Abstractions in Decision Procedures for Algebraic Data Types

Tuan-Hung Pham and Michael W. Whalen

University of Minnesota

Abstract. Reasoning about algebraic data types and functions that operate over these data types is an important problem for a large variety of applications. In this paper, we present a decision procedure for reasoning about data types using abstractions that are provided by *catamorphisms*: fold functions that map instances of algebraic data types into values in a decidable domain. We show that the procedure is sound and complete for a class of *monotonic* catamorphisms.

Our work extends a previous decision procedure that solves formulas involving algebraic data types via successive unrollings of catamorphism functions. First, we propose the categories of *monotonic* catamorphisms and *associative-commutative* catamorphisms, which we argue provide a better formal foundation than previous categorizations of catamorphisms. We use monotonic catamorphisms to fix an incompleteness in the previous unrolling algorithm (and associated proof). We then use these notions to address two open problems from previous work: (1) we provide a bound on the number of unrollings necessary for completeness, showing that it is exponentially small with respect to formula size for associative-commutative catamorphisms, and (2) we demonstrate that associative-commutative catamorphisms can be combined within a formula whilst preserving completeness.

1 Introduction

Decision procedures have been a fertile area of research in recent years, with several advances in the breadth of theories that can be decided and the speed with which substantial problems can be solved. When coupled with SMT solvers, these procedures can be combined and used to solve complex formulas relevant to software and hardware verification. An important stream of research has focused on decision procedures for algebraic data types. Algebraic data types are important for a wide variety of problems: they provide a natural representation for tree-like structures such as abstract syntax trees and XML documents; in addition, they are the fundamental representation of recursive data for functional programming languages.

Algebraic data types provide a significant challenge for decision procedures since they are recursive and usually unbounded in size. Early approaches focused on equalities and disequalities over the structure of elements of data types [2,16]. While important, these structural properties are often not expressive enough

to describe interesting properties involving the data stored in the data type. Instead, we often are interested in making statements both about the structure and contents of data within a data type. For example, one might want to express that all integers stored within a tree are positive or that the set of elements in a list does not contain a particular value.

In [22], Suter et al. described a parametric decision procedure for reasoning about algebraic data types using catamorphism (fold) functions. In the procedure, catamorphisms describe abstract views of the data type that can then be reasoned about in formulas. For example, suppose that we have a binary tree data type with functions to add and remove elements from the tree, as well as check whether an element was stored in the tree. Given a catamorphism *setOf* that computes the set of elements stored in the tree, we could describe a specification for an 'add' function as:

$$\mathit{setOf}\big(\mathrm{add}(e,t)\big) = \{e\} \cup \mathit{setOf}(t)$$

where *setOf* can be defined in an ML-like language as:

```
fun setOf t = case t of Leaf ⇒ ∅ |
                        Node(l, e, r) ⇒ setOf(l) ∪ {e} ∪ setOf(r)
```

Formulas of this sort can be decided by the algorithm in [22]. In fact, the decision procedure in [22] allows a wide range of problems to be addressed, because it is parametric in several dimensions: (1) the structure of the data type, (2) the elements stored in the data type, (3) the collection type that is the codomain of the catamorphism, and (4) the behavior of the catamorphism itself. Thus, it is possible to solve a variety of interesting problems, including:

- reasoning about the contents of XML messages,
- determining correctness of functional implementations of data types, including queues, maps, binary trees, and red-black trees.
- reasoning about structure-manipulating functions for data types, such as sort and reverse.
- computing bound variables in abstract syntax trees to support reasoning over operational semantics and type systems, and
- reasoning about simplifications and transformations of propositional logic.

The first class of problems is especially important for *guards*, devices that mediate information sharing between security domains according to a specified policy. Typical guard operations include reading field values in a packet, changing fields in a packet, transforming a packet by adding new fields, dropping fields from a packet, constructing audit messages, and removing a packet from a stream. We have built automated reasoning tools (described in [9]) based on the decision procedure to support reasoning over guard applications.

The procedure was proved sound for all catamorphisms and complete for a class of catamorphisms called *sufficiently surjective* catamorphisms, which we will describe in more detail in the remainder of the paper. Unfortunately, the algorithm in [22] was quite expensive to compute and required a specialized

predicate called M_p to be defined separately for each catamorphism and proved correct w.r.t. the catamorphism using either a hand-proof or a theorem prover.

In [23], a generalized algorithm for the decision procedure was proposed, based on unrolling the catamorphism. This algorithm had three significant advantages over the algorithm in [22]: it was much less expensive to compute, it did not require the definition of M_p, and it was claimed to be complete for all sufficiently surjective catamorphisms. Unfortunately, the algorithm in [23] is in fact not complete for all sufficiently surjective catamorphisms.

In this paper, we slightly modify the procedure of [23] to remove this incompleteness. We then address two open problems with the previous work [23]: (1) how many catamorphism unrollings are required in order to prove properties using the decision procedure? and (2) when is it possible to combine catamorphisms within a formula in a complete way? To address these issues, we introduce two further notions: *monotonic* catamorphisms, which describe an alternative formulation to the notion of *sufficiently surjective* catamorphisms for describing completeness, and *associative-commutative* (AC) catamorphisms, which can be combined within a formula while preserving completeness results. In addition, these catamorphisms have the property that they require a very small number of unrollings. This behavior explains some of the empirical success in applying catamorphism-based approaches on interesting examples from previous papers [23,9]. In short, the paper consists of the following contributions:

- We propose the notion of monotonic catamorphisms and show that all sufficiently surjective catamorphisms discussed in [22] are monotonic.
- We revise the unrolling-based decision procedure for algebraic data type [23] using monotonic catamorphisms and formally prove its completeness.
- We propose the notion of AC catamorphisms, a sub-class of monotonic catamorphisms, and show that decision procedure for algebraic data types with AC catamorphisms are *combinable* while the procedures for algebraic data types proposed by Suter et al. [22,23] only work with single catamorphisms.
- We solve the open problem of determining the maximum number of unrollings with both monotonic and AC catamorphisms.
- We show that AC catamorphisms can be automatically detected.
- We describe an implementation of the approach, called RADA [17], which accepts formulas in an extended version of the SMT-LIB2 syntax, and demonstrate it on a range of examples.

The rest of the paper is organized as follows. Section 2 presents some preliminaries about catamorphisms and the parametric logic in [22]. Section 3 discusses some properties of trees and shapes in the parametric logic. Section 4 points out some completeness issues in Suter et al.'s work [22,23]. In Section 5, we propose an unrolling-based decision procedure for algebraic data types. The decision procedure works with monotonic catamorphisms, which are discussed in Section 6, and the correctness of the algorithm for these catamorphisms is shown in Section 7. Section 8 presents AC catamorphisms, and the relationship between different types of catamorphisms is discussed in Section 9. Experimental results for our

approach are shown in Section 10. Section 11 presents related work. Finally, we conclude the paper with directions for future work in Section 12.

2 Preliminaries

We describe the parametric logic used in the decision procedures for algebraic data types proposed by Suter et al. [22,23], the definition of catamorphisms, and the idea of sufficient surjectivity, which describes situations in which the decision procedures [22,23] were claimed to be complete.

2.1 Parametric Logic

The input to the decision procedures is a formula ϕ of literals over elements of tree terms and abstractions produced by a catamorphism. The logic is *parametric* in the sense that we assume a data type τ to be reasoned about, an element theory $\mathcal{E}$ containing element types and operations, a catamorphism α that is used to abstract the data type, and a decidable theory $\mathcal{L}_\mathcal{C}$ of values in a collection domain $\mathcal{C}$ containing terms C generated by the catamorphism function. Fig. 1 shows the syntax of the parametric logic instantiated for binary trees.

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
E	::=	variables of type $\mathcal{E} \mid \mathsf{elem}(T) \mid \mathcal{T}_\mathcal{E}$	Expression
F_T	::=	$T = T \mid T \neq T$	Tree (in)equations
F_C	::=	$C = C \mid \mathcal{F}_\mathcal{C}$	Formula of $\mathcal{L}_\mathcal{C}$
F_E	::=	$E = E \mid \mathcal{F}_\mathcal{E}$	Formula of $\mathcal{L}_\mathcal{E}$
ϕ	::=	$\bigwedge F_T \wedge \bigwedge F_C \wedge \bigwedge F_E$	Conjunctions
ψ	::=	$\phi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi$	Formulas

Fig. 1. Syntax of the parametric logic. Its semantics can be found in [22].

The syntax of the logic ranges over data type terms T and $\mathcal{C}$-terms of a decidable collection theory $\mathcal{L}_\mathcal{C}$. $\mathcal{T}_\mathcal{C}$ and $\mathcal{F}_\mathcal{C}$ are arbitrary terms and formulas in $\mathcal{L}_\mathcal{C}$, as are $\mathcal{T}_\mathcal{E}$ and $\mathcal{F}_\mathcal{E}$ in $\mathcal{L}_\mathcal{E}$. Tree formulas F_T describe equalities and disequalities over tree terms. Collection formulas F_C and element formulas E_C describe equalities over collection terms C and element terms E, as well as other operations ($\mathcal{F}_\mathcal{C}$, $\mathcal{F}_\mathcal{E}$) allowed by the logic of collections $\mathcal{L}_\mathcal{C}$ and elements $\mathcal{L}_\mathcal{E}$. E defines terms in the element types $\mathcal{E}$ contained within the branches of the data types. ϕ defines conjunctions of (restricted) formulas in the tree and collection theories. The ϕ terms are the ones solved by the decision procedures in [22]; these can be generalized to arbitrary propositional formulas (ψ) through the use of a DPLL solver [8] that manages the other operators within the formula. Although the logic and unrolling procedure is parametric with respect to data types, in the sequel we focus on binary trees to illustrate the concepts and proofs.

2.2 Catamorphisms

Given a tree in the parametric logic, we can map the tree into a value in $\mathcal{C}$ using a *catamorphism*, which is a fold function of the following format:

$$\alpha(t) = \begin{cases} \mathsf{empty} & \text{if } t = \mathsf{Leaf} \\ \mathsf{combine}\big(\alpha(t_L), e, \alpha(t_R)\big) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \end{cases}$$

where empty is an element in $\mathcal{C}$ and $\mathsf{combine} : (\mathcal{C}, \mathcal{E}, \mathcal{C}) \to \mathcal{C}$ is a function that combines a triple of two values in $\mathcal{C}$ and an element in $\mathcal{E}$ into a value in $\mathcal{C}$.

Table 1. Sufficiently surjective catamorphisms in [22]

Name	$\alpha(\mathsf{Leaf})$	$\alpha(\mathsf{Node}(t_L, e, t_R))$	Example
Set	$\emptyset$	$\alpha(t_L) \cup \{e\} \cup \alpha(t_R)$	{1, 2}
Multiset	$\emptyset$	$\alpha(t_L) \uplus \{e\} \uplus \alpha(t_R)$	{1, 2}
SizeI	0	$\alpha(t_L) + 1 + \alpha(t_R)$	2
Height	0	$1 + \max\{\alpha(t_L), \alpha(t_R)\}$	2
List	List()	$\alpha(t_L)$ @ List(e) @ $\alpha(t_R)$ (in-order) List(e) @ $\alpha(t_L)$ @ $\alpha(t_R)$ (pre-order) $\alpha(t_L)$ @ $\alpha(t_R)$ @ List(e) (post-order)	(1 2) (2 1) (1 2)
Some	None	Some(e)	Some(2)
Min	None	$\min'\{\alpha(t_L), e, \alpha(t_R)\}$	1
Sortedness	(None, None, true)	(None, None, false) (if tree unsorted) (min element, max element, true) (if tree sorted)	(1, 2, true)

Catamorphisms from [22] are shown in Table 1. The first column contains catamorphism names[1]. The next two columns define $\alpha(t)$ when t is a Leaf and when it is a Node, respectively. The last column shows examples of the application of each catamorphism to $\mathsf{Node}\big(\mathsf{Node}(\mathsf{Leaf}, 1, \mathsf{Leaf}), 2, \mathsf{Leaf}\big)$. In the *Min* catamorphism, $\min'$ is the same as the usual min function except that $\min'$ ignores None in the list of its arguments, which must contain at least one non-None value. The *Sortedness* catamorphism returns a triple containing the min and max element of the subtree, and true/false depending on whether it is sorted or not.

