An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types

Clark Barrett[1] Igor Shikanian[1] Cesare Tinelli[2]
[1]New York University, barrett|igor@cs.nyu.edu
[2]The University of Iowa, tinelli@cs.uiowa.edu

November 22, 2005

New York University Technical Report: TR2005-878

Abstract

The theory of recursive data types is a valuable modeling tool for software verification. In the past, decision procedures have been proposed for both the full theory and its universal fragment. However, previous work has been limited in various ways, including an inability to deal with multiple constructors, multi-sorted logic, and mutually recursive data types. More significantly, previous algorithms for the universal case have been based on inefficient nondeterministic guesses and have been described in fairly complex procedural terms.

We present an algorithm which addresses these issues for the universal theory. The algorithm is presented declaratively as a set of abstract rules which are terminating, sound, and complete. We also describe strategies for applying the rules and explain why our recommended strategy is more efficient than those used by previous algorithms. Finally, we discuss how the algorithm can be used within a broader framework of cooperating decision procedures.

1 Introduction

Recursive 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 recursive data types thus provides an important tool for the analysis and verification of programs.

Perhaps the best-known example of a simple recursive 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. In order to retrieve 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.

More generally, we are interested in any set of (possibly mutually) recursive data types, each of which contains 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 constructer. As an example, of the more general case, consider a set of three recursive data types: *nat*, *list*, and *tree*. *nat* has two constructors: *zero*, which takes no arguments (we call such a constructor a *nullary* constructor or *constant*); and *succ*, which takes a single argument of type *nat*, and with a 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:

leaf, a constant; and *node*, which takes two arguments, the first of type *nat*, and the second of type *list*, with corresponding selectors *data* and *children* respectively. We can represent this set of types using the following convenient notation based on that used in functional programming languages:

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

The testers for this set of data types are *issucc*, *iszero*, *iscons*, *isnull*, *isnode*, and *isleaf*.

Propositions about a set of recursive 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:

$$\forall\, x : nat.\ succ(x) \neq zero,$$
$$\forall\, x : list.\ x = null \vee \exists\, y : nat, z : list.\ x = cons(y, z),$$
$$\forall\, x : tree.\ isnode(x) \rightarrow (data(x) = zero \vee issucc(data(x))).$$

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, referring the reader instead to related work such as [4, 16, 17].

There are three main contributions of this work over earlier work on the topic. First, our setting is more general: we allow mutually recursive types, each with multiple constructors, selectors, and testers, and we use the more general setting of multi-sorted logic. The rationale for a multi-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 leads to an algorithm which is more efficient than that given in previous work.

Related Work. Term algebras over constructors provide the natural intended model for recursive data types. The historically foundational decidability and quantifier elimination results for term algebras can be found in [6]. In other early work, [5] 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 [12]. Another contribution to solving satisfiability of equations over term algebras is given in [15], which extends the language with a powerful *subterm relation* predicate. In [4] two dual axiomatizations of term algebras are presented, one with constructors only, the other with selectors and testers only.

More recently, several papers by Zhang et al. [16, 17] explore decision procedures for a single recursive data type. These papers focus on ambitious schemes for quantifier elimination and combinations with other theories. Their work is largely orthogonal to ours since we focus on the quantifier-free decision problem which is only mentioned briefly in their work.

Other work with an emphasis on the quantifier-free case includes that done by Nelson and Oppen in 1980[10, 11]. In [11], Oppen gives a decision procedure for a single recursive data type with a

single constructor. In [10], the theory of lists is shown to be NP-complete when it includes the constructor *null*. Thus, an instance of the class of theories covered by the current paper already yields NP-completeness. As will be evident, the problem solved in this paper is also NP-complete.

Shostak gives an algorithm for a simple theory of lists without *null* in [13]. He also claims there is a generalization to arbitrary recursive data types. However, the claim is unsubstantiated and it is unclear how to generalize to the case of multiple constructors.

Paper Organization. Section 2 describes our formulation of the first order theory of recursive 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 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 Recursive Data Types

Previous work on recursive data types (RDTs) [16, 17] uses first-order axiomatizations in an attempt to capture the main properties of a recursive data type and reason about it. Unfortunately, the resulting axiomatization is somewhat complicated. This axiomatic approach makes the study of decision procedures for RDTs and their correctness more difficult than it needs to be.

We find it simpler and cleaner to use a semantic approach instead, as is done in algebraic specification. A set of RDTs 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 RDTs then amounts to reasoning about formulas that are true in this particular initial model.

