Decision Procedures for Algebraic Data Types with Abstractions

Philippe Suter Mirco Dotta Viktor Kuncak *

School of Computer and Communication Sciences (I&C) - École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{firstname.lastname}@epfl.ch

Abstract

We describe a family of decision procedures that extend the decision procedure for quantifier-free constraints on recursive algebraic data types (term algebras) to support recursive abstraction functions. Our abstraction functions are catamorphisms (term algebra homomorphisms) mapping algebraic data type values into values in other decidable theories (e.g. sets, multisets, lists, integers, booleans). Each instance of our decision procedure family is sound; we identify a widely applicable many-to-one condition on abstraction functions that implies the completeness. Complete instances of our decision procedure include the following correctness statements: 1) a functional data structure implementation satisfies a recursively specified invariant, 2) such data structure conforms to a contract given in terms of sets, multisets, lists, sizes, or heights, 3) a transformation of a formula (or lambda term) abstract syntax tree changes the set of free variables in the specified way.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Languages, Verification

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 [16, 6, 4] were used to verify tens of thousands of lines of imperative code [14, 3] and prove complex correctness conditions of imperative data structures [67, 50]. While much recent work on automation was invested into imperative languages, it is interesting (50 years after [44]) to consider the reach of such decision procedures when applied to *functional* programming languages, which were designed with the ease of reasoning as one of the explicit goals. Researchers have explored the uses of advanced type systems to check expressive properties [19, 66], and have recently also applied satisfiability modulo theory solvers to localized type system constraints [29, 59]. Among the recognized benefits of theorem provers is efficient support for propositional operators and arithmetic. Our paper revives another direction where theorem provers can play a prominent role: complete reasoning about certain families of functions operating on algebraic data types.

We embrace a functional language both as the implementation language and 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 [13] and bounded-exhaustive testing [10, 23]. In using the programming language as the specification language, our work is in line with soft typing approaches that originated in untyped functional languages [12], 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 [52] present a well-defined and interesting benchmark for automated reasoning about functional programs. Data structures come with well-understood specifications: they typically implement ordered or unordered collections of objects, or maps between objects. To express the desired properties of data structures, we often 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 [32, 39, 70, 18]. 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 then

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

Here α denotes an abstraction 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 [14, 67] does not explicitly define a mapping α but instead introduces a fresh *ghost* variable to represent the 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 technique yields simple verification conditions. However, this technique can impose additional annotation overhead—for the tree example above it would require supplying a set as an additional argument to each algebraic data type constructor. To eliminate this overhead and to decouple the specification from the implementation, we use recursively defined abstraction functions that compute the abstract value for each concrete data structure. As a result, our verification conditions contain user-defined function definitions that manipulate rich data types, along

* This research is supported in part by the Swiss National Science Foundation Grant #120433 “Precise and Scalable Analyses for Reliable Software”.

POPL'10, January 17–23, 2010, Madrid, Spain.

with equations and disequations involving such functions. Our goal is to develop decision procedures that can reason about interesting fragments of such a language.

We present decision procedures for reasoning about algebraic data types with user-defined abstraction functions expressed as a fold, or catamorphisms [46], over algebraic data types. These decision procedures subsume approaches for reasoning about algebraic data types [53] and add the ability to express constraints on 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 [36, 33]. 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 [65].

Our decision procedures are not limited to using sets as an abstract view of data structures. The most important condition for applicability is that the notion of a collection has a decidable theory in which the fold can be expressed. This includes in particular arrays [11], multisets with cardinality bounds [55, 56], and even option types over integer elements. Each abstract value provides different possibilities for defining the fold function.

We believe that we have identified an interesting region 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 approach works not only for sets but also for lists and multisets, and even for abstractions that encode the truth-values of 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 family of decision procedures for the quantifier-free theory of algebraic data types with different fold functions (sections 4 and 5). We establish the soundness of the decision procedures, and provide sufficient conditions on the fold function that guarantee the completeness of the decision procedure (Section 5). The intuition behind the completeness condition is to require the inverse image of the fold to have sufficiently large cardinality for sufficiently large abstract values (Section 5.3). We list several examples of interest that satisfy this condition.

2. Example

Figure 1 shows Scala [51] 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
```

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, they use the abstraction function content to view these trees as sets. The abstraction function is declared like any other function and is executable, but we assume that it obeys a syntactic restriction to make it a tree fold, as described in the next section.

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—such information would be useless to a user who has no access to the internal structure. Instead, the postconditions 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 (functions signatures and contracts) from the implementation, developers obtain better opportunities for code reuse, manual proofs become simpler [24], and automated analysis of clients becomes more tractable [32].

```
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 a set

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 the singleton set {e} (containing containing the element e). We assume the implementation of the container library to be correct, and map the container operations (e.g. ++) to the corresponding ones in the mathematical theory of finite sets (e.g. $\cup$) when we reason about the programs.