Tree shapes: The shape of a tree in the parametric logic is obtained by removing all element values in the tree.

Definition 1 (Tree shapes). *The shape of a tree is defined by constant* SLeaf *and constructor* $\mathsf{SNode}(\_, \_)$ *as follows:*

$$shape(t) = \begin{cases} \mathsf{SLeaf} & \text{if } t = \mathsf{Leaf} \\ \mathsf{SNode}\big(shape(t_L), shape(t_R)\big) & \text{if } t = \mathsf{Node}(t_L, \_, t_R) \end{cases}$$

[1] *SizeI*, which maps a tree into its number of *internal* nodes, was originally named *Size* in [22]. We rename the catamorphism to easily distinguish between itself and function *size*, which returns the total number of *all* vertices in a tree, in this paper.

Sufficiently surjective catamorphisms: The decision procedures proposed by Suter et al. [22,23] were claimed to be complete if the catamorphism used in the procedures is *sufficiently surjective* [22]. Intuitively, a catamorphism is sufficiently surjective if the inverse relation of the catamorphism has sufficiently large cardinality when tree shapes are larger than some finite bound.

Definition 2 (Sufficient surjectivity). *α is sufficiently surjective iff for each $p \in \mathbb{N}^+$, there exists, computable as a function of p, (1) a finite set of shapes S_p and (2) a closed formula M_p in the collection theory such that $M_p(c)$ implies $|\alpha^{-1}(c)| > p$, such that $M_p\big(\alpha(t)\big)$ or $shape(t) \in S_p$ for every tree term t.*

Despite its name, sufficient surjectivity has no surjectivity requirement for the codomain of α. It only requires a "sufficiently large" number of trees for values satisfying the condition M_p. Table 1 describes all sufficiently surjective catamorphisms in [22]. The only catamorphism in [22] not in Table 1 is the *Mirror* catamorphism; since the cardinality of the inversion function of the catamorphism is always 1, the sufficiently surjective condition does not hold for this catamorphism.

3 Properties of Trees and Shapes in the Parametric Logic

We present some important properties of trees and shapes in the parametric logic which will play important roles in the subsequent sections of the paper.

3.1 Properties of Trees

Property 1 follows from the syntax of the parametric logic. Properties 2 and 3 are well-known properties of full binary trees [6,18] (i.e., binary trees in which every internal node has exactly two children).

Property 1 (Type of tree). Any tree in the parametric logic is a full binary tree.

Property 2 (Size). The number of vertices in any tree in the parametric logic is odd. Also, in a tree t of size $2k + 1$ ($k \in \mathbb{N}$), we have:

$$ni(t) = k \qquad\qquad nl(t) = k + 1$$

where $ni(t)$ and $nl(t)$ are the number of internal nodes and the number of leaves in t, respectively.

Property 3 (Size vs. Height). In the parametric logic, the size of a tree of height $h \in \mathbb{N}$ must be at least $2h + 1$.

3.2 Properties of Tree Shapes

In this section, we show a special relationship between tree shapes and the well-known Catalan numbers [21], which will be used to establish some properties of monotonic and AC catamorphisms in Sections 6 and 8.

Define the size of the shape of a tree to be the size of the tree. Let $\bar{\mathbb{N}}$ be the set of odd natural numbers. Because of Property 2, the size of a shape is in $\bar{\mathbb{N}}$. Let $ns(s)$ be the number of tree shapes of size $s \in \bar{\mathbb{N}}$ and let $\mathbb{C}_n$, where $n \in \mathbb{N}$, be the n-th Catalan number [21].

Lemma 1. *The number of shapes of size $s \in \bar{\mathbb{N}}$ is the $\frac{s-1}{2}$-th Catalan number:*

$$ns(s) = \mathbb{C}_{\frac{s-1}{2}}$$

Proof. Property 1 implies that tree shapes are also full binary trees. The lemma follows since the number of full binary trees of size $s \in \bar{\mathbb{N}}$ is $\mathbb{C}_{\frac{s-1}{2}}$ [21,13]. □

Using the expression $\mathbb{C}_n = \frac{1}{n+1}\binom{2n}{n}$ [21], we could easily compute the values that function $ns : \bar{\mathbb{N}} \to \mathbb{N}^+$ returns. This function satisfies the monotonic condition in Lemma 2.

Lemma 2. $1 = ns(1) = ns(3) < ns(5) < ns(7) < ns(9) < \ldots$

Proof. According to Koshy [13], Catalan numbers can be computed as follows:

$$\mathbb{C}_0 = 1 \qquad\qquad \mathbb{C}_{n+1} = \frac{2(2n+1)}{n+2}\mathbb{C}_n \text{ (where } n \in \mathbb{N})$$

Using this expression, we obtain $\mathbb{C}_1 = 1$. When $n \geq 1$, we have:

$$\mathbb{C}_{n+1} = \frac{2(2n+1)}{n+2}\mathbb{C}_n > \frac{2(2n+1)}{4n+2}\mathbb{C}_n = \mathbb{C}_n$$

Therefore, by induction on n, we obtain:

$$1 = \mathbb{C}_0 = \mathbb{C}_1 < \mathbb{C}_2 < \mathbb{C}_3 < \mathbb{C}_4 < \ldots$$

which completes the proof because of Lemma 1. □

4 Completeness Issues in Decision Procedures in [22,23]

4.1 Problem with Treatment of Disequalities in [22]

In the unification step in the decision procedure proposed by Suter et al. [22], the treatment of disequalities is flawed. When we have disequalities involving trees $t_i \neq t_j$, [22] reduced these as follows:

> 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 $\mathcal{C}$.

Unfortunately, this step is incorrect. The construction determines the satisfiability of disequalities one at a time, but this leads to incompleteness: certain disequalities are unsatisfiable only when considered as a set. For example, suppose that we have a data type representing an enumeration:

$$\mathsf{type\ EnumType\ =\ \{\ Foo\ |\ Bar\ |\ Baz\ \};}$$

then the following is unsatisfiable:

$$(x \neq \mathsf{Foo}) \wedge (x \neq \mathsf{Bar}) \wedge (x \neq \mathsf{Baz})$$

although the individual disequalities are each satisfiable, their combination is not.

In addition, there is an assumption that the types in $\mathcal{E}$ are each infinite. If not, then similar issues arise. For example:

$$\mathsf{type\ EnumType\ =\ \{\ Foo\ [arg:\ Bool]\ |\ Bar\ |\ Baz\ \};}$$

has the same problem in that we can enumerate all the possible combinations of constructors and arguments.

Finally, the decision procedure assumes that the only elements of $\mathcal{E}$ are variables. This restriction is not realistic for the kinds of specifications that are created in Guardol [9]. For example, we could not analyze models that contained constants or arithmetic expressions. Semantically true disequalities like

$$\mathsf{Node(Leaf}, 5, \mathsf{Leaf)} \neq \mathsf{Node(Leaf}, 2+3, \mathsf{Leaf)}$$

were not correctly identified, and were tagged as satisfiable. This led to many situations in which problem instances were incorrectly characterized as *SAT*.

It is not theoretically difficult to solve these problems, and a solution is found in, for example, [2]. Several additions to unification are required:

- We distinguish between finite and infinite constructors. Infinite constructors have either recursive definitions or infinite element types. Finite constructors have either no arguments or finite argument types.
- We then lift the definitions of finite and infinite constructors to data types; a finite data type has only finite constructors.
- If a constructor is finite and contains one or more finite type elements, we split the constructor into several no-argument constructors, one for each combination.
- For each data type variable, we track the set of constructors that could potentially be used to construct the variable. For finitary constructors, a disequality (e.g., $x \neq \mathsf{Bar}$) will remove the constructor from the set.
- Disequalities between composite tree structures of the same shape are pushed down into disequalities between elements and 'leaf level' parametric variables.
- Element disequalities are preserved and passed to the element solver.

4.2 Terminating Problem in the Unrolling Procedure in [23]

In the satisfiability procedure for recursive programs in [23], the authors stated that their procedure is terminating for all sufficiently surjective abstractions. Unfortunately, this claim is incorrect. It is straightforward to create catamorphisms that satisfy the sufficient surjectivity definition that will not terminate using the procedure. For example, consider the following catamorphism, which sums the magnitudes of all the elements within the tree:

$$SumMagn(t) = \begin{cases} 0 & \text{if } t = \mathsf{Leaf} \\ SumMagn(t_L) + |e| + SumMagn(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \end{cases}$$

The catamorphism is sufficiently surjective under the construction in [22] with predicate $M_p(c) = c \geq 0$; for any value in the non-negative range of the type, there are an infinite number of trees that satisfy $\alpha^{-1}(c)$.

Suppose now that we attempt to prove the following predicate is unsatisfiable using the unrolling procedure in [23]:

$$SumMagn\big(\mathsf{Node}(t_1, 2, t_2)\big) < 2$$

for uninterpreted trees t_1 and t_2. It will not terminate with the construction in [23], because the uninterpreted function at the leaves of the unrolling can always choose an arbitrarily large negative number to assign as the value of the catamorphism, thereby creating a satisfying assignment when evaluating the input formula without control conditions. Note that negative values are not in the range of *SumMagn*. The terminating problem may occur with any catamorphism where its range and its codomain are not the same, such as *SizeI* or *Height* catamorphisms.

The issue involves the definition of sufficient surjectivity, which does not actually require that a catamorphism be surjective, i.e., defined across the entire codomain. All that is required for sufficient surjectivity is a predicate M_p that constrains the catamorphism value to represent "acceptably large" sets of trees. The *SumMagn* catamorphism is an example of a sufficiently surjective catamorphism that is not surjective.

5 Unrolling-based Decision Procedure Revisited

In this section, we restate the unrolling procedure proposed by Suter et al. [23] and propose a revised unrolling procedure, shown in Algorithms 1 and 2. The input of both algorithms is a formula ϕ written in the parametric logic and a program Π, which contains ϕ and the definitions of data type τ and catamorphism α. The decision procedure works on top of an SMT solver $\mathcal{S}$ that supports theories for $\tau, \mathcal{E}, \mathcal{C}$, and uninterpreted functions. Note that the only part of the parametric logic that is not inherently supported by $\mathcal{S}$ is the applications of the catamorphism. The main idea of the decision procedure is to approximate the behavior of the catamorphism by repeatedly unrolling it and treating the calls

to the not-yet-unrolled catamorphism instances at the leaves as calls to an uninterpreted function. However, the uninterpreted function can return any values in its codomain; thus, the presence of these uninterpreted functions can make *SAT* results untrustworthy. To address this issue, each time the catamorphism is unrolled, a set of boolean control conditions B is created to determine whether the uninterpreted functions at the bottom level are necessary to the determination of satisfiability. That is, if all control conditions are true, no uninterpreted functions play a role in the satisfiability result.

