



Coming to terms with quantified reasoning

DOI:
[10.1145/3009837.3009887](https://doi.org/10.1145/3009837.3009887)

Document Version
Accepted author manuscript

[Link to publication record in Manchester Research Explorer](#)

Citation for published version (APA):
Kovács, L., Robillard, S., & Voronkov, A. (2017). Coming to terms with quantified reasoning. In *POPL 2017: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (pp. 260-270) <https://doi.org/10.1145/3009837.3009887>

Published in:
POPL 2017: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [<http://man.ac.uk/04Y6Bo>] or contact openresearch@manchester.ac.uk providing relevant details, so we can investigate your claim.



Coming to Terms with Quantified Reasoning

Laura Kovács

TU Wien, Austria
laura.kovacs@tuwien.ac.at

Simon Robillard

Chalmers Univ. of Technology, Sweden
simon.robillard@chalmers.se

Andrei Voronkov

University of Manchester, UK
Chalmers Univ. of Technology, Sweden
andrei@voronkov.com

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.

Categories and Subject Descriptors CR-number [subcategory]: D 2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages, F 4.1. Mathematical Logic, I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods

Keywords Program analysis and verification, algebraic data types, automated reasoning, first-order theorem proving, superposition proving

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

POPL'17, January 15–21, 2017, Paris, France
© 2017 ACM. 978-1-4503-4660-3/17/01...\$15.00
<http://dx.doi.org/10.1145/3009837.3009887>

program analysis. Terms may be used to formalize the semantics of programming languages (Goguen et al. 1977; Clark 1978; Courcelle 1983); they can also themselves be the object of computation. The latter is especially obvious in the case of functional programming languages, where algebraic data structures are manipulated. Consider for example the following declaration, in the functional language ML:

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

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

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

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

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

Note that depending on the semantics of the programming language, there may exist cyclic terms such as the one satisfying the equation $x \approx \text{succ}(x)$, 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 (Mal'cev 1962). 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 (Mal'cev 1962) 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 (Barrett et al. 2007; Reynolds and Blanchette 2015) 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 (Barrett et al. 2011). 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 $x \not\approx f(x)$, $x \not\approx f(f(x))$, \dots . 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 5). This extension relies on the addition of a predicate to describe the “proper subterm” relation between terms. This approach is complete and can easily be used in any first-order theorem prover without any modification.

Unfortunately, the subterm relation is transitive, so that the number of predicates produced by saturation quickly becomes a burden for any prover. To improve the efficiency of the reasoning, we offer an alternative solution: extending the inference system of the saturation theorem prover with additional rules to treat equalities between terms (Section 6).

We implemented our new inference system, as well as the subterm relation, in the first-order theorem prover Vampire (Kovács and Voronkov 2013). 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 (Reynolds and Blanchette 2015). This set of examples were generated using the Isabelle inductive theorem prover (Nipkow et al. 2002) and translated by the Sledgehammer system (Blanchette et al. 2013). Further, we also used problems from (Colmerauer et al. 2000) with many quantifier alternations over term algebras. When compared to state-of-the-art SMT solvers, such as CVC4 and Z3 (de Moura and Bjørner 2008), 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 7).

Contributions. The main contributions of our paper are summarized below.

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

an important consequence of this conservative extension, our work yields a superposition-based decision procedure for term algebras (Section 5).

- We show how to optimize superposition-based first-order reasoning using new, term algebra specific, simplification rules, and an incomplete, but simple, replacement for a troublesome acyclicity axiom. Our new inference system provides an alternative and efficient approach to axiomatic reasoning about term algebras in first-order theorem proving and can be used with combinations of theories (Section 6).
- We implement our work in the first-order theorem prover Vampire. Our work 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 7).

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 \perp 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 $p(t_1, \dots, t_n)$, where p is a predicate symbol and t_1, \dots, 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 $L_1 \vee \dots \vee L_n$, where $n \geq 0$. When $n = 0$, we will speak of the empty clause, denoted by \square . The empty clause is always false.

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

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

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

By an *expression* E we mean a term, atom, literal, or clause. A *substitution* θ is a finite mapping from variables to terms. An *application* of this substitution to an expression (e.g. a term or a clause) E , denoted by $E\theta$, is the expression obtained from E by the simultaneous replacement of each variable x in it, such that $\theta(x)$ is defined, by $\theta(x)$. We write $E[s]$ to mean an expression E with a particular occurrence of a term s . A *unifier* of two expressions E_1 and E_2 is a substitution θ such that $E_1\theta = E_2\theta$. It is known that if two expressions have a unifier, then they have a so-called *most general unifier (mgu)* – see (Robinson 1965) for details on computing mgus.