2.1 Specifying RDTs

We formalize RDTs in the context of many-sorted equational logic (see [8] 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 RDTs. We also allow and $r \geq 0$ additional (non-RDT) sorts $\sigma_1, \ldots, \sigma_r$. We will denote by s, possibly with subscripts and superscripts, any sort in the signature other than bool, 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 RDTs under consideration. We assume for each τ_i $(1 \leq i \leq p)$ a set of $m_i \geq 1$ *constructors* of τ_i. We denote these symbols as C^i_j, where j ranges from 1 to m_i. We denote the arity of C^i_j as n^i_j (0-arity constructors are also called nullary constructors or constants) and its sort as $s^i_{j,1} \times \cdots \times s^i_{j,n^i_j} \to \tau_i$. For each constructor C^i_j, we have a set of *selectors*, which we denote as $S^i_{j,k}$, where k ranges from 1 to n^i_j, of sort $\tau_i \to s^i_{j,k}$. Finally, for each constructor, there is a *tester*.[1] $isC^i_j : \tau_i \to \mathsf{bool}$.

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 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

[1] To simplify some of the proofs, and without loss of generality, we use functions to bool instead of predicates for the testers.

could use all the numerals as the constants of sort σ_1. Having all these constants in the signature is really not necessary for our approach, but it simplifies the exposition. The real constraint is that the sorts $\sigma_1, \ldots, \sigma_r$ be infinite. We will see in Section 6, however, that our approach can be easily extended to the case in which some of these sorts are finite.

To summarize, the set of function symbols of the signature Σ consists of:

$C^i_j : s^i_{j,1} \times \cdots \times s^i_{j,n^i_j} \to \tau_i$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i$,
$S^i_{j,k} : \tau_i \to s^i_{j,k}$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i,\ k = 1, \ldots, n^i_j$,
$isC^i_j : \tau_i \to \mathsf{bool}$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i$,
$\mathsf{true} : \mathsf{bool},\ \mathsf{false} : \mathsf{bool}$,
An infinite number of constants for each σ_l, for $l = 1, \ldots, r$.

As usual in many-sorted equational logic, we also have $p + r + 1$ equality symbols (one for each sort), all written as $\approx$.

Our procedure requires one additional constraint on the set of RDTs: It must be *well-founded*. Informally, this means that each sort must contain terms that are not cyclic or infinite. More formally, we have the following definitions by simultaneous induction over constructors and sorts:

- a constructor C^i_j is well-founded if all of its argument sorts are well-founded;
- the sorts $\sigma_1, \ldots, \sigma_r$ are all well-founded;
- a sort τ_i is well-founded if at least one of its constructors is well-founded.

We require that every sort be well-founded according to the above definition.

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

- a constructor is *finite* if it is nullary or if all of its argument sorts are finite.
- a sort τ_i is *finite* if all of its constructors are finite, and is *infinite* otherwise.
- the sorts $\sigma_1, \ldots, \sigma_r$ are all infinite;

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 RDTs with functions and predicates denoted by the symbols of Σ is specified by the following set $\mathcal{E}$ of (universally quantified) equations. For reasons explained below, we assume that associated with every selector $S^i_{j,k} : \tau_i \to s^i_{j,k}$ is a distinguished ground term of sort $s^i_{j,k}$ containing no selectors (or testers), which we denote by $t^i_{j,k}$.

Equational Specification of the RDT: for $i = 1, \ldots, p$:

$$\forall x_1, \ldots, x_{n^i_j}.\ isC^i_j(C^i_j(x_1, \ldots, x_{n^i_j})) \approx \mathsf{true} \qquad (\text{for } j = 1, \ldots, m_i)$$
$$\forall x_1, \ldots, x_{n^i_{j'}}.\ isC^i_j(C^i_{j'}(x_1, \ldots, x_{n^i_{j'}})) \approx \mathsf{false} \qquad (\text{for } j, j' = 1, \ldots, m_i,\ j \neq j')$$
$$\forall x_1, \ldots, x_{n^i_j}.\ S^i_{j,k}(C^i_j(x_1, \ldots, x_{n^i_j})) \approx x_k \qquad (\text{for } k = 1, \ldots, n^i_j,\ j = 1, \ldots, m_i)$$
$$\forall x_1, \ldots, x_{n^i_{j'}}.\ S^i_{j,k}(C^i_{j'}(x_1, \ldots, x_{n^i_{j'}})) \approx t^i_{j,k} \qquad (\text{for } j, j' = 1, \ldots, m_i,\ j \neq j')$$

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 [4, 16, 17]) where the application of a wrong selector is treated as the identity function. There are several reasons for this difference. First, in a multi-sorted logic, the identity function approach does not work in general because the result may be ill-sorted. Second, by choosing a small designated term (such as a constant when possible), fewer case splits are required by the decision procedure, making the procedure more efficient. Finally, as described in Section 6.2, the difference is immaterial for formulas in which the appropriateness of the selector can be guaranteed.

