springer, 2009.
— One citation on page 20

[62] 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 71

[63] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *Computer Aided Verification*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.
— 4 citations on pages 6, 26, 33, and 67

[64] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. SMT-based verification of parameterized systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 338–348. ACM, 2016.
— 2 citations on pages 45 and 65

[65] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of philosophical logic*, pages 99–217. Springer, 2001.
— One citation on page 7

[66] Jacques Herbrand. *Recherches sur la théorie de la démonstration.* PhD thesis, Université de Paris, 1930.
— One citation on page 91

[67] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
— 2 citations on pages 1 and 3

[68] 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 33 and 35

[69] Wilfrid Hodges. *Model Theory*. Cambridge University Press, 1993.
— One citation on page 91

[70] Gérard Huet. *Résolution d'équations dans les langages d'ordre 1, 2,..., ω*. PhD thesis, Université Paris VII, 1976.
— One citation on page 91

[71] Gilles Kahn. Natural semantics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer, 1987.
— One citation on page 2

[72] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM*, 64(1):7, 2017.
— 2 citations on pages 45 and 65

[73] Egor George Karpenkov and David Monniaux. Formula slicing: Inductive invariants from preconditions. In *Haifa Verification Conference*, pages 169–185. Springer, 2016.
— One citation on page 67

[74] Abdelkader Kersani and Nicolas Peltier. Combining superposition and induction: A practical realization. In *International Symposium on Frontiers of Combining Systems*, LNCS, pages 7–22. Springer, 2013.
— One citation on page 58

[75] Abdelkader Kersani and Nicolas Peltier. Combining superposition and induction: A practical realization. In *Frontiers of Combining Systems*, pages 7–22. Springer, 2013.
— 2 citations on pages 115 and 149

[76] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.
— One citation on page 13

[77] 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 77

[78] 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 37 and 41

[79] Laura Kovács. Reasoning algebraically about P-solvable loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 249–264. Springer, 2008.
— One citation on page 6

[80] Laura Kovács and Simon Robillard. Reasoning about loops using Vampire. In *Proceedings of the 1st and 2nd Vampire Workshops*, volume 38 of *EPiC Series in Computing*, pages 52–62. EasyChair, 2016.
— 2 citations on pages 59 and 61

[81] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. In *Symposium on Principles of Programming Languages*, pages 260–270. ACM, 2017.
— 7 citations on pages 59, 97, 106, 108, 111, 115, and 146

[82] 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.
— 8 citations on pages 6, 17, 25, 32, 45, 47, 55, and 60

[83] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In *International Conference on Automated Deduction*, volume 5663 of *LNCS*, pages 199–213. Springer, 2009.
— One citation on page 30

[84] 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.
— 8 citations on pages 16, 25, 47, 73, 84, 100, 116, and 145

[85] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3-4):227–260, 1971.
— One citation on page 12

[86] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.
— One citation on page 3

[87] Shuvendu K Lahiri and Randal E Bryant. Constructing quantified invariants via predicate abstraction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 267–281. Springer, 2004.
— One citation on page 45

[88] Peter J Landin. Correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, 1965.
— One citation on page 1

[89] 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.
— 2 citations on pages 26 and 67

[90] K Rustan M Leino. This is boogie 2, 2008.
— One citation on page 20

[91] 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 25

[92] K Rustan M Leino and Michał Moskal. Co-induction simply. In *FM 2014: Formal Methods*, pages 382–398. Springer, 2014.
— One citation on page 149

[93] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
— One citation on page 3

[94] Zhen Li. *Efficient and Generic Reasoning for Modal Logics*. PhD thesis, The University of Manchester, UK, 2008.
— One citation on page 65

[95] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Symposium on Logic in Computer Science*, pages 348–357. IEEE, 1988.
— 4 citations on pages 91, 99, 107, and 110

[96] Anatoly Ivanovich Mal'cev. Axiomatizable classes of locally free algebras of certain types. *Sibirsk. Mat. Ž.*, 3:729–743, 1962.
— 2 citations on pages 72 and 91

[97] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
— 2 citations on pages 91 and 104

[98] Kenneth L McMillan. Quantified invariant generation using an interpolating saturation prover. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427. Springer, 2008.
— 2 citations on pages 45 and 67

[99] 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 134

[100] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.
— 3 citations on pages 73, 86, and 116

[101] Ernst-Rüdiger Olderog. *General equivalence of expressivity definitions using strongest postconditions resp. weakest preconditions*. Inst. für Informatik u. Prakt. Mathematik, Christian-Albrechts-Univ., 1980.
— 2 citations on pages 5 and 50

[102] Ross A. Overbeek. An implementation of hyper-resolution. *Computers & Mathematics with Applications*, 1(2):201–214, 1975.
— One citation on page 110