3. Superposition and Proof Search

We now recall some terminology related to inference systems and first-order theorem proving. Inference systems are used in the theory of superposition (Nieuwenhuis and Rubio 2001) implemented by several leading automated first-order theorem provers, including Vampire (Kovács and Voronkov 2013) and E (Schulz 2002). The material of this section is based on (Kovács and Voronkov 2013), adapted to our setting.

3.1 The Superposition Inference System

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

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

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

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

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

1. \preceq is *well-founded*: there exists no infinite sequence of terms t_0, t_1, \dots such that $t_0 \preceq t_1 \preceq \dots$;
2. \preceq is *stable under substitution*: if $s \preceq t$ then $s\theta \prec t\theta$, for every term s, t and substitution θ ;
3. \preceq is *monotonic*: if $s \preceq t$ then $l[s] \preceq l[t]$ for all terms l, s, t ;
4. \preceq has the *subterm property*: if s is a proper subterm of t , then $s \preceq t$.

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

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

We now fix a simplification ordering \preceq and a well-behaved selection function and define the *superposition inference system*. This inference system, denoted by \mathcal{S} , consists of the inference rules of Figure 1. The inference system \mathcal{S} is a sound and refutationally complete inference system for first-order logic with equality. By refutational completeness we mean that if a set S of formulas is unsatisfiable, then \square is derivable from S in \mathcal{S} .

3.2 Proof Search by Saturation

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

selection strategy must be fair: every possible inference must be selected at some step of the algorithm. A saturation algorithm with a fair inference selection strategy is called a *fair saturation algorithm*.

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

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

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

4. The Theory of Finite Term Algebras

A definition of the first-order theory of term algebras over a finite signature can be found in e.g. (Rybina and Voronkov 2001), 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.

4.1 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.

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

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

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

$$f(\bar{x}) \approx f(\bar{y}) \rightarrow \bar{x} \approx \bar{y} \tag{A3}$$

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

- Resolution

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

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

- Superposition

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

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

- Factoring

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

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

Figure 1. The superposition calculus \mathcal{S} .

$$t \not\approx x \quad (\text{A4})$$

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.

4.2 Known Results

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

Theorem 1. *The following results hold.*

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

Completeness of \mathcal{T}_{FT} is proved in a number of papers - a detailed proof can be found in, e.g., (Rybina and Voronkov 2001).

Decidability of \mathcal{T}_{FT} in Theorem 1 is implied by the completeness of \mathcal{T}_{FT} and by the fact that \mathcal{T}_{FT} has a recursive axiomatization. More precisely, completeness gives the following (slightly unusual) decision procedure: given a sentence F , run any complete first-order theorem proving procedure (e.g., a complete superposition theorem prover) simultaneously and separately on F and $\neg F$. We can get around the problem that the axiomatisation is infinite but throwing in axioms, one after one, while running the proof search — indeed, by the compactness property of first-order logic,

if a formula G is implied by an infinite set of formulas, it is also implied by a finite subset of this set. One of contributions of this paper is showing how to avoid dealing with infinite axiomatizations.

Further, the non-elementary property of \mathcal{T}_{FT} in Theorem 1 follows from a result in (Ferrante and Rackoff 1979): 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 (Vorobyov and Voronkov 1998).

Let us make the following important observation. The decidability and other results of Theorem 1 do not hold when uninterpreted functions or predicates are added to \mathcal{T}_{FT} . If we add to the Σ -term algebra uninterpreted symbols, one can for example use these symbols to provide recursive definitions of addition and multiplication, thus encoding first-order Peano arithmetic. Using the same reasoning as in (Korovin and Voronkov 2007) one can then prove the following result.

Theorem 2. *The first-order theory of Σ -algebras with uninterpreted symbols is Π_1^1 -complete, when Σ contains at least one non-constant.*

We will not give a full proof of Theorem 2 but refer to (Korovin and Voronkov 2007) for details. Here, we only show how to encode non-linear arithmetic in \mathcal{T}_{FT} using Σ -term algebra uninterpreted symbol, which is relatively straightforward. Assume, without loss of generality, that Σ contains a constant 0 and a unary function symbol s (successor). Then all ground terms, and hence all term algebra elements are of the form $s^n(0)$, where $n \geq 0$. We will identify any such term $s^n(0)$ with the non-negative integer n .

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

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

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

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

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

It is not hard to argue that in any extension of the Σ -algebra satisfying A , the functions $+$ and \cdot are interpreted as the addition

and multiplication on non-negative integers. Let now G be any sentence using only $+$, \cdot , s , 0 . Then we have that $A \rightarrow G$ is valid in the Σ -algebra if and only if G is a true formula of arithmetic.