The advantages of using an abstraction function in specifications are numerous, but they also require verification systems that can reason about these user-defined functions—these functions appear in contracts and therefore in verification conditions. For example, consider the function add in Figure 1 and apply the standard technique to replace recursive function call with the function contract. The result is a set of verification conditions including (among others) the condition:

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

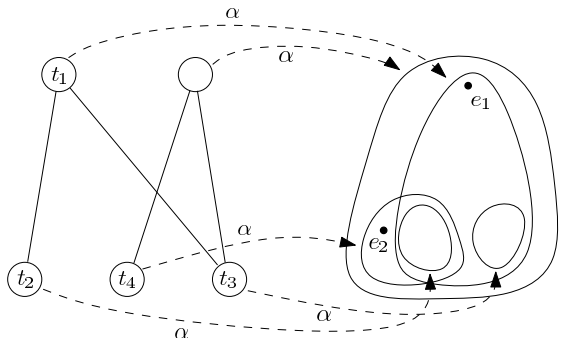

Figure 2. Illustration of (2), where edges labeled by α denote applications of the content abstraction function

This formula is graphically represented in Figure 2. It 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. It 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.

3. Reasoning about Algebraic Data Types with Abstraction Functions

Decision procedures for reasoning about algebraic data types [53, 5] are concerned with proving and disproving quantifier-free formulas that involve constructors and selectors of an algebraic data type, such as the immutable version of heterogeneous lists in LISP. They generalize the unification algorithms used in theorem proving [58] and Hindley-Milner type inference. 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* [25, 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 algebraic data types with the ability to specify an *abstract value* of the 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 [36, 56, 55, 40, 28]. 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 underspecification for interfaces of data structures. The decision procedures we describe enable proving formulas that relate data structures implemented as algebraic data types to their abstract values that specify the observable behavior of these data types. They can thus increase the automation when verifying correctness of functional data structures.

3.1 Instances of our Decision Procedure

Our decision procedures for different fold functions follow the same pattern, so we talk about them as instances of one generic decision procedure. The choice of the type of data stored in the tree in each decision procedure instance is largely unconstrained; the procedures work 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 [30]). The decision procedure is parameterized by 1) an element type $\mathcal{E}$, 2) a collection type $\mathcal{C}$, and 3) an abstraction function α (generalizing the

```
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 4. Computing the set of free variables in a λ-calculus term

content function in Figure 1). We require the abstraction function to be a catamorphism (generalized fold) [46]. We focus on the case of binary trees, so we require an abstraction function of the form

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

for some functions empty : $\mathcal{C}$ and combine : $(\mathcal{C}, \mathcal{E}, \mathcal{C}) \to \mathcal{C}$.

Figure 3 summarizes some of the instances of our decision procedure. 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 $\mathcal{L}_\mathcal{C}$. (The decision procedure for $\mathcal{L}_\mathcal{C}$ is invoked as the last step of our decision procedure.) Figure 3 shows that our decision procedure covers a wide range of collection abstractions of interest, as well as some other relevant functions definable as folds. We describe some of these cases in more detail.

Set Abstractions. The content function in Figure 1 is an example of a fold 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*.

The canonical set abstraction is not the only interesting abstraction function whose result is a set. Figure 4 shows another example, where the fold free computes a set by adding and removing elements as the tree traversal goes. Such abstraction function can then be used to prove that, e.g., a rewriting step on a λ-calculus term does not increase the set of free variables in the term.

Abstractions using Multisets and Lists. A set abstracts both the order and the multiplicity (the number of occurrences) of elements in the data structure. A more precise abstraction is a multiset (bag) (Figure 3), which preserves the multiplicity. Moreover, the decision procedure for multisets [55, 56] supports an abstraction function that abstracts a multiset into the underlying set, which enables simultaneous use of trees, multisets and sets in the same specification, giving a decision procedure for an interesting fragment of the tree-list-bag-set hierarchy [26]. Even more precise abstractions of trees use lists, supporting any chosen traversal order; they reduce to the decision procedure for the theory of lists (words) with concatenation [57].

Minimal Element. Some useful abstractions map trees into a quantity rather than into a collection. findMin in Figure 5 for instance is naturally expressed as a fold, and can be used to prove properties of data structures which maintain invariants about the position of certain particular elements (e.g. priority queues).

$\mathcal{C}$	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	[36]
Multiset	$\emptyset$	$c_1 \uplus \{e\} \uplus c_2$	$\cap, \cup, \setminus, \uplus$, setof, cardinality	NP	[55, 56]
$\mathbb{N}$	0	$c_1 + 1 + c_2$ (size)	$+, \leq$	NP	[54]
$\mathbb{N}$	0	$1 + \max(c_1, c_2)$ (height)	$+, \leq$	NP	[54]
List	List()	$a)$ c_1++ List(e)++ c_2 (in-order)	++(concat), List(_)(singleton)	PSPACE	[57]
	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	[53]
Option	None	$a)$ Some(e)	Some, None	NP	[49]
	None	$b)$ (computing minimum) see Figure 5	Some, None, $+, \leq$, if	NP	[48, 49, 54]
(Option, Option, Boolean)	(None, None, true)	$c)$ (checking sortedness) see Figure 6			

Figure 3. Example Instances of our Decision Procedure for Different Catamorphisms