[103] 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 91

[104] Lawrence C Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *The 8th International Workshop ib the Implementation of Logics*, EPiC Series in Computing, pages 1–11. EasyChair, 2012.
— One citation on page 146

[105] Tuan-Hung Pham and Michael W Whalen. RADA: A tool for reasoning about algebraic data types with abstractions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 611–614. ACM, 2013.
— One citation on page 149

[106] David A. Plaisted. The occur-check problem in Prolog. *New Generation Computing*, 2(4):309–322, 1984.
— One citation on page 110

[107] Gordon D Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.
— One citation on page 2

[108] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.
— One citation on page 7

[109] W. V. Quine. A proof procedure for quantification theory. *The Journal of Symbolic Logic*, 20(2):141–149, 1955.
— One citation on page 7

[110] Giles Reger, Nikolaj Bjørner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *GCAI*, pages 39–52, 2016.
— 3 citations on pages 15, 20, and 59

[111] Giles Reger and Martin Suda. Set of support for theory reasoning. In *IWIL Workshop and LPAR Short Presentations*, volume 1 of *Kalpa Publications in Computing*, pages 124–134. EasyChair, 2017.
— One citation on page 14

[112] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with abstraction and theory instantiation in saturation-based reasoning. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–22. Springer, 2018.
— 3 citations on pages 15, 20, and 59

[113] Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co)datatypes in SMT solvers. In *International Conference on Automated Deduction*, volume 9195 of *LNCS*, pages 197–213. Springer, 2015.
— 5 citations on pages 72, 73, 85, 86, and 92

[114] Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co)datatypes in SMT solvers. *Journal of Automated Reasoning*, 58(3):341–362, 2017.
— 7 citations on pages 97, 107, 108, 111, 146, 147, and 149

[115] Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In *16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 80–98. Springer, 2015.
— One citation on page 149

[116] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.
— One citation on page 7

[117] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1):101–115, 2003.
— 2 citations on pages 16 and 103

[118] Simon Robillard. An inference rule for the acyclicity property of term algebras. In *Proceedings of the 4th Vampire Workshop*, volume 53 of *EPiC Series in Computing*, pages 20–32. EasyChair, 2018.
— 2 citations on pages 116 and 149

[119] John Alan Robinson. Automatic deduction with hyper-resolution. *International Journal of Computing and Mathematics*, 1:227–234, 1965.
— 2 citations on pages 8 and 110

[120] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.
— 2 citations on pages 75 and 91

[121] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *International Conference on Algebraic Methodology and Software Technology*, pages 142–162. Springer, 2010.
— One citation on page 7

[122] Tatiana Rybina and Andrei Voronkov. A decision procedure for term algebras with queues. *ACM Transactions on Computational Logic*, 2(2):155–181, 2001.
— 5 citations on pages 75, 76, 91, 97, and 110

[123] Stephan Schulz. System description: E 1.8. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.
— One citation on page 16

[124] Dana S Scott and Christopher Strachey. *Toward a Mathematical Semantics for Computer Languages*. Oxford University Computing Laboratory, Programming Research Group, 1971.
— One citation on page 1

[125] R. Sekar, I.V. Ramakrishnan, and Andrei Voronkov. Term indexing. In *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier, 2001.
— 2 citations on pages 16 and 104

[126] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 59(4):483–502, 2017.
— One citation on page 61

[127] Geoff Sutcliffe and Christian Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.
— 2 citations on pages 16 and 41

[128] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. *ACM SIGPLAN Notices*, 45(1):199–210, 2010.
— 2 citations on pages 97 and 111

[129] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *International Static Analysis Symposium*, pages 298–315. Springer, 2011.
— One citation on page 149

[130] Dao Thi-Bich-Hanh. *Résolution de Contraintes du Premier Ordre dans la Théorie des Arbres Finis ou Infinis*. PhD thesis, Université Aix-Marseille 2, 2000.
— 4 citations on pages 88, 91, 97, and 111

[131] Maarten H. van Emden and John W. Lloyd. A logical reconstruction of Prolog II. *The Journal of Logic Programming*, 1(2):143–149, 1984.
— One citation on page 110

[132] 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 80

[133] 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 77

[134] Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In *International Conference on Computer Aided Verification*, volume 8559 of *LNCS*, pages 696–710. Springer, 2014.
— One citation on page 108

[135] 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 93

[136] Daniel Wand. *Superposition: Types and Induction*. PhD thesis, Saarland University, 2017.
— 4 citations on pages 58, 111, 115, and 149

[137] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5.

In *International Conference on Automated Deduction*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.
— One citation on page 16

[138] Lawrence Wos and George Robinson. Paramodulation and set of support. In *Symposium on Automatic Demonstration*, volume 125 of *LNM*, pages 276–310. Springer, 1970.
— 3 citations on pages 10, 12, and 110

[139] Lawrence Wos, George A. Robinson, and Daniel F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM (JACM)*, 12(4):536–541, 1965.
— One citation on page 14