An Abstract Decision Procedure for a Theory of Inductive Data Types

Clark Barrett barrett@cs.nyu.edu
Igor Shikanian igor@cs.nyu.edu
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

Cesare Tinelli* tinelli@cs.uiowa.edu
Department of Computer Science
University of Iowa

Abstract

Inductive data types are a valuable modeling tool for software verification. In the past, decision procedures have been proposed for various theories of inductive data types, some focused on the universal fragment, and some focused on handling arbitrary quantifiers. Because of the complexity of the full theory, previous work on the full theory has not focused on strategies for practical implementation. However, even for the universal fragment, previous work has been limited in several significant ways. In this paper, we present a general and practical algorithm for the universal fragment. The algorithm is presented declaratively as a set of abstract rules which we show to be terminating, sound, and complete. We show how other algorithms can be realized as strategies within our general framework, and we propose a new strategy and give experimental results indicating that it performs well in practice. We conclude with a discussion of several useful ways the algorithm can be extended.

KEYWORDS: *inductive data types, decision procedures, term algebras, satisfiability modulo theories*

1. Introduction

Inductive data types are commonly used in programming. In particular, functional languages support such structures explicitly. The same notion is also a convenient abstraction for common data types such as records and data structures such as linked lists used in more conventional programming languages. The ability to reason automatically and efficiently about inductive data types thus provides an important tool for the analysis and verification of programs.

Perhaps the best-known example of a simple inductive data type is the *list* type used in LISP. Lists are either the *null* list or are constructed from other lists using the *constructor cons*. This constructor takes two arguments and returns the result of prepending its first argument to the list in its second argument. To access the elements of a list, a pair of *selectors* is provided: *car* returns the first element of a list and *cdr* returns the rest of the list.

∗ Partially supported by the National Science Foundation grant #0237422.

More generally, we are interested in any set of (possibly mutually recursive) inductive data types, each of which is built with one or more constructors. Each constructor has selectors that can be used to retrieve the original arguments as well as a *tester* which indicates whether a given term was constructed using that constructor. As an example of the more general case, suppose we want to model lists of trees of natural numbers. Consider a set of three inductive data types: *nat*, *list*, and *tree*. The type *nat* has two constructors: *zero*, which takes no arguments; and *succ*, which takes a single argument of type *nat* and has the corresponding selector *pred*. The *list* type is as before, except that we now specify that the elements of the list are of type *tree*. The *tree* type in turn has two constructors: *node*, which takes an argument of type *list* and has the corresponding selector *children*, and *leaf*, which takes an argument of type *nat* and has the corresponding selector *data*. We can represent this set of types using the following convenient notation based on that used in functional programming languages:

$$\begin{array}{lcl} \mathit{nat} & := & \mathit{succ}(\mathit{pred} : \mathit{nat}) \mid \mathit{zero}; \\ \mathit{list} & := & \mathit{cons}(\mathit{car} : \mathit{tree},\ \mathit{cdr} : \mathit{list}) \mid \mathit{null}; \\ \mathit{tree} & := & \mathit{node}(\mathit{children} : \mathit{list}) \mid \mathit{leaf}(\mathit{data} : \mathit{nat}); \end{array}$$

The testers for this set of data types are *is_succ*, *is_zero*, *is_cons*, *is_null*, *is_node*, and *is_leaf*.

Propositions about a set of inductive data types can be captured in a sorted first-order language which closely resembles the structure of the data types themselves in that it has function symbols for each constructor and selector, and a predicate symbol for each tester. For instance, propositions that we would expect to be true for the example above include the following:

1. $\forall\, x : \mathit{nat}.\ \mathit{succ}(x) \not\approx \mathit{zero}$,
2. $\forall\, x : \mathit{list}.\ x \approx \mathit{null} \vee \mathit{is\_cons}(x)$, and
3. $\forall\, x : \mathit{tree}.\ \mathit{is\_leaf}(x) \rightarrow (\mathit{data}(x) \approx \mathit{zero} \vee \mathit{is\_succ}(\mathit{data}(x)))$.

In this paper, we discuss a procedure for deciding such formulas. We focus on satisfiability of a set of literals, which (through well-known reductions) can be used to decide the validity of universal formulas. We do not consider quantifier elimination, which can be used to decide the full theory, referring the reader instead to related work such as [7, 9, 20, 21].

There are three main contributions of this work over earlier work on the topic. First, our setting is more general: we allow mutually recursive inductive types each with multiple constructors, selectors, and testers, and we use the more general setting of many-sorted logic. The rationale for a many-sorted approach is that it more closely corresponds to potential applications such as analysis of programming languages. In particular, the well-sortedness requirements rule out many syntactical constructs that would not make sense in practice.

The second contribution is in presentation. We present the theory itself in terms of an initial model rather than axiomatically as is often done. Also, the presentation of the decision procedure is given as abstract rewrite rules, making it more flexible and easier to analyze than if it were given imperatively.

Finally, as described in Section 5, the flexibility provided by the abstract algorithm allows us to describe a new strategy with significantly improved practical efficiency.

Related Work. Term algebras over constructors provide the natural intended model for inductive data types. The historically foundational decidability and quantifier elimination results for term algebras can be found in [10]. In other early work, [8] addresses the problem of satisfiability of one equation in a term algebra, modulo other equations. The applications and extension of the quantifier elimination procedure to term algebras with queues is handled in [16]. Another contribution to solving satisfiability of equations over term algebras is given in [19], which extends the language with a powerful *sub-term relation* predicate. In [7] two dual axiomatizations of term algebras are presented, one with constructors only, the other with selectors and testers only.

An often-cited reference for the quantifier-free case is the treatment by Oppen in 1980 [15]. Oppen's algorithm gives a detailed decision procedure for a single inductive data type with a single constructor. The algorithm is linear for conjunctions of literals and NP-complete for arbitrary quantifier-free formulas. The case of multiple constructors is not addressed. In [14], Nelson and Oppen show that for a simple list data type with two constructors, satisfiability of conjunctions of literals is NP-complete. However, no decision procedure is given. Shostak gives an algorithm for a simple theory of lists without *null* in [17]. He also claims there is a generalization to arbitrary inductive data types. However, the claim is unsubstantiated and it is unclear how to generalize to the case of multiple constructors.

More recently, several papers [9, 20, 21] explore decision procedures for a single inductive data type. These papers focus on ambitious schemes for quantifier elimination and combinations with other theories rather than the question of a simple and efficient algorithm for the quantifier-free case. One possible extension of Oppen's algorithm to the case of multiple constructors is discussed briefly in [20]. A comparison of our algorithm with that of [20] is made in Section 5.

Finally, a recent approach based on first-order reasoning with the superposition calculus is described in [6]. This work shows how a decision procedure for an inductive data type with a single constructor can be automatically inferred from the first-order axioms, even though the axiomatization is infinite. While the algorithm as given is worst-case exponential, it has the advantage of being easily implementable (any existing superposition-based theorem prover can be used to implement the strategy) and can be easily combined with other theories that have been shown to be decidable using superposition. We are also interested in being able to combine with other theories (a topic we address in Section 6.3). However, as far as the theory decision procedure is concerned, our focus is on generality and efficiency rather than immediacy of implementation. The multiple-constructor case as well as an investigation of practical efficiency are listed as future work in [6]. Success in these directions would offer an interesting alternative to our approach.

Paper Organization. The paper, which improves and expands on a preliminary version presented at the PDPAR'06 workshop [3] is organized as follows.[1] Section 2 describes our formulation of the first-order theory of inductive data types. In Section 3, we present the algorithm as a set of abstract rules. The correctness of the algorithm is shown in Section 4. In Section 5, we discuss the efficiency of the algorithm and show, in particular, that it

1. The improvements include simpler notation, simplifications to and expanded explanations of the rules, more detailed examples, and a section on correctness with complete proofs.

can be exponentially more efficient than previous naive algorithms. Finally, in Section 6, we discuss how the algorithm can be extended, including how to handle finite sorts.

2. The Theory of Inductive Data Types

Previous work on inductive data types (IDTs) [20, 21] uses first-order axiomatizations in an attempt to capture the main properties of an inductive data type and reason about it. We find it simpler and cleaner to use a semantic approach instead, as is done in algebraic specification. A set of IDTs can be given a simple equational specification over a suitable signature. The intended model for our theory can be formally, and uniquely, defined as the initial model of this specification. Reasoning about a set of IDTs then amounts to reasoning about formulas that are true in this particular initial model.

2.1 Specifying IDTs

We formalize IDTs in the context of many-sorted equational logic (see [12] among others). We will assume that the reader is familiar with the basic notions in this logic, and also with basic notions of term rewriting.

We start with the theory signature. We assume a many-sorted signature Σ whose set of sorts consists of a distinguished sort bool for the Booleans, and $p \geq 1$ sorts $\tau_1, \ldots, \tau_p$ for the IDTs. We also allow $r \geq 0$ additional (non-IDT) sorts $\sigma_1, \ldots, \sigma_r$. We will denote by s, possibly with subscripts, any sort in the signature other than bool, by τ any sort in $\{\tau_1, \ldots, \tau_p\}$, and by σ any sort in $\{\sigma_1, \ldots, \sigma_r\}$.

As mentioned earlier, the function symbols in our theory signature correspond to the constructors, selectors, and testers of the set of IDTs under consideration. We assume for each τ a set $\mathcal{C}_\tau$ of $m_\tau \geq 1$ *constructors* of τ. We will denote constructors by the letter C, possibly primed or with subscripts. We will write $C : s_1 \cdots s_n \to \tau$ to denote that the constructor C takes $n \geq 0$ arguments of respective sort $s_1, \ldots, s_n$ and returns a value of sort τ. Constructors with arity 0 are called *nullary constructors* or *constants*. For each constructor $C : s_1 \cdots s_n \to \tau$, we assume n corresponding *selector* symbols denoted by $S_C^{(1)}, \ldots, S_C^{(n)}$ with $S_C^{(i)} : \tau \to s_i$, and a *tester* predicate symbol denoted by is_C. To simplify some of the proofs, and without loss of generality, we treat is_C as a function symbol of type $\tau \to \mathsf{bool}$. We write $S^{(i)}$ instead of $S_C^{(i)}$ when C is clear from context or not important.