By standard results in universal algebra we know that $\mathcal{E}$ admits an *initial model* $\mathcal{R}$. We refer the reader to [8] 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 [8]. □

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). □

Informally, the previous lemma means that $\mathcal{R}$ does in fact capture the set of RDTs 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_i is generated using just its constructors, and that distinct ground constructor terms of sort τ_i 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_i denotes an infinite data type if and only if τ_i 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

Before giving a formal description of the algorithm, which is quite technical, we start with an informal overview based on examples. Our procedure builds on the algorithm by Oppen [11] for

a single type with a single constructor. Consider, for example, the *list* datatype without *null* and the following set of literals: $\{cons(x,y) \approx z, car(w) \approx x, cdr(w) \approx y, w \not\approx z\}$. The idea of Oppen's algorithm is to use a graph which relates terms according to their meaning in the intended model. Thus, $cons(x,y)$ is a parent of x and y and $car(w)$ and $cdr(w)$ are children of w. The equations induce an equivalence relation on the nodes of the graph. The Oppen algorithm proceeds by performing *upwards* (congruence) and *downwards* (unification) closure on the graph and then checking for cycles[2] or for a violation of any disequalities. For our example, upwards closure results in the conclusion $w \approx z$, which contradicts the disequality $w \not\approx z$.

Suppose we replace $w \not\approx z$ with $v \approx w$ and $y \not\approx cdr(v)$ in the previous set. The new graph has a node for v, with $car(v)$ as its left child. A right child node with $cdr(v)$ is then added for completeness. Now, downwards closure forces $car(v) \approx car(w) \approx x$ and $cdr(v) \approx cdr(w) \approx y$, contradicting the disequality $y \not\approx cdr(v)$.

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, for the first set of literals above, 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 $\{cons(x,y) \approx z, w \approx cons(s,t), x \approx s, t \approx y, w \not\approx z\}$. In general, this approach only requires downwards closure.

Unfortunately, with the addition of 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* whether the value denoted by a given variable is built with one constructor or another. A simple extention of Oppen's algorithm for the case of multiple constructors is proposed [16]. 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. The problem is that the type completion guess is very expensive.

Our strategy combines ideas from all of these algorithms. 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. As we will see in Sections 4 and 5, the flexibility of our rules allows our algorithm to be both complete and efficient.

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 [7].

3.1 Definitions and Notation

In the following, we will consider well-sorted formulas over the signature Σ above and an infinite set X of 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 (i.e., equations or negated equations) over Σ and variables from X, we wish to determine the satisfiability of Γ in the algebra $\mathcal{R}$.[3] 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

[2] A simple example of a cycle is: $cons(x,y) \approx z, car(x) \approx z$.

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

$isC_k^j(t) \approx \mathsf{true}$, which we will write just as $isC_k^j(t)$. We will abbreviate negated equations $\neg(t_1 \approx t_2)$ between non-Boolean terms as $t_1 \not\approx t_2$.

Following [1], we will make use of the sets V_{τ_i} (V_{σ_i}) of *abstraction* variables of sort τ_i (σ_i); abstraction variables are disjoint from input variables (variables in Γ) and function as equivalence class representatives for the terms in Γ. We denote the set of all variables (both input and abstraction) in E as $\mathcal{V}ar(E)$. We will use the expression $labels(\tau_i)$ for the set $\{C_1^i, \ldots, C_{m_i}^i\}$ and define $labels(\sigma_l)$ to be the empty set of labels for each σ_l. 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. Given a variable assignment α into the elements of $\mathcal{R}$, we say that α satisfies $t \rightarrow v$ in $\mathcal{R}$ iff α satisfies the equation $t \approx v$ in $\mathcal{R}$.

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_i variable.[4] A variable assignment α satisfies a labeling pair $v \mapsto \{C_{j_1}^i, \ldots, C_{j_n}^i\}$ in $\mathcal{R}$ if α satisfies the formula $isC_{j_1}^i(v) \vee \cdots \vee isC_{j_n}^i(v)$ in $\mathcal{R}$.

Finally, the *Inst* construct is used to track applications of the Instantiate rules given below. It is needed to ensure termination by preventing multiple applications of the same Instantiate rule. It is a unary predicate that is applied only to abstraction variables. It is always satisfied by every variable assignment.

Let Σ^{C} denote the set of all constant symbols in Σ, including 0-arity 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, 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.

In the derivation rules we assume an arbitrary, but fixed, well-founded ordering $\succ$ on the abstraction variables that is total on variables of the same sort. Each rule consists of a premise and one or more conclusions. Each premise is made up of a multiset of literals, 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 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 $\mathcal{V}ar(E')$.

3.2 The derivation rules

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

[4] 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.

Abstraction rules

Abstract 1 $$\frac{p[c],\ E}{c \to v,\ v \mapsto \mathit{labels}(s),\ p[v],\ E} \quad \text{if} \quad \begin{array}{l} p \in \Lambda,\ c : s, \\ v \text{ fresh from } V_s \end{array}$$

Abstract 2 $$\frac{p[C^i_j \mathbf{u}],\ E}{C^i_j \mathbf{u} \to v,\ p[v],\ v \mapsto \{C^i_j\},\ E} \quad \text{if } p \in \Lambda,\ v \text{ fresh from } V_{\tau_i}$$

