On Decision Procedures for Algebraic Data Types with Abstractions

LARA-REPORT-2009-003

Philippe Suter Mirco Dotta Viktor Kuncak [*]

EPFL School of Computer and Communication Sciences, Switzerland

{firstname.lastname}@epfl.ch

Abstract

We describe a parameterized decision procedure that extends the decision procedure for functional recursive algebraic data types (trees) with the ability to specify and reason about abstractions of data structures. The abstract values are specified using recursive abstraction functions that map trees into other data types that have decidable theories. Our result yields a decidable logic which can be used to prove that implementations of functional data structures satisfy recursively specified invariants and conform to interfaces given in terms of sets, multisets, or lists, or to increase the automation in proof assistants.

1. Introduction

Decision procedures for proving verification conditions have seen great practical success in recent years. Systems using such decision procedures incorporated into SMT provers [15, 6] were used to verify tens of thousands of lines of imperative code [13, 3] and prove complex correctness conditions of imperative data structures [62, 46]. While much recent work on automation was invested into imperative languages, it is interesting to consider the reach of such decision procedures when applied in *functional* programming languages, which were designed with the ease of reasoning as one of the explicit goals [40]. Researchers have explored the uses of advanced type systems to check expressive properties [17, 61], and have recently also applied automated provers to solve localized type system constraints [24, 54]. The benefits of incorporating automated provers include the ability to efficiently deal with arithmetic constraints and propositionally complex constraints.

In this paper we embrace a functional language as an implementation language, but also as the specification language. In fact, our properties are expressed as executable assertions. Among the immediate benefits is that the developer need not learn a new notation for properties. Moreover, the developers can debug the properties and the code using popular testing approaches such as Quickcheck [12, 56]. The use of executable specification language avoids the difficulties of adapting classical higher-order logic to run-time checking [63]. In using programming language as the specification language, our work is in line with soft typing approaches that originated in untyped functional languages [11], although we use ML-like type system as a starting point, focusing on properties that go beyond the ML types.

Purely functional implementations of data structures [48] present a well-defined and interesting benchmark for automated reasoning about functional programs. Data structures come with well-understood specifications: they typically implement an ordered or unordered collection of objects or maps between objects. When expressing the desired properties of data structures, it quickly becomes evident that we need a rich set of data types to write specifications. In particular, it is desirable to have in the language not only algebraic data types, but also finite sets and multisets. These data types can be used to concisely specify the observable behavior of data structures with the desired level of under-specification [27, 35, 66, 16]. For example, if neither the order nor the repetitions of elements in the tree matter, an appropriate abstract value is a set. An abstract description of an add operation that inserts into a data structure is the following identity:

$$\alpha(\mathsf{add}(e,t)) = \{e\} \cup \alpha(t) \tag{1}$$

Here α denotes a function mapping a tree into the set of elements stored in the tree. Other variants of the specification can use multisets or lists instead of sets.

An important design choice is how to specify such mappings α between the concrete and abstract data structure values. A popular approach [13, 63] does not explicitly define a mapping α but instead introduces a fresh ghost variable to represent values $\alpha(t)$. It then uses invariants to relate the ghost variable to the concrete value of the data structure. Because developers explicitly specify values of ghost variables, such approach yields simple verification conditions. However, this approach can impose additional annotation overhead. To eliminate such overhead we choose to use recursively defined abstraction functions to map concrete data structures to their abstract view. As a result, our verification conditions contain user-defined function definitions that manipulate rich data types, along with equations and disequations involving such functions. Our goal is to develop decision procedures that can reason about interesting fragments of such language.

We present a decision procedure for reasoning about recursive data types with user-defined abstraction functions, expressed as a fold over trees, or a catamorphisms [42], over algebraic data types. This decision procedure subsumes approaches for reasoning about recursive data types [49]. It adds the ability to express constraints on the abstract view of the data structure, as well as user-defined mapping between the concrete and the abstract view of the data structure. When using sets as the abstract view, our decision procedure can naturally be combined with decision procedures for reasoning about sets of elements in the presence of cardinality bounds [32, 28]. It also presents a new example of a theory that fits in the recently described approach for combining decision procedures that share sets of elements [60].

Our decision procedure is not limited to using sets as an abstract view of data structure. The most important condition for applicability is that the notion of a collection has a decidable theory

[*] This research is supported in part by the Swiss National Science Foundation Grant "Precise and Scalable Analyses for Reliable Software".

```
object BSTSet {
  type E = Int
  type C = Set[E]
  sealed abstract class Tree
  private case class Leaf() extends Tree
  private case class Node(left: Tree, value: E, right: Tree)
                extends Tree

  // abstraction function
  def content(t: Tree): C = t match {
    case Leaf() => Set.empty
    case Node(l,e,r) => content(l) ++ Set(e) ++ content(r)
  }

  // returns an empty set
  def empty: Tree = {
    Leaf()
  } ensuring (res => content(res) == Set.empty)

  // adds an element to a set
  def add(e: E, t: Tree): Tree = (t match {
    case Leaf() => Node(Leaf(), e, Leaf())
    case t @ Node(l,v,r) =>
      if (e < v) Node(add(e, l), v, r)
      else if (e == v) t
      else Node(l, v, add(e, r))
  }) ensuring (res => content(res) == content(t) ++ Set(e))

  // user−defined equality on abstract data type (congruence)
  def equals(t1 : Tree, t2 : Tree) : Boolean =
    (content(t1) == content(t2))

}
```

Figure 1. A part of a binary search tree implementation of set

in which the catamorphism can be expressed. This includes in particular arrays [10], multisets with cardinality bounds [51, 52], and even option types over integer elements. Each abstract value provides different possibilities for defining fold over trees.

We believe that we have identified an interesting point in the space of automated reasoning approaches, because the technique turned out to be applicable more widely than we had expected. We intended to use the technique to verify the abstraction of values of functional data structures using sets. It turned out that the decision procedure is often complete not only for set abstraction but also for lists and multisets, and even for boolean abstraction that encodes data structure invariants. Beyond data structures used to implement sets and maps, we have found that computing bound variables, a common operation on the representations of lambda terms and formulas, is also amenable to our approach. We thus expect that our decision procedure can help increase the automation of reasoning about operational semantics and type systems of programming languages.

Contribution. This paper presents a parameterized decision procedure for the quantifier-free theory of (purely functional) recursive tree data structures with a catamorphism function. We establish soundness of the general procedure and provide conditions on the catamorphism under which the procedure is complete.

2. Example

Figure 1 shows Scala [47] code for a partial implementation of a set of integers using a binary search tree. The class hierarchy

```
sealed abstract class Tree
private case class Leaf() extends Tree
private case class Node(left:Tree, value:E, right:Tree) extends Tree
```

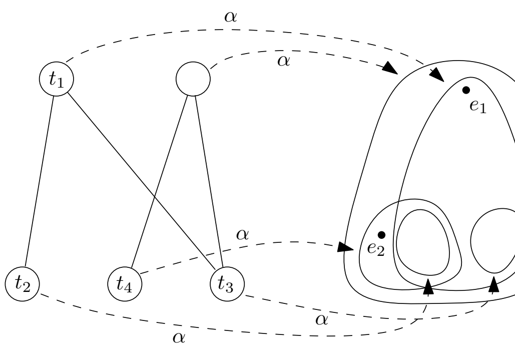

Figure 2. Graphical representation of the constraint (2), where edges labeled by α denote applications of content

describes an algebraic data type Tree with the alternatives Node and Leaf. The use of the private keyword implies that the alternatives are not visible outside of the module BSTSet. The keyword sealed means that the hierarchy cannot be extended outside of the module.

The module BSTSet provides its clients with functions to create empty sets and to insert elements into existing sets. Because the client has no information on the type Tree, he uses the abstraction function content to view these trees as sets. Note that we call it an *abstraction* function because of its conceptual role, but it is declared and written like any other function, and therefore is executable as well.

Additionally, the functions empty and add are annotated with postconditions on which the client can rely, without knowing anything about their concrete implementation. These postconditions do not give any information about the inner structure of binary search trees, as such information would be useless to a user who has no access to this structure. Instead, they express properties on their result in terms of the abstraction function. The advantages of such an abstraction mechanism are well-known; by separating the specification, the functions signatures and their contracts, from the implementation, the concrete code, one keeps programs modular, allowing for simpler reasoning, better opportunities for code reuse and easier replacement of modules.

The parametrized type Set[E], accessed through the type alias C and used in the abstraction function and the specifications, refers to the Scala library class for immutable sets. The operator ++ computes a set consisting of the union of two sets, and the constructor Set(e) constructs a singleton containing the element e. We assume the implementation of the library to be correct, and map these operations to the corresponding ones in the theory of finite sets when we reason about the programs.