In addition to these symbols, we also assume that the signature contains two constants, true and false of sort bool, and an infinite number of distinct constants of each sort σ. The constants are meant to be names for the elements of that sort, so for instance if σ_1 were a sort for the natural numbers, we could use all the numerals as the constants of sort σ_1. Having all these constants in the signature is not necessary for our approach, but in the following exposition it provides an easy way of ensuring that the sorts in σ are infinite. Section 6.1 shows that our approach can be easily extended to the case in which some of these sorts are finite. As usual in many-sorted equational logic, we also have $p + r + 1$ equality symbols (one for each sort mentioned above), all written as $\approx$.

Our procedure requires one additional constraint on the set of IDTs: It must be *well-founded.* A sort s is well-founded iff there exist ground (i.e., variable-free) Σ-terms of sort s. Informally, each sort must contain terms that do not denote cyclic or otherwise infinite

data types. Note that because we assume the existence of constants of sort σ_i (for each i), these sorts are automatically well-founded.

In some cases, it will be necessary to distinguish between *finite* and *infinite* sorts and constructors:

- A sort s is *finite* iff there are only finitely many ground Σ-terms of sort s;
- a constructor C is *finite* if it is nullary or if all of its argument sorts are finite.

As we will see, consistent with the above terminology, our semantics will interpret finite, resp. infinite, τ-sorts indeed as finite, resp. infinite, sets.

We denote by $\mathcal{T}(\Sigma)$ the set of (well-sorted) ground terms of signature Σ or, equivalently, the many-sorted term algebra over that signature. The IDTs with functions and predicates denoted by the symbols of Σ are specified by the set of universally quantified equations given below. For reasons explained below, we assume that associated with every selector $S_C^{(i)} : \tau \to s$ is a distinguished ground term t_C^i of sort s containing no selectors (or testers).

Equational Specification of IDTs. Given a signature Σ of the form above, the associated inductive data type is specified by the following set $\mathcal{E}$ of axiom schemas for each sort τ in Σ and distinct constructors $C : s_1 \cdots s_n \to \tau$ and $C' : s'_1 \cdots s'_{n'} \to \tau$:

$$\forall x_1, \ldots, x_n.\ is_C(C(x_1, \ldots, x_n)) \approx \mathsf{true}$$
$$\forall x_1, \ldots, x_n.\ is_{C'}(C(x_1, \ldots, x_n)) \approx \mathsf{false}$$
$$\forall x_1, \ldots, x_n.\ S_C^{(i)}(C(x_1, \ldots, x_n)) \approx x_i \qquad \text{for all } i = 1, \ldots, n$$
$$\forall x_1, \ldots, x_n.\ S_{C'}^{(i)}(C(x_1, \ldots, x_n)) \approx t_{C'}^i \qquad \text{for all } i = 1, \ldots, n'$$

The last axiom specifies what happens when a selector is applied to the "wrong" constructor. Note that there is no obviously correct thing to do in this case since it would correspond to an error condition in a real application. Our axiom specifies that in this case, the result is the designated ground term for that selector. This is different from other treatments (such as [7, 20, 21]) where the application of a selector to the wrong constructor is treated as the identity function. The main reason for this difference is that the identity function would not always be well-sorted in many-sorted logic. It is important to notice that as a result, our procedure may give counter-intuitive results if given as input a formula whose satisfiability depends on the application of a selector to the wrong constructor. One possible approach for dealing with this difficulty is discussed in Section 6.2.

By standard results in universal algebra we know that $\mathcal{E}$ admits an *initial model* $\mathcal{R}$. We refer the reader to [12] for a thorough treatment of initial models. For our purposes, it will be enough to mention the following properties that $\mathcal{R}$ enjoys by virtue of being an initial model.

Lemma 2.1. *Where $\approx_{\mathcal{E}}$ is the equivalence relation on Σ-terms induced by $\mathcal{E}$, let $\mathcal{T}(\Sigma)/\approx_{\mathcal{E}}$ be the quotient of the term algebra $\mathcal{T}(\Sigma)$ by $\approx_{\mathcal{E}}$.*

1. *For all ground Σ-terms t_1, t_2 of the same sort, $t_1 \approx_{\mathcal{E}} t_2$ iff $\mathcal{R}$ satisfies $t_1 \approx t_2$.*
2. *$\mathcal{R}$ is isomorphic to $\mathcal{T}(\Sigma)/\approx_{\mathcal{E}}$.*

Proof. These are applications to $\mathcal{R}$ of standard results about initial models. See, for instance Theorem 5.2.11 and Theorem 5.2.17 of [12]. □

Lemma 2.2. *Let Ω be the signature obtained from Σ by removing the selectors and the testers. The reduct of $\mathcal{R}$ to Ω is isomorphic to $\mathcal{T}(\Omega)$.*

Proof. By Lemma 2.1(2) we can take $\mathcal{R}$ to coincide with $\mathcal{T}(\Sigma)/\approx_{\mathcal{E}}$, whose elements are the equivalence classes of $\approx_{\mathcal{E}}$ on the ground Σ-terms. To prove the claim then it is enough to show that (i) every ground Σ-term is equivalent in $\mathcal{E}$ to a ground Ω-term, and (ii) no two distinct ground Ω-terms belong to the same equivalence class.

Consider the rewrite system R obtained by orienting the equations in $\mathcal{E}$ left to right. It is easy to show that R is terminating. It is also immediate that R contains no critical pairs and so it is confluent. It follows by basic results in term rewriting that R is canonical: every Σ-term has a unique normal form (wrt. R), and two Σ-terms are equivalent in $\mathcal{E}$ iff they have the same normal form.

Now, by a simple inductive argument, one can show that the normal form of each ground Σ-term is a ground Ω-term, which proves (i) above. It is trivial that every ground Ω-term is irreducible by R. This entails that distinct ground Ω-terms are inequivalent in $\mathcal{E}$, proving (ii). □

We will call *ground constructor terms* the elements of the set $\mathcal{T}(\Omega)$ defined in the previous lemma. Informally, the lemma means that $\mathcal{R}$ does in fact capture the set of IDTs in question, as we can take the carrier of $\mathcal{R}$ to be the term algebra $\mathcal{T}(\Omega)$. This also shows that in $\mathcal{R}$ each data type τ is generated using just its constructors, and that distinct ground constructor terms of sort τ are distinct elements of the data type. Using the two lemmas one can also easily show that in $\mathcal{R}$ the sort bool denotes a two-element set, the sorts $\sigma_1, \ldots, \sigma_r$ denote infinite sets, and each sort τ denotes an infinite data type if and only if τ is infinite in the sense specified earlier. From a more formal point of view, these lemmas will be useful in proving the correctness of the decision procedure.

3. The Decision Procedure

In this section, we present a decision procedure for the satisfiability of sets of Σ-literals over $\mathcal{R}$. Before giving a formal description of the algorithm, which is quite technical, we start with an informal overview based on examples.

3.1 Overview and Examples

Our procedure builds on the algorithm by Oppen [15] for a single type with a single constructor. Let us first look at how Oppen's procedure works on a simple example.

Example 3.1. *Consider the list data type without the null constructor*[2] *and the following set of literals:* $\{cons(x, y) \approx z, car(w) \approx x, cdr(w) \approx y, w \not\approx z\}$.

2. Note that this data type is not well-founded. Indeed, because Oppen only considers data types with a single constructor, there is no base case for terms (unless the constructor has arity 0), so his semantics are over models with infinite terms. In contrast, we choose to disallow models with infinite terms while allowing multiple constructors, a combination that we feel is more intuitive and corresponds better to actual uses of IDTs.

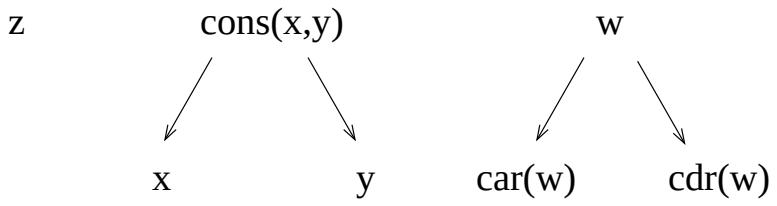

Figure 1. Term graph for Example 3.1

Oppen's procedure works as follows: first, a graph is constructed that relates terms according to their meaning in the intended model. The graph for Example 3.1 is shown in Figure 1. Notice that $cons(x, y)$ is a parent of x and y and $car(w)$ and $cdr(w)$ are children of w. The Oppen algorithm next computes the equivalence relation on nodes of the graph induced by the set of all equations. It then proceeds by performing an *upwards* (congruence) and *downwards* (unification) closure on the graph and then checking for cycles or for a violation of disequalities. A cycle occurs if there exists a sequence of nodes beginning and ending with the same node such that adjacent nodes are either distinct nodes in the same equivalence class or are adjacent in the graph.[3] For Example 3.1, upwards closure implies that $w \approx cons(x, y)$. But since we also have $cons(x, y) \approx z$, this contradicts the disequality $w \not\approx z$, indicating that the set of literals is unsatisfiable.

An alternative algorithm for the case of a single constructor is to introduce new terms and variables to replace variables that are inside of selectors. For Example 3.1, we would introduce $w \approx cons(s, t)$ where s, t are new variables. Now, by substituting and collapsing applications of selectors to constructors, we get $\{cons(x, y) \approx z, w \approx cons(s, t), x \approx s, t \approx y, w \not\approx z\}$. This approach, advocated in [17], only requires downwards closure.

Unfortunately, if a data type has more than one constructor, things are not quite as simple. In particular, the simple approach of replacing variables with constructor terms does not work because one cannot establish *a priori* which constructor should be used to build the value denoted by a given variable.

