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

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. In this paper, we present a general algorithm for the universal fragment. The algorithm is presented declaratively as a set of abstract rules which are terminating, sound, and complete. We show how other algorithms can be realized as strategies within our general framework. Finally, we propose a new strategy and give experimental results showing that it is significantly faster than other strategies.

1 Introduction

Recursive data types are commonly used in programming. 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 constructor. 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 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:

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

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 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: (i) $\forall x : nat.\ succ(x) \neq zero$; (ii) $\forall x : list.\ x = null \vee \exists y : nat, z : list.\ x = cons(y, z)$; and (iii) $\forall x : tree.\ is\_leaf(x) \rightarrow (data(x) = zero \vee is\_succ(data(x)))$.

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.

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 and multiple constructors. 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 4, 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 recursive data types. 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. [9, 10] 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. An often-cited reference for the quantifier-free case is the treatment by Nelson and Oppen in 1980[7, 8] (where the problem is also shown to be NP-complete). In particular, Oppen's algorithm in [8] gives a detailed decision procedure for a single recursive data type with a single constructor. Unfortunately, the case of multiple constructors is discussed only briefly and not rigorously. A possible extension of Oppen's algorithm to the case of multiple constructors is discussed briefly in [9]. We will discuss the relationship of our algorithm to these in more detail in Section 4.

2 The Theory of Recursive Data Types

Previous work on recursive data types (RDTs) [9, 10] uses first-order axiomatizations in an attempt to capture the main properties of a recursive 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 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 [6] 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 $r \geq 0$ 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 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 **??** shows 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: (i) a constructor C^i_j is well-founded if all of its argument sorts are well-founded; (ii) the sorts $\sigma_1, \ldots, \sigma_r$ are all well-founded; (iii) 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: (i) a constructor is *finite* if it is nullary or if all of its argument sorts are finite; (ii) a sort τ_i is *finite* if all of its constructors are finite, and is *infinite* otherwise; (iii) 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.

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

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 are 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, 9, 10]) where the application of a wrong selector is treated as the identity function. The main reason for this difference is that identity function would not always be well-sorted in multi-sorted logic.

By standard results in universal algebra we know that $\mathcal{E}$ admits an *initial model* $\mathcal{R}$. By standard results in universal algebra we can show the following result:[2] Let Ω be the signature obtained from Σ by removing the selectors and the testers; then, the reduct of $\mathcal{R}$ to Ω is isomorphic to $\mathcal{T}(\Omega)$. Informally, this 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 $\mathcal{T}(\Omega)$.

3 The Decision Procedure

We start with an informal overview based on examples. Our procedure builds on the algorithm by Oppen [8] for a single type with a single constructor. Consider, for example, the *list* data type 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[3] 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

[2] Proofs of all results in this paper can be found in [3].

[3] A simple example of a cycle is: $cdr(x) \approx x$.

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 extension of Oppen's algorithm for the case of multiple constructors is proposed in [9]. 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.

Our presentation combines ideas from all of these algorithms 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 4 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 [5].

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 $\mathcal{R}$.[4] 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 $isC_k^j(t) \approx \mathsf{true}$, 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 $lbls(\tau_i)$ for the set $\{C_1^i, \ldots, C_{m_i}^i\}$ and define $lbls(\sigma_l)$ 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*.

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

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 *labellings* of abstraction variables with sets of constructor symbols. It is used to keep track of possible constructors for instantiating a τ_i variable.[5] A variable assignment α satisfies a labeling pair $v \mapsto \{C^i_{j_1}, \ldots, C^i_{j_n}\}$ in $\mathcal{R}$ if α satisfies the formula $isC^i_{j_1}(v) \vee \cdots \vee isC^i_{j_n}(v)$ 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.

Abstraction rules

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

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

[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.

Abstract 3 $$\frac{p[S^i_{j,\kappa}u],\ E}{S^i_{j,1}u \to v_1,\ \ldots,\ S^i_{j,n^i_j}u \to v_{n^i_j},\ p[v_\kappa],\ v_1 \mapsto lbls(s_1),\ \ldots, v_{n^i_j} \mapsto lbls(s_{n^i_j}),\ E}$$ if $p \in \Lambda, S^i_{j,k} : \tau_i \to s_k$, each v_ι fresh from V_{s_ι}

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 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 $S^i_{j,k}$ 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 $$\frac{u \approx v,\ E}{u \to v,\ E}$$ if $u \succ v$

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

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

Remove 2 $$\frac{\neg isC^i_j\, v,\ E}{v \mapsto lbls(sort(v)) \setminus \{C^i_j\},\ E}$$

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 $$\frac{t \to u,\ t \to v,\ E}{u \to v,\ t \to v,\ E}$$ 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 $$\frac{C^i_j\mathbf{u} \to v,\ C^i_j\mathbf{v} \to v,\ E}{C^i_j\mathbf{u} \to v,\ \mathbf{u} \to \mathbf{v},\ E}$$

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

Cycle $$\frac{C^{i_n}_{j_n}\mathbf{u_n}u\mathbf{v_n} \to u_{n-1}, \ldots, C^{i_2}_{j_2}\mathbf{u_2}u_2\mathbf{v_2} \to u_1, C^{i_1}_{j_1}\mathbf{u_1}u_1\mathbf{v_1} \to u, E}{\bot} \quad \text{if } \ n \geq 1$$

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** rule detects 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 $$\frac{S^i_{j,1}u \to u_1,\ \ldots,\ S^i_{j,n^i_j}u \to u_{n^i_j},\ u \mapsto \{C^i_j\},\ E}{C^i_j u_1 \cdots u_{n^i_j} \to u,\ u \mapsto \{C^i_j\},\ \mathit{Inst}(u),\ E} \quad \text{if } \ \mathit{Inst}(u) \notin E$$

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

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

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

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

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

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

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

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 4, below.