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

4.3 Other Formalizations

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

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

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

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

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.

4.4 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 $\tau_1 \times \dots \times \tau_n \rightarrow \tau$. The requirement that there is at least one constant should then be replaced by the requirement that for every sort, there exists a ground term of this sort.

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

Consider an example with the following term algebra signature:

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

This signature defines an algebra of binary trees, where every node and leaf is decorated by an element of a (non-constructor) sort τ . In this case term algebra axioms are only using sorts with constructors. The axioms of this theory of trees, as defined previously, are shown in Figure 2.

5. A Conservative Extension of the Theory of Term Algebras

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

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

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

$$Sub(x_i, f(x_1, \dots, x_i, \dots, x_n)), \quad (\text{B1})$$

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

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

$$\neg Sub(x, x) \quad (\text{B3})$$

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

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

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

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

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

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

Note that the full first-order theory of term algebras with the subterm predicate is undecidable (Venkataraman 1987).

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

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

While it is hard to expect that proving term algebra formulas by superposition will result in a better decision procedure compared to those described in the literature, see e.g. (Colmerauer et al. 2000), 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

$$\begin{aligned}
& \exists y(x \approx \text{leaf}(y)) \vee \exists y_1, y_2, y_3(x \approx \text{node}(y_1, y_2, y_3)) \\
& \text{node}(x_1, x_2, x_3) \not\approx \text{leaf}(y_1) \\
& \text{leaf}(x) \approx \text{leaf}(y) \rightarrow x \approx y \\
& \text{node}(x_1, x_2, x_3) \approx \text{node}(y_1, y_2, y_3) \rightarrow x_1 \approx y_1 \wedge x_2 \approx y_2 \wedge x_3 \approx y_3 \\
& x \not\approx \text{node}(x, y_1, y_2) \quad x \not\approx \text{node}(y_1, y_2, x) \quad x \not\approx \text{node}(\text{node}(x, y_1, y_2), y_3, y_4) \quad \dots
\end{aligned}$$

Figure 2. The instantiation of the theory axioms for the signature Σ_{Bin} .

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 7 give an evidence that it is also efficient in practice.

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

$$\text{Sub}(x, x_i) \rightarrow \text{Sub}(x, f(x_1, \dots, x_i, \dots, x_n)).$$

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.

6. An Extended Calculus

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

6.1 A naive calculus

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

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

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

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

Take for example two ground unit clauses $f(a) \approx b$ and $g(a) \approx b$, where all symbols apart from b are constructors. This set of clauses is unsatisfiable in the theory of term algebras. However, if we replace the axiom $f(x) \not\approx g(y)$ by a simplification rule, there

are no inferences that can be done between these clauses (assuming we are using the standard Knuth-Bendix ordering).

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

6.2 The Distinctness Rule

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

The simplification rule is

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

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

The deletion rule is

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

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

6.3 The Injectivity Rule

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

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

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

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

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

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

6.4 The Acyclicity Rule

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

The simplification rule for acyclicity is

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

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

The deletion rule is

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

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

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

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

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

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

7. Experimental Results

7.1 Implementation

We implemented the subterm relation of Section 5 and simplification rules of Section 6 in the first-order theorem prover Vampire (Kovács and Voronkov 2013). 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 solvers¹ and the quantified theory division of the CASC 2016 competition of first-order provers². 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 download³.

7.2 Input Syntax and Tool Usage

In our work, we used an extended SMTLIB syntax (Barrett et al. 2016) to describe term constructors. Although not yet part of the official SMTLIB standard, this syntax is already supported by the

¹<http://smtcomp.sourceforge.net/2016/>

²<http://www.cs.miami.edu/~tptp/CASC/J8/>

³<http://www.cse.chalmers.se/~simrob/tools.html>

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 6. 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 5, can also be deactivated.

Note that the SMTLIB syntax also provides the not yet standardized command `declare-codatatypes` to declare types of potentially cyclic or infinite data structures. The theory underlying the semantics of such types is almost identical to that of finite term algebras, except that the acyclicity axiom is replaced by a uniqueness rule that asserts that observationally equal terms are indeed equal (Reynolds and Blanchette 2015). 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.