Example 3.2. *Consider again the list data type, this time with both the cons and the null constructor, and the following set of literals:* $\{cons(x, y) \approx w, cdr(w) \approx cdr(y), y \not\approx null\}$.

The graph for Example 3.2 is shown in Figure 2. Observe that the new graph has nodes for both children of w and y, even though these terms do not all appear in the given set of literals. For the sake of simplicity, we follow Oppen in requiring that every node with at least one child has a complete set of children.

A simple extension of Oppen's algorithm for the case of multiple constructors is proposed in [20]. The idea is to first guess a *type completion*, that is, a labeling of every variable by a constructor, which is meant to constrain a variable to take only values built with the associated constructor. Once all variables are labeled by a single constructor, the Oppen algorithm can be used to determine if the constraints can be satisfied under that labeling.

Unfortunately, the type completion guess can be very expensive in practice. In Example 3.2, there are 7 terms that are not constructor terms and thus could potentially have been

3. A simple example of a cycle is: $cons(x, y) \approx y$.

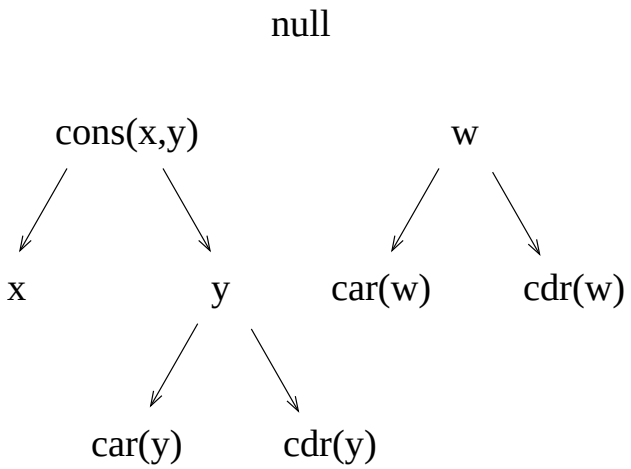

Figure 2. Term graph for Example 3.2

constructed using either constructor. A naive type completion guess would require 2^7 cases. However, most of these cases need not be considered. In fact, we only need to consider which constructor is used to construct the value of y. If y is constructed with *null*, then this contradicts the disequality $y \not\approx \mathit{null}$. On the other hand, if y is constructed with *cons*, then downward closure requires $y \approx \mathit{cdr}(w) \approx \mathit{cdr}(y)$, creating a cycle.

Our presentation combines ideas from previous work as well as introducing some new ones. There is a set of upward and downward closure rules to mimic Oppen's algorithm. The idea of a type completion is replaced by a set of labeling rules that can be used to refine the set of possible constructors for each term (in particular, this allows us to delay guessing as long as possible). And the notion of introducing constructors and eliminating selectors is captured by a set of selector rules. In addition to the presentation, one of our key contributions is to provide precise side-conditions for when case splitting is necessary as opposed to when it can be delayed. The results given in Section 5 show that with the right strategy, significant gains in efficiency can be obtained.

We describe our procedure formally in the following, as a set of derivation rules. We build on and adopt the style of similar rules for abstract congruence closure [1] and syntactic unification [11].

3.2 Definitions and Notation

In the following, we will consider well-sorted formulas over the signature Σ above and an infinite set X of implicitly existential variables. To distinguish these variables, which can occur in formulas given to the decision procedure described below, from other internal variables used by the decision procedure, we will sometimes call the elements of X *input* variables.

Given a set Γ of literals over Σ and variables from X, we wish to determine the satisfiability of Γ in the algebra $\mathcal{R}$.[4] That is, we wish to determine whether there exists a variable

4. In both theory and practice, the satisfiability of arbitrary quantifier-free formulas can be easily determined given a decision procedure for a set of literals. Using the fact that a universal formula $\forall \mathbf{x}\varphi(\mathbf{x})$ is

assignment α, a mapping of input variables to ground terms, such that applying α to Γ results in a set of ground literals all of which are true in $\mathcal{R}$. We will assume for simplicity, and with no loss of generality, that the only occurrences of terms of sort bool are in atoms of the form $is_C(t) \approx \mathsf{true}$, which we will write just as $is_C(t)$.

Following [1], for each sort τ (σ) we will make use of the sets V_τ (V_σ) of *abstraction* variables of sort τ (σ); abstraction variables are disjoint from input variables (variables in Γ) and function as equivalence class representatives for the terms in Γ. We assume an arbitrary, but fixed, well-founded ordering $\succ$ on the abstraction variables that is total on variables of the same sort. We denote the set of all variables (both input and abstraction) in Γ as $\mathcal{V}ar(\Gamma)$. Recall that for each sort τ the set $\mathcal{C}_\tau$ denotes the set of τ's constructors. To simplify the notation we will write $\mathcal{C}_s$ regardless of whether s is a τ-sort or a σ-sort. In the latter case $\mathcal{C}_s$ will denote the empty set. We will write $sort(t)$ to denote the sort of the term t.

The rules make use of three additional constructs that are not in the language of Σ: $\rightarrow$, $\mapsto$, and *Inst*. The symbol $\rightarrow$ is used to represent *oriented* equations. Its left-hand side is a Σ-term t and its right-hand side is an abstraction variable v. The symbol $\mapsto$ denotes *labelings* of abstraction variables with sets of constructor symbols. It is used to keep track of possible constructors for instantiating a τ variable.[5] Finally, the *Inst* construct is used to track applications of the **Instantiate 2** rule given below. It is needed to ensure termination by preventing multiple applications of the rule. It is a unary predicate that is applied only to abstraction variables.

Let Σ^{C} denote the set of all constant symbols in Σ, including nullary constructors. We will denote by Λ the set of all possible literals over Σ and input variables X. Note that this does not include oriented equations ($t \rightarrow v$), labeling pairs ($v \mapsto L$), or applications of *Inst*. In contrast, we will denote by E multisets of literals of Λ, oriented equations, and labeling pairs, and applications of *Inst*. To simplify the presentation, we will consistently use the following meta-variables: c, d denote constants (elements of Σ^{C}) or input variables from X; u, v, w denote abstraction variables; t denotes a *flat term*—i.e., a term all of whose proper sub-terms are abstraction variables—or a label set, depending on the context. $\mathbf{u}, \mathbf{v}$ denote possibly empty sequences of abstraction variables; and $\mathbf{u} \rightarrow \mathbf{v}$ is shorthand for the set of oriented equations resulting from pairing corresponding elements from $\mathbf{u}$ and $\mathbf{v}$ and orienting them so that the left hand variable is greater than the right hand variable according to $\succ$. Finally, $v \bowtie t$ denotes any of $v \approx t$, $t \approx v$, $v \not\approx t$, $t \not\approx v$, or $v \mapsto t$. To streamline the notation, we will sometimes denote function application simply by juxtaposition.

Each rule consists of a premise and one or more conclusions. Each premise is made up of a multiset of literals from Λ, oriented equations, labeling pairs, and applications of *Inst*. Conclusions are either similar multisets or $\bot$, where $\bot$ represents a trivially unsatisfiable formula. As we show later, the soundness of our rule-based procedure depends on the fact that the premise E of a rule is satisfied in $\mathcal{R}$ by a valuation of $\mathcal{V}ar(E)$ iff one of the conclusions E' of the rule is satisfied in $\mathcal{R}$ by an extension of that valuation.

true in a model exactly when $\neg\varphi(\mathbf{x})$ is unsatisfiable in the model, this also provides a decision procedure for universal formulas.

5. To simplify the writing of the rules, some rules may introduce labeling pairs for variables with a non-τ sort, even though these play no role.

3.3 The derivation rules

Our decision procedure consists of the following derivation rules on multisets E.

Abstraction rules

$$\textbf{Abstract 1}\quad \frac{p[c],\ E}{c \to v,\ v \mapsto \mathcal{C}_s,\ p[v],\ E}\quad \text{if}\quad \begin{array}{l} p \in \Lambda,\ c : s, \\ v \text{ fresh from } V_s \end{array}$$

$$\textbf{Abstract 2}\quad \frac{p[C\,\mathbf{u}],\ E}{C\,\mathbf{u} \to v,\ p[v],\ v \mapsto \{C\},\ E}\quad \text{if}\ \ p \in \Lambda,\ C \in \mathcal{C}_\tau\ v \text{ fresh from } V_\tau$$

$$\textbf{Abstract 3}\quad \frac{p[S_C^{(k)}\,u],\ E}{\begin{array}{l} S_C^{(1)}\,u \to v_1, \ldots, S_C^{(n)}\,u \to v_n,\ p[v_k], \\ v_1 \mapsto \mathcal{C}_{s_1}, \ldots, v_n \mapsto \mathcal{C}_{s_n}, E \end{array}}\quad \text{if}\quad \begin{array}{l} p \in \Lambda, \\ C : s_1 \cdots s_n \to \tau, \\ \text{each } v_i \text{ fresh from } V_{s_i} \end{array}$$

The abstraction or *flattening* rules assign a new abstraction variable to every sub-term in the original set of literals. Each rule contains a literal of the form $p[t]$ in the premise and $p[v]$ in the conclusion. The meaning of this notation is that $p[t]$ is some literal containing the term t and $p[v]$ is the literal obtained by replacing every occurrence of t in $p[t]$ with the abstraction variable v. Abstraction variables are used as place-holders or equivalence class representatives for the sub-terms they replace. While we would not expect a practical implementation to actually introduce these variables, it greatly simplifies the presentation of the remaining rules.

The **Abstract 1** rule replaces input variables or constants. **Abstract 2** replaces constructor terms, and **Abstract 3** replaces selector terms. Notice that in each case, a labeling pair for the introduced variables is also created. This corresponds to labeling each sub-term with the set of possible constructors with which it could have been constructed. Also notice that in the **Abstract 3** rule, whenever a selector is applied, we effectively introduce all possible applications of selectors associated with the same constructor. This simplifies the later selector rules and corresponds to the step in the Oppen algorithm which ensures that in the term graph, any node with children has a complete set of children.