Abstract 3 $$\frac{p[S^i_{j,\kappa} u],\ E}{\begin{array}{l} S^i_{j,1} u \to v_1,\ \ldots,\ S^i_{j,n^i_j} u \to v_{n^i_j},\ p[v_\kappa], \\ v_1 \mapsto \mathit{labels}(s_1),\ \ldots, v_{n^i_j} \mapsto \mathit{labels}(s_{n^i_j}),\ E \end{array}} \quad \text{if} \quad \begin{array}{l} p \in \Lambda, S^i_{j,k} : \tau_i \to s_k, \\ \text{each } v_\iota \text{ fresh from } V_{s_\iota} \end{array}$$

The abstraction or *flattening* rules essentially perform a pre-processing step, assigning a new abstraction variable to every sub-term in the original set of literals. Abstraction variables are then used as place-holders or equivalence class representatives for those sub-terms. 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 constuctor 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 $S^i_{j,k}$ 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

Orient $$\frac{u \approx v,\ E}{u \to v,\ E} \quad \text{if } u \succ v$$

Inconsistent $$\frac{v \not\approx v,\ E}{\bot}$$

Remove 1 $$\frac{\mathit{is}C^i_j\, v,\ E}{v \mapsto \{C^i_j\},\ E}$$

Remove 2 $$\frac{\neg \mathit{is}C^i_j\, v,\ E}{v \mapsto \mathit{labels}(\mathit{sort}(v)) \setminus \{C^i_j\},\ E}$$

The simple literal level rules are almost 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 reflexivity. 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 $$\frac{u \bowtie t,\ u \to v,\ E}{v \bowtie t,\ u \to v,\ E}$$

Simplify 2 $$\frac{f \mathbf{u} u \mathbf{v} \to w,\ u \to v,\ E}{f \mathbf{u} v \mathbf{v} \to w,\ u \to v,\ E}$$

Superpose $$\frac{t \to u,\ t \to v,\ E}{u \to v,\ t \to v,\ E} \quad \text{if } u \succ v$$

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

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 known to be equal. 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 $\dfrac{C^i_j\mathbf{u} \to v,\ C^i_j\mathbf{v} \to v,\ E}{C^i_j\mathbf{u} \to v,\ \mathbf{u} \to \mathbf{v},\ E}$ **Clash 1** $\dfrac{C^i_j\mathbf{u} \to v,\ C^i_{j'}\mathbf{v} \to v,\ E}{\bot}$ if $j \neq j'$

Clash 2 $\dfrac{c \to v,\ d \to v,\ E}{\bot}$ if $c, d \in \Sigma^C,\ c : \sigma,\ d : \sigma,\ c \neq d$

Cycle $\dfrac{C^{i_n}_{j_n}\mathbf{u_n}u\mathbf{v_n} \to u_{n-1},\ \ldots,\ C^{i_2}_{j_2}\mathbf{u_2}u_2\mathbf{v_2} \to u_1,\ C^{i_1}_{j_1}\mathbf{u_1}u_1\mathbf{v_1} \to u,\ E}{\bot}$ 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. The **Clash** rules simply detect instances of terms that are in the same equivalence class that must be disequal in the intended model. The **Cycle** rule detects the (inconsistent) cases in which a term would have to be cyclical.

Selector rules

Instantiate 1 $\dfrac{S^i_{j,1}u \to u_1,\ \ldots,\ S^i_{j,n^i_j}u \to u_{n^i_j},\ u \mapsto \{C^i_j\},\ E}{C^i_j u_1 \cdots u_{n^i_j} \to u,\ u \mapsto \{C^i_j\},\ \mathit{Inst}(u),\ E}$ if $\mathit{Inst}(u) \notin E$

Instantiate 2 $\dfrac{v \mapsto \{C^i_j\},\ E}{\begin{array}{l} C^i_j u_1 \cdots u_{n^i_j} \to v,\ \mathit{Inst}(v),\ E \\ u_1 \mapsto \mathit{labels}(s^i_{j,1}),\ \ldots,\ u_{n^i_j} \mapsto \mathit{labels}(s^i_{j,n^i_j}) \end{array}}$ if $\begin{array}{l} \mathit{Inst}(v) \notin E, \\ v \mapsto L \notin E, \\ C^i_j \text{ finite constructor}, \\ S^a_{b,c}(v) \to v' \notin E, \\ u_k \text{ fresh from } V_{s^i_{j,k}} \end{array}$

Collapse 1 $\dfrac{C^i_j u_1 \cdots u_{n^i_j} \to u,\ S^i_{j,k}u \to v,\ E}{C^i_j u_1 \cdots u_{n^i_j} \to u,\ u_k \approx v,\ E}$

Collapse 2 $\dfrac{S^i_{j,k}u \to v,\ u \mapsto L,\ E}{t^i_{j,k} \approx v,\ u \mapsto L,\ E}$ if $C^i_j \notin L$

Rule **Instantiate 1** is used to eliminate selectors by replacing the argument of the selectors with a new term constructed using the appropriate constructor. Notice that only terms that have selectors applied to them can be instantiated and then only once they are unambiguously labeled. 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. Rule **Instantiate 2** is used for finite constructors. For completeness, terms labeled with finite constructors must always be instantiated, even when no selectors are applied to them.