7.3 Benchmarks

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

- A (parametrized) game theory problem originally described in (Colmerauer et al. 2000). This problem relies on the term algebra of natural numbers to describe winning and losing positions of a game. It is possible to encode, for a given positive integer k , a predicate $winning_k$ over positions, such that $winning_k(p)$ holds iff there exists a winning strategy from the position p in k or fewer moves. The satisfiability of the resulting first-order formula can be checked by term algebra decision procedures, since it does not use symbols other than those of the term algebra, but it includes $2k$ alternating universal and existential quantifiers. This heavy use of quantifiers makes it an interesting and challenging problem for provers. An example of this problem encoded in the SMTLIB syntax is given in Figure 3.
- Problems about functional programs, generated by the Isabelle interactive theorem prover (Nipkow et al. 2002) and translated by the Sledgehammer system (Blanchette et al. 2013). 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 (Reynolds and Blanchette 2015) and, to the best of our knowledge, represent the set of all publicly available benchmarks on algebraic data types.

7.4 Evaluation

Our experiments were carried out on a cluster on which each node is equipped with two quad core Intel processors running at 2.4 GHz and 24GiB of memory. To compare our work to other state-of-the-art systems, we include the results of running the SMT solvers Z3 and CVC4 on the Isabelle problems, as previously reported in (Reynolds and Blanchette 2015), and also add the results of running these two solvers on the game theory problem.

```

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

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

(check-sat)

```

Figure 3. An instance of the game theory problem from (Colmerauer et al. 2000), 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 $winning_1(s(z))$.

Game theory problems. The times required to solve the game theory problem for different values of the parameter k are shown in Table 1. The first column indicates the time required by Vampire using the theory axioms (A) described in Section 5, and the second and third columns give the time needed when the simplification rules (R) are also activated in Vampire (Section 6). For this particular problem, the acyclicity rule plays no role in the proof, but in order to assess its impact on performance, the third column shows the times needed to solve the problem when the subterm relation axioms (S) are also included in the input. The fourth and fifth columns of Table 1 respectively indicate the times needed by CVC4 and Z3 for solving the corresponding problem. Where no value is given, the prover was unable to solve the problem. Despite belonging to a decidable class, this problem is quite challenging for theorem provers and SMT solvers, which is easily explained by the presence of a formula with many quantifier alternations. The SMT solver CVC4 is able to disprove the negated conjecture only for $k = 1$, and Z3 can disprove it only for $k = 1$ or $k = 2$. SMT solvers can also consider the (non-negated) conjecture and try to satisfy it, but this does not produce better results. In comparison, our implementation in Vampire can solve the problem for $k = 6$, that is for formulas with 12 alternated existential and universal quantifiers, in 8.19 seconds. In (Colmerauer et al. 2000), the authors are able to solve the problem for k as high as 80, using an implementation of the decision procedure presented in (Dao 2000). 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 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 (De Moura and Bjørner 2007)). Table 1 also shows that adding simplification rules as described in Section 6 improves the behavior of the theorem prover.

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 1. Time required to prove unsatisfiability of different instances of the game theory problem from (Colmerauer et al. 2000).

Isabelle problems about functional programs. Our results on evaluating Vampire on the Isabelle problems are shown in Table 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, 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 2 summarizes our results on this set of benchmarks, using a single best strategy in Vampire. For each solver, we also show the number of problems solved uniquely only by that solver.

We also ran Vampire with a combination of strategies with a total time limit of 120 seconds. Table 3 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 than CVC4 and 7.3% more than Z3, which is a significant improvement. 50 problems are uniquely solved by Vampire, as listed in column six Table 3. When compared to Vampire, only 4 problems were proved by CVC4 alone, while Z3 cannot prove any problem that was not proved by Vampire – see columns seven and eight of Table 3. Summarizing, Table 2 shows that Vampire outperforms the best existing solvers so far. The experimental results of Tables 1–2 provide an evidence that our methods for proving properties of algebraic data types outperform methods currently used by SMT solvers.

7.5 Comparison of Option Values

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

1. the Boolean value selecting whether term algebra rules are used;

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

Table 2. Number of problems solved among the 6282 Isabelle problems translated by SledgeHammer.

- 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:

- we use a set of problems obtained by discarding problems that are too easy or currently unsolvable;
- 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.

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

The results of these experiments are shown in Table 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 6) instead of theory axioms (Section 5) results in many more problems solved. This gives us an evidence that the axiomatization based on the subterm relation results in much larger search spaces. This also means that the value resulting in an incomplete strategy in this case generally behaves better.

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

8. Related Work

The problem of reasoning over term algebras first appears in the restricted form of syntactic unification, mentioned in (Herbrand 1930). The algorithm for syntactic unification was later described in (Robinson 1965), and later refined into quasi-linear (Baxter 1976; Huet 1976; Martelli and Montanari 1982) and linear algorithms (Paterson and Wegman 1976).