Literal level rules

$$\textbf{Orient}\quad \frac{u \approx v,\ E}{u \to v,\ E}\quad \text{if}\ \ u \succ v \qquad\qquad \textbf{Remove 1}\quad \frac{is_C\,v,\ E}{v \mapsto \{C\},\ E}$$

$$\textbf{Inconsistent}\quad \frac{v \not\approx v,\ E}{\bot} \qquad\qquad \textbf{Remove 2}\quad \frac{\neg is_C\,v,\ E}{v \mapsto \mathcal{C}_{sort(v)} \setminus \{C\},\ E}$$

The simple literal level rules are mostly self-explanatory. The **Orient** rule is used to replace an equation between abstraction variables (which every equation eventually becomes after applying the abstraction rules) with an oriented equation. Oriented equations are used in the remaining rules below. The **Inconsistent** rule detects violations of the reflexivity of equality. The **Remove** rules remove applications of testers and replace them with labeling pairs that impose the same constraints.

Upward (i.e., congruence) closure rules

Simplify 1 $\dfrac{u \bowtie t,\ u \to v,\ E}{v \bowtie t,\ u \to v,\ E}$ **Simplify 2** $\dfrac{f\mathbf{u}u\mathbf{v} \to w,\ u \to v,\ E}{f\mathbf{u}v\mathbf{v} \to w,\ u \to v,\ E}$

Superpose $\dfrac{t \to u,\ t \to v,\ E}{u \to v,\ t \to v,\ E}$ if $u \succ v$

Compose $\dfrac{t \to v,\ v \to w,\ E}{t \to w,\ v \to w,\ E}$

These rules are modeled after similar rules for abstract congruence closure in [1]. The **Simplify** and **Compose** rules essentially provide a way to replace any abstraction variable with a smaller (according to $\succ$) one if the two are constrained to be equal. Note that the symbol f in the **Simplify 2** rule refers to an arbitrary function symbol from Σ. The **Superpose** rule merges two equivalence classes if they contain the same term. Congruence closure is achieved by these rules because if two terms are congruent, then after repeated applications of the first set of rules, they will become syntactically identical. Then the **Superpose** rule will merge their two equivalence classes.

Downward (i.e., unification) closure rules

Decompose $\dfrac{C\,\mathbf{u} \to v,\ C\,\mathbf{v} \to v,\ E}{C\,\mathbf{u} \to v,\ \mathbf{u} \to \mathbf{v},\ E}$

Clash $\dfrac{c \to v,\ d \to v,\ E}{\bot}$ if $c, d \in \Sigma^{\mathrm{C}},\ c : \sigma,\ d : \sigma,\ c \neq d$

Cycle $\dfrac{C_n\,\mathbf{u_n}u\mathbf{v_n} \to u_{n-1},\ \ldots,\ C_2\,\mathbf{u_2}u_2\mathbf{v_2} \to u_1,\ C_1\,\mathbf{u_1}u_1\mathbf{v_1} \to u,\ E}{\bot}$ if $n \geq 1$

The main downward closure rule is the **Decompose** rule: whenever two terms with the same constructor are in the same equivalence class, their arguments must be equal. Recall that $\mathbf{u} \to \mathbf{v}$ is shorthand for the set of oriented equations resulting from pairing corresponding elements from $\mathbf{u}$ and $\mathbf{v}$ and orienting them so that the left hand variable is greater than the right hand variable according to $\succ$. The **Clash** rule detects constants that are in the same equivalence class despite the fact that they are disequal in the intended model. The **Cycle** rule detects an inconsistency when a constructor term would have to be equivalent to one of its sub-terms.

Selector rules

Instantiate 1 $\dfrac{S_C^{(1)}\,u \to u_1,\ \ldots,\ S_C^{(n)}\,u \to u_n,\ u \mapsto \{C\},\ E}{C\,u_1 \cdots u_n \to u,\ u \mapsto \{C\},\ E}$ if $C : s_1 \cdots s_n \to \tau,\ n \geq 1$

Instantiate 2 $\dfrac{u \mapsto \{C\},\ E}{\begin{array}{l} C\,u_1 \cdots u_n \to u,\ u \mapsto \{C\},\ \mathit{Inst}(u), \\ u_1 \mapsto \mathcal{C}_{s_1},\ \ldots,\ u_n \mapsto \mathcal{C}_{s_n}, E \end{array}}$ if $\begin{array}{l} C \text{ finite constructor}, \\ C : s_1 \cdots s_n \to \tau, \\ \mathit{Inst}(u) \notin E, \\ u_i \text{ fresh from } V_{s_i} \end{array}$

Collapse 1 $$\frac{C\,u_1 \cdots u_n \to u,\ S_C^{(i)}\,u \to v,\ E}{C\,u_1 \cdots u_n \to u,\ u_i \approx v,\ E}$$

Collapse 2 $$\frac{S_C^{(i)} u \to v,\ u \mapsto L,\ E}{t_C^i \approx v,\ u \mapsto L,\ E} \quad \text{if } C \notin L$$

Rule **Instantiate 1** is used to eliminate selectors by replacing the argument of the selectors with a new term constructed using the appropriate constructor. Only terms that have selectors applied to them can be instantiated and then only once they are uniquely labeled. Notice that all of the selectors applied to the term are eliminated at the same time. This is why the entire set of selectors is introduced in the **Abstract 3** rule.

For completeness, a term labeled with a finite constructor must be instantiated even if no selectors are applied to that term. This is accomplished by rule **Instantiate 2**. The side conditions are similar to those in **Instantiate 1**, except that this rule *only* applies to terms labeled with finite constructors. The *Inst* predicate ensures that the rule is applied at most once for each such term.

The **Collapse** rules eliminate selectors when the result of their application can be determined. In **Collapse 1**, a selector $S_C^{(i)}$ is applied to a term constructed with constructor C. In this case, the selector expression is replaced by the appropriate argument of the constructed term. In **Collapse 2**, a selector $S_C^{(i)}$ is applied to a term which must have been constructed with a constructor other than C. In this case, the designated term t_C^i for the selector replaces the selector expression.

Labeling rules

Refine $$\frac{v \mapsto L_1,\ v \mapsto L_2,\ E}{v \mapsto L_1 \cap L_2,\ E}$$ **Empty** $$\frac{v \mapsto \emptyset,\ E}{\bot} \quad \text{if } v : \tau$$

Split 1 $$\frac{S_C^{(i)}\,u \to v,\ u \mapsto \{C\} \cup L,\ E}{S_C^{(i)}\,u \to v,\ u \mapsto \{C\},\ E \qquad S_C^{(i)}\,u \to v,\ u \mapsto L,\ E} \quad \text{if } L \neq \emptyset$$

Split 2 $$\frac{u \mapsto \{C\} \cup L,\ E}{u \mapsto \{C\},\ E \qquad u \mapsto L,\ E} \quad \text{if } \begin{array}{l} L \neq \emptyset, \\ \{C\} \cup L \text{ all finite constructors} \end{array}$$

The **Refine** rule simply combines labeling constraints that may arise from different sources for the same abstraction variable. **Empty** enforces the constraint that every τ-term must be constructed by some constructor. The splitting rules are used to refine the set of possible constructors for a term and are the only rules that cause branching. If a term labeled with only finite constructors cannot be eliminated in some other way, **Split 2** must be applied until it is labeled with a single constructor. For other terms, the **Split 1** rule only needs to be applied to distinguish the case of a selector being applied to the "right" constructor from a selector being applied to the "wrong" constructor. On either branch, one of the **Collapse** rules will apply immediately. We discuss this further in Section 5, below.

$null \to v_1$	$v_1 \mapsto \{null\}$	$v_5 \to v_4$
$x \to v_2$	$v_2 \mapsto \{cons, null\}$	$v_9 \to v_7$
$y \to v_3$	$v_3 \mapsto \{cons, null\}$	$v_3 \not\approx v_1$
$cons(v_2, v_3) \to v_4$	$v_4 \mapsto \{cons\}$	
$w \to v_5$	$v_5 \mapsto \{cons, null\}$	
$car(v_5) \to v_6$	$v_6 \mapsto \{cons, null\}$	
$cdr(v_5) \to v_7$	$v_7 \mapsto \{cons, null\}$	
$car(v_3) \to v_8$	$v_8 \mapsto \{cons, null\}$	
$cdr(v_3) \to v_9$	$v_9 \mapsto \{cons, null\}$	

Figure 3. Example 3.2 after **Abstraction** and **Orient**

$null \to v_1$	$v_1 \mapsto \{null\}$	$v_5 \to v_4$
$x \to v_2$	$v_2 \mapsto \{cons, null\}$	$v_9 \to v_7$
$y \to v_3$	$v_3 \mapsto \{cons, null\}$	$v_3 \not\approx v_1$
$cons(v_2, v_3) \to v_4$	$v_4 \mapsto \{cons\}$	
$w \to v_4$	$v_6 \mapsto \{cons, null\}$	
$cons(v_6, v_7) \to v_4$	$v_7 \mapsto \{cons, null\}$	
$car(v_3) \to v_8$	$v_8 \mapsto \{cons, null\}$	
$cdr(v_3) \to v_7$		

Figure 4. Figure 3 after congruence rules, **Refine**, and **Instantiate 1**

3.4 An Example Using the Rules

Let us revisit Example 3.2 and see how the rules work on this example. Recall that we have the following set of literals: $\{cons(x, y) \approx w, cdr(w) \approx cdr(y), y \not\approx null\}$. After applying the **Abstraction** and **Orient** rules, we have the set of literals shown in Figure 3. Next, the **Simplify** and **Compose** rules can be used to replace all occurrences in the first two columns of v_5 and v_9 with v_4 and v_7 respectively. Then, **Refine** can be used to eliminate two of the labeling pairs. Notice that after replacing v_5 with v_4, v_4 can be instantiated (the side conditions of **Instantiate 1** are satisfied). The resulting set of literals is shown in Figure 4. At this point, there are two *cons* terms equivalent to v_4, so the **Decompose** rule applies, yielding two new oriented equations: $v_6 \to v_2$ and $v_7 \to v_3$. These can again be used together with the congruence rules and **Refine** to simplify the other literals. The resulting set is shown in Figure 5.