The advantages of using an abstraction function in the specifications are clear, but it also makes the task of verification systems harder, since they need to reason about these user-defined functions which can appear in contracts, and therefore in verification conditions. For example, using standard techniques to generate verification conditions for functional programs, we would create for the function add the following condition (among others):

$$\begin{array}{l}\forall t_1, t_2, t_3, t_4 : \mathsf{Tree},\ e_1, e_2 : \mathsf{Int} \\ \quad t_1 = \mathsf{Node}(t_2, e_1, t_3) \Rightarrow \\ \quad \mathsf{content}(t_4) = \mathsf{content}(t_2) \cup \{e_2\} \Rightarrow \\ \quad \mathsf{content}(\mathsf{Node}(t_4, e_1, t_3)) = \mathsf{content}(t_1) \cup \{e_2\}\end{array} \tag{2}$$

Such a formula combines constraints over algebraic data types and over finite sets, as well as a non-trivial connection given by the recursively defined abstraction function content. Such formula is therefore beyond the reach of currently known decision procedures. In the following sections, we present a new decision procedure which can handle such formulas. As an illustration, Section 4.5 applies our decision procedure precisely to the verification condition (2).

3. Reasoning about Recursive Data Structures and Abstraction Functions

Decision procedures for reasoning about recursive data structures [49, 5] are concerned with proving and disproving quantifier-free formulas that involve constructors and selectors of a *tree* algebraic data type, such as the immutable version of heterogeneous lists in LISP. Using the terminology of model theory, this problem can be described as the satisfiability of quantifier-free first-order formulas in the theory of *term algebras* [21, Page 14, Page 67]. A term algebra structure has as a domain of interpretation ground terms over some set of function symbols, called *constructors*. The language of term algebras includes application of constructors to build larger terms from smaller ones, and the only atomic formulas are comparing terms for equality.

In this paper, we extend the decision procedure for such recursive data types with the ability to specify an *abstract value* of the recursive data type. The abstract value can be, for example, a set, relation, multiset (bag), or a list. A number of decision procedures are known for theories of such abstract values, [32, 52, 51, 36, 23]. Such values purposely ignore issues such as tree shape, ordering, or even the exact number of times an element appears in the data structure. In return, they come with powerful algebraic laws and decidability properties that are often not available for algebraic data types themselves, and they often provide the desired amount of under-specification for interfaces of data structures.

Our decision procedure enables proving formulas that relate data structures implemented as algebraic data types to their abstract values that specify the observable behavior of these data types. It can thus greatly increase the automation when verifying correctness of functional data structures.

3.1 Instances of the Decision Procedure

The choice of the type of data stored in the tree in our decision procedure is largely unconstrained; the procedure works for any infinitely countable parameterized data type, which we will denote by $\mathcal{E}$ in our discussion (it could be extended to finite data types using techniques from [25]). The decision procedure is parameterized by

1. the element type $\mathcal{E}$
2. a collection type $\mathcal{C}$ and
3. an abstraction function α (generalizing the content function in Figure 1).

We require the abstraction function to be a catamorphism (generalized fold) [42]. In the case of binary trees on which we focus, for some functions

$$\begin{aligned} \mathsf{empty} &: \mathcal{C} \\ \mathsf{combine} &: (\mathcal{C}, \mathcal{E}, \mathcal{C}) \rightarrow \mathcal{C} \end{aligned}$$

the definition of the abstraction function is:

```
def α(t: Tree): C = t match {
  case Leaf() ⇒ empty
  case Node(l,e,r) ⇒ combine(α(l), e, α(r))
}
```

We next show that this requirement is naturally satisfied for many abstraction functions over recursive data types.

```
object Lambda {
  type ID = String
  type C = Set[String]

  sealed abstract class Term
  case class Var(id: ID) extends Term
  case class App(fun: Term, arg: Term) extends Term
  case class Abs(bound: ID, body: Term) extends Term

  def free(t: Term): C = t match {
    case Var(id) => Set(id)
    case App(fun, arg) => free(fun) ++ free(arg)
    case Abs(bound, body) => free(body) -- Set(bound)
  }
}
```

Figure 3. Computing the set of free variables in a λ-calculus term

```
object MinElement {
  type E = Int

  sealed abstract class Tree
  case class Node(left: Tree, value: E, right: Tree) extends Tree
  case class Leaf() extends Tree

  def findMin(t: Tree): Option[E] = t match {
    case Leaf() => None
    case Node(l,v,r) =>
      (findMin(l),findMin(r)) match {
        case (None,None) => Some(v)
        case (Some(vl),None) => Some(min(v, vl))
        case (None,Some(vr)) => Some(min(v, vr))
        case (Some(vl),Some(vr)) => Some(min(v, vl, vr))
      }
  }
}
```

Figure 4. Using the minimal element as an abstraction

Canonical Set Abstraction. The content function in Figure 1 is an example of a catamorphism used as an abstraction function. In this case, empty $= \emptyset$ and combine$(t_1, e, t_2) = c_1 \cup \{e\} \cup c_2$. We found this example to be particularly useful and well-behaved, so we refer to it as the *canonical set abstraction*.

Free Variables of Lambda Calculus Terms. Canonical abstraction is not the only interesting abstraction function whose result is a set. Figure 3 shows another example, where the catamorphism free computes a set by adding and removing elements as the tree traversal goes. Such abstraction function can then be used to verify that rewriting applied to a λ-calculus term preserves the set of free variables in the term.

Minimal Element. Some useful abstractions map trees into a single element rather than into a collection which depends on the size of the tree. findMin in Figure 4 for instance is naturally expressed as a catamorphism, and can be used to prove properties of data structures which maintain invariants about the position of certain particular elements (e.g. priority queues).

Sortedness of Binary Search Trees. Catamorphisms can also compute properties about tree structures which apply to the complete set of nodes and go beyond the expression of a container in terms of another. Figure 5 shows the abstraction function sorted which, when applied to a binary tree, returns a triple whose third element is a boolean indicating whether the tree is sorted. While this specification style may feel unnatural in that case, we consider

```
object SortedSet {
  type E = Int

  sealed abstract class Tree
  case class Leaf() extends Tree
  case class Node(left: Tree, value: E, right: Tree) extends Tree

  def sorted(t: Tree): (Option[Int],Option[Int],Boolean) =
   t match {
    case Leaf() => (None, None, true)
    case Node(l, v, r) => {
      (sorted(l),sorted(r)) match {
        case ((_,_,false),_) => (None, None, false)
        case (_,(_,_,false)) => (None, None, false)
        case ((None,None,_),(None,None,_)) =>
          (Some(v), Some(v), true)
        case ((Some(minL),Some(maxL),_),(None,None,_))
          if (maxL < v) => (Some(minL),Some(v),true)
        case ((None,None,_),(Some(minR),Some(maxR),_))
          if (minR > v) => (Some(v), Some(maxR), true)
        case ((Some(minL),Some(maxL),_),
              (Some(minR),Some(maxR),_))
          if (maxL < v && minR > v) =>
            (Some(minL),Some(maxR),true)
        case _ => (None,None,false)
      }
    }
  }
}
```

Figure 5. Using an abstraction function to check the sorted property

this function to be a good example of the broad application spectrum of our decision procedure.

Figure 6 summarizes some of the specific cases to which our decision procedure applies. It shows the type of the abstract value $\mathcal{C}$, the definition of the functions empty and combine that define the catamorphism, some of the operations available on the logic $\mathcal{L}_\mathcal{C}$ of $\mathcal{C}$ values, and points to one of the references that can be used to show the decidability of $\mathcal{L}_\mathcal{C}$. Figure 6 illustrates that our decision procedure covers a wide range of collection abstractions of interest, as well as some other relevant functions definable as catamorphisms.

4. The Decision Procedure

4.1 Preliminaries

To simplify the presentation, we present our decision procedure for the specific algebraic data type of binary trees, corresponding to the case classes in Figure 1, which in ML syntax, would correspond to the algebraic data type

datatype Tree = Leaf | Node **of** Tree $*$ E $*$ Tree

Our procedure naturally extends to data types with more constructors.

If t_1 and t_2 denote values of type Tree, by $t_1 = t_2$ we denote that t_1 and t_2 are structurally equal that is, either they are both leaves, or they are both nodes with equal values and equal subtrees.

For the purpose of soundness, we leave the collection type $\mathcal{C}$ and the language $\mathcal{L}_\mathcal{C}$ of decidable constraints on $\mathcal{C}$ largely unconstrained. As explained below, the conditions for completeness are relatively easy to satisfy when the image are sets and become somewhat more involved for multisets and lists.

In our exposition, we use the notation

$$\mathsf{distinct}(x_1^1, x_2^1, \ldots, x_{I(1)}^1; \ldots; x_1^n, \ldots, x_{I(n)}^n)$$

as a syntactic shorthand for the following conjunction of disequalities

$$\bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} \bigwedge_{k=1}^{I(i)} \bigwedge_{l=1}^{I(j)} x_k^i \neq x_l^j$$

For example, $\mathsf{distinct}(x, y; z)$ reads as $x \neq z \wedge y \neq z$, and $\mathsf{distinct}(x_1; \ldots; x_n)$ means that all x_i are different.

4.2 Overview of the Decision Procedure

The high-level summary of our procedure is that it solves the constraints over trees using unification, then derives all relevant consequences on the type $\mathcal{C}$ of collections that abstracts the trees. It thus effectively reduces a formula over trees and their contents expressed using terms of $\mathcal{L}_\mathcal{C}$ to a formula in $\mathcal{L}_\mathcal{C}$, for which a decision procedure is assumed to be available. Figure 7 gives a high-level view of the process. We next define our parametrized decision problem more precisely, present our decision procedure, and show its soundness and completeness.