Algorithm 1: Unrolling decision procedure in [23] with *sufficiently surjective catamorphisms*

```
1  (φ, B) ← unrollStep(φ, Π, ∅)
2  while true do
3      switch decide(φ ∧ ⋀_{b∈B} b) do
4          case SAT
5              return "SAT"
6          case UNSAT
7              switch decide(φ) do
8                  case UNSAT
9                      return "UNSAT"
10                 case SAT
11                     (φ, B) ← unrollStep(φ, Π, B)
```

Algorithm 2: Revised unrolling procedure with *monotonic catamorphisms*

```
1  (φ, B) ← unrollStep(φ, Π, ∅)
2  while true do
3      switch decide(φ ∧ ⋀_{b∈B} b) do
4          case SAT
5              return "SAT"
6          case UNSAT
7              switch decide(φ ∧ R_α) do
8                  case UNSAT
9                      return "UNSAT"
10                 case SAT
11                     (φ, B) ← unrollStep(φ, Π, B)
```

The unrollings without control conditions represent an over-approximation of the formula with the semantics of the program with respect to the parametric logic, in that it accepts all models accepted by the parametric logic plus some others (due to the uninterpreted functions). The unrollings with control conditions represent an under-approximation: all models accepted by this model will be accepted by the parametric logic with the catamorphism.

The algorithm determines the satisfiability of ϕ through repeated unrolling α using the *unrollStep* function. Given a formula ϕ_i generated from the original ϕ after unrolling the catamorphism i times and the set of control conditions B_i of ϕ_i, function $unrollStep(\phi_i, \Pi, B_i)$ unrolls the catamorphim one more time and returns a pair (ϕ_{i+1}, B_{i+1}) containing the unrolled version ϕ_{i+1} of ϕ_i and a set of control conditions B_{i+1} for ϕ_{i+1}. Function $decide(\varphi)$ simply calls $\mathcal{S}$ to check the satisfiability of φ and returns *SAT*/*UNSAT* accordingly. The algorithm either terminates when ϕ is proved to be satisfiable without the use of uninterpreted functions (line 5) or ϕ is proved to be unsatisfiable when assigning any values to uninterpreted functions still cannot make the problem satisfiable (line 9).

The central problem of Algorithm 1 is that its termination is not guaranteed. For example, non-termination can occur if the uninterpreted function U_α representing α can return values outside the range of α. Consider an unsatisfiable input problem: $SizeI\ (t) < 0$, for an uninterpreted tree t when *SizeI* is defined over the integers in an SMT solver. Although *SizeI* is sufficiently surjective, Algorithm 1 will not terminate since each uninterpreted function at the leaves of the unrolling can always choose an arbitrarily large negative number to assign as the value of the catamorphism, thereby creating a satisfying assignment when

evaluating the input formula without control conditions. These negative values are outside the range of *SizeI*, and this termination problem can occur for any catamorphism that is not surjective. Unless an underlying solver supports predicate subtyping, such catamorphisms are easily constructed, and in fact *SizeI* or *Height* catamorphisms are not surjective when defined against SMT-LIB 2.0 [3].

To address the non-termination issue, we need to constrain the assignments to uninterpreted functions $U_\alpha(t)$ representing $\alpha(t)$ to return only values inside the range of α. Let R_α be a condition that, together with $U_\alpha(t)$, represents the range of α. The collection of values that $U_\alpha(t)$ can return can be constrained by R_α. In Algorithm 2, the user-provided range R_α is included in the *decide* function to make sure that any values that $U_\alpha(t)$ returns could be mapped to some "real" tree $t' \in \tau$ such that $\alpha(t') = U_\alpha(t)$:

$$\forall c \in \mathcal{C} : \big(c = U_\alpha(t) \wedge R_\alpha(c)\big) \Rightarrow \big(\exists t' \in \tau : \alpha(t') = c\big) \tag{1}$$

Formula (1) defines a correctness condition for R_α. Unfortunately, it is difficult to prove this without the aid of a theorem prover. On the other hand, it is straightforward to determine whether R_α is a sound approximation of the range of R (that is, all values in the range of R are accepted by R_α) using induction and an SMT solver. To do so, we simply unroll α a single time over an uninterpreted tree t. We assume R_α is true for the uninterpreted functions in the unrolling but that R_α is false over the unrolling. If an SMT solver can prove that the formula is *UNSAT*, then R_α soundly approximates the range; this unrolling encodes both the base and inductive case.

6 Monotonic Catamorphisms

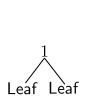
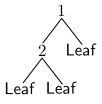
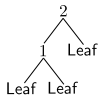
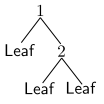
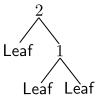
In the rest of the paper, we propose *monotonic* catamorphisms and prove that Algorithm 2 is complete for this class, provided that R_α accurately characterizes the range of α. We show that this class is a subset of sufficiently surjective catamorphisms, but general enough to include all catamorphisms described in [22,23] and all those that we have run into in industrial experience. Monotonic catamorphisms admit a termination argument in terms of the number of unrollings, which is an open problem in [23]. Moreover, a subclass of monotonic catamorphisms, *associative-commutative* (AC) catamorphisms can be combined while preserving completeness of the formula, addressing another open problem in [23].

6.1 Definition

Given a catamorphism α and a tree t, $\beta(t)$ is the size of the set of trees that map to $\alpha(t)$:

$$\beta(t) = |\alpha^{-1}\big(\alpha(t)\big)|$$

Table 2. Examples of $\beta(t)$ with the *Multiset* catamorphism

t					
$\alpha(t)$	$\{1\}$	$\{1,2\}$	$\{1,2\}$	$\{1,2\}$	$\{1,2\}$
$\beta(t)$	1	4	4	4	4

Examples of $\beta(t)$: The value of $\beta(t)$, where $t \in \tau$, clearly depends on α. For example, if α is the *Set* catamorphism, $\beta(\mathsf{Leaf}) = 1$; also, $\forall t \in \tau, t \neq \mathsf{Leaf} : \beta(t) = \infty$ since there are an infinite number of trees that have the same set of element values.

Table 2 shows some examples of $\beta(t)$ with the *Multiset* catamorphism. The value of β(first tree) is 1 because it is the only tree in the parametric logic that can map to $\{1\}$ by catamorphism *Multiset*. The last four trees are distinct and they are all the trees that can map to the same multiset $\{1,2\}$; hence, their $\beta(t)$ are equal to 4.

Definition 3 (Monotonic catamorphism). *A catamorphism $\alpha : \tau \to \mathcal{C}$ is monotonic iff there exists $h_\alpha \in \mathbb{N}^+$ such that:*

$$\forall t \in \tau : height(t) \geq h_\alpha \Rightarrow \big(\beta(t) = \infty \vee \exists t_0 \in \tau : height(t_0) = height(t) - 1 \wedge \beta(t_0) < \beta(t)\big)$$

Note that if α is monotonic with h_α, it is also monotonic with any $h'_\alpha \in \mathbb{N}^+$ bigger than h_α.

6.2 Examples of Monotonic Catamorphisms

This section proves that all sufficiently surjective catamorphims introduced by Suter et al. [22] are monotonic. These catamorphisms are listed in Table 1. Note that the *Sortedness* catamorphism can be defined to allow or not allow duplicate elements [22]; we define $Sortedness_{dup}$ and $Sortedness_{nodup}$ for the *Sortedness* catamorphism where duplications are allowed and disallowed, respectively.

The monotonicity of *Set*, *SizeI*, *Height*, *Some*, *Min*, and $Sortedness_{dup}$ catamorphisms is easily proved by showing the relationship between infinitely surjective abstractions [22] and monotonic catamorphisms.

Lemma 3. *Infinitely surjective abstractions are monotonic.*

Proof. According to Suter et al. [22], α is infinitely surjective S-abstraction, where S is a set of trees, if and only if $\beta(t)$ is finite for $t \in S$ and infinite for $t \notin S$. Therefore, α is monotonic with $h_\alpha = \max\{height(t) \mid t \in S\} + 1$. □

Theorem 1. *Set, SizeI, Height, Some, Min, and $Sortedness_{dup}$ are monotonic.*

Proof. [22] showed that *Set*, *SizeI*, *Height*, and $Sortedness_{dup}$ are infinitely surjective abstractions. Also, *Some* and *Min* have the properties of infinitely surjective $\{\mathsf{Leaf}\}$-abstractions. Therefore, the theorem follows from Lemma 3. □

It is more challenging to prove that *Multiset*, *List*, and $Sortedness_{nodup}$ catamorphisms are monotonic since they are not infinitely surjective abstractions. First, we define the notion of strict subtrees and supertrees.

Definition 4 (Strict subtree). *Given two trees t_1 and t_2 in the tree domain τ, tree t_1 is a subtree of tree t_2, denoted by $t_1 \preceq t_2$, iff:*

$$t_1 = \mathsf{Leaf} \vee$$
$$t_1 = \mathsf{Node}(t_{1L}, e, t_{1R}) \wedge t_2 = \mathsf{Node}(t_{2L}, e, t_{2R}) \wedge t_{1L} \preceq t_{2L} \wedge t_{1R} \preceq t_{2R}$$

Tree t_1 is a strict subtree of tree t_2, denoted by $t_1 \precneqq t_2$, iff

$$t_1 \preceq t_2 \wedge size(t_1) < size(t_2)$$

Similarly, we define the notion $\succneqq$ of strict supertrees as the inverse of $\precneqq$. Next, we state Lemma 4 before proving that *Multiset*, *List*, and $Sortedness_{nodup}$ catamorphisms are monotonic. The proof of Lemma 4 is omitted since it is obvious.

Lemma 4. *For all $h \in \mathbb{N}^+$, any tree of height h must be a strict supertree of at least one tree of height $h-1$.*

Theorem 2. *List catamorphisms are monotonic.*

Proof. Let $h_\alpha = 2$. For any tree t such that $height(t) \geq h_\alpha$, there are exactly $ns\big(size(t)\big)$ distinct trees that can map to $\alpha(t)$. Thus, $\beta(t) = ns\big(size(t)\big)$. Due to Lemma 4, there exists t_0 such that $t_0 \precneqq t \wedge height(t_0) = height(t)-1$, which leads to $size(t_0) < size(t)$. From Property 3, $height(t) \geq h_\alpha = 2$ implies $size(t) \geq 5$. From Lemma 2, $ns\big(size(t_0)\big) < ns\big(size(t)\big)$, which means $\beta(t_0) < \beta(t)$. □

Theorem 3. *Multiset catamorphisms are monotonic.*

Proof. Let $h_\alpha = 2$. Consider any tree t such that $height(t) \geq h_\alpha$. Let $P(t)$ be the number of distinct permutations of multiset $\alpha(t)$. P is monotonic since $\forall t_1, t_2$ where $t_1 \precneqq t_2$, $\alpha(t_1)$ is a sub multiset of $\alpha(t_2)$, which implies $P(t_1) \leq P(t_2)$.

For each permutation of multiset $\alpha(t)$, there are $ns\big(size(t)\big)$ distinct trees corresponding to this permutation and those trees are all mapped to $\alpha(t)$. Thus, $\beta(t) = P(t) \times ns\big(size(t)\big)$.