At this point, the only rule that can be applied is the **Split 1** rule. And only v_3 satisfies the necessary condition of having a selector applied to it. There are two cases. Consider first the case where $v_3 \mapsto \{cons\}$. In this case, **Instantiate 1** applies, yielding $cons(v_8, v_3) \to v_3$ which yields $\bot$ by the **Cycle** rule. In the other case, we have $v_3 \mapsto \{null\}$. This time, since *null* is a finite constructor, we can apply **Instantiate 2** to get $null \to v_3$. The **Superpose** rule then gives $v_3 \to v_1$. This can be used together with $v_3 \not\approx v_1$ to deduce $\bot$ (via the **Simplify 1** and **Inconsistent** rules).

$null \to v_1$	$v_1 \mapsto \{null\}$	$v_5 \to v_4$
$x \to v_2$	$v_2 \mapsto \{cons, null\}$	$v_9 \to v_3$
$y \to v_3$	$v_3 \mapsto \{cons, null\}$	$v_3 \not\approx v_1$
$cons(v_2, v_3) \to v_4$	$v_4 \mapsto \{cons\}$	
$car(v_3) \to v_8$	$v_8 \mapsto \{cons, null\}$	
$cdr(v_3) \to v_3$		
$v_6 \to v_2$		
$v_7 \to v_3$		

Figure 5. Figure 4 after **Decompose** and congruence rules

4. Correctness

The satisfiability in $\mathcal{R}$ of a set Γ of Σ-literals with variables in X can be checked by applying exhaustively to Γ the derivation rules in the previous section. This set of rules is very flexible in that the rules can be applied in *any* order and still yield a decision procedure for the satisfiability in $\mathcal{R}$. No specific rule application strategy is needed to achieve termination, soundness or completeness. We formalize this in the following in terms of a suitable notion of *derivation* for these rules.

A *derivation tree* (for a set Γ of Σ-literals with variables in X) is a finite tree with root Γ such that for each internal node E of the tree, its children are the conclusions of some rule whose premise is E. A *refutation tree* (for Γ) is a derivation tree all of whose leaves are $\bot$. We say that a node in a derivation tree is *(ir)reducible* if (n)one of the derivation rules applies to it. A *derivation* is a sequence of derivation trees starting with the single-node tree containing Γ, where each tree is derived from the previous one by the application of a rule to one of its leaves. A *refutation* is a finite derivation ending with a refutation tree.

For a multiset E of literals, a variable assignment α is a mapping from $\mathcal{V}ar(E)$ into the elements of $\mathcal{R}$ that is well-sorted (i.e., $sort(x) = sort(\alpha(x))$ for every $x \in \mathcal{V}ar(E)$). If α is a variable assignment, then we denote by $\overline{\alpha}$ the homomorphic extension of α that maps arbitrary terms into elements of $\mathcal{R}$. We say that α satisfies $s \approx t$ iff $\overline{\alpha}(s)$ equals $\overline{\alpha}(t)$.

For convenience, we extend the notion of satisfiability and well-sortedness to the extra-logical constructs. The oriented equation $t \to v$ is well-sorted iff t and v have the same sort. Furthermore, α satisfies $t \to v$ in $\mathcal{R}$ iff α satisfies the equation $t \approx v$ in $\mathcal{R}$. The expression $v \mapsto L$, labeling a variable v of sort s with the set L of constructor symbols, is considered to be well-sorted if $L \subseteq \mathcal{C}_s$. The valuation α satisfies a labeling pair $v \mapsto L$ in $\mathcal{R}$ if v is of a non-τ sort or α satisfies the formula $is_C(v) \approx \mathsf{true}$ for some $C \in L$. An application of *Inst* is always well-sorted and satisfied by every variable assignment. We start with a lemma that gives a couple of useful invariants.

Lemma 4.1. *Let $E_0, E_1, \ldots,$ be a branch on a derivation tree. Then the following holds for all $i \geq 0$.*

1. *If E_0 is well-sorted, then for all i, E_i is well-sorted.*
2. *For all $u \to v \in E_i$, we have $u \succ v$.*

Proof. A simple examination of each of the rules confirms that these invariants are maintained. □

Before proving termination, we need the following additional notation. For each constructor $C \in \Sigma$, let $|C|$ denote 0 if C is infinite and otherwise denote the size of the (finite) set containing all ground constructor terms whose top symbol is C, and all of their sub-terms.

Proposition 4.2 (Termination)**.** *Every derivation is finite.*

Proof. Given a derivation tree, let $E_0, E_1, \ldots$ be any branch of the tree that does not end with $\bot$. It is enough to show that the branch can be mapped to a strictly descending sequence in a well-founded ordering. The ordering $\succ_l$ we will use is a lexicographic ordering over tuples of the form (s, t, S, T, M, A, n) where s, t, T, and n are natural numbers, S is a multiset of naturals, M is a multiset of symbols from Σ and variables from X, and A is a multiset of abstraction variables. The ordering $\succ_l$ is the one induced by the well-founded orderings $>, >, >_{\mathrm{m}}, >, \sqsupset_{\mathrm{m}}, \succ_{\mathrm{m}}, >$ where

- $>$ is the usual ordering of the natural numbers,
- $>_{\mathrm{m}}$ is the multiset ordering induced by $>$,
- $\sqsupset_{\mathrm{m}}$ is the multiset ordering induced by some arbitrary well-founded ordering of the set $\Sigma \cup X$, and
- $\succ_{\mathrm{m}}$ is the multiset ordering induced by the given ordering $\succ$ over the abstraction variables.

The descending sequence $(s_i, t_i, S_i, T_i, M_i, A_i, n_i)$ for $i = 0, 1, \ldots$ is defined as follows. Recall that Σ-literals do not include oriented equations, labeling pairs, or applications of *Inst*.

- s_i is the number of selector symbols in the Σ-literals of E_i;
- t_i is total number of selector symbols appearing in E_i;
- S_i is the multiset consisting of the *sizes* of the Σ-literals of E_i, where by size we mean the number of occurrences of symbols from Σ (including $\approx$) and input variables, but not of abstraction variables;
- T_i is the sum of all $|v|_i$ for all abstraction variables $v \in \mathcal{V}ar(E_i)$ that do not appear as an argument to *Inst* in E_i where, for each v, $|v|_i = \sum_{C \in L_i} |C|$ and L_i is the union of all label sets for v in E_i;
- M_i is the multiset of occurrences of symbols from Σ and input variables from X in Σ-literals or oriented equations of E_i;
- A_i is the multiset of all the occurrences of abstraction variables in E_i;
- n_i is the number of label occurrences in E_i, that is, occurrences of the constructor symbols in labeling pairs of E_i.

We show that for all consecutive nodes E_i, E_{i+1} in the branch $(s_i, t_i, S_i, T_i, M_i, A_i, n_i) \succ_1 (s_{i+1}, t_{i+1}, S_{i+1}, T_{i+1}, M_{i+1}, A_{i+1}, n_{i+1})$. The proof is by cases, depending on the rule used to derive E_{i+1} from E_i.

1. The cases corresponding to the rules **Inconsistent**, **Clash**, **Cycle**, and **Empty** do not apply since they all have conclusion $\bot$.

2. Suppose one of the rules **Abstract 1**, **Abstract 2**, **Orient**, **Remove 1**, or **Remove 2** was applied. Each of these rules leaves s_i and t_i unchanged while removing at least one Σ-symbol or input variable from a literal (without changing the other literals). In each of these cases, $S_i >_{\mathrm{m}} S_{i+1}$.

3. With **Abstract 3**, the number of selector symbols appearing in literals is reduced by one, so $s_i > s_{i+1}$.

4. With all the congruence closure rules except for **Superpose** when the term t in the rule is not an abstraction variable, the only change is the replacement of an abstraction variable by another abstraction variable which is smaller by Lemma 4.1(2). Thus, s_i, t_i, S_i, T_i, and M_i remain the same, while $A_i \succ_{\mathrm{m}} A_{i+1}$. In the case where **Superpose** is applied and t is not an abstraction variable, t must contain a symbol from $\Sigma \cup X$. If t contains a selector, then $s_i = s_{i+1}$ and $t_i > t_{i+1}$. Otherwise, $M_i \sqsupset_{\mathrm{m}} M_{i+1}$ (it is easy to see that s_i, t_i, S_i, and T_i remain the same in this case).

5. The **Decompose** rule does not change the values of s_i, t_i, S_i, or T_i. However, it does eliminate one occurrence of a constructor symbol. Hence, $M_i \sqsupset_{\mathrm{m}} M_{i+1}$.

6. Now consider the selector rules. With **Instantiate 1**, since the constructor C in the rule has positive arity (i.e., $n \geq 1$) then $s_i = s_{i+1}$ and $t_i > t_{i+1}$. With **Instantiate 2**, s_i, t_i and S_i are unchanged but

$$T_{i+1} = (T_i - |u|_i) + \sum_{k=1}^{n} |u_k|_{i+1} \ .$$

   It is not difficult to see that $|u|_i > \sum_{k=1}^{n} |u_k|_{i+1}$. Thus, $T_i > T_{i+1}$.

7. With the collapse rules, exactly one selector symbol is eliminated from (a non-literal of) E_i, so $s_i = s_{i+1}$ and $t_i > t_{i+1}$.[6.]