```
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 5. Using the minimal element as an abstraction

```
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 6. A fold that checks that a tree is sorted

Sortedness of Binary Search Trees. Fold functions 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 6 shows the abstraction function sorted which, when applied to a binary tree, returns a triple containing a lower and upper bound on the set of elements, and a boolean indicating whether the tree is sorted. Although alternative specifications of sortedness are possible, this one directly conforms to the form of a fold function; at the same time it is efficiently executable.

The code in Figure 6 allows for trees with repeated elements. By replacing the occurrences of $\leq$ by the stricter $<$ we obtain the definition of sorted trees with distinct elements, which can also be handled by our decision procedure (the strict inequality turns out to be a more complicated instance of the decision procedure, see Section 5.3).

This example also illustrates fold functions that return n-tuples, which is a useful strategy to represent multiple mutually recursive functions. We will therefore assume that we work with a single fold function in our decision procedure.

4. The Decision Procedure

To simplify the presentation, we describe our decision procedure for the specific algebraic data type of binary trees, corresponding to the case classes in Figure 1. The 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.

As far as soundness is concerned, we can leave the collection type $\mathcal{C}$ and the language $\mathcal{L}_\mathcal{C}$ of decidable constraints on $\mathcal{C}$ largely unconstrained. As explained in Section 5, the conditions for completeness are relatively easy to satisfy when the image are sets; they become somewhat more involved for e.g. 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, $\mathsf{distinct}(x, y; z)$ means $x \neq z \wedge y \neq z$, whereas $\mathsf{distinct}(x_1; \ldots; x_n)$ means that all x_i are different.

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, detecting possible unsatisfiability within the term algebra theory
4. if unification did not fail, project the constraints on tree terms obtained from unification to the formula ϕ_C in the collection theory, yielding a new formula ϕ'_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

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

$\mathcal{T}_\mathcal{C}$ 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

4.1 Overview of the Decision Procedure

Figure 7 gives a high-level summary of the decision procedure. It solves the constraints over trees using unification, then derives all relevant consequences on the type $\mathcal{C}$ of collections that abstracts the trees. In this way it reduces a formula over trees and their abstract $\mathcal{L}_\mathcal{C}$-values to a $\mathcal{L}_\mathcal{C}$ formula, for which a decision procedure is assumed to be available. We next define our decision problem more precisely, then present the core steps of our decision procedure and show its soundness. Section 5 provides remaining subtle steps of the decision procedure and proves its completeness for a certain class of abstraction functions.

4.2 Syntax and Semantics of our Logic

Figure 8 shows the syntax of our logic. Figure 9 describes its semantics. The description refers to the catamorphism α, as well as the semantics $[\![\ \ ]\!]_\mathcal{C}$ of the parameter theory $\mathcal{L}_\mathcal{C}$.

4.3 Key Steps of the Decision Procedure

We describe a decision procedure for conjunctions of literals in our parametric theory. To lift the decision procedure to formulas of arbitrary boolean structure it suffices to apply the DPLL(T) approach [21].

Purification. In the first step of our decision procedure, we separate the conjuncts of our formula into literals over tree terms on one side, literals of $\mathcal{L}_\mathcal{C}$ on the other side, and finally the literals containing the catamorphism to connect the two sides. By the syntax of formulas, a literal in the formula can only combine tree terms with

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

Figure 9. Semantics of the parametric logic

terms of $\mathcal{L}_\mathcal{C}$ when the tree terms occur as arguments of the abstraction function α. It therefore suffices to replace all such applications by fresh variables of $\mathcal{L}_\mathcal{C}$ and add the appropriate binding equalities to the formula:

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

In the rewrite rule above, T denotes any tree term, c_F and t_F are fresh in the new formula.

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 $\{=,\neq\}$):

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

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:

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

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

Trivial:

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

Symbol Clash:

$$\frac{\mathsf{Leaf} \stackrel{?}{=} \mathsf{Node}(\ldots) \cup P'; S}{\bot} \quad \frac{\mathsf{Node}(\ldots) \stackrel{?}{=} \mathsf{Leaf} \cup P'; S}{\bot}$$

Orient:

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

Occurs Check:

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

Term Variable Elimination:

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

Element Variable Elimination:

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

Decomposition:

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

Figure 10. Unification Rules