From Lemma 4, there exists t_0 such that $t_0 \precneqq t$ and $height(t_0) = height(t)-1$, which leads to $size(t_0) < size(t)$. From Property 3, $height(t) \geq h_\alpha = 2$ implies $size(t) \geq 5$. From Lemma 2, $ns\big(size(t_0)\big) < ns\big(size(t)\big)$. Since P is monotonic and $t_0 \precneqq t$, $P(t_0) \leq P(t)$. Thus, $ns\big(size(t_0)\big) \times P(t_0) < ns\big(size(t)\big) \times P(t)$, which causes $\beta(t_0) < \beta(t)$. □

Theorem 4. *$Sortedness_{nodup}$ catamorphisms over integer trees are monotonic.*

Proof. Let $h_\alpha = 2$. Consider any tree t of height 2 or more. If t is unsorted, we have $\beta(t) = \infty$ since there are an infinite number of unsorted trees. As a result, the monotonic property of the catamorphism holds.

On the other hand, consider the case when t is sorted. We have $\alpha(t) = (a, b, \mathsf{true})$, where $a = \min(t)$ and $b = \max(t)$. Since there are only a finite number of sorted trees whose elements are distinct and in the finite range $[a, b]$, we have $\beta(t) < \infty$. From Lemma 4, there exists t_0 such that $t_0 \precneqq t$ and $height(t_0) = height(t) - 1$. Since $height(t_0) = height(t) - 1 \geq 1$, tree t_0 has at least one internal node; in other words, $ni(t_0) \geq 1$. Because $t_0 \precneqq t$, we have $ni(t_0) < ni(t)$. Because duplications are not allowed, $ni(t) \leq b - a + 1$. Therefore,

$$1 \leq ni(t_0) < b - a + 1 \tag{2}$$

From t_0 we construct a tree t'_0 such that $height(t'_0) = height(t) - 1$ and $\beta(t'_0) < \beta(t)$ as follows. Initially, t'_0 is set to be t_0. Let *seq* be the sequence of internal nodes obtained from the in-order traversal of t'_0, excluding all leaves. There are exactly $ni(t_0)$ internal nodes in *seq*. For the i-th internal node in *seq*, where $1 \leq i \leq ni(t_0)$, we re-set the value of its element to $a + i - 1$. After this process, t'_0 is sorted, $height(t'_0) = height(t_0) = height(t) - 1$, $\min(t'_0) = a$, $\max(t'_0) = a + ni(t_0) - 1$ and the range of the element values in t'_0 is in $[a, a + ni(t_0) - 1]$. From Equation (2), this range is a strict sub-range of $[a, b]$. As a result, $|\alpha^{-1}\big((a, a + ni(t_0) - 1, \mathsf{true})\big)| < |\alpha^{-1}\big((a, b, \mathsf{true})\big)|$ since the bigger range for elements we have, the more number of sorted trees with distinct elements we can construct from the range. Consequently, $\beta(t'_0) < \beta(t)$. □

7 Unrolling Decision Procedure - Proof of Correctness

We now prove the correctness of the unrolling decision procedure in Algorithm 2. We start with some properties of monotonic catamorphisms in Section 7.1 and then discuss the main proofs in Section 7.2. In this section, p stands for the number of disequalities between tree terms in the input formula.

7.1 Some Properties of Monotonic Catamorphisms

In the following α is assumed to be a monotonic catamorphism with h_α and β as defined earlier.

Definition 5 ($\mathcal{M}_\beta$). *$\mathcal{M}_\beta(h)$ is the minimum value of $\beta(t)$ of all trees t of height h:*

$$\forall h \in \mathbb{N} : \mathcal{M}_\beta(h) = \min\{\beta(t) \mid t \in \tau, height(t) = h\}$$

Corollary 1. *$\mathcal{M}_\beta(h)$ is always greater or equal to 1.*

Proof. $\forall h \in \mathbb{N} : \mathcal{M}_\beta(h) \geq 1$ since $\forall t \in \tau : \beta(t) = |\alpha^{-1}\big(\alpha(t)\big)| \geq 1$. □

Lemma 5 (Monotonic Property of $\mathcal{M}_\beta$). *Function $\mathcal{M}_\beta : \mathbb{N} \to \mathbb{N}$ satisfies the following monotonic property:*

$$\forall h \in \mathbb{N}, h \geq h_\alpha : \mathcal{M}_\beta(h) = \infty \Rightarrow \mathcal{M}_\beta(h+1) = \infty \quad \vee \\ \mathcal{M}_\beta(h) < \infty \Rightarrow \mathcal{M}_\beta(h) < \mathcal{M}_\beta(h+1)$$

Proof. Consider any $h \in \mathbb{N}$ such that $h \geq h_\alpha$. Consider the case where $\mathcal{M}_\beta(h) = \infty$. From Definition 5, every tree t_h of height h has $\beta(t_h) = \infty$. Hence, every tree t_{h+1} of height $h+1$ has $\beta(t_{h+1}) = \infty$ from Definition 3. Thus, $\mathcal{M}_\beta(h+1) = \infty$.

Consider the other case where $\mathcal{M}_\beta(h) < \infty$. Let t_{h+1} be any tree of height $h+1$. From Definition 3, there are two sub-cases as follows.

Sub-case 1 [$\beta(t_{h+1}) = \infty$]: Because $\mathcal{M}_\beta(h) < \infty$, we have $\mathcal{M}_\beta(h) < \beta(t_{h+1})$.

Sub-case 2 [there exists t_h of height h such that $\beta(t_h) < \beta(t_{h+1})$]: From Definition 5, $\mathcal{M}_\beta(h) < \beta(t_{h+1})$.

In both sub-cases, we have $\mathcal{M}_\beta(h) < \beta(t_{h+1})$. Since t_{h+1} can be any tree of height $h+1$, we have $\mathcal{M}_\beta(h) < \mathcal{M}_\beta(h+1)$ from Definition 5. □

Corollary 2. *For any natural number $p > 0$, $\mathcal{M}_\beta(h_\alpha + p) > p$.*

Proof. By induction on h based on Lemma 5 and Corollary 1. □

Theorem 5. *For every number $p \in \mathbb{N}^+$, there exists some height $h_p \geq h_\alpha$, computable as a function of p, such that for every height $h \geq h_p$ and for every tree t_h of height h, we have $\beta(t_h) > p$.*

Proof. Let $h_p = h_\alpha + p$. From Corollary 2, $\mathcal{M}_\beta(h_p) > p$. Based on Lemma 5, we could show by induction on h that $\forall h \geq h_p : \mathcal{M}_\beta(h) > p$. Hence, this theorem follows from Definition 5. □

Lemma 6. *For all number $p \in \mathbb{N}^+$ and for all tree $t \in \tau$, we have:*

$$\beta(t) > p \Rightarrow \beta\big(\mathsf{Node}(\_, \_, t)\big) > p \wedge \beta\big(\mathsf{Node}(t, \_, \_)\big) > p$$

Proof. Consider tree $t' = \mathsf{Node}(t_L, e, t)$. The value of $\alpha(t')$ is computed in terms of $\alpha(t_L)$, e, and $\alpha(t)$. There are $\beta(t)$ trees that can map to $\alpha(t)$ and we can substitute any of these trees for t in t' without changing the value of $\alpha(t')$. Hence, $\beta(t) > p$ implies $\beta(t') > p$. In other words, $\beta(t) > p \Rightarrow \beta\big(\mathsf{Node}(\_, \_, t)\big) > p$. Similarly, we can show that $\beta(t) > p \Rightarrow \beta\big(\mathsf{Node}(t, \_, \_)\big) > p$. □

7.2 Proof of Correctness of the Unrolling-based Decision Procedure

We claim that our unrolling-based decision procedure with monotonic catamorphisms is (1) sound for proofs, (2) sound for models, (3) terminating for satisfiable formulas, and (4) terminating for unsatisfiable formulas. Due to space limitations, we do not present the proofs for the first three properties, which can be adapted with minor changes from similar proofs in [23]. Rather, we focus on proving that Algorithm 2 is terminating for unsatisfiable formulas. As defined in Section 2.1, the logic is described over only conjunctions, but this can easily be generalized to arbitrary formulas using DPLL(T) [8]. The structure of the proof in this case is the same. The outline of the proof is as follows:

1. Given an input formula ϕ_{in}, without loss of generality, we perform purification and unification on ϕ_{in} to yield a formula ϕ_P. We then define a maximum unrolling depth $\mathfrak{D}$ and formula $\phi_{\mathfrak{D}}$, in which all catamorphism instances in $\phi_{\mathfrak{D}}$ are unrolled to depth $\mathfrak{D}$ as described in Algorithm 2. Note that the formulas differ only in the treatment of catamorphism terms.
2. Given an unrolling $\phi_{\mathfrak{D}}$, if all control conditions are true, then the catamorphism functions are completely determined. Therefore, any model for $\phi_{\mathfrak{D}}$ can be easily converted into a model for ϕ_{in}.
3. If at least one control condition for the unrolling is false, we may have a tree t where $\alpha_{\mathfrak{D}}(t)$ does not match $\alpha(t)$ since the computation of $\alpha_{\mathfrak{D}}(t)$ depends on an uninterpreted function. In this case, we show that it is always possible to replace t with a concrete tree t' that satisfies the other constraints of the formula and that yields the same catamorphism value.
4. To construct t', we first note that past a certain depth of unrolling $depth^{\max}_{\phi_{in}} + 1$, the value chosen for any catamorphism applications will satisfy all constraints other than disequalities between tree terms. We then note that all tree disequality constraints can be satisfied if we continue to unroll the catamorphism h_p times.

Now, let ϕ_{in} be an input formula of Algorithm 2. Without loss of generality, we purify the formula ϕ_{in} (as in [22]) and then perform tree unification (as in [2]) on the resulting formula. If there is a clash during the unification process, ϕ_{in} must be unsatisfiable; otherwise, we obtain a substitution function $\sigma = \{t^1_{var} \mapsto T_1, \ldots, t^m_{var} \mapsto T_m\}$ where each tree variable t^i_{var}, where $1 \leq i \leq m$, does not appear anywhere in tree terms $T_1, \ldots, T_m$. Following [22], the remaining variables (which unify only with themselves and occur only at the leaves of tree terms) we label *parametric variables*.

We substitute for tree variables and obtain a formula $\phi_P = \phi_t \wedge \phi_c \wedge \phi_e \wedge \phi_b$ that is equisatisfiable with ϕ_{in}, where ϕ_t contains disequalities over tree terms (tree equalities have been removed through unification), ϕ_c contains formulas in the collections theory, ϕ_e contains formulas in the element theory, and ϕ_b is a set of formulas of the form $c = \alpha(t)$, where c is a variable in the collections theory and t is a tree term. We observe that given σ and any model for ϕ_P, it is straightforward to create a model for ϕ_{in}.

Given ϕ_P, Algorithm 2 produces formulas $\phi_{\mathfrak{D}}$ which are the same as ϕ_P except that each term $c = \alpha(t)$ in ϕ_b is replaced by $c = \alpha_{\mathfrak{D}}(t)$, where $\alpha_{\mathfrak{D}}$ is the catamorphism unrolled $\mathfrak{D}$ times.