The **Collapse** rules eliminate selectors when the result of their application can be determined. In **Collapse 1**, a selector is applied to a term known to be equal to a constructor of the "right" type.

In this case, the selector expression is replaced by the appropriate argument of the constructor. In **Collapse 2**, a selector is applied to a term which must have been constructed with the "wrong" constructor. In this case, the designated term $t^i_{j,k}$ for the selector replaces the selector expression.

Labeling rules

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

$$\textbf{Split 1}\quad \frac{S^i_{j,k}(u) \to v,\ u \mapsto \{C^i_j\} \cup L,\ E}{S^i_{j,k}(u) \to v,\ u \mapsto \{C^i_j\},\ E \qquad S^i_{j,k}(u) \to v,\ u \mapsto L,\ E}\quad \text{if } L \neq \emptyset$$

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

The **Refine** rule simply combines labeling constraints that may arise from different sources for the same equivalence class. **Empty** enforces the constraint that every τ-term must be constructed by some constructor. The **Split** 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 unambiguously. 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 vs 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.

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) is a finite tree with root Γ and 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.

Before proving correctness, we start with a lemma that gives a few useful invariants. Since the first property below deals with well-sortedness, we first define what it means for the extra-logical constructs to be well-sorted: The oriented equation $t \to v$ is well-sorted if t and v have the same sort. The expression $v \mapsto L$, labeling the variable v with the set L of constructor symbols, is considered to be well-sorted if $L \subseteq \mathit{labels}(\mathit{sort}(v))$. Applications of *Inst* are always well-sorted.

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 introduce the following definitions. For an infinite constructor C, define $|C| = 0$. For a finite constructor C^i_j, define $|C^i_j|$ to be 1 if C^i_j is nullary and $\sum_{k=1}^{n^i_j} |s^i_{j,k}|$ otherwise, where for a finite sort τ_i, we define $|\tau_i| = \sum_{j=1}^{m_i} |C^i_j| + 1$.

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

Proof. It is enough to show that each branch $E_0, E_1, \ldots$ of a derivation tree can be mapped to a (strictly) descending sequence in a well-founded ordering.

Let Ξ be the set of constructor, selector, and constant symbols from Σ together with the input variables from X. Then let $\sqsupset$ be any well-founded ordering of the elements of Ξ.

For $i \geq 0$, Let S_i be a pair consisting of first, the number of selector symbols in the Σ-literals of E_i and second, the total number of selector symbols appearing in E_i. Let N_i be the multiset consisting of the sizes of the Σ-literals of E_i, where by size we mean the number of occurrences of both symbols from Σ (including $\approx$) and input variables, but not of abstraction variables.

Now, for each abstraction variable v, let $|v|_i = \sum_{C \in L} |C| + 1$, where L is the intersection of all labels for v in E_i. Define V_i to be 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.

Let M_i be the multiset of occurrences of symbols from Ξ in either Σ-literals of E_i or in oriented equations from E_i. Let O_i be the multiset of all the occurrences of abstraction variables in E_i. Finally, let n_i be the number of label occurrences in E_i, that is, occurrences of the constructor symbols in labeling pairs of E_i.

Let $>_{\mathrm{m}}$, $\sqsupset_{\mathrm{m}}$, and $\succ_{\mathrm{m}}$ be the multiset orderings induced respectively by the usual ordering $>$ over the natural numbers, the ordering $\sqsupset$ above, and the given ordering $\succ$ over the abstraction variables. Let $>_2$ be the lexicographic ordering of pairs of naturals induced by $>$. Let $\succ_{\mathrm{l}}$ be the lexicographic ordering of pairs of naturals, tuples of naturals, multisets of symbols of Ξ, multisets of naturals, multisets of abstraction variables, and naturals induced by $>_2, >_{\mathrm{m}}, >, \sqsupset_{\mathrm{m}}, \succ_{\mathrm{m}}, >$. Observe that $\succ_{\mathrm{l}}$ is well-founded. We will show that given some E_i, either $E_{i+1} = \bot$ (in which case the branch terminates trivially) or $(S_i, N_i, V_i, M_i, O_i, n_i) \succ_{\mathrm{l}} (S_{i+1}, N_{i+1}, V_{i+1}, M_{i+1}, O_{i+1}, n_{i+1})$. The proof is by cases, considering each of the rules.

The Inconsistent, Clash 1, Clash 2, Cycle, and Empty rules are trivial, since they have the conclusion $\bot$.

Suppose Abstract 1, Abstract 2, Orient, Remove 1, or Remove 2 is applied. Each of these rules leaves S_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, $N_i >_{\mathrm{m}} N_{i+1}$. For the case of Abstract 3, the number of selector symbols appearing in literals is reduced by 1, so $S_i >_2 S_{i+1}$.