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 corresponding to 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 $T_i = T_j$ if $T_i \sim T_j$, and $T_i \neq T_j$ otherwise. 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, adding 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 S of solution equations. Equations range over tree variables and element variables. The special system $\bot$ represents failure. The set of equations S has the property that it is of the form $\{t_1 = T_1, \ldots, t_n = T_n, e_1 = e_i, \ldots, e_m = e_j\}$, 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 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 n variables t_i for which $\sigma_S(t_i) = t_i$; we call such variables *parameter variables*. Let $\vec{u}$ denote the vector of the remaining m 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:

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

where

1. $\vec{T}$ are vectors of expressions in the language of algebraic data types, 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 $T_j(\vec{t})$ is not identical to t_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 $\vec{u} = u_1, \ldots, u_m$ and $\vec{t} = t_1, \ldots, t_n$, then the constraint $N(\vec{u}, \vec{t})$ can be denoted by $\mathsf{distinct}(u_1, \ldots, u_m; t_1; \ldots; t_n; \mathsf{Leaf})$;
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 $M(\vec{u}, \vec{t}, \vec{c})$:

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

After this transformation, α applies only to parameter variables. We introduce a variable c_i of $\mathcal{L}_\mathcal{C}$ 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, with-

out 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 $\mathcal{L}_\mathcal{C}$.

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, $\mathsf{combine}$ 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 Section 5 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 $\mathcal{L}_\mathcal{C}$.

4.4 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.

4.5 Complexity of the Reduction

Our decision procedure reduces formulas to normal form in nondeterministic 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 algebraic data types [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.

5. Completeness

We next describe the the strategy for computing the formula D_M from Section 4 for a broad class of catamorphisms. We prove that a computation following our strategy results in a sound and *complete* overall decision procedure.

5.1 Canonical Set Abstraction

We first 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 with the formula D in (4).

Definition 1. *If $c_1, \ldots, c_n$ are the free set 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 simple choice of D_M gives a complete decision procedure, we prove the following.

Lemma 2. *Let D_0 be a conjunction of n disequalities of terms built from tree variables $t_1, \ldots, t_m$ and symbols* Node, Leaf. *Suppose that D_0 does not contain a trivial disequality $T \neq T$ for any term T. If $A_1, \ldots, A_m$ are sets of trees such that $|A_j| > n$ for all $1 \leq j \leq m$, then D_0 has a satisfying assignment such that for each j, the value t_j belongs to A_j.*

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

We start by rewriting each disequality $T_i \neq T_i{}'$ in the form:

$$\neg \left(\begin{array}{ll} & \ldots \\ \wedge & t_a = C_a(t_{a1}, \ldots, t_{ak_a}) \\ \wedge & \ldots \end{array} \right)$$

where the conjunction of equalities is obtained by unifying the terms T_i and $T_i{}'$. The conjunction is non-empty because the statement of the lemma assumes that T_i and $T_i{}'$ are not syntactically identical. Here, the expressions of the form $C(t_{a1}, \ldots, t_{ak_a})$ denote terms built using Node, Leaf and the variables $t_{a1}, \ldots, t_{ak_a}$, where each of the variables appears 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

$$\bigvee_s L_{s1} \wedge \ldots \wedge L_{sn}$$

Note that in each conjunction, there is exactly one conjunct of the form $t_a \neq C_a(t_{a1}, \ldots, t_{ak_a})$ for each of the n disequalities $T_i \neq T_i{}'$ of the original problem. Notice as well that each variable t_a can be on the left-hand side of several disequalities in the same conjunction. From the form of the equations obtained using unification, we know that the set of variables $\{t_{a1}, \ldots, t_{ak_a}\}$ never contains t_a. This formula is logically equivalent to the original one from the statement of the lemma. To show that it is satisfiable, we pick an arbitrary disjunct and show that it is satisfiable.

We construct a satisfying assignment for such a conjunction as follows. For the first step, we start by collecting the set P_1 of 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 because there are n disequalities in the conjunction and $|A_1| > n$. We substitute in the entire formula T_1 instead of t_1. Because t_1 cannot appear on both left and right-hand side of an equation, in the resulting formula, all ground disequalities result from the grounding of the disequalities in P_1 and reduce to true. We eliminate the disequalities of P_1 from the set of disequalities.

For all indices $j \in \{2, \ldots, m\}$ do the following. Collect the set P_j of all disequalities of the form $t_j \neq T$ and $T \neq C(t_j)$ (in the second form, $C(t_j)$ denotes a term built with Leaf, Node, at least one occurrence of t_j, and no other variable). There are clearly no more than n such disequalities in P_j. For each of these disequality literals L_k, there is *at most* one value v_k^j for t_j which contradicts it: it is either the ground term T or its subterm. Because $|A_j| > n$, there exists a term $T_j \in A_j \setminus \{v_k^j\}_k$. Substitute T_j instead of t_j in the entire conjunction. This ensures that all disequalities in P_j hold. Remove the disequalities in P_j from the conjunction. We then proceed with t_{j+1}. The procedure terminates in m steps with an assignment mapping t_j to T_j for $1 \leq j \leq m$. Moreover, at this point there are no ground equations left and no variables left, so all conjuncts have been eliminated and satisfied. ■