The rules are proved sound, complete and terminating in our full report [3]. The proofs must be omitted here due to space considerations.

4 Strategies and Efficiency

4.1 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 Split rules: because Oppen's algorithm assumes a single constructor, the Split rules are never used. A generalization of Oppen's algorithm is mentioned in [9].[6] They add the step of initially guessing a "type completion". To model this, we first add the following simple Split rule which is invoked greedily (after abstraction) until it no longer applies, at which point the basic strategy is followed.

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

We will call this strategy the *eager splitting* strategy. One of the key contributions of this paper is to recognize that the eager splitting strategy can be improved in two significant ways. First, the split rule should be delayed as long as possible, and second, the simple split rule should be replaced with the smarter Split 1 and Split 2 rules. We call this the *lazy splitting* strategy. The two modifications work together to reduce the size of the resulting derivation in two ways. First, unless an abstract variable u is labeled with all

[6] Unfortunately, there is not enough detail in [9] to be sure that this is an accurate characterization of their algorithm, but this reflects our best understanding of it.

Worst Case Number of Splits	Benchmarks	Sat	Unsat	Eager		Lazy	
				Splits	Time (s)	Splits	Time (s)
0	4416	306	4110	0	24.4	0	24.9
1-5	2523	2217	306	6893	17.1	2419	16.4
6-10	692	570	122	4967	6.1	1600	5.5
11-20	176	112	64	2399	2.2	509	1.5
21-100	144	73	71	6282	4.4	334	1.2
101+	49	11	38	16593	9.9	73	0.3

Table 1: Eager vs Lazy Splitting

finite constructors, Split 1 is only enabled when some selector is applied to u. By itself, this eliminates many needless case splits. Second, by applying the split rules *lazily* (in particular by first applying selector rules), it may be possible to eliminate selectors and thus eliminate additional case splits.

Example. Consider the following simple *tree* data type. It has a binary constructor $node : tree \times tree \rightarrow tree$ with two associated selectors, $left : tree \rightarrow tree$ and $right : tree \rightarrow tree$. 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 abstraction, there are $n + 2$ abstraction variables labeled with two labels each. If we eagerly apply the simple Split rule at this point, the derivation tree would reach size $O(2^{n+2})$. On the other hand, if we follow the lazy strategy, the worst case is $O(n^2)$. This is because whenever we do a case split that assigns a constructor other than *node* to a term, it induces a cascade of Collapse 2 and cuts off a large piece of the search space.

4.2 Experimental Results

We have implemented both the lazy and the eager splitting strategies in the theorem prover CVC Lite [2]. Using the mutually recursive data types *nat*, *list*, and *tree* mentioned in the introduction, we randomly generated 8000 benchmarks within the following parameters: each benchmark tested the satisfiability of a conjunction of literals over the three data types; the number of variables of each type varied from 1 to 10; the number of literals varied from 2 to 10. In addition, for half of the benchmarks, we ensured that selectors were only applied to the "appropriate" constructor (as would be expected in a real application). For the other half, we relaxed this constraint.[7]

As might be expected with a large random set, most of the benchmarks are quite uninteresting. 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). The table below shows the total time and case splits required to solve the benchmarks. The benchmarks are divided into categories based on the the maximum number of case splits required to solve the benchmark. The results are shown in Table 1.

The table shows that for easy benchmarks that don't require many splits, the two algorithms perform almost identically, though the eager strategy typically still needs more

[7] See `http://www.cs.nyu.edu/~barrett/datatypes` for more details on the benchmarks and results.

case splits. In fact, on only 3 of 8000 benchmarks did the eager strategy require fewer case splits than the lazy strategy.[8] However, as the difficulty increases, the time and number of case splits required by the eager strategy increases significantly while the lazy strategy is much more robust.

5 Extending the Algorithm

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 $null_i$.

References

[1] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *JAR*, 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 CAV '04*, volume 3114 of *LNCS*, pages 515–518. Springer-Verlag, July 2004.

[3] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. Technical Report TR2005-878, Department of Computer Science, New York University, Nov. 2005.

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

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

[6] 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.

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

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

[9] 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.

[10] T. Zhang, H. B. Sipma, and Z. Manna. Term algebras with length function and bounded quantifier alternation. In *Proceedings of TPHOLs '04*, volume 3223 of *LNCS*, pages 321–336, 2004.

[8] The 3 benchmarks are not hard: both strategies can solve them with less than 10 splits and in negligible time.