To prove the completeness result, we compute $depth^{\max}_{\phi_{in}}$, which is, intuitively, the maximum depth of any tree term in ϕ_P. Let $depth^{\max}_{\phi_{in}} = \max\{depth_{\phi_P}(t) \mid \text{tree term } t \in \phi_P\}$ where $depth_{\phi_P}(t)$ is defined as follows:

$$depth_{\phi_P}(t) = \begin{cases} 1 + \max\{depth_{\phi_P}(t_L), depth_{\phi_P}(t_R)\} & \text{if } t = \mathsf{Node}(t_L, \_, t_R) \\ 0 & \text{if } t = \mathsf{Leaf} \mid \text{tree variable} \end{cases}$$

We next define a lemma that states that assignments to catamorphisms are *compatible* with all formula constraints other than structural disequalities be-

tween trees. We define ϕ_P^* to be the formula obtained by removing all the tree disequality constraints ϕ_t in ϕ_P.

Lemma 7. *Given a formula ϕ_P^* with monotonic catamorphism α and correct range predicate R_α, after $\mathfrak{D} \geq depth_{\phi_{in}}^{\max} + 1$ unrollings, if $\phi_{\mathfrak{D}}$ has a model, then ϕ_P^* also has a model.*

Proof. It is sufficient to prove the lemma for $\mathfrak{D} = depth_{\phi_{in}}^{\max} + 1$. Assume $\mathcal{M}_{\mathfrak{D}}$ is a model for $\phi_{\mathfrak{D}}$. We claim that we can construct from $\mathcal{M}_{\mathfrak{D}}$ a model $\mathcal{M}^*$ for ϕ_P^*. We note that the assignments to model variables are compatible with ϕ_e and ϕ_c, as they are unchanged from $\phi_{\mathfrak{D}}$ to ϕ_P^*. We now modify $\mathcal{M}_{\mathfrak{D}}$ to ensure that all constraints in ϕ_b are satisfied. For each tree t_d in $\mathcal{M}_{\mathfrak{D}}$, we replace it with a tree t^* in $\mathcal{M}^*$ constructed as follows:

- If $height(t_d) < depth_{\phi_{in}}^{\max} + 1$, t_d belongs to a set of complete trees generated by Algorithm 2. In this case, $\alpha_{\mathfrak{D}}(t_d) = \alpha(t_d)$, so we can set $t^* = t_d$.
- If $height(t_d) \geq depth_{\phi_{in}}^{\max} + 1$, then the calculation of $\alpha_{\mathfrak{D}}(t_d)$ involves at least one uninterpreted function $U_\alpha(t_{uif})$ over some subtree[2] t_{uif} of t_d, and it is possible that $\alpha(t_{uif}) \neq U_\alpha(t_{uif})$. Since we have unrolled the catamorphism $depth_{\phi_{in}}^{\max} + 1$ times, tree t_{uif} corresponds to a subtree of a parametric variable in the original problem. The only constraints on the structure of such subtrees are those on the value of $U_\alpha(t_{uif})$; we call them uninterpreted-constrained subtrees. Furthermore, by the condition of R_α, there exists a concrete subtree t' such that $U_\alpha(t_{uif}) = \alpha(t')$. We construct t^* by replacing all such uninterpreted-constrained subtrees t_{uif} with concrete subtrees t' such that $U_\alpha(t_{uif}) = \alpha(t')$. The structure of t^* matches the structure of t_d up to the level of the deepest tree term in ϕ_P^* (whose depth cannot be deeper than $depth_{\phi_{in}}^{\max}$), so all other formula constraints on t_d are preserved.

After this process, we obtain a model $\mathcal{M}^*$ for ϕ_P^*. □

Theorem 6. *Given a formula ϕ_{in} with monotonic catamorphism α and correct range predicate R_α, after $\mathfrak{D} = depth_{\phi_{in}}^{\max} + 1 + h_p$ unrollings, if $\phi_{\mathfrak{D}}$ has a model, then ϕ_{in} also has a model.*

Proof. We first note that ϕ_{in} is trivially equisatisfiable with ϕ_P, so we focus on showing the equisatisfiability between ϕ_P and $\phi_{\mathfrak{D}}$.

Assume $\mathcal{M}_{\mathfrak{D}}$ is a model for $\phi_{\mathfrak{D}}$. Our goal is to construct an extension $\mathcal{M}$ of $\mathcal{M}_{\mathfrak{D}}$ to make ϕ_P hold. $\mathcal{M}_{\mathfrak{D}}$ specifies values for element variables in $\mathcal{E}$, collection variables in $\mathcal{C}$, tree variables in τ, and values for uninterpreted functions $U_\alpha(t_{uif})$ for some trees t_{uif}. Note that any model $\mathcal{M}$ for ϕ_P does not contain values for uninterpreted functions $U_\alpha(t_{uif})$: these uninterpreted functions are created by the unrolling algorithm to approximate the behaviors of the applications of α. Thus, to construct a model for ϕ_P, we need to get rid of uninterpreted functions

[2] We use the standard definition of subtrees in the proof of Lemma 7. That is, t_1 is a subtree of t_2 if and only if the root of t_1 is a vertex of t_2. Note that this definition is different from the notion of subtrees in Definition 4.

$U_\alpha(t_{uif})$ in $\mathcal{M}_\mathfrak{D}$. We assume that the base solver $\mathcal{S}$ can return complete tree models for any tree terms in $\phi_\mathfrak{D}$.

If all the control conditions in B are true, all the values for uninterpreted functions $U_\alpha(t_{uif})$ in $\mathcal{M}_\mathfrak{D}$ are unnecessary for the determination of satisfiability. We construct $\mathcal{M}$ as follows: we set $\mathcal{M}$ to be $\mathcal{M}_\mathfrak{D}$, remove from $\mathcal{M}$ all the values for $U_\alpha(t_{uif})$, and pick any valid values for the remaining variables that are in ϕ_P but have not been added to $\mathcal{M}$.

On the other hand, consider the case when at least one control condition in B is false. In this case, some values for uninterpreted functions $U_\alpha(t_{uif})$ for some trees t_{uif} in $\mathcal{M}_\mathfrak{D}$ are required for the determination of satisfiability.

From Lemma 7, as long as we can construct a concrete tree $t' \in \tau$ such that $\alpha(t') = U_\alpha(t_{uif})$ for each t_{uif} where $U_\alpha(t_{uif})$ is in $\mathcal{M}_\mathfrak{D}$ while still maintaining the disequalities between tree terms in $\phi_\mathfrak{D}$, we can have a corresponding model $\mathcal{M}$ for ϕ_P. There are two cases for each tree term t_{term} in $\phi_\mathfrak{D}$ as follows.

Case 1 [$height(t_{term}) < depth^{\max}_{\phi_{in}} + 1 + h_p$]: Tree t_{term} is complete because uninterpreted functions $U_\alpha(t_{uif})$ can only appear at depth $depth^{\max}_{\phi_{in}} + 1 + h_p$. For every tree term $t'_{term} \in \phi_\mathfrak{D}$ such that the disequality $t_{term} \neq t'_{term}$ is in $\phi_\mathfrak{D}$, there are two cases to consider: (1) if $height(t'_{term}) < depth^{\max}_{\phi_{in}} + 1 + h_p$, tree term t'_{term} is also complete; hence, the two tree terms must be different since $\phi_\mathfrak{D}$ has a model $\mathcal{M}_\mathfrak{D}$ for them, and (2) if $height(t'_{term}) \geq depth^{\max}_{\phi_{in}} + 1 + h_p$, the two tree terms must also be different since their heights are different. Hence, the disequality constraints in $\phi_\mathfrak{D}$ between t_{term} and other tree terms are satisfied.

Case 2 [$height(t_{term}) \geq depth^{\max}_{\phi_{in}} + 1 + h_p$]: Every vertex at depth $depth^{\max}_{\phi_{in}} + 1 + h_p$ of t_{term} must represent for the root of a tree t_{uif} where $U_\alpha(t_{uif})$ is in $\mathcal{M}_\mathfrak{D}$. For each such tree t_{uif}, by the condition of R_α, there exists a concrete tree $t' \in \tau$ such that $\alpha(t') = U_\alpha(t_{uif})$. We replace t_{uif} with t' and remove the corresponding $U_\alpha(t_{uif})$ from $\mathcal{M}_\mathfrak{D}$. Note that at this point there is no guarantee that all the disequality constraints between t_{term} and other tree terms are satisfied because we replace every t_{uif} with *any* t' such that $\alpha(t') = U_\alpha(t_{uif})$.

Next, we construct a set ST_{h_p} of trees of heights at least h_p as follows. Initially, ST_{h_p} is empty. Starting from each vertex u at depth $depth^{\max}_{\phi_{in}} + 1 + h_p$ of t_{term}, we go bottom-up h_p steps to reach a node u_{h_p}, then add t_{h_p} to ST_{h_p} if $t_{h_p} \notin ST_{h_p}$ where t_{h_p} is the tree rooted at u_{h_p}. After this process, ST_{h_p} is not empty since $height(t_{term}) \geq depth^{\max}_{\phi_{in}} + 1 + h_p$. We visualize this step in Fig. 2.

For each tree $t_{h_p} \in ST_{h_p}$, the relative location of its root u_{h_p} in t_{term} is at depth $depth^{\max}_{\phi_{in}} + 1$. Therefore, there is no disequality constraint between t_{h_p} and any original tree terms in $\phi_\mathfrak{D}$, which cannot locate at a depth bigger than $depth^{\max}_{\phi_{in}}$. Hence, the choice of t_{h_p} only depends on $\alpha(t_{h_p})$. Because $height(t_{h_p}) \geq h_p$, we have $\beta(t_{h_p}) > p$ from Theorem 5. From Lemma 6, we have $\beta(t_{term}) > p$.

Fix all vertices in t_{term} except for those in the trees in ST_{h_p}. Since $\beta(t_{h_p}) > p$ for each $t_{h_p} \in ST_{h_p}$, there exists an assignment $\{t_{h_p} \mapsto t'_{h_p} \mid t_{h_p} \in ST_{h_p}, \alpha(t_{h_p}) = \alpha(t'_{h_p})\}$ that satisfies all the disequality constraints between t_{term} and other tree terms (see Lemma 2 in [22]). We obtain $\hat{t}_{term}$ by applying this assignment to t_{term}.

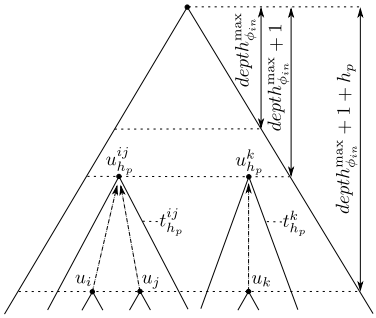


Fig. 2. Construct a set of trees ST_{hp} from t_{term}

We construct $\mathcal{M}$ as follows: First, we set $\mathcal{M}$ to be $\mathcal{M}_{\mathfrak{D}}$. Next, we remove from $\mathcal{M}$ all values for uninterpreted functions $U_\alpha(t_{uif})$. For each value of t_{term} in $\mathcal{M}$, if t_{term} belongs to Case 1, we keep its value; otherwise, we replace t_{term} with $\hat{t}_{term}$ as discussed before. Finally, we assign any valid values to the remaining variables that are in ϕ_P but have not been added to $\mathcal{M}$. □

Corollary 3. *Given a formula ϕ_{in} with monotonic catamorphism α and correct range predicate R_α, Algorithm 2 is terminating for unsatisfiable formulas.*

Proof. This is the immediate contrapositive of Theorem 6. Suppose ϕ_{in} does not have a model. In this case, $\phi_{\mathfrak{D}}$ also does not have a model and the SMT solver $\mathcal{S}$ will return *UNSAT*. □

This proof implies that Algorithm 2 terminates after no more than $depth^{\max}_{\phi_{in}} + 1 + h_p$ number of unrollings for unsatisfiable formulas. If the number of unrollings exceeds $depth^{\max}_{\phi_{in}} + 1 + h_p$, we conclude that ϕ_{in} is satisfiable; note that if we are interested in complete tree models, we still need to keep unrolling until we reach line 5 in Algorithm 2.