Remark. Lemma 2 above is a strengthening of the Independence of Disequations Lemma [15, Page 178], [42]. Namely, the statement in [15, Page 178] requires the sets A_j to be infinite, whereas we showed above (using a new, more complex proof) that it suffices for A_j to have more elements than there are disequalities. While the original weaker version suffices for Lemma 3, we need our stronger statement in Section 5.3.

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 $t_i \neq$ Leaf holds because this conjunct is in D. Therefore, $\alpha(t_i) \neq \emptyset$ by the above observations about α. 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 above observations about α. By Lemma 2 there are values $t_i \in A_i$ for $1 \leq i \leq n$ such that the disequalities 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 the 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 [35] 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 [36]. 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 algebraic data types with the canonical set abstraction belongs to NP.

5.2 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 if and only if $\alpha^{-1}(\alpha(t))$ is finite for $t \in S$ and infinite for $t \notin S$.*

The canonical set abstraction is an infinitely surjective $\{$Leaf$\}$-abstraction. Other infinitely surjective $\{$Leaf$\}$-abstractions are the tree size abstraction, which for a given tree computes its size as the number of internal nodes, the tree height abstraction, and the sortedness abstraction of Figure 6.

An example of infinitely surjective $\emptyset$-abstractions is the function FV(t) that computes the set of free variables in an abstract syntax tree t representing a lambda expression or a formula. Indeed, for each finite set s of variables (including $s = \emptyset$), there exist infinitely many terms t such that FV$(t) = s$.

We can compute D_M for an infinitely surjective S-abstraction whenever S is finite. The general idea 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, the terms containing variables are distinct from all T_i. The formula

D_M then states the condition $\bigwedge_{t\in S} c_i \neq \alpha(t)$. We omit the details because they are subsumed by the more general construction below, but we note that the above algorithm for $\{\mathsf{Leaf}\}$-abstractions also works for $\emptyset$-abstractions.

5.3 Sufficiently Surjective Abstractions

We next present a more general completeness result, which requires collections to be classified either as being an image of sufficiently many terms, or as having one of finitely many shapes.

Definition 5 (Tree Shape and Size)**.** *Let* SLeaf *be a new constant symbol and* $\mathsf{SNode}(t_1, t_2)$ *a new constructor symbol. The* shape *of a tree* t*, denoted* $\check{s}(t)$*, is a ground term built from* SLeaf *and* $\mathsf{SNode}(\_, \_)$ *as follows:*

$$\check{s}(\mathsf{Leaf}) = \mathsf{SLeaf}$$
$$\check{s}(\mathsf{Node}(T_1, e, T_2)) = \mathsf{SNode}(\check{s}(T_1), \check{s}(T_2))$$

We define the size *of a shape as:*

$$\mathrm{size}(\mathsf{SLeaf}) = 0$$
$$\mathrm{size}(\mathsf{SNode}(s_1, s_2)) = 1 + \mathrm{size}(s_1) + \mathrm{size}(s_2)$$

By extension, we define the size of a tree t *to be the size of its shape.*

Definition 6 (Shape Instantiation)**.** *The* instantiation *of the shape of a tree* t *produces a copy of* t *where the values stored in the nodes are replaced by fresh variables:*

$$\mathrm{inst}(t, i) = \mathrm{inst}'(t, i, 1)$$
$$\mathrm{inst}'(\mathsf{SNode}(s_1, s_2), i, j) = \mathsf{Node}(\mathrm{inst}'(s_1, i, 1 + j), v^i_j, \mathrm{inst}'(s_2, i, 1 + j + \mathrm{size}(s_1)))$$
$$\mathrm{inst}'(\mathsf{SLeaf}, i, j) = \mathsf{Leaf}$$

In the instantiation function, i determines the names of the fresh variables: the variables introduced by the instantiation $\mathrm{inst}(s, i)$ range from v^i_1 to $v^i_{\mathrm{size}(s)}$. Consequently, if $i \neq j$, then the terms $\mathrm{inst}(t, i)$ and $\mathrm{inst}(t, j)$ have no common variables. Note that for an abstraction function α and a tree shape s, the term $\alpha(\mathrm{inst}(s, i))$ contains no tree variables, so it can be rewritten (by completely evaluating α) into a term in the collection theory with the free variables $v^i_1, \ldots, v^i_{\mathrm{size}(s)}$. Note finally that for every tree term T, the formula $\mathrm{inst}(\check{s}(T), i) = T$ is satisfiable.

Definition 7 (Sufficient Surjectivity)**.** *We call an abstraction function* ***sufficiently surjective*** *if and only if, for each natural number* $p > 0$ *there exist, computable as a function of* p*,*

- *a finite set of shapes* S_p
- *a closed formula* M_p *in the collection theory such that* $M_p(c)$ *implies* $|\alpha^{-1}(c)| > p$

such that, for every term t*,* $M_p(\alpha(t))$ *or* $\check{s}(t) \in S_p$*.*

Note that M_p can introduce fresh variables as long as it is existentially closed and the decision procedure for the collection theory can handle positive occurrences of existential quantifiers.

The definition above implies:

$$\lim_{p\to\infty} \inf_{\check{s}(t)\notin S_p} |\alpha^{-1}(\alpha(t))| = \infty$$

We now show how we can build a formula D_M equisatisfiable with the formula D of (4) provided the aforementioned assumptions hold. We keep the notational convention that the parameter variables $\vec{t}$ range from t_1 to t_n and that the terms $\vec{T}(\vec{t})$ built around them range from $T_1(\vec{t})$ to $T_m(\vec{t})$. We also assume that for all variables t_i, the conjunct $c_i = \alpha(t_i)$ is present in D. This is consistent with the normal form we presented earlier, up to renaming of the variables.

In the following we take $p = \binom{n}{2} + n \cdot m$, where n is the dimension of the vector of term variables in $\vec{t}$ in D, and m is the dimension of the vector $\vec{T}(\vec{t})$ for derived terms in D. Consider the formula:

$$P \equiv \bigwedge_{i=1}^{n} \left(M_p(c_i) \vee \bigvee_{s\in S_p} \mathrm{inst}(s, i) = t_i \right) \wedge \bigwedge_{j=1}^{m} \left(M_p(\alpha(T_j(\vec{t}))) \vee \bigvee_{s\in S_p} \mathrm{inst}(s, j + n) = T_j(\vec{t}) \right)$$