The full-first order theory of term algebras over a finite signature was first studied in (Mal'cev 1962), where its decidability was proved by quantifier elimination. Other quantifier elimination procedures appeared in (Comon 1988; Maher 1988; Hodges 1993; Rybina and Voronkov 2001). (Ferrante and Rackoff 1979) 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 (Colmerauer et al. 2000). The theory of finite or infinite trees is also studied in (Dao 2000) 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 (Barrett et al. 2007) and later extended to a decision procedure for co-algebraic data types in (Reynolds and Blanchette 2015). These decision procedures exploit SMT-style reasoning and are supported by CVC4. Z3 also supports proving properties about algebraic data types (Bjorner et al. 2013). 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.

9. Conclusion

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

- simplifying more clauses;
- 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 (Reynolds and Blanchette 2015). 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.

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

Table 3. Distribution of solved problems according to the data types they feature

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

Table 4. Comparison of proof option values for acyclicity in Vampire.

The work presented here should be a useful development for the verification of functional programs. For example it would benefit the tool HALO (Vytiniotis et al. 2013), 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 (Claessen et al. 2013) and automated reasoning tools for higher-order logic, e.g. Sledgehammer (Blanchette et al. 2013).

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.

Acknowledgments

We acknowledge funding from the ERC Starting Grant 2014 SYM-CAR 639270, the Wallenberg Academy Fellowship 2014 TheProSE, the Swedish VR grant GenPro D0497701, the Austrian FWF research project RiSE S11409-N23, and the EPSRC grant ReVeS: Reasoning for Verification and Security.

References

C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.

C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, volume 6806 of *LNCIS*. Springer, 2011.

C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

L. D. Baxter. *The complexity of unification*. PhD thesis, University of Waterloo Waterloo, Ontario, 1976.

N. Björner, K. McMillan, and A. Rybalchenko. Higher-order program verification as satisfiability modulo theories with algebraic data-types. *arXiv preprint arXiv:1306.5264*, 2013.

J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *Automated Deduction–CADE-23*, 6803:116–130, 2013.

K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *Automated Deduction–CADE-24*, volume 7898 of *LNCIS*, pages 392–406. Springer, 2013.

K. L. Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

A. Colmerauer et al. Expressiveness of full first order constraints in the algebra of finite or infinite trees. In *Principles and Practice of Constraint Programming–CP 2000*, volume 1894 of *LNCIS*, pages 172–186. Springer, 2000.

H. Comon. *Unification et disunification: Théorie et applications*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 1988.

B. Courcelle. Fundamental properties of infinite trees. *Theoretical computer science*, 25(2):95–169, 1983.

T. B. H. Dao. *Résolution de contraintes du premier ordre dans la théorie des arbres finis ou infinis*. PhD thesis, Université Aix-Marseille 2, 2000.

L. De Moura and N. Björner. Efficient e-matching for smt solvers. In *International Conference on Automated Deduction*, volume 4603 of *LNCIS*, pages 183–198. Springer, 2007.

L. M. de Moura and N. Björner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCIS*, pages 337–340. Springer, 2008.

J. Ferrante and C. W. Rackoff. *The computational complexity of logical theories*, volume 718 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979.

J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM (JACM)*, 24(1): 68–95, 1977.

J. Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Université de Paris, 1930.

W. Hodges. *Model Theory*. Cambridge University Press, 1993.

G. Huet. *Résolution d’équations dans des langages d’ordre 1, 2...* PhD thesis, Université Paris VII, 1976.

K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2007.

L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of CAV*, volume 8044 of *LNCIS*, pages 1–35, 2013.

M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 348–357. IEEE Computer Society, 1988.

A. I. Mal’cev. Axiomatizable classes of locally free algebras of certain types. *Sibirsk. Mat. Zh.*, 3:729–743, 1962.

A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2): 258–282, 1982.

R. Nieuwenhuis and A. 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.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCIS*. Springer, 2002.

M. S. Paterson and M. N. Wegman. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186. ACM, 1976.

A. Reynolds and J. C. Blanchette. A decision procedure for (co)datatypes in SMT solvers. In *Automated Deduction–CADE-25*, volume 9195 of *LNCIS*, pages 197–213. Springer, 2015.

- J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.
- T. Rybina and A. Voronkov. A decision procedure for term algebras with queues. *ACM Transactions on Computational Logic*, 2(2):155–181, 2001. doi: 10.1145/371316.371494.
- S. Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2-3): 111–126, 2002.
- K. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM (JACM)*, 34(2):492–510, 1987.
- S. G. Vorobyov and A. 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 Press, 1998.
- D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 431–442. ACM, 2013.