Corollary 4. *Monotonic catamorphisms are sufficiently surjective.*

Proof. Recall the definition of sufficient surjectivity in Definition 2. We define $\alpha_{h_p}(t)$ as the unrolling of catamorphism α to depth h_p for tree t, and $CC\big(\alpha_{h_p}(t)\big)$ as the set of control conditions for the unrolling.

Then, we define the set of base shapes S_p and predicate M_p as follows:

- $S_p = \{s \mid height(s) \leq h_p\}$
- $M_p(c) = \exists t.\big(c = \alpha_{h_p}(t)\big) \wedge \neg\big(\bigwedge_{b \in CC(\alpha_{h_p}(t))}\ b\big)$

First, we show that each tree is either in S_p or $M_p\big(\alpha(t)\big)$ holds for the tree. We partition trees by height. Either a tree is shorter or equal in height to h_p, or it is larger. If it is shorter or equal, it is captured by S_p. If it is larger, then we substitute this tree for t. At least one control condition $b \in CC\big(\alpha_{h_p}(t)\big)$ must be false. Since at the leaves of the unrolling, we use uninterpreted functions whose

values are constrained only by R_α, the actual catamorphism value of the tree will be one of the possible values of the unrolled catamorphism $\alpha_{h_p}(t)$.

We note that the set S_p is clearly finite.

Finally, we show that $M_p(c)$ implies $|\alpha^{-1}(c)| > p$. Given a catamorphism value c, by virtue of the restriction that at least one control condition $b \in CC\big(\alpha_{h_p}(t)\big)$ is false and R_α, we can map that value to some tree t such that the height of t is greater than h_p. If c satisfies $M_p(c)$, then by Theorem 5, $|\alpha^{-1}(c)| > p$. □

8 Associative-Commutative (AC) Catamorphisms

This section presents associative-commutative (AC) catamorphisms, a sub-class of monotonic catamorphisms that have some important properties. They are detectable, combinable, and impose an exponentially small upper bound of the number of unrollings. The question whether these results extend to the full class of sufficiently surjective catamorphisms is still open.

8.1 Definition

Definition 6 (AC catamorphism). *Catamorphism $\alpha : \tau \to \mathcal{C}$ is AC if*

$$\alpha(t) = \begin{cases} id_\oplus & \text{if } t = \mathsf{Leaf} \\ \alpha(t_L) \ \oplus \ \delta(e) \ \oplus \ \alpha(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \end{cases}$$

where $\oplus : (\mathcal{C}, \mathcal{C}) \to \mathcal{C}$ is an associative and commutative binary operator with an identity element $id_\oplus \in \mathcal{C}$ (i.e., $\forall x \in \mathcal{C} : x \ \oplus \ id_\oplus = id_\oplus \ \oplus \ x = x$) and $\delta : \mathcal{E} \to \mathcal{C}$ is a function that maps[3] an element value in $\mathcal{E}$ into a corresponding value in $\mathcal{C}$.

Like catamorphisms defined in [22,23], AC catamorphisms are fold functions mapping the content of a tree in the parametric logic into a value in a collection domain where a decision procedure is assumed to be available. There are two main differences in the presentation between AC catamorphisms and those in [22,23]. First, the combine function is replaced by an associative, commutative operator $\oplus$ and function δ. Second, Leaf is mapped to the identity value of operator $\oplus$ instead of the empty value of $\mathcal{C}$ (though the two quantities are usually the same in practice).

Detection: Unlike sufficiently surjective catamorphisms, AC catamorphisms are detectable. A catamorphism, written in the format in Definition 6, is AC if the following conditions hold:

- $\oplus$ is an associative and commutative operator over $\mathcal{C}$.
- $id_\oplus$ is an identity element of $\oplus$.

These conditions can be easily proved by SMT solvers [1,5] or theorem provers such as ACL2 [11].

[3] For instance, if $\mathcal{E}$ is Int and $\mathcal{C}$ is IntSet, we can have $\delta(e) = \{e\}$.

Signature: An AC catamorphism α is completely defined if we know its collection domain $\mathcal{C}$, element domain $\mathcal{E}$, AC operator $\oplus$, and function $\delta : \mathcal{E} \to \mathcal{C}$. In other words, the 4-tuple $\langle \mathcal{C}, \mathcal{E}, \oplus, \delta \rangle$ is the *signature* of α. It is unnecessary to include tree domain τ and identity element $id_{\oplus}$ in the signature since the former depends only on $\mathcal{E}$ and the latter must be specified in the definition of $\oplus$.

Definition 7 (Signature of AC catamorphisms). *The signature of an AC catamorphism α is defined as follows:*

$$sig(\alpha) = \langle \mathcal{C}, \mathcal{E}, \oplus, \delta \rangle$$

Values: Because of the associative and commutative operator $\oplus$, the value of an AC catamorphism for a tree has an important property: it is independent of the structure of the tree.

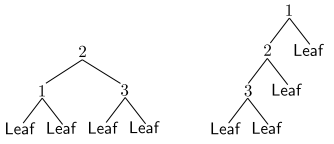
Corollary 5 (Values of AC catamorphisms). *The value of $\alpha(t)$, where α is an AC catamorphism, only depends on the values of elements in t. Furthermore, the value of $\alpha(t)$ does not depend on the relative positions of the element values.*

$$\alpha(t) = \begin{cases} id_{\oplus} & \text{if } t = \mathit{Leaf} \\ \delta(e_1) \ \oplus \ \delta(e_2) \ \oplus \ \ldots \ \oplus \ \delta(e_{ni(t)}) & \text{otherwise} \end{cases}$$

where $e_1, e_2, \ldots, e_{ni(t)}$ are all element values stored in $ni(t)$ internal nodes of t.

Examples: In Table 1, *Height*, *List*, *Some*, and *Sortedness* are not AC because their values depend on the positions of tree elements. Table 3 shows some examples to demonstrate that the values of these catamorphisms depend on the relative locations of element values in trees. The table contains two different trees t_1 and t_2 that have the same collection of element values. Hence, if α is an AC catamorphism, $\alpha(t_1)$ must be equal to $\alpha(t_2)$. Since this condition does not hold with *Height*, *List*, *Some*, and *Sortedness*, they are not AC.

Table 3. Examples of monotonic but not associative-commutative catamorphisms

	t_1	t_2
		
Height	2	3
List (in-order)	(1 2 3)	(3 2 1)
Some	Some(2)	Some(1)
Sortedness	(1, 3, true)	(None, None, false)

Other catamorphisms in Table 1, including *Set*, *Multiset*, *SizeI*, and *Min* are AC. Furthermore, we could define other AC catamorphisms based on well-known associative and commutative operators such as $+, \cap, \max, \vee, \wedge$, etc. We could also use user-defined functions as the operators in AC catamorphisms; in this case, we will need the help of an additional analysis tool to verify that all conditions for AC catamorphims are met.

8.2 AC Catamorphisms are Monotonic

AC catamorphisms are not only automatically detectable but also monotonic. Thus, they can be used in Algorithm 2.

Lemma 8. *If α is an AC catamorphism then*

$$\forall t \in \tau : \beta(t) \geq ns\big(size(t)\big)$$

Proof. Consider any tree $t \in \tau$. Let L be a list of size $ni(t)$ such that L_j, where $1 \leq j \leq ni(t)$, is equal to the value stored in the j-th internal node in t.

Property 2 implies that any shape of size $size(t)$ must have exactly $ni(t)$ SNode(s) and $nl(t)$ SLeaf(s). Let $sh_1, \ldots, sh_{ns(size(t))}$ be all shapes of size $size(t)$. From sh_i, where $1 \leq i \leq ns\big(size(t)\big)$, we construct a tree t_i by converting every SLeaf in sh_i into a Leaf and converting the j-th SNode in sh_i into a structurally corresponding Node with element value L_j, where $1 \leq j \leq ni(t)$. After this process, $t_1, \ldots, t_{ns(size(t))}$ are mutually different because their shapes $sh_1, \ldots, sh_{ns(size(t))}$ are distinct. From Corollary 5, we obtain

$$\alpha(t) = \alpha(t_1) = \ldots = \alpha(t_{ns(size(t))}) = \delta(L_1) \ \oplus \ \delta(L_2) \ \oplus \ \ldots \ \oplus \ \delta(L_{ni(t)})$$

As a result, $\beta(t) \geq ns\big(size(t)\big)$. □

Theorem 7. *AC catamorphisms are monotonic.*

Proof. Let α be an AC catamorphism. Let $h_\alpha = 4$. Consider any tree t such that $height(t) \geq h_\alpha = 4$. If $\beta(t) = \infty$, the monotonic condition for t in Definition 3 holds.

Suppose on the other hand that $\beta(t) < \infty$. Due to Lemma 4, there exists t_0 such that $t_0 \precneqq t \wedge height(t_0) = height(t) - 1 \geq 3$, which implies $size(t_0) \geq 7$ due to Property 3. Due to Lemma 2, $ns\big(size(t_0)\big) \geq ns(7) > 2$. From Lemma 8,

$$\beta(t_0) > 2 \tag{3}$$

which is mathematically equivalent to

$$\beta(t_0) < 2 \times \big(\beta(t_0) - 1\big) \tag{4}$$

Let Q be the collection of internal nodes that are in t but not in t_0. Q is not empty because $t_0 \precneqq t$. Let $e_1, \ldots, e_{|Q|}$ be the elements stored in $|Q|$ nodes in Q. From Corollary 5, we have

$$\alpha(t) = \alpha(t_0) \ \oplus \ \delta(e_1) \ \oplus \ \delta(e_2) \ \oplus \ \ldots \ \oplus \ \delta(e_{|Q|}) \tag{5}$$

Next, we construct a tree t_Q from $e_1, \ldots, e_{|Q|}$. The construction of t_Q is shown in Fig. 3. Let node_i, where $1 \leq i \leq |Q|$, be the node corresponding to e_i in t_Q. We build t_Q in a bottom-up fashion as follows: $\mathsf{node}_{|Q|} = \mathsf{Node}(\mathsf{Leaf}, e_{|Q|}, \mathsf{Leaf})$ and $\mathsf{node}_j = \mathsf{Node}(\mathsf{node}_{j+1}, e_j, \mathsf{Leaf})$, where $Q > j \geq 1$. Let leaf_{Q1} and leaf_{Q2} be the left and right leaves of $\mathsf{node}_{|Q|}$, respectively.