For a conjunction ϕ of literals over the theory of trees parametrized by $\mathcal{L}_\mathcal{C}$ and α:

1. apply purification to separate ϕ into $\phi_T \wedge \phi_B \wedge \phi_C$ where:
   - ϕ_T contains only literals over tree terms
   - ϕ_C contains only literals over terms from $\mathcal{L}_\mathcal{C}$
   - ϕ_B contains only literals of the form $c = \alpha(t)$ where c is a variable from $\mathcal{L}_\mathcal{C}$ and t is a tree variable
2. flatten all terms and eliminate the selectors left and right
3. apply unification on the tree terms, potentially detecting unsatisfiability
4. if unification did not fail, propagate the result from unification to the relevant parts of ϕ_C, yielding a new formula ϕ'_C in $\mathcal{L}_\mathcal{C}$
5. establish the satisfiability of ϕ with a decision procedure for $\mathcal{L}_\mathcal{C}$ applied to ϕ'_C

Figure 7. Overview of the decision procedure

T	$::=$	$t \mid \mathsf{Leaf} \mid \mathsf{Node}(T, E, T)$ $\mid \mathsf{left}(T) \mid \mathsf{right}(T)$	Tree terms
C	$::=$	$c \mid \alpha(t) \mid \mathcal{T}_\mathcal{C}$	$\mathcal{C}$-terms
F_T	$::=$	$T = T \mid T \neq T$	Equations over trees
F_C	$::=$	$C = C \mid \mathcal{F}_\mathcal{C}$	Formulas of $\mathcal{L}_\mathcal{C}$
E	$::=$	variables or constants of type $\mathcal{E}$	
ϕ	$::=$	$\bigwedge F_T \wedge \bigwedge F_C$	Conjunctions
ψ	$::=$	$\phi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$ $\mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi$	Formulas

$\mathcal{T}_\mathcal{C}$ and $\mathcal{F}_\mathcal{C}$ represent terms and formulas of $\mathcal{L}_\mathcal{C}$ respectively. Formulas are assumed to be closed under negation.

Figure 8. Syntax of the parametric logic

4.3 Syntax and Semantics of Recursive Data Structures with Abstraction Functions

The syntax of our logic is given in Figure 8, and Figure 9 gives a description of its semantics. The description refers to the semantics of the parameter theory $\mathcal{L}_\mathcal{C}$, which we write $[\![\ ]\!]_\mathcal{C}$, as well as the definition of the catamorphism α.

$\mathcal{C}$	empty	combine(c_1, e, c_2)	abstract operations (apart from $\wedge, \neg, =$)	complexity	follows from
Set	$\emptyset$	$c_1 \cup \{e\} \cup c_2$	$\cup, \cap, \setminus$, cardinality	NP	[32]
Multiset	$\emptyset$	$c_1 \uplus \{e\} \uplus c_2$	$\cap, \cup, \setminus, \uplus$, setof, cardinality	NP	[51, 52]
$\mathbb{N}$	0	$c_1 + 1 + c_2$	$+, \leq$	NP	[50]
List	List()	*a)* c_1++ List(e)++ c_2 (in-order)	++(concat), List(_)(singleton)	PSPACE	[53]
	List()	*b)* List(e)++c_1++ c_2 (pre-order)			
	List()	*c)* c_1++ c_2++ List(e) (post-order)			
Tree	Leaf	Node(c_2, e, c_1) (mirror)	Node, Leaf	NP	[49]
Option	None	*a)* Some(e)	Some, None	NP	[45]
	None	*b)* (computing minimum) see Figure 4	Some, None, $+, \leq$, if	NP	[44, 45, 50]
(Option, Option, Boolean)	(None, None, true)	*c)* (checking sortedness) see Figure 5			

Figure 6. Example Instances of our Decision Procedure for Different Catamorphisms

$$
\begin{array}{rcl}
[\![\mathsf{Node}(T_1, e, T_2)]\!] & = & \mathsf{Node}([\![T_1]\!], [\![e]\!]_{\mathcal{C}}, [\![T_2]\!]) \\
[\![\mathsf{Leaf}]\!] & = & \mathsf{Leaf} \\
[\![\mathsf{left}(\mathsf{Node}(T_1, e, T_2))]\!] & = & [\![T_1]\!] \\
[\![\mathsf{right}(\mathsf{Node}(T_1, e, T_2))]\!] & = & [\![T_2]\!] \\
[\![\alpha(t)]\!] & & \text{given by the catamorphism} \\
[\![T_1 = T_2]\!] & = & [\![T_1]\!] = [\![T_2]\!] \\
[\![T_1 \neq T_2]\!] & = & [\![T_1]\!] \neq [\![T_2]\!] \\
[\![C_1 = C_2]\!] & = & [\![C_1]\!]_{\mathcal{C}} = [\![C_2]\!]_{\mathcal{C}} \\
[\![\mathcal{F}_{\mathcal{C}}]\!] & = & [\![\mathcal{F}_{\mathcal{C}}]\!]_{\mathcal{C}} \\
[\![\neg\phi]\!] & = & \neg[\![\phi]\!] \\
[\![\phi_1 \star \phi_2]\!] & = & [\![\phi_1]\!] \star [\![\phi_2]\!] \\
 & & \text{where } \star \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}
\end{array}
$$

Figure 9. Semantics of the parametric logic

4.4 Key Steps of the Decision Procedure

We describe here a decision procedure for a conjunction of literals in our parametric theory. Such a decision procedure can be adapted to handle formulas of arbitrary boolean structure [19].

Purification. In the first step of our decision procedure, we separate all tree terms from literals from $\mathcal{L}_{\mathcal{C}}$. By construction, such terms can only occur as the argument of the abstraction function α. It therefore suffices to replaces all such applications by fresh variables of $\mathcal{L}_{\mathcal{C}}$ and add the appropriate equalities (T denotes any tree term, c_{F} and t_{F} are fresh in the new formula):

$$\mathcal{F}_{\mathcal{C}} \rightsquigarrow t_{\mathrm{F}} = T \wedge c_{\mathrm{F}} = \alpha(t_{\mathrm{F}}) \wedge \mathcal{F}_{\mathcal{C}}[\alpha(T) \mapsto c_{\mathrm{F}}]$$

Flattening of Tree Terms. We then flatten tree terms in a straightforward way. If t and t_{F} denote tree variables, T_1 and T_2 non-variable tree terms and T an arbitrary tree term, we repeatedly apply the following five rewrite rules until none applies ($\doteq$ denotes one of $\{=, \neq\}$):

$$
\begin{array}{rcl}
T \doteq \mathsf{Node}(T_1, E, T_2) & \rightsquigarrow & t_{\mathrm{F}} = T_1 \wedge T \doteq \mathsf{Node}(t_{\mathrm{F}}, E, T_2) \\
T \doteq \mathsf{Node}(t, E, T_2) & \rightsquigarrow & t_{\mathrm{F}} = T_2 \wedge T \doteq \mathsf{Node}(t, E, t_{\mathrm{F}}) \\
T \doteq \mathsf{left}(T_1) & \rightsquigarrow & t_{\mathrm{F}} = T_1 \wedge T \doteq \mathsf{left}(t_{\mathrm{F}}) \\
T \doteq \mathsf{right}(T_1) & \rightsquigarrow & t_{\mathrm{F}} = T_1 \wedge T \doteq \mathsf{right}(t_{\mathrm{F}}) \\
T_1 \doteq t & \rightsquigarrow & t \doteq T_1 \\
t \neq T_1 & \rightsquigarrow & t_{\mathrm{F}} = T_1 \wedge t \neq t_{\mathrm{F}}
\end{array}
$$

where t_{F} is always a fresh variable. It is straightforward to see that this rewriting always terminates.

Elimination of Selectors. The next step is to eliminate terms of the form $\mathsf{left}(t)$ and $\mathsf{right}(t)$. We do this by applying the following rewrite rules:

$$
\begin{array}{rcl}
t = \mathsf{left}(t_1) & \rightsquigarrow & t_1 = \mathsf{Node}(t_{\mathrm{L}}, e, t_{\mathrm{R}}) \wedge t = t_{\mathrm{L}} \\
t = \mathsf{right}(t_1) & \rightsquigarrow & t_1 = \mathsf{Node}(t_{\mathrm{L}}, e, t_{\mathrm{R}}) \wedge t = t_{\mathrm{R}}
\end{array}
$$

Here we use an assumption that the original formula was well-typed, which ensures that selectors are not applied to Leaf nodes. Again, e, t_{L} and t_{R} denote fresh variables of the proper types.