Note that $\alpha(T_j(\vec{t}))$ can be rewritten as a term in the collection theory using the variables $\vec{c}$. Note that existentially quantifying P over the variables introduced by inst gives a formula that is always true, by the assumptions on M_p and S_p. Let P' be the disjunctive normal form of P. For every disjunct P^d of P', observe that for each t_i, either $M_p(\alpha(t_i))$ is a conjunct of P^d, or $\mathrm{inst}(s, i) = t_i$ is a conjunct for exactly one s in S_p. The same observation holds for the terms $T_j(\vec{t})$.

We proceed as follows for each disjunct P^d of P'. We run unification over the equalities between terms. This can either result in a clash (because the shape assigned to a term $T_j(\vec{t})$ is in contradiction with the shapes assigned to the variables of $\vec{t}$), or produce new equalities between the freshly introduced element variables v. If there was a clash, we simply replace P^d by false and eliminate it from the formula. Otherwise, we add to P^d the new equalities produced by unification, yielding a disjunct P^d_U.

We next add additional conjuncts to P^d_U to obtain a formula D^d equisatisfiable with $D \wedge P^d_U$, as follows. Recall that D contains conjuncts of the forms:

- $t_i \neq t_j$ as part of $N(\vec{T}(\vec{t}), \vec{t})$,
- $t_i \neq T_j(\vec{t})$ as part of $N(\vec{T}(\vec{t}), \vec{t})$, and
- $c_i = \alpha(t_i)$ as part of $M^1(\vec{t}, \vec{c})$.

Initially, we set D^d to be the formula P^d_U. Then, for each disequality $T \neq T'$ in D (where T and T' can represent either variables or constructed terms), if in P^d_U we have $\mathrm{inst}(s, i) = T$ and $\mathrm{inst}(s, j) = T'$ for the same shape s, we add as a conjunct to D^d the disjunction $\bigvee_{1\leq k\leq \mathrm{size}(s)} v^i_k \neq v^j_k$. Finally, we replace in D^d all the equalities of the form $\mathrm{inst}(s, i) = T$ by $\alpha(\mathrm{inst}(s, i)) = \alpha(T)$. As we already observed, $\alpha(\mathrm{inst}(s, i))$ can always be rewritten to a term in the collection theory by evaluating α. In the case where T is a variable t_i, $\alpha(T)$ is simply c_i. If it is a term $T_j(\vec{t})$, $\alpha(T)$ can be rewritten in terms of $\vec{c}$ by partially evaluating α.

The resulting formula D_M is $\bigvee_d D^d$. We claim that D_M is equisatisfiable with D.

Proof. *(Preliminary transformations)* Conjoining P to D does not change the satisfiability of the formula, and neither does the transformation to disjunctive normal form, so D is equisatisfiable with $D \wedge \bigvee_p P^d$. The unification procedure is equivalence preserving, so the formula after unification is still equisatisfiable. It therefore suffices to show that $D \wedge P^d_U$ is equisatisfiable with D^d.

(From trees to collections) First, observe that D^d is a consequence of $D \wedge P^d_U$. Indeed, $\bigvee v^i_k \neq v^j_k$ follows from $T \neq T'$, $\mathrm{inst}(s, i) = T$, and $\mathrm{inst}(s, j) = T'$. Also, $\alpha(\mathrm{inst}(s, i)) = \alpha(T)$ follows from $\mathrm{inst}(s, i) = T$, and partial evaluation of α is equivalence preserving. Therefore, if $D \wedge P^d_U$ has a model, then D^d as a consequence holds in this model. Thus D^d has a model. It remains to show the converse.

(From collections to trees) Assume $\mathcal{M}$ is a model for D^d, which specifies the values for element and collection variables. We construct an extension of $\mathcal{M}$ with values for tree variables $\vec{t}$ such that $D \wedge P^d_U$ holds.

For those terms T for which P_U^d contains a conjunct $\mathrm{inst}(s,i) = T$, we assign T to be the value of $\mathrm{inst}(s,i)$ in $\mathcal{M}$ (indeed, $\mathcal{M}$ specifies the values of all free variables v_j^i in $\mathrm{inst}(s,i)$). In this assignment, the literals in P_U^d of the form $\alpha(T) = c_i$ are true for such terms T. Furthermore, all disequalities between such terms hold. Indeed, terms of different shape are distinct, and for terms of equal shape the formula D^d contains a disjunct $\bigvee v_k^i \neq v_k^j$ ensuring that the terms differ in at least one element.

It remains to define values for the trees T for which P_U^d does not contain a conjunct of the form $\mathrm{inst}(s,i) = T$ in such a way that the literals containing these trees are true. These are disequality literals, as well as literals of the form $\alpha(T) = c_i$, when T is a variable t_i. For each such tree T, the formula P_U^d contains the conjunct $M_p(\alpha(T))$, by construction of the disjunctive normal form. From the assumptions on M_p, from $M_p(\alpha(T))$ we conclude $|\alpha^{-1}(\alpha(T))| > p$. Therefore, there are at least $p+1$ trees T_k such that $\alpha(T_k) = \alpha(T)$. The number of disequalities in D is at most $\binom{n}{2} + n \cdot m$. Because $p = \binom{n}{2} + n \cdot m$, we can apply Lemma 2 to choose values for (at most) n trees satisfying (at most) p disequations from sets of size at least $p+1$. This choice of trees completes the assignment for the remaining tree variables such that all conjuncts of D^d hold. ■

Model Construction. Lemma 2 is constructive, so the proof above also gives model construction whenever 1) the underlying decision procedure for the collection provides model construction, and 2) there is an algorithm to compute, for each c where $M_p(c)$, a finite set of containing p elements t such that $\alpha(t) = c$.

Worst-Case Complexity of the Decision Problem. The reduction from the starting formula to the theory of collections is a nondeterministic polynomial-time algorithm that invokes the computation of the set S_p and the formula M_p. When S_p and M_p can be computed in polynomial time, then each of the disjuncts considered is of polynomial size. Our decision procedure is this case an NP reduction. This case applies to the three examples below. When the satisfiability for the collection theory is in NP (e.g. for multisets and sortedness), the overall satisfiability problem is also in NP.

5.4 Application to Multisets, Lists, and Sortedness