8. Finally, consider the labeling rules. The **Refine** rule eliminates an occurrence of an abstraction variable. Hence certainly $A_i \succ_{\mathrm{m}} A_{i+1}$. All the preceding components of the tuple are unchanged with the possible exception of T_i which may get smaller when $L_1 \neq L_2$. The split rules both produce two conclusions, each of which has fewer constructors appearing in labels than in the premise. Furthermore, this is the only change, so T_i either decreases or is unchanged, $n_i > n_{i+1}$ and everything else is unchanged.

□

6. Note that $s_i = s_{i+1}$ with **Collapse 2** because, by definition, t_C^i is a ground term with no selectors.

The soundness of the decision procedure is based on the following result.

Lemma 4.3. *The premise E of a derivation rule is satisfied in $\mathcal{R}$ by a valuation α of $\mathcal{V}ar(E)$ iff one of the conclusions E' of the rule is satisfied in $\mathcal{R}$ by an extension of α to $\mathcal{V}ar(E')$.*

Proof. Again, the proof is by cases.

(Abstraction rules) The if direction is immediate. For the other direction, for **Abstract 1**, suppose that the premise is satisfied by α in $\mathcal{R}$. We extend α by setting v to the value of c under $\mathcal{R}, \alpha$. Consider the labeling pair $v \mapsto \mathcal{C}_s$ in the conclusion. It is trivially satisfied if v is of a non-τ sort. When v is of sort τ, it is satisfied as a consequence of the first axiom (schema) in $\mathcal{R}$'s specification and the fact that $\alpha(v)$ is a constructor term by Lemma 2.2. With this observation, it is clear that the extended variable assignment satisfies the conclusion. For **Abstract 2**, a similar argument shows that an extended variable assignment which assigns v to the value of $C\,\mathbf{u}$ under $\mathcal{R}, \alpha$ must satisfy the conclusion. For **Abstract 3**, the argument is again similar, but this time we must extend α to map each v_i to the value of $S_C^{(i)}\,u$ under $\mathcal{R}, \alpha$.

(Literal level rules) The case of **Orient** and **Inconsistent** is obvious. For **Remove 1** the claim follows by definition of satisfaction for labeling pairs. For **Remove 2** we rely on the fact that $\mathcal{R}, \alpha$ satisfies $is_C\,v$ exactly when it satisfies $v \mapsto \{C\}$, for any C. This follows from Lemma 2.2 and the first and second axiom schemas.

(Upward closure rules) The claim follows from basic properties of equality.

(Downward closure rules) The result follows from Lemma 2.2 and basic properties of the term algebra $\mathcal{T}(\Omega)$.

(Selector rules) In case of **Instantiate 1** and **2** the claims follow from the definition of satisfaction for labeling pairs, the *Inst* predicate, the first three axiom schemas, and Lemma 2.2. For **Collapse 1** the result follows by the third axiom schema; for **Collapse 2** by the fourth schema, Lemma 2.2 and the definition of satisfaction for labeling pairs.

(Labeling rules) The claim follows by simple Boolean reasoning and the definition of satisfaction for labeling pairs. □

Proposition 4.4 (Soundness)**.** *If a set E_0 has a refutation tree, then it is unsatisfiable in $\mathcal{R}$.*

Proof. By structural induction on refutation trees and the previous lemma. □

To prove completeness we will rely on the next three lemmas. First we need a couple of definitions. If E is a multiset of literals, we write $\sim_E$ for the equivalence relation induced by oriented equations in E. We also define $lbls_E(u)$ as the intersection of all label sets L where $v \mapsto L$ appears in E for some $v \sim_E u$.

Lemma 4.5. *Suppose E is a node in a derivation tree and that E contains an oriented equation of the form $S_C^{(i)}\,u \to v$ for some C (of arity n), u, v, and i, where $1 \leq i \leq n$. We will call this an* oriented selector equation*. Then at least one of the following is also true:*

(i) E also contains an oriented equation of the form $C\,\mathbf{w} \to u'$ for some $\mathbf{w}$ and u' where $u' \sim_E u$.

(ii) $C \notin \mathit{lbls}_E(u)$

(iii) *There exist* $u_1, \ldots, u_n$ *and* $v_1, \ldots, v_n$ *such that for each* $1 \leq k \leq n$, $S_C^{(k)}\, u_k \to v_k \in E$ *and* $u_k \sim_E u$.

Proof. The proof is by induction on derivation trees. The base case is trivial since the root of a derivation tree has no oriented equations. For the inductive case, we consider each of the rules. First note that if a rule does not introduce, change, or delete any oriented selector equations and furthermore does not delete or change any oriented equations of the form $C\,\mathbf{w} \to u'$, then the property is trivially preserved. This covers the following rules: **Abstract 1**, **Abstract 2**, the literal level rules, **Simplify 1**, **Clash**, **Cycle**, **Instantiate 2**, and the labeling rules. We now consider the others:

Abstract 3. This rule introduces new oriented selector equations. For these, it is easy to see that condition (iii) is satisfied. It is also easy to see that the property is preserved for any other oriented selector equations.

Simplify 2. This rule may change an oriented selector equation from $S_C^{(i)}\, u \to v$ to $S_C^{(i)}\, u' \to v$ when $u \to u'$. However, in this case, we have $u \sim_E u'$, and it follows that the property is preserved.

Superpose. If we have two oriented selector equations: $S_C^{(i)}\, u \to v$ and $S_C^{(i)}\, u \to v'$, with $v \succ v'$, then the first of these may be eliminated by the **Superpose** rule. If the eliminated oriented selector equation was needed to fulfill condition (iii) for some other oriented selector equation in the premise, then we must ensure that the property still holds in the conclusion. However, notice that $S_C^{(i)}\, u \to v'$ may be used instead and so the property holds.

Compose. Suppose $S_C^{(i)}\, u \to v$ is rewritten to $S_C^{(i)}\, u \to v'$. It is easy to see that the property holds for the new oriented selector equation for the same reasons as it did for the old. Also, if the old oriented selector equation was used to fulfill condition (iii) for some other oriented selector equation, then the new one does so as well.

Decompose. This rule may eliminate an oriented equation of the form $C\,\mathbf{w} \to u'$ which might affect condition (i) for some oriented selector equation. However, it only does so when there exists another oriented equation of the form $C\,\mathbf{v} \to u'$ that is not eliminated. This can be used to satisfy condition (i) instead.

Instantiate 1. This rule eliminates oriented selector equations which could affect condition (iii) for some other oriented selector equation. However, it also introduces an oriented equation of the form $C\,\mathbf{w} \to u$, so condition (i) will now apply to such oriented selector equations.

Collapse 1. This rule eliminates an oriented selector equation which could affect condition (iii) for some other oriented selector equation. However, it is easy to see that because we have an oriented equation of the form $C\,\mathbf{w} \to u$, condition (i) must apply to such oriented selector equations.

Collapse 2. This rule eliminates an oriented selector equation which could affect condition (iii) for some other oriented selector equation. However, it is easy to see that because $C \notin \mathit{lbls}_E(u)$, condition (ii) must apply to such oriented selector equations. □

Lemma 4.6. *No irreducible leaf* E *in a derivation tree contains occurrences of selector symbols.*

Proof. The claim is trivially true if $E = \{\bot\}$, so assume that $E \neq \{\bot\}$. Since E is irreducible, by the rule **Abstract 3** and Lemma 4.1(1), every occurrence of a selector in E must be in an oriented equation of the form $S_C^{(i)}\, u \to v$, for some constructor $C : s_1 \cdots s_n \to \tau$ and an abstraction variable u of sort τ. So assume that $S_C^{(i)}\, u \to v \in E$. By Lemma 4.5, we know that one of three conditions applies. The first case is that condition (i) holds: E also contains an oriented equation of the form $C\,\mathbf{w} \to u'$ for some $\mathbf{w}$ and u' where $u' \sim_E u$. Since E is irreducible, we must have that $u' = u$, but then **Collapse 1** applies, contradicting the irreducibility of E. The second case is (ii): $C \notin \mathit{lbls}_E(u)$. Again, because E is irreducible, this means that E contains $u \mapsto L$ and $C \notin L$. Thus, **Collapse 2** applies, again a contradiction. Finally, the third case is (iii): there exist $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ such that for each $1 \leq k \leq n$, $S_C^{(k)}\, u_k \to v_k \in E$ and $u_k \sim_E u$. Again, because E is irreducible, we must have that $u_k = u$ for each k. Also, since (ii) does not apply and **Split 1** cannot be applied, E must contain $u \mapsto \{C\}$. But this means that **Instantiate 1** applies, again yielding a contradiction.

□

Lemma 4.7. *Every irreducible leaf E other than $\{\bot\}$ in a derivation tree is satisfiable in $\mathcal{R}$.*

Proof. We build a valuation α of $\mathcal{V}ar(E)$ that satisfies E in $\mathcal{R}$. To start, let

$$\begin{aligned} V &= \{v \mid t \to v \in E \text{ for some } t\} \\ T_v &= \{t \mid t \to v \in E\} \text{ for all } v \in V \end{aligned}$$

Observe that the sets T_u and T_v are disjoint for all distinct u and v, otherwise E would contain two equations of the form $t \to u$ and $t \to v$, and so would be reducible by **Superpose**. Furthermore, for all $v \in V$, T_v contains at most one non-variable term. To see that, recalling that E contains no occurrences of selector symbols by Lemma 4.6, assume that T_v contains a constant symbol c of sort σ. Clearly it cannot contain a term t of sort other than σ because otherwise either $c \to v$ or $t \to v$ would be ill-sorted, which is not possible by Lemma 4.1(1). The only other possible terms of sort σ are other constant symbols d. But then, if $d \to v$ were in E, **Clash** would apply to E. Now assume that T_v contains a term of the form $C\,\mathbf{u}$. Again by well-sortedness, it is enough to argue that T_v contains no additional terms of the form $C'\,\mathbf{u}'$ of the same sort as v's. But such terms cannot be in T_v. If $C = C'$ then **Decompose** applies. If $C \neq C'$, notice that whenever an oriented equation of the form $C\,\mathbf{u} \to v$ is introduced, we also have $v \mapsto \{C\}$. Since label sets never grow, at some point we have to have had both $v \mapsto \{C\}$ and $v \mapsto \{C'\}$. Since **Refine** must have been applied to these two labeling pairs, E must now contain $v \mapsto \emptyset$ and is thus reducible by **Empty**.