These first three steps yield a normalized conjunctive formula where all literals are in exactly one of following three categories:

- literals over tree terms, which are of one of the following forms:

  $$t_1 = t_2, \quad t = \mathsf{Node}(t_1, E, t_2), \quad t_1 \neq t_2$$

  (Note that disequalities are always between variables.)
- binding literals, which are of the form:

  $$c = \alpha(t)$$

- literals over terms of $\mathcal{L}_{\mathcal{C}}$, which do not contain tree variables or applications of α, and whose specific form depends on the parameter theory $\mathcal{L}_{\mathcal{C}}$

Case Splitting. For simplicity of the presentation, we describe our procedure non-deterministically by splitting the decision problem into a collection of problems of simpler structure (this is a non-deterministic polynomial process). Consider the set $\{t_1, \ldots, t_n, \mathsf{Leaf}\}$ of tree variables appearing in the normalized formula, augmented with the constant term Leaf. We solve the decision problem for each possible partitioning of this set into equivalence classes. Let $\sim$ denote an equivalence relation defining such a partitioning. We generate our subproblem by adding to the original problem, for each pair of terms (T_i, T_j) in the set, the constraint:

$$\begin{cases} T_i = T_j & \text{if } T_i \sim T_j \\ T_i \neq T_j & \text{otherwise} \end{cases}$$

Consider now the set $\{e_1, \ldots, e_m\}$ of variables denoting elements of type $\mathcal{E}$. We again decompose our subproblem according to all possible partitionings over this set, that is we add equalities and disequalities for all pairs (e_i, e_j) in the same way as for tree variables.

The original problem is satisfiable if and only if any of these subproblems is satisfiable. The remaining steps of the decision procedure are applied to each subproblem separately.

Unification. At this point, we apply unification on the positive tree literals. Following [2], we describe the process using inference rules consisting of transformations on *systems*. A system is the pair, denoted $P; S$, of a set P of equations to unify, and a set

Trivial:

$$\frac{T \overset{?}{=} T \cup P'; S}{P'; S}$$

Symbol Clash:

$$\frac{\mathsf{Leaf} \overset{?}{=} \mathsf{Node}(\dots) \cup P'; S}{\perp} \quad \frac{\mathsf{Node}(\dots) \overset{?}{=} \mathsf{Leaf} \cup P'; S}{\perp}$$

Orient:

$$\frac{\{T_1 \overset{?}{=} t\} \cup P'; S}{\{t \overset{?}{=} T_1\} \cup P'; S} \text{ if } T_1 \text{ is not a variable}$$

Occurs Check:

$$\frac{\{t \overset{?}{=} T\} \cup P'; S}{\perp} \text{ if } t \text{ appears in } T \text{ but } t \neq T$$

Term Variable Elimination:

$$\frac{\{t \overset{?}{=} T\} \cup P'; S}{P'[t \mapsto T]; S[t \mapsto T] \cup \{t = T\}} \text{ if } t \text{ does not appear in } T$$

Element Variable Elimination:

$$\frac{\{e_1 \overset{?}{=} e_2\} \cup P'; S}{P'[e_1 \mapsto e_2]; S[e_1 \mapsto e_2] \cup \{e_1 = e_2\}}$$

Decomposition:

$$\frac{\{\mathsf{Node}(T_1, e, T_2) \overset{?}{=} \mathsf{Node}(T_1', e', T_2')\} \cup P'; S}{\{T_1 \overset{?}{=} T_1', T_2 \overset{?}{=} T_2', e \overset{?}{=} e'\} \cup P'; S}$$

Figure 10. Unification Rules

S of solution equations. Equations range over tree variables and element variables. The special system $\perp$ represents failure. The set of equations S has the property that it is of the form $\{t_1 = T_1, \dots, t_n = T_n, e_1 = e_i, \dots, e_m = e_j\}$, where each tree variable t_i and each element variable e_i on the left-hand side of an equality does not appear anywhere else in S. Such a set is said to be in *solved form*, and we associate to it a substitution function σ_S. Over tree terms, it is defined by $\sigma_S = \{t \mapsto T \mid (t = T) \in S\}$. The definition over element variables is similar. The inference rules are the usual rules for unification adapted to our particular case, and are shown in Figure 10.

Any concrete algorithm implementing the described inference system will have the property that on a set of equations to unify, it will either fail, or terminate with no more equations to unify and a system $\emptyset; S$ describing a solution and its associate function σ_S.

If for any disequality $t_i \neq t_j$ or $e_i \neq e_j$, we have that respectively $\sigma_S(t_i) = \sigma_S(t_j)$ or $\sigma_S(e_i) = \sigma_S(e_j)$, then our (sub)problem is unsatisfiable. Otherwise, the tree constraints are satisfiable and we move on to the constraints on the collection type $\mathcal{C}$.

Normal Form After Unification. After applying unification, we can represent the original formula as a disjunction of formulas in a normal form. Let σ_S be the substitution function obtained from unification. Let $\vec{t}$ be the vector of variables t_i for which $\sigma_S(t_i) = t_i$; we call such variables *parameter variables*. Let $\vec{u}$ denote the remaining tree variables; for these variable $\sigma_S(u_j)$ is an expression built from $\vec{t}$ variables using Node and Leaf, they are thus uniquely given as a function of parameter variables. By the symbol v_i we denote a term variable that is either a parameter variable t_i or a non-parameter variable u_i. Using this notation, we can represent (a disjunct of) the original formula in the form:

$$\vec{u} = \vec{T}(\vec{t}) \wedge N(\vec{u}, \vec{t}) \wedge M(\vec{u}, \vec{t}, \vec{c}) \wedge F_E \wedge F_C \qquad (3)$$

where

1. $\vec{T}$ are vectors of expressions in the language of recursive data structures, expressing non-parameter term variables $\vec{u}$ in terms of the parameter variables $\vec{t}$;
2. $N(\vec{u}, \vec{t})$ denotes a conjunction of disequalities of term variables u_i, t_i that, along with $\vec{T}$, completely characterize the equalities and disequalities between the term variables. Specifically, N contains:
   (a) a disequality $t_i \neq t_j$ for every pair of distinct parameter variables;
   (b) a disequality $t_i \neq u_j$ for every pair of a parameter variable and a non-parameter variable for which the term $T_i(\vec{t})$ is not identical to u_i
   (c) a disequality $t_i \neq \mathsf{Leaf}$ for each parameter variable t_i.

   Note that for the remaining pairs of variables u_i and u_j, either the equality holds and $T_i(\vec{t}) = T_j(\vec{t})$ or the disequality holds and follows from the other disequalities and the fact that $T_i \neq T_j$. Note that, if $\vec{u} = u_1, \dots, u_m$ and $\vec{t} = t_1, \dots, t_n$, then the constraint $N(\vec{u}, \vec{t})$ can be denoted by $\mathsf{distinct}(u_1, \dots, u_m; t_1; \dots; t_n; \mathsf{Leaf})$;
3. $M(\vec{u}, \vec{t}, \vec{c})$ denotes a conjunction of formulas $c_i = \alpha(v_i)$ where v_i is a term variable and c_i is a collection variable;
4. F_E is a conjunction of literals of the form $e_i = e_j$ and $e_i \neq e_j$ for some element variables e_i, e_j;
5. F_C is a formula of the logic of collections (Figure 8).

Partial Evaluation of the Catamorphism. We next partially evaluate the catamorphism α with respect to the substitution σ_S obtained from unification. More precisely, we repeatedly apply the following rewriting on terms to terms contained in the subformula $M(\vec{u}, \vec{t}, \vec{c})$:

$$\begin{array}{rcl} \alpha(u) & \rightsquigarrow & \alpha(\sigma_S(u)) \\ \alpha(\mathsf{Node}(t_1, e, t_2)) & \rightsquigarrow & \mathsf{combine}(\alpha(t_1), e, \alpha(t_2)) \\ \alpha(\mathsf{Leaf}) & \rightsquigarrow & \mathsf{empty} \end{array}$$

After this transformation, α applies only to parameter variables. We introduce a variable c_i of $\mathcal{L}_C$ to ensure that for each parameter t_i we have an equality of the form $c_i = \alpha(t_i)$, unless such conjunct is already present. After adding conjuncts $c_i = \alpha(t_i)$ we can replace all occurrences of $\alpha(t_i)$ with c_i. We can thus replace, without changing the satisfiability of the formula (3), the subformula $M(\vec{u}, \vec{t}, \vec{c})$ with

$$M^1(\vec{t}, \vec{c}) \wedge F_C^1$$

where M^1 contains only conjunctions of the form $c_i = \alpha(t_i)$ and F_C^1 is a formula in $\mathcal{L}_C$.

Example. This is a crucial step of our decision procedure, and we illustrate it with a simple example. If $\vec{u} = \vec{T}(\vec{t})$ is simply the formula $u = \mathsf{Node}(t_1, e, t_2)$, then a possible formula N is

$$\mathsf{distinct}(t_1; t_2; u; \mathsf{Leaf})$$

A possible formula M is

$$c = \alpha(u) \wedge c_1 = \alpha(t_1)$$

After the partial evaluation of the catamorphism and introducing variable c_2 for $\alpha(t_2)$, we can replace M with

$$c_1 = \alpha(t_1) \wedge c_2 = \alpha(t_2) \wedge c = \mathsf{combine}(c_1, e, c_2)$$

where we denote the first two conjuncts by $M^1(c_1, c_2)$ and the third conjunct by F_C^1. (Here, combine is an expression in $\mathcal{L}_\mathcal{C}$ defining the catamorphism.)