We now show that the list and multiset abstraction are sufficiently surjective abstractions, as is the sortedness abstraction for trees with distinct elements. (The set of these examples is not meant to be exhaustive.) In the following, let C_n denote the number of binary trees with n elements, and let K_m denote a its inverse, that is, the smallest natural number n such that $C_n > m$. The functions C_n and K_m are monotonic and computable.

Lists. Consider the catamorphism for infix traversal of the tree, for which we have $\mathsf{empty} = \mathsf{List}()$ and $\mathsf{combine}(c_1, e, c_2) = c_1 \mathbin{+\!\!+} \mathsf{List}(e) \mathbin{+\!\!+} c_2$. (Catamorphisms for pre-order and post-order traversal can be handled analogously.) We can use the following definitions for S_p and M_p:

- $S_p = \{s \mid \mathrm{size}(s) < K_p\}$
- $M_p(c) \equiv \exists e_1, \ldots, e_{K_p} \,.\, \exists c' \,.\, c = \mathsf{List}(e_1, \ldots, e_{K_p}) \mathbin{+\!\!+} c'$

S_p is the set of shapes with less than K_p nodes, while $M_p(c)$ expresses that the list c has at least K_p elements, so clearly for any tree t, either its shape $\mathsf{s}(t)$ in is S_p, or it has more than K_p nodes and therefore $M_p(\alpha(t))$ holds. Finally, observe that for a list c of n elements, α maps exactly C_n distinct trees to c. Therefore, for any c such that $M_p(c)$ holds, we have $|\alpha^{-1}(c)| = C_{K_p}$, and $C_{K_p} > p$ by construction. Therefore, the infix traversal abstraction is sufficiently surjective and our completeness argument applies.

Multisets. Consider the multiplicity-preserving multiset abstraction, which is given by $\mathsf{empty} = \emptyset$ and $\mathsf{combine}(c_1, e, c_2) = c_1 \uplus \{e\} \uplus c_2$. We then take

- $S_p = \{s \mid \mathrm{size}(s) < K_p\}$
- $M_p(c) \equiv \exists e_1, \ldots, e_{K_p} \,.\, \exists c' \,.\, c = \{e_1, \ldots, e_{K_p}\} \uplus c'$

For a multiset c with n elements (counting repetitions), there are *at least* C_n trees mapped by α to c, so the same argumentation as for lists applies.

Sortedness. Finally, consider the abstraction function described in Section 3.1 that checks the sortedness of trees. We mentioned in Section 5.2 that the version which allowed repeated elements is infinitely surjective. In contrast, in the case where the elements of the trees have to be distinct, it is not infinitely surjective. The reason is that the catamorphism also computes the minimal and maximal elements of the tree, and there are only finitely many sorted trees with distinct elements between a given minimum and maximum. The catamorphism is nevertheless sufficiently surjective. Indeed, we can take

- $S_p = \{s \mid \mathrm{size}(s) < K_p\}$
- $M_p((a, b, \varsigma)) \equiv 1 + b - a \geq K_p$

where (a, b, ς) is the triple of the minimum, the maximum, and the sortedness of the tree, as computed by the catamorphism. Here, M_p essentially says that the range of values in the tree is sufficiently wide, so that enough distinct trees mapping to (a, b, ς) can be constructed. In conclusion, the catamorphisms that map trees into lists, multisets, or sortedness property are also instances for which our decision procedure is complete.

6. Related Work

One reason why we find our result useful is that it can leverage a number of existing decidability results. In [5] the authors present an abstract approach that can be used to obtain efficient strategies for reasoning about algebraic data types (without abstraction functions). For reasoning about sets and multisets one expressive approach is the use of the decidable array fragment [11]. Optimal complexity bounds for reasoning about sets and multisets in the presence of cardinality constraints have been established in [36, 56]. Building on these results, extensions to certain operations on vectors has been presented in [40]. Reasoning about lists with concatenation can be done using Makanin's algorithm [41] and its improvements [57]. A different class of constraints uses rich string operations but imposes bounds on string length [8]. Researchers have identified a number of laws in the area of manual program derivation, including laws that relate trees, lists, bags, and sets [26]. The present paper can be viewed as a step towards automating some of these laws. Another example in this direction is the combinatory array logic [17], which supports map operations on arrays but does not support the cardinality operator.

Our parameterized decision procedure is an example of an approach to combine logics (e.g. the logic of algebraic data types and a logic of collections). Standard results in this field are Nelson-Oppen combination [48]. 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 an algebraic data type; 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 [62, 30, 20] as well as disjointness condition subject to a local finiteness condition [22]. An approach that allows theories to share set algebra with cardinalities is presented in [65]. None

of these results by itself handles the problem of reasoning about a catamorphism from the theory of algebraic data types. That said, our canonical set abstraction of algebraic data types is a new BAPA-reducible theory that fits into the framework [65].

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, which are not locally finite.

Related to our partial evaluation of the catamorphism is the phenomenon of local theory extensions [27], 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 the extented version of the local theory framework is needed. Concurrently with our result, the machinery of local theory extensions has been extended to certain homomorphisms in term algebras [61], although without considering homomorphisms that compute sets, multisets, or lists. We plan to investigate unifying the results in [61] with our notion of sufficiently surjective abstraction.