Suppose one of the congruence closure rules is applied. In each case, with the exception of Superpose when t is not an abstraction variable, the only change is the replacement of an abstraction variable by another smaller abstraction variable. We know the replacement is smaller by Lemma 4.1(2). Thus, S_i, N_i, V_i, and M_i remain the same, while $O_i \succ_{\mathrm{m}} O_{i+1}$. In the case where Superpose is applied and t is not an abstraction variable, t must contain a symbol from Ξ. If t contains a selector, then clearly $S_i >_2 S_{i+1}$. Otherwise, $M_i \sqsupset_{\mathrm{m}} M_{i+1}$ (it is easy to see that S_i, N_i and V_i remain the same in this case).

Now consider the Decompose rule. Decompose does not change the values of S_i, N_i or V_i. However, it does eliminate one instance of the C^i_j symbol so that $M_i \sqsupset_{\mathrm{m}} M_{i+1}$.

Now consider the selector rules. For the Instantiate 1 rule, if $k_j > 0$, then $S_i >_2 S_{i+1}$. If $k_j = 0$, S_i and N_i are unchanged but $V_{i+1} = V_i - |u|_i$. By definition, $|u|_i$ must be positive. For the Instantiate 2 rule, S_i and N_i are unchanged. But $V_{i+1} = V_i - |v|_i + \sum_{k=1}^{n_j^i} |u_k|_{i+1}$. By definition, $|v|_i = |C_j^i| + 1 = \sum_{k=1}^{n_j^i} |s_{j,k}^i| + 1 = \sum_{k=1}^{n_j^i} |u_k|_{i+1} + 1$. Thus, $V_{i+1} < V_i$.

For the collapse rules, exactly one selector symbol is eliminated, so that $S_i >_2 S_{i+1}$. Note that in particular, for rule Collapse 2, by definition t_i^j must be a ground term containing no selectors, so the symbols introduced by t_i^j can only be constructor and constant symbols.

Finally, consider the labeling rules. The Refine rule eliminates an occurrence of an abstraction variable (so that $O_i \succ_{\mathrm{m}} O_{i+1}$) while leaving S_i, N_i, V_i, and M_i unchanged. 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 S_i, N_i, M_i, and O_i are unchanged, V_i either decreases or is unchanged, and $n_i > n_{i+1}$.

Since each rule either terminates the branch or moves downward in a well-founded ordering, every branch must be finite. □

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. For each of the Abstract rules, the if direction is immediate. In the other direction, for the Abstract 1 rule, suppose that the premise is satisfied by α in $\mathcal{R}$. We extend α by setting v to the value of c under $\mathcal{R}, \alpha$. Notice that the labeling pair in the conclusion must be satisfied with any assignment. This is trivially the case if v is not of non-τ sort. When v is of sort τ_i, it is a consequence of the Axiom (schema) 1 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 the Abstract 2 rule, a similar argument shows that an extended variable assignment which assigns v to the value of $C_j^i\mathbf{u}$ under $\mathcal{R}, \alpha$ must satisfy the conclusion. For the Abstract 3 rule, the argument is again similar, but this time we must extend α to map each v_ι to the value of $S_{j,\iota}^i u$ under $\mathcal{R}, \alpha$.

Now consider the literal level rules. The Orient and Inconsistent rules are obvious. Remove 1 follows by definition of satisfaction for labeling pairs. The Remove 2 rule relies on the fact that for any v of sort τ_i, $IsC_j^i v$ holds for exactly one pair $\langle i, j\rangle$. This follows from Lemma 2.2 and Axioms 1 and 2.

In each of the congruence rules, the result follows from basic properties of equality. For the downward closure rules, the result follows from Lemma 2.2 and basic properties of the term algebra $\mathcal{T}(\Omega)$.

For the Instantiate rules, the result follows from the definition of satisfaction for labeling pairs and the *Inst* predicate, and Axioms 1, 2, and 3. For Collapse 1 the result follows by Axiom 3, and for Collapse 2 by Axiom 4, Lemma 2.2 and the definition of satisfaction for labeling pairs.

Finally, the labeling rules follow 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. The proof is immediate by structural induction and the previous lemma. □

To prove completeness we will rely on the next two lemmas.

Lemma 4.5. *No irreducible leaf E in a derivation tree contains occurrences of selector symbols.*