Normal form After Evaluating Catamorphism. We next replace $\vec{u}$ by $\vec{T}(\vec{t})$ in (3) and obtain formula of the form

$$D \wedge E \tag{4}$$

where

1. $D \equiv N(\vec{T}(\vec{t}), \vec{t}) \wedge M^1(\vec{t}, \vec{c})$
2. $E \equiv F_E \wedge F_C \wedge F_C^1$

Expressing Existence of Distinct Terms. Note that E already belongs to the logic of collection $\mathcal{L}_\mathcal{C}$. To reduce (4) to a formula in $\mathcal{L}_\mathcal{C}$, it therefore suffices to have a mapping from D to some $\mathcal{L}_\mathcal{C}$-formula D_M. Observe that by using true as D_M we obtain a sound procedure for proving unsatisfiability. While useful, such procedure is not complete. To ensure completeness, we require that D and D_M are equisatisfiable. The appropriate mapping from D to D_M depends on $\mathcal{L}_\mathcal{C}$, and the properties of α. In sections 4.8, 4.9, 4.10 we give such mappings that ensure completeness for a number of logics $\mathcal{L}_\mathcal{C}$ and catamorphisms α.

Invoking Decision Procedure for Collections. Having reduced the problem to a formula in $\mathcal{L}_\mathcal{C}$ we invoke a decision procedure for $\mathcal{L}_\mathcal{C}$.

4.5 Example of Decision Procedure Run

We next give an example of the application of our decision procedure for the canonical set abstraction. Recall (2), our example from Section 2, now written as

$$\begin{array}{l} \forall t_1, t_2, t_3, t_4 : \mathsf{Tree},\ e_1, e_2 : \mathcal{E} \\ \quad t_1 = \mathsf{Node}(t_2, e_1, t_3) \Rightarrow \\ \quad \alpha(t_4) = \alpha(t_2) \cup \{e_2\} \Rightarrow \\ \quad \alpha(\mathsf{Node}(t_4, e_1, t_3)) = \alpha(t_1) \cup \{e_2\} \end{array} \tag{5}$$

To prove the validity of (5), we show the unsatisfiability of the following quantifier free formula

$$\begin{array}{l} t_1 = \mathsf{Node}(t_2, e_1, t_3) \\ \wedge\ \alpha(t_4) = \alpha(t_2) \cup \{e_2\} \\ \wedge\ \alpha(\mathsf{Node}(t_4, e_1, t_3)) \neq \alpha(t_1) \cup \{e_2\} \end{array} \tag{6}$$

After the application of purification and flattening, we get the following

$$\begin{array}{l} t_1 = \mathsf{Node}(t_2, e_1, t_3) \wedge t_5 = \mathsf{Node}(t_4, e_1, t_3) \\ \wedge\ c_1 = \alpha(t_4) \wedge c_2 = \alpha(t_2) \wedge c_3 = \alpha(t_5) \wedge c_4 = \alpha(t_1) \\ \wedge\ c_1 = c_2 \cup \{e_2\} \wedge c_3 \neq c_4 \cup \{e_2\} \end{array}$$

We now have to guess a partitioning over the set $\{t_1, \ldots, t_5, \mathsf{Leaf}\}$. Out of those we can pick, many will fail at unification, for instance if $t_1 = t_2$, or if $t_1 = t_5 \wedge t_2 \neq t_4$. As a more interesting case, we pick

$$\bigwedge_{1 \leq i < j \leq 5} t_i \neq t_j$$

We also pick that $e_1 \neq e_2$, although this has no consequence on the rest of this example.

Unification does not merge any term or variables and we obtain the following substitution function σ_S

$$\begin{array}{l} \sigma_S(t_1) = \mathsf{Node}(t_2, e_1, t_3) \\ \sigma_S(t_5) = \mathsf{Node}(t_4, e_1, t_3) \end{array}$$

Applying σ_S on the disequalities over trees does not yield any contradiction, so the constraints over tree terms are satisfiable.

For the sake of clarity, we now rename our tree variables so that the letters used in the description of the decision procedure and in this example match

$$t_1 \mapsto u_1 \quad t_2 \mapsto t_1 \quad t_3 \mapsto t_2 \quad t_4 \mapsto t_3 \quad t_5 \mapsto u_2$$

Our formula in normal form now reads as

$$\begin{array}{l} u_1 = \mathsf{Node}(t_1, e_1, t_2) \wedge u_2 = \mathsf{Node}(t_3, e_1, t_2) \\ \wedge\ \mathsf{distinct}(u_1, u_2; t_1; t_2; t_3; \mathsf{Leaf}) \\ \wedge\ c_1 = \alpha(t_3) \wedge c_2 = \alpha(t_1) \wedge c_3 = \alpha(u_2) \wedge c_4 = \alpha(u_1) \\ \wedge\ e_1 \neq e_2 \\ \wedge\ c_1 = c_2 \cup \{e_2\} \wedge c_3 \neq c_4 \cup \{e_2\} \end{array}$$

We can now partially apply α. The transformation on $c_3 = \alpha(u_2)$, for instance, is as follows

$$\begin{array}{rl} & c_3 = \alpha(u_2) \\ \rightsquigarrow & c_3 = \alpha(\mathsf{Node}(t_3, e_1, t_2)) \\ \rightsquigarrow & c_3 = \alpha(t_3) \cup \{e_1\} \cup \alpha(t_2) \end{array}$$

The completed application yields (note that we introduced c_5)

$$\begin{array}{l} \mathsf{distinct}(\mathsf{Node}(t_1, e_1, t_2), \mathsf{Node}(t_3, e_1, t_2); t_1; t_2; t_3; \mathsf{Leaf}) \\ \wedge\ c_1 = \alpha(t_3) \wedge c_2 = \alpha(t_1) \wedge c_5 = \alpha(t_2) \\ \wedge\ c_3 = c_1 \cup \{e_1\} \cup c_5 \wedge c_4 = c_2 \cup \{e_1\} \cup c_5 \\ \wedge\ e_1 \neq e_2 \\ \wedge\ c_1 = c_2 \cup \{e_2\} \wedge c_3 \neq c_4 \cup \{e_2\} \end{array}$$

Notice that replacing the first two lines of this formula (which correspond to D in the normal form (4)) by true gives us a formula in the theory of finite sets which is unsatisfiable. This subproblem is therefore unsatisfiable. To show that (6) is unsatisfiable, we would still have to try all other possible arrangements of equalities on the tree variables. In this case though, it is not difficult to see that adding more equalities between tree terms cannot make the formula satisfiable. We therefore conclude that our verification condition (5) is valid.

4.6 Soundness of the Decision Procedure

We show that each of our reasoning steps results in a logically sound conclusion. The soundness of the purification and flattening steps is straightforward: each time a fresh variable is introduced, it is constrained by an equality, so any model of the original formula will naturally extend to a model for the rewritten formula which contains additional fresh variables. Conversely, the restriction of any model for the rewritten formula to the initial set of variables will be a model for the original formula.

Our decision procedure relies on two case splittings. We will give an argument for the splitting on the partitioning of tree variables. The argument for the splitting on the partitioning of content variables is then essentially the same. Let us call ϕ the formula before case splitting. Observe that for each partitioning, the resulting subproblem contains a strict superset of the constraints of the original problem, that is, each subproblem is expressible as a formula $\phi \wedge \psi$, where ψ does not contain variables not appearing in ϕ. Therefore, if, for any of the subproblems, there exists a model $\mathcal{M}$ such that $\mathcal{M} \models \phi \wedge \psi$, then $\mathcal{M} \models \phi$ and $\mathcal{M}$ is also a model for the original problem. For the converse, assume the existence of a model $\mathcal{M}$ for the original problem. Construct the relation $\sim$ over the tree variables $t_1, \ldots, t_n$ of ϕ as follows:

$$t_i \sim t_j \iff \mathcal{M} \models t_i = t_j$$

Clearly, $\sim$ is an equivalence relation and thus there is a subproblem for which the equality over the tree variables is determined by $\sim$. It is not hard to see that $\mathcal{M}$ is a model for that subproblem. It is therefore sound to reduce the satisfiability of the main problem to the satisfiability of at least one of the subproblems.

Our unification procedure is a straightforward adaptation from a textbook exposition of the algorithm and the soundness arguments can be lifted from there [2, Page 451].

The soundness of the evaluation of α follows from its definition in terms of empty and $\mathsf{combine}$. Introducing fresh variables c_i in the form of equalities $c_i = \alpha(t_i)$ is again sound, following the same argument as for the introduction of tree variables during flattening. The subsequent replacement of terms of the form $\alpha(t_i)$ by their representative variable c_i is sound: any model for the formula without the terms $\alpha(t_i)$ can be trivially extended to include a valuation for them. Finally, the replacement of the tree variables $\vec{u}$ by the terms $\vec{T}(\vec{t})$ is sound, because unification enforces that any model for the formula before the substitution must have the same valuation for u_i and the corresponding term T_i. Therefore, there is a direct mapping between models for the formula before and after the substitution.