Now consider the relation $\lessdot$ over V defined as follows:

$$u \lessdot v \text{ iff } E \text{ contains an equation of the form } C\,\mathbf{u}u\mathbf{u}' \to v.$$

By the **Cycle** rule and the assumptions on E, the finite relation $\lessdot$ is acyclic and hence well founded. We can define a valuation α of V into $\mathcal{R}$[7] by well founded induction on $\lessdot$.

7. Whose universe, recall, is the term algebra $\mathcal{T}(\Omega)$.

Let $\{v_1, \ldots, v_n\}$ be the set of all the $\lessdot$-minimal elements of V such that for $i = 1, \ldots, n$, $c_i \to v_i \in E$ with c_i a constant symbol—possibly a nullary constructor. For $i = 1, \ldots, n$ we define $\alpha(v_i) = c_i$. Now let $\{v_{n+1}, \ldots, v_{n+k}\}$ be the remaining $\lessdot$-minimal elements of V. For $i = n+1, \ldots, n+k$, if v_i is of sort σ, we define $\alpha(v_i) = d_i$ where d_i is some constant of sort σ in $\mathcal{T}(\Omega) \setminus \{\alpha(v_1), \ldots, \alpha(v_{n+i-1})\}$ [8]. If v_i is of some sort τ, we know by a previous observation that $v_i \mapsto L \in E$. Note that by the **Empty** and the **Split** rules, $C \in L$ for some non-nullary C. Moreover, C must be an infinite constructor, or otherwise an equation of the form $C\,\mathbf{u} \to v_k$ would be in E by **Instantiate 2**, making v_k non-$\lessdot$-minimal. We then define $\alpha(v_k) = C\,t_1 \cdots t_m$ where C is some infinite constructor in L of arity $m > 0$ and $C\,t_1 \cdots t_m$ is some term in $\mathcal{T}(\Omega) \setminus \{\alpha(v_1), \ldots, \alpha(v_{n+k-1})\}$.

We are now left with defining $\alpha(v)$ for all non-minimal $v \in V$. If v is non-minimal, then there must be an equation of the form $C\,u_1 \cdots u_k \to v$ in E for some constructor C. Furthermore, $k \geq 1$ (otherwise v would be minimal) and $u_i \lessdot v$ for all $i = 1, \ldots, k$. We then define $\alpha(v) = C\,\alpha(u_1) \cdots \alpha(u_k)$.

We now show by induction on $\lessdot$ that the valuation α just defined is an injection of V into $\mathcal{T}(\Omega)$. Let u, v be two distinct elements of V of the same sort.

If u and v are both $\lessdot$-minimal in the set $\{v_1, \ldots, v_n\}$ defined earlier, then $\alpha(u) \neq \alpha(v)$ because the sets $T_{v_1}, \ldots, T_{v_n}$ are mutually disjoint. If one (or both) of them is in $\{v_{n+1}, \ldots, v_{n+k}\}$ then $\alpha(u) \neq \alpha(v)$ by construction.

If u, say, is not $\lessdot$-minimal, then both u and v must be of some sort τ. It follows that $\alpha(u), \alpha(v)$ are terms of the form $C\,\alpha(u_1) \cdots \alpha(u_n)$, $C'\,\alpha(v_1) \cdots \alpha(v_{n'})$, respectively, with $n, n' \geq 1$. Now, if $C \neq C'$, then $\alpha(u)$ and $\alpha(v)$ are trivially distinct terms. If $C = C'$, then $n = n'$; however, $u_i \neq v_i$ for some i otherwise $C\,u_1 \cdots u_n \to u$ and $C\,u_1 \cdots u_n \to v$ would be in E and **Superpose** would apply. If u_i and v_i are distinct then by induction $\alpha(u_i)$ and $\alpha(v_i)$ are distinct, therefore $\alpha(u)$ and $\alpha(v)$ are distinct as well.

Now we can extend α to the whole $\mathcal{V}ar(E)$ by defining it for the remaining (input or abstraction) variables of E. Each such variable x occurs in an equation of the form $x \to v$ in E. Hence we define $\alpha(x) = \alpha(v)$. For later reference, let α' be the homomorphic extension of α to the set of Σ-terms over $\mathcal{V}ar(E)$.

The valuation α satisfies every element e of E. This is immediate if e has the form $v \approx v$ or the form $v \mapsto L$ with $v : \sigma$. If e has the form $u \not\approx v$ with u, v distinct, then α satisfies e for being injective over the abstraction variables of E. If e has the form $t \to v$, then α satisfies e because $\alpha(v) = \alpha'(t)$ by construction. If e has the form $v \mapsto L$ where v has sort τ consider the following two cases. If $C\,u_1 \cdots u_k \to v \in E$ for some $C\,u_1 \cdots u_k$ then it is not difficult to show that L must be $\{C\}$. But then $\alpha(v) = C\,\alpha(u_1) \cdots \alpha(u_k)$ by construction. If there is no $C\,u_1 \cdots u_k \to v \in E$, then $\alpha(v)$ is defined as some term $C\,t_1 \cdots t_k$ where $C \in L$. In both cases, it is then immediate that α satisfies $v \mapsto L$.

To conclude the proof it is enough to observe that, for being irreducible, E can only contain elements of the forms listed above. □

Proposition 4.8 (Completeness)**.** *If a set E_0 is unsatisfiable in $\mathcal{R}$, then it has a refutation.*

Proof. We prove the contrapositive of the proposition. Assume that E_0 has no refutations. By Proposition 4.2, there is a derivation tree for E_0 with an irreducible leaf $E \neq \{\bot\}$. By

8. Using the assumption that all sorts σ are infinite.

Lemma 4.7, E is satisfiable in $\mathcal{R}$. It follows by a repeated application of Lemma 4.3 that E_0 is satisfiable in $\mathcal{R}$ as well. □

5. Strategies and Efficiency

It is not difficult to see that the problem of determining the satisfiability of an arbitrary set of literals is NP-complete. A subset of the problem (a simple case with two constructors) was shown to be NP-hard in [14]. To see that it is in NP, we note that given a type completion, no additional splits are necessary, and the remaining rules can be carried out in polynomial time. However, as with other NP-complete problems (Boolean satisfiability being the most obvious example), the right strategy can make a significant difference in practical efficiency.

5.1 Strategies

A *strategy* is a predetermined methodology for applying the rules. Before discussing our recommended strategy, it is instructive to look at the closest related work. Oppen's original algorithm is roughly equivalent to the following: After abstraction, apply the selector rules to eliminate all instances of selector symbols. Next, apply upward and downward closure rules (the bidirectional closure). As you go, check for conflicts using the rules that can derive $\bot$. We will call this the *basic strategy*. Note that it excludes the splitting rules: because Oppen's algorithm assumes a single constructor, the splitting rules are never used. A generalization of Oppen's algorithm is mentioned in [20]. They add the step of initially guessing a "type completion". To model this, consider the following simple **Split** rule:

$$\textbf{Split} \quad \frac{u \mapsto \{C\} \cup L,\ E}{u \mapsto \{C\},\ E \qquad\qquad u \mapsto L,\ E} \quad \text{if} \quad L \neq \emptyset$$

Now consider a strategy which invokes **Split** greedily (after abstraction) until it no longer applies and then follows the basic strategy. We will call this strategy the *greedy splitting* strategy.

One of the key contributions of this paper is to recognize that the greedy splitting strategy can be improved in two significant ways. First, the simple **Split** rule should be replaced with the smarter **Split 1** and **Split 2** rules. Second, these rules should be delayed as long as possible. We call this the *lazy splitting* strategy. The lazy strategy reduces the size of the resulting derivation in two ways. First, notice that **Split 1** is only enabled when some selector is applied to u. By itself, this eliminates many needless case splits. Second, by applying the splitting rules *lazily* (in particular by first applying selector rules), it may be possible to avoid splitting completely in many cases. We already saw in Section 3 that Example 3.2 can be solved using only a single case split, instead of the 2^7 splits required by a naive type completion. Here, we look at another example that emphasizes the advantages of lazy splitting.

Example 5.1. *Suppose we have the following simple tree data type:*

$$\mathit{tree} \quad := \quad \mathit{node}(\mathit{left} : \mathit{tree},\ \mathit{right} : \mathit{tree}) \mid \mathit{leaf};$$

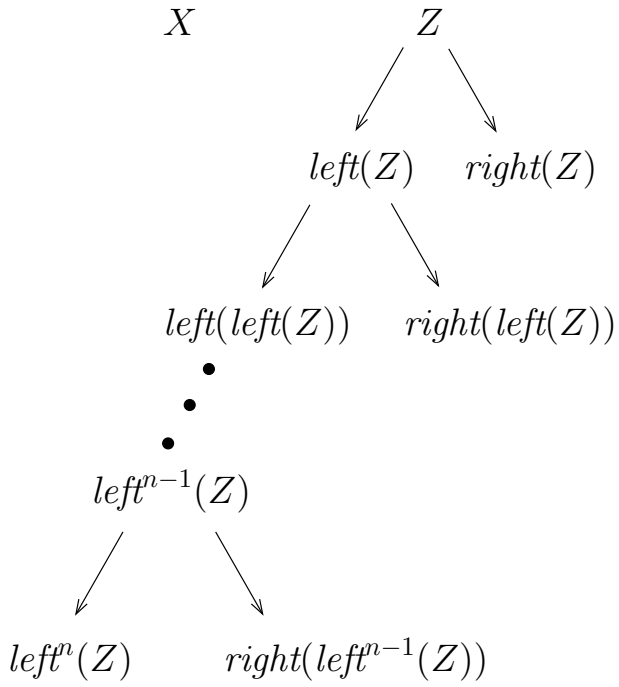

Figure 6. Term graph for Example 5.1

Let leaf be the designated term for both selectors and then consider the following set of literals: $\{left^n(Z) \approx X, is\_node(Z), Z \approx X\}$.