Proof. The claim is trivially true is $E = \{\bot\}$, so assume that $E \neq \{\bot\}$. Since E is irreducible, by the Abstract 3 rule and Lemma 4.1(1), any occurrence of a selector in E must be in an oriented equation of the form $S^i_{j,k}(u) \to v_k$, where u is an abstraction variable of sort τ_i. So assume by contradiction that $S^i_{j,k}(u) \to v_k \in E$. By the Abstract, Refine, Simplify 1, and Empty rules we also know that u has at least one label in E, i.e., $u \mapsto L \in E$ with $L \neq \emptyset$. Furthermore, by the Split 1 rule, L must be a singleton, and in particular, by the Collapse 2 rule, it must be $\{C^i_j\}$. We also know that no equation of the form $C^i_j\mathbf{u} \to u$ (with u fixed) is in E or in any predecessor node of E. In fact, an equation of that form, once introduced, is either replaced by the rules by one of the same form (i.e., $C^i_j\mathbf{u}' \to u$, for some $\mathbf{u}'$) or by one of the form $C^i_j\mathbf{u}' \to u'$. The latter case can only happen as a consequence of the Superpose or Compose rules, which however then introduce the oriented equation $u \to u'$. Such an equation in turn can only be replaced by one the form $u \to u''$. Therefore, if $C^i_j\mathbf{u} \to u$ occurred in one of the ancestors of E in the derivation tree, then either some $C^i_j\mathbf{w} \to u$ or some $C^i_j\mathbf{w} \to w$ and $u \to w$ would occur in E. But this is not possible because then either Collapse 1 or Simplify 2 rule would respectively apply to $S^i_{j,k}(u) \to v_k$.

Now, observe that $S^i_{j,k}(u) \to v_k$ can only be the result of a sequence of upward closure rules applied to an equation of the form $S^i_{j,k}(u') \to v'_k$ introduced by the Abstract 3 rule. It is easy to see that such closure rules apply in the same way to *all* the equations $S^i_{j,1}(u') \to v'_1, \ldots, S^i_{j,n^i_j}(u') \to v'_{n^i_j}$ introduced by Abstract 3. From the absence of equations of the form $C^i_j\mathbf{u} \to u$ in the branch it follows that E must contain $S^i_{j,1}(u) \to v_1, \ldots, S^i_{j,n^i_j}(u) \to v_{n^i_j}$. But then the Instantiate rule applies to E, again contradicting the assumption that E is irreducible. □

Lemma 4.6. *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 the Superpose rule. 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 4.5, 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 2 would apply to E. Now assume that T_v contains a term of the form $C^i_j\mathbf{u}$. Again by well-sortedness, it is enough to argue that T_v contains no additional terms of the form $C^i_{j'}\mathbf{v}$. But such terms cannot be in T_v because otherwise either the Decompose or the Clash 1 rule would apply.

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

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

By the Cycle rule and the assumptions on E, the relation $\lessdot$ is acyclic and hence well founded. We can define a valuation α of V into $\mathcal{R}$[5] by well founded induction on $\lessdot$.

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.[6] For $i = 1, \ldots, n$ we define $\alpha(v_i) = c_i$. Now let $\{v_{n+1}, \ldots, v_{n+k}\}$

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

[6] This includes the case in which c_i is a constructor of 0-arity.

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 $\mathcal{T}(\Omega) \setminus \{\alpha(v_1), \ldots, \alpha(v_{n+i-1})\}$ [7]. If v_i is of some sort τ_j, we know by a previous observation that $v_i \mapsto L \in E$ with $L \neq \emptyset$. Note that L must contain at least one non-finite constructor. Suppose all constructors are finite: if L is not a singleton, then Split 2 applies, contradicting irreducibility of E. If L is a singleton, and $C \in L$ is nullary, then by the Instantiate 1 rule, an equation of the form $C \to v_k$ is in E. If $C \in L$ is non-nullary, then by the Instantiate 2 rule, an equation of the form $C\mathbf{u} \to v_k$ is in E contradicting $\lessdot$-minimality of v_k. We then define $\alpha(v_k) = C^i_j t_1 \cdots t_{n^i_j}$ where C^i_j is some non-zero-arity constructor in L and $C^i_j t_1 \cdots t_{n^i_j}$ 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 $Cu_1 \cdots u_k \to v$ in E. 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_i. It follows that $\alpha(u), \alpha(v)$ are terms of the form $C^i_j \alpha(u_1) \cdots \alpha(u_{n^i_j})$, $C^i_{j'} \alpha(v_1) \cdots \alpha(v_{n^i_{j'}})$, respectively, with $n^i_j \geq 1$ and $n^i_{j'} \geq 1$. Now, if $j \neq j'$, then $\alpha(u)$ and $\alpha(v)$ are trivially distinct terms. If $j = j'$, then by induction $\alpha(u_1)$ and $\alpha(v_1)$, say, 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 $v : \tau_i$ consider the following two cases. If $C^i_j u_1 \cdots u_k \to v \in E$ for some $C^i_j u_1 \cdots u_k$ then it is not difficult to show that L must be $\{C^i_j\}$. But then $\alpha(v) = C^i_j \alpha(u_1) \cdots \alpha(u_k)$ by construction. If there is no $C^i_j u_1 \cdots u_k \to v \in E$, then $\alpha(v)$ is defined as some term $C^i_j t_1 \cdots t_k$ where $C^i_j \in L$. 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 4.7 (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 Lemma 4.6, 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

A *strategy* is a predetermined methodology for applying the rules. Different strategies may be more or less efficient. Before discussing our recommended strategy, it is instructive to look at the

[7] Using the assumption that all sorts σ are infinite.

closest related work. A naive algorithm for universal formulas is discussed in [16]. Although the presentation there is somewhat different, the essence of their algorithm can be mimicked by our rules[8] with one small modification: replacing the Split 1 and Split 2 rules with the following basic Split rule:

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

There are four steps in their naive algorithm: guess a "type completion"; simplify; compute the bidirectional closure; and check for conflicts. These steps are roughly equivalent to the following strategy: after abstraction, apply the Split rule until it can no longer be applied (this corresponds to guessing a type completion). Next, apply the selector rules to eliminate all instances of selector symbols. Then, apply upward and downward closure rules (the bidirectional closure). Finally, check for conflicts using the remaining rules.