4.7 Complexity of the Reduction

Our decision procedure reduces formulas to normal form in non-deterministic polynomial time because it performs guesses of equivalence relations on polynomially many variables, runs the unification algorithm, and does partial evaluation of the catamorphism at most once for each appropriate term in the formula. The reduction is therefore in the same complexity class as the pure theory of recursive data structures [5]. In addition to the reduction, the overall complexity of the decision problem also depends on the formula D_M, and on the complexity of solving the resulting constraints in the collection theory.

4.8 Canonical Set Abstraction

We next give a complete procedure for the canonical set abstraction, where $\mathcal{C}$ is the structure of all finite sets with standard set algebra operations, and α is given by

$$\mathsf{empty} = \emptyset$$
$$\mathsf{combine}(c_1, e, c_2) = c_1 \cup \{e\} \cup c_2$$

Observations about α. Note that, for each term $t \neq \mathsf{Leaf}$, $\alpha(t) \neq \emptyset$. Let $e \in \mathcal{E}$ and consider the set $S = \alpha^{-1}(\{e\})$ of terms that map to $\{e\}$. Then S is the set of all non-leaf trees that have e as the only stored element, that, is, S is the least set such that

1. $\mathsf{Node}(\mathsf{Leaf}, e, \mathsf{Leaf}) \in S$, and
2. $t_1, t_2 \in S \rightarrow \mathsf{Node}(t_1, e, t_2) \in S$.

Thus, $\alpha^{-1}(\{e\})$ is infinite. More generally, $\alpha^{-1}(c)$ is infinite for every $c \neq \emptyset$, because each tree that maps into a one-element subset of c extends into some tree that maps into c.

Expressing Existence of Distinct Terms using Sets. We can now specify the formula D_M that is equisatisfiable to the formula D in (4).

Definition 1. *If $c_1, \ldots, c_n$ are the free variables in D, then (for theory $\mathcal{C}$ and α given above) define D_M as*

$$\bigwedge_{i=1}^{n} c_i \neq \emptyset$$

To argue why this choice gives a complete decision procedure, it will be useful to review the following.

Lemma 2. *(Independence of Disequations Lemma, variant of [38], [14, Page 178]) Let D_0 be a conjunction of disequations of terms built from tree variables $t_1, \ldots, t_m$ and symbols Node, Leaf. Suppose that D_0 does not contain a trivial disequation $T \neq T$ for any term T. If $A_1, \ldots, A_m$ are infinite sets of trees, then D_0 has a satisfying assignment such that for each i where $1 \leq i \leq m$, the value t_i belongs to A_i.*

Proof. We first show that we can reduce the problem to a simpler one where the disequalities all have the form $t_i \neq T_j$, then show how we can construct a satisfying assignment for a conjunction of such disequalities.

We start by rewriting each disequality $\neg(T = T')$ in the form:

$$\neg \left(\begin{array}{ll} & \ldots \\ \wedge & t_i = C_i(t_{i,k}, \ldots, t_{i,l}) \\ \wedge & t_j = C_j(t_{j,k}, \ldots, t_{j,l}) \\ \wedge & \ldots \end{array} \right)$$

where the conjunction of equalities is obtained by unifying the terms T and T'. Here, the expressions of the form $C(t_{i,k}, \ldots, t_{i,l})$ denote terms built using Node, Leaf and the variables $t_{i,k}, \ldots, t_{i,l}$. Note that each of these variables does appear at least once in the term. After applying this rewriting to all disequalities and converting the resulting formula to disjunctive normal form, we obtain a problem of the form

$$\bigvee \bigwedge K$$

where each conjunct K is of the form

$$t_i \neq C_{i,j}(t_{i,j,k}, \ldots, t_{i,j,l})$$

In other words, each variable t_i can be on the left-hand side of several disequalities in the same conjunction. Due to the form of the equations obtained from unification, the set of variables $\{t_{i,j,k}, \ldots, t_{i,j,l}\}$ never contains t_i. This formula is logically equivalent to the original one from the statement of the lemma. To show that it is satisfiable, we now need to show that one of its disjuncts is satisfiable.

We construct a satisfying assignment for such a conjunction as follows: we start by collecting all disequalities of the form $t_1 \neq T$, where T is a ground term. We pick for t_1 a value T_1 in A_1 different from all such Ts. This is always possible as there are finitely many disequalities in the conjunction and infinitely many trees in A_1. We substitute in all disequalities T_1 for t_1. For all indices $i \in \{2, \ldots, m\}$ we now do the following: we collect all disequalities of the forms $t_i = T$ and $T = C(t_i)$ (in the second form, $C(t_i)$ denotes a term built with Leaf, Node, at least one occurrence of t_i, and no other variable). Note that for each of these disequalities, there is *at most* one value for t_i which contradicts it. Therefore, we can always pick a value T_i in A_i for t_i which satisfies all the disequalities. We then substitute T_i for t_i in the conjunction and proceed with t_{i+1}. It is not hard to see that this procedure terminates once all variables have been assigned a value and that the complete assignment is a satisfying one. ■

Lemma 3. *For $\mathcal{C}$ denoting the structure of finite sets and α given as above, $\exists \vec{t}.D$ is equivalent to D_M.*

Proof. Let $\vec{t}$ be $t_1, \ldots, t_n$. Fix values $c_1, \ldots, c_n$. We first show $\exists \vec{t}.D$ implies D_M. Pick values $t_1, \ldots, t_n$ for which D holds. Then $t_i \neq \mathsf{Leaf}$ holds because this conjunct is in D. Therefore, $\alpha(t_i) \neq \emptyset$ by property of α above. Because $c_i = \alpha(t_i)$ is a conjunct in D, we conclude $c_i \neq \emptyset$. Therefore, D_M holds as well.

Conversely, suppose D_M holds. This means that $c_i \neq \emptyset$ for $1 \leq i \leq n$. Let $A_i = \alpha^{-1}(c_i)$ for $1 \leq i \leq n$. Then the sets A_i are all infinite by the observation above. By Lemma 2 there are values $t_i \in A_i$ for $1 \leq i \leq n$ such that the disequations in $N(\vec{T}(\vec{t}), \vec{t})$ hold. By definition of A_i, $M^1(\vec{t}, \vec{c})$ is also true. Therefore, D is true in this assignment. ■

Complexity for the Canonical Set Abstraction. We have observed earlier that reduction to $\mathcal{L}_{\mathcal{C}}$ is an NP process. There are several decision procedures that support reasoning about sets of elements and support standard set operations. One of the most direct approaches to obtain such a decision procedure [31] is to use an encoding into first-order logic, and observe that the resulting formulas belong to the Bernays-Schönfinkel-Ramsey class of first-order logic with a single universal quantifier. Checking satisfiability of such formulas is NP-complete [9]. It is also possible to extend this logic to allow stating that two sets have the same cardinality, and the resulting logic is still within NP [32]. Because the reduction, the generation of D_M and the decision problem for $\mathcal{L}_\mathcal{C}$ are all in NP, we conclude that the decision problem for recursive data structures with the canonical set abstraction belongs to NP.

4.9 Infinitely Surjective Abstractions

The canonical set abstraction is a special case of what we call *infinitely surjective abstractions*, for which we can compute the formula D_M.

Definition 4 (Infinitely Surjective Abstraction)**.** *If S is a set of trees, we call a domain $\mathcal{C}$ and a catamorphism α an infinitely surjective S-abstraction iff for every tree $t \notin S$, the set $\alpha^{-1}(\alpha(t))$ is infinite.*

We can compute D_M for an infinitely surjective S-abstraction whenever S is finite.

Canonical set abstraction is an infinitely surjective $\{\mathsf{Leaf}\}$-abstraction. Another infinitely surjective $\{\mathsf{Leaf}\}$-abstraction is the term size abstraction, which for a given tree computes its size, as the number of internal nodes.

An example of infinitely surjective $\emptyset$-abstractions is the function that computes the set of free variables in an abstract syntax tree representing a lambda expression or a formula. Note that for each finite set of variables, there exist infinitely many terms that have these bound variables.

Among other important examples of infinitely surjective $\emptyset$-abstractions are most non-trivial recursively defined invariants, such as the property that a tree is sorted or that it has the heap ordering.

The general idea for computing D_M for an infinitely surjective S-abstraction is to add the elements $T_1, \ldots, T_m$ of S into the unification algorithm and guess arrangements over them. This will ensure that, in the resulting formula, all parameters are distinct from any T_i. The formula D_M then states the condition $c_i \notin \alpha[S]$ where $\alpha[S]$ denotes the image of S under the catamorphism. We omit the details, but we note that the algorithm for $\{\mathsf{Leaf}\}$-abstractions works also for $\emptyset$-abstractions.

4.10 Completeness for Lists and Multisets

We finally examine certain natural abstractions of trees using lists and multisets. The common feature for these catamorphisms is that the $\mathsf{combine}$ function always produces a collection whose size is strictly larger than the sizes of its first and third argument. Because there are only finitely many binary trees of a given size, it follows that there are only finitely many trees that map to any given collection. Consequently, such catamorphisms are not infinitely surjective and the method of the previous section does not apply.