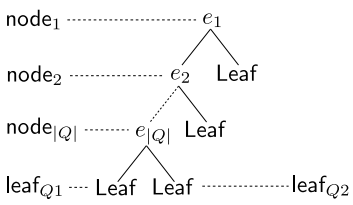


Fig. 3. Construct t_Q from $e_1, \ldots, e_{|Q|}$

From the definition of β, the set of distinct trees that can map to $\alpha(t_0)$ (i.e. α(each tree in this set) $= \alpha(t_0)$) has exactly $\beta(t_0)$ trees. Let num_{t_0} be the number of bigger-than-Leaf trees in this set. Since there is at most one Leaf tree in the set of these $\beta(t_0)$ distinct trees, we have

$$\beta(t_0) - 1 \leq num_{t_0} \leq \beta(t_0) \tag{6}$$

From Equations (3) and (6), we have $num_{t_0} \geq 2$, which means there are at least 2 distinct bigger-than-Leaf trees that can map to $\alpha(t_0)$. Let t_0' and t_0'' be any two of them. That is, t_0' and t_0'' are two different bigger-than-Leaf trees and

$$\alpha(t_0') = \alpha(t_0'') = \alpha(t_0) \tag{7}$$

Let us consider t_0'. Let t_{01}' and t_{02}' be the trees obtained by replacing leaf_{Q1} and leaf_{Q2} in t_Q with t_0', respectively. Since $t_0' \neq \mathsf{Leaf}$, we have $t_{01}' \neq t_{02}'$. From Corollary 5, we have

$$\alpha(t_{01}') = \alpha(t_{02}') = \alpha(t_0') \ \oplus \ \delta(e_1) \ \oplus \ \delta(e_2) \ \oplus \ \ldots \ \oplus \ \delta(e_{|Q|})$$

From Equation (5), Equation (7), and the above equation, we obtain

$$\alpha(t_{01}') = \alpha(t_{02}') = \alpha(t)$$

Hence, *from any bigger-than-*Leaf *tree that can map to* $\alpha(t_0)$*, we could generate at least two different trees that can map to* $\alpha(t)$.

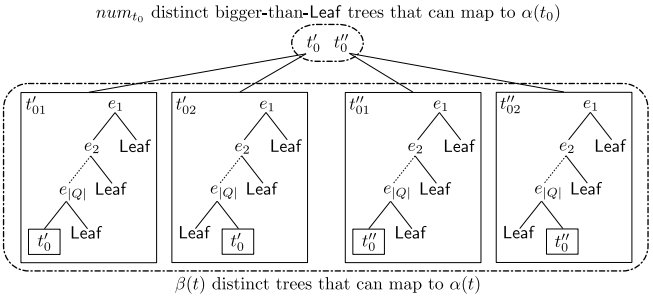


Fig. 4. Relationship between t_0', t_0'' and $t_{01}', t_{02}', t_{01}'', t_{02}''$

Let us consider t''_0. We construct t''_{01} and t''_{02} from t''_0 and t_Q such that $\alpha(t''_{01}) = \alpha(t''_{02}) = \alpha(t)$ using the same method as above. It is clear that $t''_{01} \neq t'_{01}$ since $t''_0 \neq t'_0$. Also, $t''_{02} \neq t'_{01}$ since t''_{02} has leaf_{Q1} but does not have leaf_{Q2} while t'_{01} has leaf_{Q2} but does not have leaf_{Q1}. Similarly, we can show that $t''_{01} \neq t'_{02}$ and $t''_{02} \neq t'_{02}$. Thus, four trees $t'_{01}, t'_{02}, t''_{01}$, and t''_{02} obtained from t'_0 and t''_0 are mutually different. The relationship between them is shown in Fig. 4.

Moreover, t'_0 and t''_0 are any pair of different bigger-than-Leaf trees that can map to $\alpha(t_0)$. Thus, *from all* num_{t_0} *bigger-than-*Leaf *distinct trees that can map to* $\alpha(t_0)$*, we have at least* $2 \times num_{t_0}$ *distinct trees that can map to* $\alpha(t)$. Hence,

$$\begin{aligned} 2 \times num_{t_0} &\leq \beta(t) \\ \therefore 2 \times \big(\beta(t_0) - 1\big) &\leq \beta(t) \quad [\text{ From Equation (6) }] \\ \therefore \beta(t_0) &< \beta(t) \quad [\text{ From Equation (4) }] \end{aligned}$$

As a result, α is monotonic based on Definition 3. □

8.3 Exponentially Small Upper Bound of the Number of Unrollings

In the proof of Theorem 5, we use $h_p = h_\alpha + p$ to guarantee that the algorithm terminates after unrolling no more than $depth^{\max}_{\phi_{in}}+1+h_p$ times. The upper bound implies that the number of unrollings may be large when p is large, leading to a high complexity for the algorithm with monotonic catamorphisms.

In this section, we demonstrate that in the case of AC catamorphims, we could choose a different value for h_p such that not only the termination of the algorithm is guaranteed with h_p, but also the growth of h_p is *exponentially small* compared with that of p. Recall from the proof of Theorem 5 that as long as we can choose $h_p \geq h_\alpha$ such that $\mathcal{M}_\beta(h_p) > p$, Theorem 5 will follow. We will define such h_p after proving the following important lemma.

Lemma 9. *If* α *is AC then* $\forall h \in \mathbb{N} : \mathcal{M}_\beta(h) \geq \mathbb{C}_h$.

Proof. Let $t_h \in \tau$ be any tree of height h. We have $\beta(t_h) \geq ns\big(size(t_h)\big)$ from Lemma 8. Thus, $\beta(t_h) \geq ns(2h+1)$ due to Property 3 and Lemma 2. Therefore, $\beta(t_h) \geq \mathbb{C}_h$ by Lemma 1. As a result, $\mathcal{M}_\beta(h) \geq \mathbb{C}_h$ from Definition 5. □

Let $h_p = \max\big\{h_\alpha, \min\{h \mid \mathbb{C}_h > p\}\big\}$. By construction, $h_p \geq h_\alpha$ and $\mathbb{C}_{h_p} > p$. From Lemma 9, $\mathcal{M}_\beta(h_p) \geq \mathbb{C}_{h_p} > p$. Thus, Theorem 5 follows.

The growth of $\mathbb{C}_n$ is exponential [7]. Thus, h_p is exponentially smaller than p since $\mathbb{C}_{h_p} > p$. For example, when $p = 10^4$, we can choose $h_p = 10$ since $\mathbb{C}_{10} > 10^4$. Similarly, when $p = 5 \times 10^4$, we can choose $h_p = 11$. In the example, we assume that $h_\alpha \leq 10$, which is true for all catamorphisms in this paper.

8.4 Combining AC Catamorphisms

Let $\alpha_1, \ldots, \alpha_m$ be m AC catamorphisms where the signature of the i-th catamorphim ($1 \leq i \leq m$) is $sig(\alpha_i) = \langle \mathcal{C}_i, \mathcal{E}, \oplus_i, \delta_i \rangle$. Catamorphism α with signature $sig(\alpha) = \langle \mathcal{C}, \mathcal{E}, \oplus, \delta \rangle$ is a combination of $\alpha_1, \ldots, \alpha_m$ if

- $\mathcal{C}$ is the domain of m-tuples, where the ith element of each tuple is in $\mathcal{C}_i$.
- $\oplus : (\mathcal{C}, \mathcal{C}) \to \mathcal{C}$ is defined as follows, given $\langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_m \rangle \in \mathcal{C}$:

$$id_{\oplus} = \langle id_{\oplus_1}, id_{\oplus_2}, \ldots, id_{\oplus_m} \rangle$$
$$\langle x_1, x_2, \ldots, x_m \rangle \ \oplus \ \langle y_1, y_2, \ldots, y_m \rangle = \langle x_1 \ \oplus_1 \ y_1, x_2 \ \oplus_2 \ y_2, \ldots, x_m \ \oplus_m \ y_m \rangle$$

- $\delta : \mathcal{E} \to \mathcal{C}$ is defined as follows: $\delta(e) = \big\langle \delta_1(e), \delta_2(e), \ldots, \delta_m(e) \big\rangle$
- α is defined as in Definition 6.

Theorem 8. *Every catamorphism obtained from the combination of AC catamorphims is also AC.*

Proof. Let α be a combination of m AC catamorphisms $\alpha_1, \ldots, \alpha_m$. We prove the AC property of α by showing that $\oplus$ is an associative and commutative operator with identity element $id_{\oplus}$.

Given $\langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_m \rangle, \langle z_1, \ldots, z_m \rangle \in \mathcal{C}$, operator $\oplus$ is commutative because operators $\oplus_1, \ldots, \oplus_m$ are commutative:

$$\begin{aligned}
\langle x_1, x_2, \ldots, x_m \rangle \ \oplus \ \langle y_1, y_2, \ldots, y_m \rangle &= \langle x_1 \ \oplus_1 \ y_1, x_2 \ \oplus_2 \ y_2, \ldots, x_m \ \oplus_m \ y_m \rangle \\
&= \langle y_1 \ \oplus_1 \ x_1, y_2 \ \oplus_2 \ x_2, \ldots, y_m \ \oplus_m \ x_m \rangle \\
&= \langle y_1, y_2, \ldots, y_m \rangle \ \oplus \ \langle x_1, x_2, \ldots, x_m \rangle
\end{aligned}$$

Also, operator $\oplus$ is associative since operators $\oplus_1, \ldots, \oplus_m$ are associative:

$$\begin{aligned}
&\big(\langle x_1, x_2, \ldots, x_m \rangle \ \oplus \ \langle y_1, y_2, \ldots, y_m \rangle\big) \ \oplus \ \langle z_1, z_2, \ldots, z_m \rangle \\
=&\langle x_1 \ \oplus_1 \ y_1, x_2 \ \oplus_2 \ y_2, \ldots, x_m \ \oplus_m \ y_m \rangle \ \oplus \ \langle z_1, z_2, \ldots, z_m \rangle \\
=&\big\langle (x_1 \ \oplus_1 \ y_1) \ \oplus_1 \ z_1, (x_2 \ \oplus_2 \ y_2) \ \oplus_2 \ z_2, \ldots, (x_m \ \oplus_m \ y_m) \ \oplus_m \ z_m \big\rangle \\
=&\big\langle x_1 \ \oplus_1 \ (y_1 \ \oplus_1 \ z_1), x_2 \ \oplus_2 \ (y_2 \ \oplus_2 \ z_2), \ldots, x_m \ \oplus_m \ (y_m \ \oplus_m \ z_m) \big\rangle \\
=&\langle x_1, x_2, \ldots, x_m \rangle \ \oplus \ \langle y_1 \ \oplus_1 \ z_1, y_2 \ \oplus_2 \ z_2, \ldots, y_m \ \oplus_m \ z_m \rangle \\
=&\langle x_1, x_2, \ldots, x_m \rangle \ \oplus \ \big(\langle y_1, y_2, \ldots, y_m \rangle \ \oplus \ \langle z_1, z_2, \ldots, z_m \rangle\big)
\end{aligned}$$

Finally, $id_{\oplus}$ is the identity element of $\oplus$ since $id_{\oplus_1}, \ldots, id_{\oplus_m}$ are the identity elements of $\oplus_1, \ldots, \oplus_m$, respectively:

$$\begin{aligned}
\langle x_1, x_2, \ldots, x_m \rangle \ \oplus \ id_{\oplus} &= \langle x_1, x_2, \ldots, x_m \rangle \ \oplus \ \langle id_{\oplus_1}, id_{\oplus_2}, \ldots, id_{\oplus_m} \rangle \\
&= \langle x_1 \ \oplus_1 \ id_{\oplus_1}, x_2 \ \oplus_2 \ id_{\oplus_2}, \ldots, x_m \ \oplus_m \ id_{\oplus_m} \rangle \\
&= \langle x_1, x_2, \ldots, x_m \rangle
\end{aligned}$$

□

Note that while it is easy to combine AC catamorphims, it might be challenging to compute R_α, where α is a combination of AC catamorphisms.