One of the key contributions of this paper is to recognize that this naive strategy can be improved in two significant ways. First, the split rule should be delayed as long as possible, and second, the naive split rule can be replaced with the smarter Split 1 and Split 2 rules. These two modifications work together and have the potential to dramatically reduce the size of the resulting derivation. Notice that with the smarter splitting rules, unless an abstract variable u is labeled with all finite constructors, Split 1 is only enabled when some selector is applied to u. By itself, this eliminates many needless case splits. But by delaying the Split rules (in particular by first applying selector rules), it may be possible to eliminate selectors and thus eliminate additional case splits.

Suppose we have a simple *tree* data type. It has a binary constructor $node : tree \times tree \to tree$ with two associated selectors, $left : tree \to tree$ and $right : tree \to tree$. There is also a 0-arity constructor *leaf* which is also the designated term for both selectors. Now, consider the following input:

$$left^n(Z) \approx X \ \wedge \ isnode(Z) \ \wedge \ Z \approx X$$

After applying all available rules except for the Split rules, the result will look something 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 follow our strategy. Split 1 can only be applied to some $u_i, (1 < i < n)$, so let's say 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 $\perp$ via the Empty and Refine rules. The other branch contains $u_i \mapsto \{node\}$ and results in the application of the Instantiate 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 $\perp$ via the Cycle rule.

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, we can do better. If we start at u_{n-1} and work our way

[8] Unfortunately, there is not enough detail in [16] to be sure that this is an accurate characterization of their algorithm, but this reflects our best understanding of it.

down, each split will take only $O(1)$, so the total size of the derivation tree will be $O(n)$.[9] This is not a coincidence and leads to a final strategy suggestion: split on nodes that correspond to the least deeply nested terms first.

Of course, in the worst case, our strategy will still be exponential because the problem is NP-complete, but with this example as evidence, we claim that our strategy is never worse than the naive strategy, and is often far superior.

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 imposed before that each of $\sigma \in \{\sigma_1, \ldots, \sigma_r\}$ is infinite valued. Since we have no such restrictions on sorts τ_i, 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 way to play it safe is to modify the model $\mathcal{R}$ so that selectors are interpreted as partial functions. An evaluation of a formula in this model 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. Fortunately, this approach can easily be implemented as described in [3]: 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 3-valued semantics.

6.3 Cooperating with other Decision Procedures

A final point is that that our procedure has been designed to easily integrate into a Nelson-Oppen-style framework for cooperating decision procedures [9]. In the many-sorted case, the key theoretical requirements (see [14]) 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.[10] A key operational requirement is that the decision procedure is also able to easily compute and communicate equality information.

[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 [11], 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] 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.

The theory of $\mathcal{R}$ (i.e., the set of sententences 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.

7 Conclusion

We have presented an algorithm for deciding the theory of recursive data types. Novel features of our treatment include the ability to handle mutually recursive multi-sorted data types, a simpler presentation of the theory, an abstract declarative algorithm, and smarter splitting rules which can greatly enhance efficiency. As future work, we propose to treat the subjects mentioned briefly in the last section in more detail. Also, though a prototype implementation has been completed, we have begun work on implementing the decision procedure within the theorem prover CVC Lite [2].

References

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

[2] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[3] S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. In *Proceedings of the* 2^{nd} *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '04)*, July 2004. Cork, Ireland.

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

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

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

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

[8] 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.

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

[10] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[11] D. C. Oppen. Reasoning about recursively defined data structures. *Journal of the Association for Computing Machinery*, 27(3):403–411, July 1980.

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

[13] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[14] C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In J. Alferes and J. Leite, editors, *Proceedings of the 9th European Conference on Logic in Artificial Intelligence (JELIA'04), Lisbon, Portugal*, volume 3229 of *Lecture Notes in Artificial Intelligence*, pages 641–653. Springer, 2004.

[15] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the Association of Computing Machinery*, 34(2):492–510, April 1987.

[16] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. In *Proceedings of the* 2^{nd} *International Joint Conference on Automated Reasoning (IJCAR '04) LNCS 3097*, pages 152–167, 2004.

[17] T. Zhang, H. B. Sipma, and Z. Manna. Term algebras with length function and bounded quantifier alternation. In *Proceedings of the* 17^{th} *International Conference on Theorem Proving in Higher Order Logics (TPHOLs '04)*, volume 3223 of *Lecture Notes in Computer Science*, pages 321–336, 2004.