The proof decidability for term powers [34] 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 [69] researchers describe a decision procedure for algebraic data types with size constraints and in [43] 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 [69] covers also the case of a finite number of atoms, whereas we have chosen to focus on the case of infinite set of elements $\mathcal{E}$. Term algebras have an extensively developed theory, and enjoy many desirable properties, including quantifier elimination [42]; quantifier elimination also carries over to many extensions of term algebras [15, 34, 60, 69, 68]. Note that in examples such as multisets with cardinality, we cannot expect quantifier elimination to hold because the quantified theory is undecidable [55, Section 6].

Some aspects of our decision procedure are similar to folding and unfolding performed when using types to reason about data structures [29, 63, 50, 59, 66]. 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 [38, 37, 45, 64, 47]. 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 reason about imperative programs [3, 7, 31, 67] typically use declarative constructs and data types within specification annotations. Our decision procedure extends the expressive power of imperative specifications that can be handled in a predictable way.

The SMT-LIB standard [4] for SMT provers currently does not support algebraic data types, even though several provers support it in their native input languages [6, 16]. 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.

7. 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*. Elsevier, 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*, 2007.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 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] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

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

[12] R. Cartwright and M. Fagan. Soft typing. In *PLDI*, 1991.

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

[14] 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 *TPHOLs*, 2009.

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

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

[17] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, 2009.

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

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

[20] P. Fontaine. Combinations of theories and the Bernays-Schönfinkel-Ramsey class. In *VERIFY*, 2007.

[21] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV*, 2004.

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

[23] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. On test generation through programming in UDITA. Technical Report LARA-REPORT-2009-05, EPFL, Sep. 2009.

[24] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1971.

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

[26] P. F. Hoogendijk and R. C. Backhouse. Relational programming laws in the tree, list, bag, set hierarchy. *Sci. Comput. Program.*, 22(1-2), 1994.

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

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

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

[30] S. Krstić, A. Goel, J. Grundy, and C. Tinelli. Combined satisfiability modulo parametric theories. In *TACAS*, 2007.

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

[32] 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.

[33] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding boolean algebra with presburger arithmetic. *J. Automated Reasoning*, 2006.

[34] V. Kuncak and M. Rinard. Structural subtyping of non-recursive types is decidable. In *LICS*, 2003.

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

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

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

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

[39] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *VMCAI*, 2005.

[40] P. Maier. Deciding extensions of the theories of vectors and bags. In *VMCAI*, 2009.

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

[42] A. I. Mal'cev. Chapter 23: Axiomatizable classes of locally free algebras of various types. In *The Metamathematics of Algebraic Systems*, volume 66. North Holland, 1971.

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

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

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

[46] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, volume 523 of *LNCS*, 1991.

[47] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. PLDI*, 2001.

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

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

[50] 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.

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

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

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

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

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

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

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

[58] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1), 1965.

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

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

[61] V. Sofronie-Stokkermans. Locality results for certain extensions of theories with bridging functions. In *CADE*, 2009.

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

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

[64] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *VMCAI*, 2006.

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

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

[67] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.

[68] T. Zhang, H. B. Sipma, and Z. Manna. The decidability of the first-order theory of Knuth-Bendix order. In *CADE*, 2005.

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

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