Nonetheless, we can still obtain decidability in these cases because the number of terms that map to a given collection grows as the size of the collection grows. That is, if $|c|$ denotes some notion of size of a collection c, then

$$\lim_{n\to\infty}\left(\inf_{c:|c|\geq n}|\alpha^{-1}(c)|\right) = \infty$$

For a tree t, let $|t|$ be defined by $|\mathsf{Leaf}| = 0$, $|\mathsf{Node}(t_1, e, t_2)| = |t_1| + 1 + |t_2|$.

Lists with concatenation. Consider the structure of all finite lists with the concatenation operator $\mathtt{++}$ and the operation $\mathsf{List}(e_1, \ldots, e_n)$ to denote the finite list with the specified elements $e_1, \ldots, e_n$. Consider the catamorphism for infix traversal of the tree, given by

$$\mathsf{empty} = \mathsf{List}()$$
$$\mathsf{combine}(c_1, e, c_2) = c_1 \mathbin{+\!+} \mathsf{List}(e) \mathbin{+\!+} c_2$$

(catamorphisms for pre-order and post-order traversal can be handled analogously). Let $|c|$ denote the number of elements in the list. Clearly $|t| = |\alpha(t)|$.

For a fixed c, the number of trees t for which $\alpha(t) = c$ is equal to the number of binary trees with $k = |c|$ nodes (denoted C_k) and grows exponentially with k.

Consider a formula $N(\vec{T}(\vec{t}), \vec{t})$ where $\vec{T}$ is $T_1, \ldots, T_m$ and $\vec{t}$ is $t_1, \ldots, t_n$. Let k be the least positive integer such that $C_k \geq m + n$.[1] For each $1 \leq i \leq n$ let $e^i_1, \ldots, e^i_k$ be fresh element variables, and let $r^i_1, \ldots, r^i_p$ be the enumeration over all expressions denoting finite trees r^i with $\alpha(r^i) = \mathsf{List}(e^i_1, \ldots, e^i_l)$ for $l < k$. Consider the formula

$$D \wedge \bigwedge_{i=1}^{n}(|t_i| \geq k \vee \bigvee_{j=1}^{p} t_i = r^i_j) \tag{7}$$

Let D^w be a disjunct in the disjunctive normal form of (7). Observe that in D^w, every variable t_i is either assigned to some constant tree r^i_j, or the constraint $|t_i| \geq k$ is present. For all t_i assigned to a tree, substitute r^i_j for t_i in D (D always appears as a conjunct of D^w) and partially evaluate the catamorphism over r^i_j. The remaining tree variables in D^w must satisfy the constraint $|t_i| \geq k$.

In the language of lists, we represent $|t_i| \geq k$ by $c_i = \mathsf{List}(e^i_1, \ldots, e^i_k) \mathbin{+\!+} c'_i$ (note that $c_i = \alpha(t_i)$ is a conjunct in D^w because it is a conjunct of D). Notice that the number of disequations between parameters does not increase, since they only appear in D. Because t_i is sufficiently large, even for the satisfying assignment where all $\alpha(t_i)$ are assigned to the same collection, it is possible to pick the values for t_i to be distinct. It then remains to check whether D^w is satisfiable and derive equalities between element variables e^i_j, similarly to the unification algorithm in Section 4.4. Since we can do this for any disjunct D^w, we can take as the disjunct of D_M the satisfiable disjuncts D^w_M containing the appropriate conditions $c_i = \mathsf{List}(e^i_1, \ldots, e^i_k) \mathbin{+\!+} c'_i$ and the result of partially evaluating the catamorphisms over the substituted terms r^i_j. By construction, these satisfiable disjuncts will cover all possible assignments to the variables t_i and therefore D_M will be equisatisfiable to D, as required by our completeness argument.

Multisets with disjoint union. Consider now similarly the structure of all finite multisets with the disjoint union operator $\uplus$, such as the one in [52, 51]. Consider the catamorphism given by $\mathsf{empty} = \emptyset$ and $\mathsf{combine}(c_1, e, c_2) = c_1 \uplus \{e\} \uplus c_2$ that is similar to canonical set abstraction but preserves the number of occurrences of each element in the data structure. Let $|c|$ denote the number of elements in the multiset, counting each element as many times as it appears. Then also $|t| = |\alpha(t)|$. Note that if a multiset c has the total number of occurrences k then the number of trees that map into c is the number C_k of trees that map into one fixed list corresponding to the multiset times the number of distinct lists that map into the same multiset. Thus, there are at least C_k trees that map into the multiset c. Therefore, the previous argument and the definition of D_M applies in this case as well. The differences is simply that the catamorphism is different and that the bound $|t| \geq k$ is expressed in D_M by $\alpha(c_i) = \{e^i_1, \ldots, e^i_k\} \uplus c'_i$. Besides, the bound p in the enumeration of finite trees mapping to a given collection is larger.

[1] Note that $k \sim O(\log(m + n))$.

Example Consider the following formula where α is the multiset abstraction:

$$\begin{aligned} & c_1 = \alpha(t_1) \wedge c_2 = \alpha(t_2) \wedge c_3 = \alpha(t_3) \\ \wedge\ & \mathsf{distinct}(t_1; t_2; t_3; \mathsf{Leaf}) \\ \wedge\ & c_1 = c_2 \wedge c_2 = c_3 \wedge c_3 = c_1 \wedge |c_1| < 2 \end{aligned}$$

The formula is already in its normal form after partial evaluation as described in Section 4.4. We have that:

$$\begin{aligned} N &\equiv \mathsf{distinct}(t_1; t_2; t_3; \mathsf{Leaf}) \\ M^1 &\equiv c_1 = \alpha(t_1) \wedge c_2 = \alpha(t_2) \wedge c_3 = \alpha(t_3) \\ F_C &\equiv c_1 = c_2 \wedge c_2 = c_3 \wedge c_3 = c_1 \wedge |c_1| < 2 \end{aligned}$$

We need to pick the smallest k such that $C_k \geq 4$, which is 3 ($C_2 = 2$, $C_3 = 5$). The enumeration over the finite trees r is as follows:

$$\begin{aligned} r_1^1 &= \mathsf{Node}(\mathsf{Leaf}, e_1^1, \mathsf{Leaf}) \\ r_2^1 &= \mathsf{Node}(\mathsf{Node}(\mathsf{Leaf}, e_2^1, \mathsf{Leaf}), e_1^1, \mathsf{Leaf}) \\ r_3^1 &= \mathsf{Node}(\mathsf{Node}(\mathsf{Leaf}, e_1^1, \mathsf{Leaf}), e_2^1, \mathsf{Leaf}) \\ r_4^1 &= \mathsf{Node}(\mathsf{Leaf}, e_1^1, \mathsf{Node}(\mathsf{Leaf}, e_2^1, \mathsf{Leaf})) \\ r_5^1 &= \mathsf{Node}(\mathsf{Leaf}, e_2^1, \mathsf{Node}(\mathsf{Leaf}, e_1^1, \mathsf{Leaf})) \end{aligned}$$

...and similarly for the trees r_j^2 and r_j^3.

The disjunctive normal form of (7) for our example looks as follows:

$$\begin{aligned} D_M \equiv (& N \wedge M^1 \wedge t_1 = r_1^1 \wedge t_2 = r_1^2 \wedge t_3 = r_1^3 \\ \vee\ & N \wedge M^1 \wedge t_1 = r_2^1 \wedge t_2 = r_1^2 \wedge t_3 = r_1^3 \\ \vee\ & N \wedge M^1 \wedge \ldots \\ \vee\ & N \wedge M^1 \wedge |t_1| \geq 3 \wedge t_2 = r_1^2 \wedge t_3 = r_1^3 \\ \vee\ & N \wedge M^1 \wedge \ldots \\ \vee\ & N \wedge M^1 \wedge |t_1| \geq 3 \wedge |t_2| \geq 3 \wedge |t_3| \geq 3) \end{aligned}$$

Note that all (216) disjuncts of D_M are satisfiable, since we can always pick the values e_j^i to be distinct for different values of i and we have no constraints on the trees except that they need to be distinct.

However, the complete formula $D_M \wedge F_C$ can now be shown to be unsatisfiable: indeed none of the assignments of constant trees to the variables t_i can satisfy the additional constraints that the distinct trees have the same multiset abstraction yet are of size less than 2. (If we replace the constraint $|c_1| < 2$ by $|c_1| \leq 2$, for example, the formula becomes satisfiable.) Note that the incomplete procedure mentioned at the end of Section 4.4 could not have detected the unsatisfiability (or the satisfiability in the second case), since the key here is to consider the exhaustive set of assignments with sufficiently small trees.

Complexity for the Multiset Abstraction. The construction of D_M that we exposed does not immediately yield an algorithm in NP. Note however that a non-deterministic algorithm would only have to guess the values needed to build a satisfying assignment for the tree variables. There are only polynomially many of these, since the size of each tree is polynomial function of k, which itself grows logarithmically with the number of distinct terms. The complexity of the decision procedure for the multiset abstraction is therefore NP.

Complexity for the List Abstraction. The best known decision procedure for reasoning about lists with concatenation is already in PSPACE and our combined decision procedure for that abstraction is therefore in the same complexity class.