A term graph for Example 5.1 is shown in Figure 6. After applying all available rules except for the splitting rules, the resulting set of literals looks like this:

$$\begin{array}{ll}\{ & Z \to u_0, X \to u_0, u_0 \mapsto \{node\}, node(u_1, v_1) \to u_0, u_n \to u_0, \\ & left(u_1) \to u_2, \ldots, left(u_{n-1}) \to u_n, u_1 \mapsto \{leaf, node\}, \ldots, u_n \mapsto \{leaf, node\}, \\ & right(u_1) \to v_2, \ldots, right(u_{n-1}) \to v_n, v_1 \mapsto \{leaf, node\}, \ldots, v_n \mapsto \{leaf, node\}\},\end{array}$$

Notice that there are $2n$ abstraction variables labeled with two labels each. If we eagerly applied the naive **Split** rule at this point, the derivation tree would reach size $O(2^{2n})$.

Suppose, on the other hand, that we use the lazy strategy. First notice that **Split 1** can only be applied to n of the abstraction variables ($u_i, 1 \leq i \leq n$). Thus the more restrictive side-conditions of **Split 1** already reduce the size of the problem to at worst $O(2^n)$ instead of $O(2^{2n})$. However, by only applying it lazily, we do even better: suppose we split on u_i. The result is two branches, one with $u_i \mapsto \{node\}$ and the other with $u_i \mapsto \{leaf\}$. The second branch induces a cascade of (at most n) applications of Collapse 2 which in turn results in $u_k \mapsto \{leaf\}$ for each $k > i$. This eventually results in $\bot$ via the Empty and Refine rules. The other branch contains $u_i \mapsto \{node\}$ and results in the application of the **Instantiate 1** rule, but little else, and so we will have to split again, this time on a different u_i. This process will have to be repeated until we have split on all of the u_i. At that point, there will be a cycle from u_0 back to u_0, and so we will derive $\bot$ via the Cycle rule.

Table 1. Greedy vs. Lazy Splitting

Worst Case Splits	Num. of Tests	Sat	Unsat	Greedy		Lazy	
				Splits	Time (s)	Splits	Time (s)
0	4416	306	4110	0	24.6	0	24.6
1-5	2520	2216	304	6887	16.8	2414	17.0
6-10	692	571	121	4967	5.8	1597	5.7
11-20	178	112	66	2422	2.3	517	1.6
21-100	145	73	72	6326	4.5	334	1.1
101+	49	11	38	16593	9.8	73	0.3

Because each split only requires at most $O(n)$ rules and there are $n-1$ splits, the total size of the derivation tree will be $O(n^2)$. In fact, if we start at u_{n-1} and work our way down, each split will take only $O(1)$, so the total size of the derivation tree will be $O(n)$.[9.]

5.2 Experimental Results

We have implemented both the lazy and the greedy splitting strategies in the theorem prover CVC3 [4]. We are not aware of any application-based benchmarks for this theory, but fortunately this is not necessary for comparing the two splitting strategies. What is necessary is to have some benchmarks that require non-trivial amounts of splitting. To produce such benchmarks, we randomly generated conjunctions of literals over the mutually recursive inductive data types *nat*, *list*, and *tree* mentioned in the introduction.[10.]

As expected, most of the benchmarks are quite easy. In fact, over half of them are solved without any case splitting at all. However, a few of them did prove to be somewhat challenging, at least in terms of the number of splits required. We tried both the greedy and lazy strategies on all benchmarks and categorized the benchmarks according to how many case splits were required in the worst case by either strategy.

Table 1 shows the results. As expected, for easy benchmarks that don't require many splits, the two algorithms perform almost identically. However, as the difficulty increases, the lazy strategy performs much better. For the hardest benchmarks, the lazy strategy outperforms the greedy strategy by more than an order of magnitude. Notice that the disparity in case splits is even greater: for nontrivial benchmarks, the number of case splits taken by the lazy strategy is always much less than that taken by the greedy strategy: over two orders of magnitude for the hardest benchmarks.

9. This does not mean the total time is necessarily $O(n)$. In general, processing a node includes bidirectional closure and checking for cycles which requires $O(n)$ steps (see [15], for example). So the total processing time is bounded by $O(n \cdot m)$, where m is the size of the derivation tree. In this case, the total time is bounded by $O(n^2)$.
10. See http://www.cs.nyu.edu/~barrett/datatypes for details on the benchmarks and results.

6. Extending the Algorithm

In this section we briefly discuss several ways in which our algorithm can be used as a component in solving a larger or related problem.

6.1 Finite Sorts

Here we consider how to lift the limitation that each of $\sigma \in \{\sigma_1, \ldots, \sigma_r\}$ is infinite valued. Since we have no such restrictions on τ-sorts, the idea is to simply replace such a σ by a new τ-like sort τ_σ, whose set of constructors (all of which will be nullary) will match the domain of σ. For example, if σ is a finite scalar of the form $\{1, \ldots, n\}$, then we can let

$$\tau_\sigma ::== null_1 \mid \ldots \mid null_n$$

We then proceed as before, after replacing all occurrences of σ by τ_σ and each i by $null_i$.

6.2 Simulating Partial Function Semantics

As mentioned earlier, it is not clear how best to interpret the application of a selector to the wrong constructor. One compelling approach is to interpret selectors as partial functions. An evaluation of a formula then has three possible outcomes: true, false, or undefined. This approach may be especially valuable in a verification application in which application of selectors is required to be guarded so that no formula should ever be undefined. This can easily be implemented by employing the techniques described in [5]: given a formula to check, a special additional formula called a type-correctness condition is computed (which can be done in time and space linear in the size of the input formula). These two formulas can then be checked using a decision procedure that interprets the partial functions (in this case, the selectors) in some arbitrary way over the undefined part of the domain. The result can then be interpreted to reveal whether the formula would have been true, false, or undefined under the partial function semantics. A similar approach is advocated in [9].

6.3 Cooperating with other Decision Procedures

A final point is that that our procedure has been designed to integrate easily into a Nelson-Oppen-style framework for cooperating decision procedures [13]. In the many-sorted case, the key theoretical requirements (see [18]) for two decision procedures to be combined are that the signatures of their theories share at most sort symbols and each theory is *stably infinite* over the shared sorts.[11] A key operational requirement is that the decision procedure is also able to easily compute and communicate equality information.

The theory of $\mathcal{R}$ (i.e., the set of sentences true in $\mathcal{R}$) is trivially stably infinite over the sorts $\sigma_1, \ldots, \sigma_r$ and over any τ-sort containing a non-finite constructor—as all such sorts denote infinite sets in $\mathcal{R}$. Also, in our procedure the equality information is eventually completely captured by the oriented equations produced by the derivation rules, and so entailed equalities can be easily detected and reported.

For a detailed and formal discussion of how to integrate a rule-based decision procedure such as this one into a general framework combining Boolean reasoning and multiple decision

11. A many-sorted theory T is stably infinite over a subset S of its sorts if every quantifier-free formula satisfiable in T is satisfiable in a model of T where the sorts of S denote infinite sets.

procedures, we refer the reader to our related work in [2]. Note that, in particular, this work shows how the internal theory case splits can be delegated on demand to the Boolean engine; this is the implementation strategy followed in CVC3.

7. Conclusion

We have presented an algorithm for deciding a theory of inductive data types. Novel features of our treatment include the ability to handle mutually recursive, many-sorted types, a simpler presentation of the theory, an abstract declarative algorithm, and smarter splitting rules which can greatly enhance efficiency. The algorithm has been proved correct and is implemented in the theorem prover CVC3.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *JAR*, 31:129–168, 2003.

[2] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in sat modulo theories. In M. Hermann and A. Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06), Phnom Penh, Cambodia*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, 2006.

[3] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. In *Proceedings of PDPAR*, Aug. 2006.

[4] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany*, Lecture Notes in Computer Science. Springer, 2007. (to appear).

[5] S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. In *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR '04)*, volume 125(3) of *ENTCS*, pages 13–23, July 2005.

[6] M. P. Bonacina and M. Echenim. Generic theorem proving for decision procedures. Technical report, Università degli studi di Verona, 2006. Available at `http://profs.sci.univr.it/~echenim/`.

[7] W. Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[8] D. Kozen. Complexity of finitely presented algebras. In *Proceedings of the 9-th Annual ACM Symposium on Theory of Computing*, pages 164–177, 1977.

[9] V. Kuncak and M. Rinard. On the theory of structural subtyping. Technical Report MIT-LCS-TR-879, Massachusetts Institute of Technology, 2003.

[10] A. I. Mal'cev. On elementary theories of locally free universal algebras. *Soviet Mathematical Doklady*, 2(3):768–771, 1961.

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

[12] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky, D. V. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Claredon Press, 1992.

[13] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[14] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–364, April 1980.

[15] D. C. Oppen. Reasoning about recursively defined data structures. *JACM*, 27(3):403–411, July 1980.

[16] T. Rybina and A. Voronkov. A decision procedure for term algebras with queues. *ACM Transactions on Computational Logic*, 2(2):155–181, Apr. 2001.

[17] R. Shostak. Deciding combinations of theories. *JACM*, 31(1):1–12, 1984.

[18] C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In J. Alferes and J. Leite, editors, *Proceedings of JELIA '04*, volume 3229 of *LNAI*, pages 641–653. Springer, 2004.

[19] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *JACM*, 34(2):492–510, Apr. 1987.

[20] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. In *Proceedings of IJCAR '04 LNCS 3097*, pages 152–167, 2004.

[21] T. Zhang, H. B. Sipma, and Z. Manna. Term algebras with length function and bounded quantifier alternation. In *Proceedings of TPHOLs*, 2004.