9 The Relationship between Abstractions

This section discusses the relationship between different types of catamorphisms, including sufficiently surjective, infinitely surjective, monotonic, and AC catamorphisms. Their relationship is shown in Fig. 5.

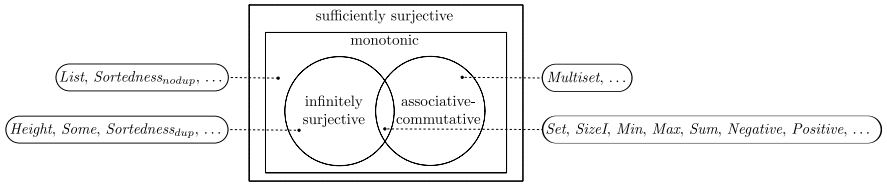


Fig. 5. Relationship between different types of catamorphisms

Monotonic and sufficiently surjective catamorphisms: Corollary 4 shows that all monotonic catamorphisms are sufficiently surjective. Theoretically, the set of sufficiently surjective catamorphisms is a super-set of that of monotonic catamorphisms. In practice, however, we are not aware of any sufficiently surjective catamorphisms that are not monotonic.

Infinitely surjective and monotonic catamorphisms: All infinitely surjective catamorphisms are monotonic, as proved in Lemma 3.

AC and monotonic catamorphisms: All AC catamorphisms are monotonic, as proved in Theorem 7.

Infinitely surjective and AC catamorphisms: The sets of the two types of catamorphisms are intersecting, as shown in Fig. 5.

10 Experimental Results

We have implemented our algorithm in RADA [17], a verification tool used in the Guardol system [9], and evaluated the tool with a collection of 38 benchmark guard examples listed in Table 4. The results are very promising: all of them were automatically verified in a very short amount of time.

Table 4. Experimental results

Benchmark	Result	# unrollings	Time (s)
sumtree(01|02|03|05|06|07|10|11|13)	sat	$1-4$	$0.52-1.02$
sumtree(04|08|09|12|14)	unsat	$0-2$	$0.52-0.98$
negative_positive(01|02)	unsat	$1-6$	$0.33-0.82$
min_max(01|02)	unsat	$1-6$	$0.74-1.44$
mut_rec1	sat	2	0.78
mut_rec(3|4)	unsat	$1-2$	$0.72-1.03$
Email_Guard_Correct_(01| . . . |17)	unsat	$1-2$	$0.72-0.99$

The collection of benchmarks is divided into four sets. The benchmarks in the first three sets were manually designed and those in the last set were automatically generated from Guardol [9]. The first set consists of examples related

to *Sum*, an AC catamorphism that computes the sum of all element values in a tree. The second set contains combinations of AC catamorphisms that are used to verify some interesting properties such as (1) there does not exist a tree with at least one element value that is both positive and negative and (2) the minimum value in a tree can not be bigger than the maximum value in the tree. The definitions of the AC catamorphisms used in the first two sets of benchmarks are as follows.

- *Sum* maps a tree to the sum of all element values in the tree. We assume that $\mathcal{E}$ is a numeric type.

$$Sum(t) = \begin{cases} 0 & \text{if } t = \mathsf{Leaf} \\ Sum(t_L) + e + Sum(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \end{cases}$$

- *Max* is defined in a similar way to *Min* in Table 1.
- *Negative* maps a tree to true if all the element values in the tree are negative and false otherwise. We assume that $\mathcal{E}$ is a numeric type and Leaf is both positive and negative.

$$Negative(t) = \begin{cases} \mathsf{true} & \text{if } t = \mathsf{Leaf} \\ Negative(t_L) \wedge (e < 0) \wedge Negative(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \end{cases}$$

- *Positive* maps a tree to true if all the element values in the tree are positive and false otherwise. We assume that $\mathcal{E}$ is a numeric type and Leaf is both positive and negative.

$$Positive(t) = \begin{cases} \mathsf{true} & \text{if } t = \mathsf{Leaf} \\ Positive(t_L) \wedge (e > 0) \wedge Positive(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \end{cases}$$

To further evaluate the performance of our algorithm, we have conducted some experiments with non-monotonic catamorphisms in the last two sets of benchmarks. In particular, the third set contains simple mutually recursive catamorphisms. Each of the Guardol benchmarks in the last set has 8 mutually recursive data types, 6 catamorphisms, and complex verification conditions.

All benchmarks were run on a machine using an Intel Core I3 running at 2.13 GHz with 2GB RAM with Z3 [5] as the underlying solver ($\mathcal{S}$) in the experiments.

11 Related Work

We discuss some work that is closest to ours. Our approach extends the work by Suter et al. [22,23]. In [22], the authors propose a family of procedures for algebraic data types where catamorphisms are used to abstract tree terms. Their procedures are complete with sufficiently surjective catamorphisms, which are closely related to the notion of monotonic catamorphisms in this paper. We have shown that all monotonic catamorphisms are sufficiently surjective and

all sufficiently surjective catamorphisms described in [22] are monotonic. Moreover, there are a number of advantages of using monotonic catamorphisms, as discussed in Sections 6 and 8. In the early phase of the Guardol project [9], we implemented the decision procedures [22] on top of OpenSMT [4] and found some flaws in the treatment of disequalities in the unification step of the procedures; fortunately, the flaws can be fixed using the techniques in [2].

Our unrolling-based decision procedure is based on the work by Suter et al. [23]. As pointed out in Section 5, their work has a non-terminating issue involving the use of uninterpreted functions. Also, their method works with sufficiently surjective catamorphisms while ours is designed for monotonic catamorphisms.

One work that is close to ours is that of Madhusudan et al. [15], where a sound, incomplete, and automated method is proposed to achieve recursive proofs for inductive tree data-structures while still maintaining a balance between expressiveness and decidability. The method is based on DRYAD, a recursive extension of the first-order logic. DRYAD has some limitations: the element values in DRYAD must be of type int and only four classes of abstractions are allowed in DRYAD. In addition to the sound procedure, [15] shows a decidable fragment of verification conditions that can be expressed in STRAND$_{dec}$ [14]. However, this decidable fragment does not allow us to reason about some important properties such as the height or size of a tree. However, the class of data structures that [15] can work with is richer than that of our approach.

Using abstractions to summarize recursively defined data structures is one of the popular ways to reason about algebraic data types. This idea is used in the Jahob system [24,25] and in some procedures for algebraic data types [20,23,10,15]. However, it is often challenging to directly reason about the abstractions. One approach to overcome the difficulty, which is used in [23,15], is to approximate the behaviors of the abstractions using uninterpreted functions and then send the functions to SMT solvers [5,1] that have built-in support for uninterpreted functions and recursive data types (although recursive data types are not officially defined in the SMT-LIB 2.0 format [3]).

Recently, Sato et al. [19] proposes a verification technique that has support for recursive data structures. The technique is based on higher-order model checking, predicate abstraction, and counterexample-guided abstraction refinements. Given a program with recursive data structures, we encode the structures as functions on lists, which are then encoded as functions on integers before sending the resulting program to the verification tool described in [12]. Their method can work with higher-order functions while ours cannot. On the other hand, their method cannot verify some properties of recursive data structures while ours can thanks to the use of catamorphisms. An example of such a property is as follows: after inserting an element to a binary tree, the set of all element values in the new tree must be a super set of that of the original tree.

12 Conclusion

We have proposed a revised unrolling decision procedure for algebraic data types with monotonic catamorphisms. Like sufficiently surjective catamorphisms, monotonic catamorphisms are fold functions that map abstract data types into values in a decidable domain. We have showed that all sufficiently surjective catamorphisms known in the literature to date [22] are actually monotonic. We have also established an upper bound of the number of unrollings with monotonic catamorphisms. Furthermore, we have pointed out a sub-class of monotonic catamorphisms, namely associative-commutative (AC) catamorphisms, which are proved to be detectable, combinable, and guarantee an exponentially small maximum number of unrollings thanks to their close relationship with Catalan numbers. Our combination results extend the set of problems that can easily be reasoned about using the catamorphism-based approach.

In the future, we would like to generalize the notion of catamorphisms to allow additional inputs related to either control conditions (e.g. *member*) or leaf values (e.g. *fold* functions), while preserving completeness guarantees. Also, we would like to generalize the completeness argument for mutually recursive data types involving multiple catamorphisms.

In addition, our decision procedure assumes a correct R_α value, and may diverge if this value is not correct. We believe that it is possible to check the R_α value during unrolling and to return *error* if the value is incorrect by examining the soundness of R_α after removing a value chosen for U_α within the problem (call this $R_{\alpha-U}$). If this is sound, then R is incorrect, and we should return *error*.

Acknowledgements. We thank David Hardin, Konrad Slind, Andrew Gacek, Sanjai Rayadurgam, and Mats Heimdahl for their feedback on early drafts of this paper. We thank Philippe Suter and Viktor Kuncak for discussions about their decision procedures [22,23]. This work was sponsored in part by NSF grant CNS-1035715 and by a subcontract from Rockwell Collins.

References

1. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, pages 171–177, 2011.
2. 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.
3. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT*, 2010.
4. R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT solver. In *TACAS*, pages 150–153, 2010.
5. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.
6. S. S. Epp. *Discrete Mathematics with Applications*. Brooks/Cole Publishing Co., 4th edition, 2010.
7. P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

8. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *CAV*, pages 175–188, 2004.
9. D. Hardin, K. Slind, M. Whalen, and T.-H. Pham. The Guardol Language and Verification System. In *TACAS*, pages 18–32, 2012.
10. S. Jacobs and V. Kuncak. Towards Complete Reasoning about Axiomatic Specifications. In *VMCAI*, pages 278–293, 2011.
11. M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Springer, 2000.
12. N. Kobayashi, R. Sato, and H. Unno. Predicate Abstraction and CEGAR for Higher-Order Model Checking. In *PLDI*, pages 222–233, 2011.
13. T. Koshy. *Catalan Numbers with Applications*. Oxford University Press, 2009.
14. P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. In *POPL*, pages 611–622, 2011.
15. P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive Proofs for Inductive Tree Data-Structures. In *POPL*, pages 123–136, 2012.
16. D. C. Oppen. Reasoning About Recursively Defined Data Structures. *J. ACM*, 27(3):403–411, July 1980.
17. T.-H. Pham and M. W. Whalen. RADA: A Tool for Reasoning about Algebraic Data Types with Abstractions. In *ESEC/FSE*, 2013 (to appear).
18. K. H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 7th edition, 2012.
19. R. Sato, H. Unno, and N. Kobayashi. Towards a Scalable Software Model Checker for Higher-Order Programs. In *PEPM*, pages 53–62, 2013.
20. V. Sofronie-Stokkermans. Locality Results for Certain Extensions of Theories with Bridging Functions. In *CADE*, pages 67–83, 2009.
21. R. P. Stanley. *Enumerative Combinatorics, Volume 2*. Cambridge University Press, 2001.
22. P. Suter, M. Dotta, and V. Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *POPL*, pages 199–210, 2010.
23. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability Modulo Recursive Programs. In *SAS*, pages 298–315, 2011.
24. K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *PLDI*, pages 349–361, 2008.
25. K. Zee, V. Kuncak, and M. C. Rinard. An Integrated Proof Language for Imperative Programs. In *PLDI*, pages 338–351, 2009.