5. Related Work

One reason why we find our result useful is that it can leverage a number of existing decidability results. In the area of recursive data structures [5] presents an abstract approach that can be used to obtain more efficient strategies for recursive data structures than the one that we chose to present. For reasoning about sets and multisets one expressive approach is the use of the decidable array fragment [10]. Optimal complexity bounds for reasoning about sets and multisets in the presence of cardinality constraints have been established in [32, 52]. Building on these results, extensions to certain operations on vectors has been presented in [36]. Reasoning about lists with concatenation can be done using Makanin's algorithm [37] and its improvements [53]. A different class of constraints uses rich string operations but imposes bounds on string length [8].

Our parameterized decision procedure presents one particular approach to combining logics. Standard results in this field are Nelson-Oppen combination [44]. Nelson-Oppen combination is not sufficient to encode catamorphisms because the disjointness conditions are not satisfied, but is very useful in obtaining interesting decidable theories to which the catamorphism can map a recursive data structure; such compound domains are especially of interest when using catamorphisms to encode invariants. There are combination results that lift the stable infiniteness restriction of the Nelson-Oppen approach [57, 25, 18] as well as disjointness condition subject to a local finiteness condition [20]. An approach that allows theories to share set algebra with cardinalities is presented in [60]. None of these approaches handle the problem of reasoning about a catamorphism from the theory of algebraic data types.

A technique for connecting two theories through homomorphic functions has been explored in [1]. We were not able to derive our decision procedure from [1], because the combination technique in [1] requires the homomorphism to hold between two copies of some shared theory Ω_0 that is locally finite, but our homomorphisms (i.e. catamorphisms) are defined on term algebras (i.e. recursive structures), which are not locally finite.

Related to our partial evaluation of the catamorphism is the phenomenon of local theory extensions [22], where axioms are instantiated only to terms that already exist syntactically in the formula. In our case of tree data types, the decision procedure must apply the axioms also to some consequences of the formula, obtained using unification, so an extension of the basic local theory framework is needed. To the best of our knowledge, the machinery of local theory extensions has not been applied the theory of algebraic data types.

The proof decidability for term powers [29, 30] introduces homomorphic functions that map a term into 1) a simplified "shape" term that ignores the stored elements and 2) the set of elements stored in the term. However, this language was meant to address reasoning about structural subtyping and not transformation of algebraic data types. Therefore, it does not support the comparison of the set of elements stored in distinct terms, and it would not be applicable to the verification conditions we consider in this paper. Furthermore, it does not apply to multisets or lists.

In [65] researchers describe a decision procedure for recursive data structures with size constraints and in [39] a decision procedure for trees with numeric constraints that model invariants of red-black trees. Our decision procedure supports reasoning about not only size, but also the content of the data structure. We remark that [65] covers also the case of a finite number of atoms, whereas we have chosen to focus on the case of infinite set of elements $\mathcal{E}$. Overall term algebras have an extensively developed theory, and enjoy many desirable properties, including quantifier elimination [38]; quantifier elimination also carries over to many extensions of term algebras [14, 30, 55, 65, 64]. We remark that in examples such as multisets we cannot expect quantifier elimination to hold.

Some aspects of our decision procedure are similar to folding and unfolding performed when using types to reason about data structures [24, 58, 46, 54, 61]. One of our goals was to understand the completeness or possible sources of incompleteness of such techniques. We do not aim to replace the high-level guidance available in such successful systems, but expect that our results can be used to further improve such techniques.

Several decision procedures are suitable for data structures in imperative programs [34, 33, 41, 59, 43]. There is seemingly nothing preventing such logics to be used also for functional programs. However, these logics alone fail to describe algebraic data types because they cannot express extensional equality and disequality of entire tree data structure instances, or the construction of new data structures from smaller ones. Even verification systems that aim to reason about imperative programs but use annotations [3, 7, 26, 62] typically rely on declarative languages. In such situations, the ability to use recursive functions in the specification language is very valuable.

The SMT-LIB standard [4] for SMT provers currently does not support recursive data structures, even though several provers support it in their native input languages [6, 15]. By providing new opportunities to use decision procedures based on algebraic data types, our results present a case in favor of incorporating such data types into standard formats. Our new decidability results also support the idea of using rich specification languages that admit certain recursively defined functions.

6. Conclusions

We have presented a decision procedure that extends the well-known decision procedure for algebraic data types. The extension enables reasoning about the relationship between the values of the data structure and the values of a recursive function (catamorphism) applied to the data structure. The presence of catamorphisms gives great expressive power and provides connections to other decidable theories, such as sets, multisets, lists. It also enables the computation of certain recursive invariants. Our decision procedure has several phases: the first phase performs unification and solves the recursive data structure parts, the second applies the recursive function to the structure generated by unification. The final phase is more subtle, is optional from the perspective of soundness, but ensures completeness of the decision procedure.

Automated decision procedures are widely used for reasoning about imperative programs. Functional programs are claimed to be more amenable to automated reasoning—this was among the original design goals of functional programming, and has been supported by experience from type systems and interactive proof assistants. Our decision procedure further supports this claim, by showing a wide range of properties that can be predictably proved about functional data structures.

References

[1] F. Baader and S. Ghilardi. Connecting many-sorted theories. In *CADE*, pages 278–294, 2005.

[2] F. Baader and W. Snyder. Unification Theory. In *Handbook of Automated Reasoning*, pages 445–532. 2001.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[4] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2009.

[5] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. *Electronic Notes in Theoretical Computer Science*, 174(8):23–37, 2007.

[6] C. Barrett and C. Tinelli. CVC3. In *CAV*, volume 4590 of *LNCS*, 2007.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[8] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321, 2009.

[9] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[10] A. R. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.

[11] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, 1991.

[12] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[13] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *Conf. Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, 2009.

[14] H. Comon and C. Delor. Equational formulae with membership constraints. *Information and Computation*, 112(2):167–216, 1994.

[15] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[16] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.

[17] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *POPL*, pages 281–292, 2004.

[18] P. Fontaine. Combinations of theories and the bernays-schönfinkel-ramsey class. In *VERIFY*, 2007.

[19] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *16th Conf. Computer Aided Verification*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[20] S. Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 33(3-4):221–249, 2005.

[21] W. Hodges. *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993.

[22] C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On local reasoning in verification. In *TACAS*, pages 265–281, 2008.

[23] J. Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.

[24] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.

[25] S. Krstic, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In *TACAS*, volume 4424 of *LNCS*, pages 602–617, 2007.

[26] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[27] V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), December 2006.

[28] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. `http://dx.doi.org/10.1007/s10817-006-9042-1`.

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

[30] V. Kuncak and M. Rinard. Structural subtyping of non-recursive types is decidable. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science*, 2003.

[31] V. Kuncak and M. Rinard. Decision procedures for set-valued fields. In *1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, 2005.

[32] V. Kuncak and M. Rinard. Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In *CADE-21*, 2007.

[33] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*, 2008.

[34] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, 2006.

[35] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th Int. Conf. Verification, Model Checking and Abstract Interpretation*, 2005.

[36] P. Maier. Deciding extensions of the theories of vectors and bags. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403, 2009.

[37] G. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, pages 129–198, 1977. (In AMS, (1979)).

[38] A. I. Mal'cev. Chapter 23: Axiomatizable classes of locally free algebras of various types. In *The Metamathematics of Algebraic Systems*, volume 66, page 262. North Holland, 1971. (Translation, original in Doklady, 1961).

[39] Z. Manna, H. B. Sipma, and T. Zhang. Verifying balanced trees. In *LFCS*, pages 363–378, 2007.

[40] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Comm. A.C.M.*, 3:184–195, 1960.

[41] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.

[42] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, volume 523 of *LNCS*, 1991.

[43] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[44] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.

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

[46] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape, size and bag properties via separation logic. In *VMCAI*, 2007.

[47] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[48] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[49] D. C. Oppen. Reasoning about recursively defined data structures. In *POPL*, pages 151–157, 1978.

[50] C. H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.

[51] R. Piskac and V. Kuncak. Decision procedures for multisets with cardinality constraints. In *VMCAI*, number 4905 in LNCS, 2008.

[52] R. Piskac and V. Kuncak. Linear arithmetic with stars. In *CAV*, 2008.

[53] W. Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3), 2004.

[54] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

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

[56] Scalacheck - a powerful tool for automatic unit testing. `http://code.google.com/p/scalacheck/`.

[57] C. Tinelli and C. Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3), 2005.

[58] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, 2000.

[59] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.

[60] T. Wies, R. Piskac, and V. Kuncak. Combining theories with shared set operations. In *FroCoS: Frontiers in Combining Systems*, 2009.

[61] H. Xi. Dependently typed pattern matching. *Journal of Universal Computer Science*, 9(8):851–872, 2003.

[62] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2008.

[63] K. Zee, V. Kuncak, M. Taylor, and M. Rinard. Runtime checking for program verification. In *Workshop on Runtime Verification*, volume 4839 of *LNCS*, 2007.

[64] T. Zhang, H. B. Sipma, and Z. Manna. The decidability of the first-order theory of knuth-bendix order. In *CADE*, pages 131–148, 2005.

[65] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. *Inf. Comput.*, 204(10):1526–1574, 2006.

[66] D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *PADL*, 2005.