J Autom Reasoning (2016) 57:281–318
DOI 10.1007/s10817-016-9368-2

Reasoning About Algebraic Data Types with Abstractions

Tuan-Hung Pham[1] · **Andrew Gacek**[2] · **Michael W. Whalen**[1]

Received: 15 March 2014 / Accepted: 9 March 2016 / Published online: 29 March 2016

Abstract Reasoning about functions that operate over algebraic data types is an important problem for a large variety of applications. One application of particular interest is network applications that manipulate or reason about complex message structures, such as XML messages. This paper presents a decision procedure for reasoning about algebraic data types using abstractions that are provided by catamorphisms: fold functions that map instances of algebraic data types to values in a decidable domain. We show that the procedure is sound and complete for a class of catamorphisms that satisfy a generalized sufficient surjectivity condition. Our work extends a previous decision procedure that unrolls catamorphism functions until a solution is found. We use the generalized sufficient surjectivity condition to address an incompleteness in the previous unrolling algorithm (and associated proof). We then propose the categories of monotonic and associative catamorphisms, which we argue provide a more intuitive inclusion test than the generalized sufficient surjectivity condition. We use these notions to address two open problems from previous work: (1) we provide a bound, with respect to formula size, on the number of unrollings necessary for completeness, showing that it is linear for monotonic catamorphisms and exponentially small for associative catamorphisms, and (2) we demonstrate that associative catamorphisms can be combined within a formula while preserving completeness. Our combination results extend the set of problems that can be reasoned about using the catamorphism-based approach. We also describe an implementation of the approach, called RADA, which accepts formulas in an extended version of the SMT-LIB 2.0 syntax. The procedure is quite general and is central to the reasoning infrastructure for Guardol, a domain-specific language for reasoning about network guards.

✉ Tuan-Hung Pham
hung@cs.umn.edu

Andrew Gacek
andrew.gacek@gmail.com

Michael W. Whalen
whalen@cs.umn.edu

[1] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

[2] Rockwell Collins, Advanced Technology Center, Cedar Rapids, IA, USA

Keywords Decision procedures · Algebraic data types · SMT solvers

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; they are also 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 [2,17]. 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.

Suter et al. [27] described a parametric decision procedure for reasoning about algebraic data types using catamorphism (fold) functions. In the procedure, catamorphisms describe the 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:

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

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)
```

The work in [27,28] provides a foundation towards reasoning about such formulas. The approach 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.

Example 1 Suppose we have a catamorphism *remDirtyWords* that removes from an XML message m all the words in a given blacklist. Also suppose we want to verify the following idempotent property of the catamorphism: the result obtained after applying the catamorphism to a message m twice is the same as the result obtained after applying the catamorphism to m once. We can write this property as a formula that can be decided by the decision procedure in [27] as follows:

$$\mathit{remDirtyWords}(m) = \mathit{remDirtyWords}\big(\mathit{remDirtyWords}(m)\big)$$

We can also use the decision procedure to verify properties of programs that manipulate algebraic data structures. First, we turn the program into *verification conditions* that are formulas in our logic (c.f., [8]), then use the decision procedure to solve these conditions. A sample verification condition for the *add* function is:

$$(t_1 = \mathsf{Node}(t_2, e_1, t_3) \wedge \mathit{setOf}(t_4) = \mathit{setOf}(t_2) \cup \{e_2\}) \implies \\ \mathit{setOf}(\mathsf{Node}(t_4, e_1, t_3)) = \mathit{setOf}(t_1) \cup \{e_2\}$$

△

The procedure [27] was proved sound for all catamorphisms and claimed to be complete for a class of catamorphisms called *sufficiently surjective* catamorphisms, which we will describe in more detail in Sect. 3.1. The original algorithm in [27] was quite expensive to compute and required a specialized predicates M_p and S_p to be defined separately for each catamorphism and proved correct with respect to the catamorphism using either a hand-proof or a theorem prover. In [28], a generalized algorithm for the decision procedure was proposed, based on unrolling the catamorphism. This algorithm had three significant advantages over the algorithm in [27]: 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, both algorithms are *incomplete* for some sufficiently surjective catamorphisms. In [27], the proposed algorithms are incomplete for problems involving finite types and formulas involving inequalities that are non-structural (e.g.: $5 + 3 \neq 8$). In [28], the proposed algorithm is incomplete because of missing assumptions about the range of the catamorphism function.

In this paper, we propose a complete unrolling-based decision procedure for catamorphisms that satisfy a *generalized sufficient surjectivity* condition. We also demonstrate that our unrolling procedure is complete for sufficiently surjective catamorphisms, given suitable S_p and M_p predicates.

We then address two open problems with the previous work [28]: (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? We introduce *monotonic* catamorphisms and prove that our decision procedure is complete with monotonic catamorphisms, and this class of catamorphisms gives a linear unrolling bound for the procedure. While monotonic catamorphisms include all catamorphisms introduced by [27,28], we show that monotonic catamorphisms are a strict subset of sufficiently surjective catamorphisms. To answer the second question, we introduce *associative* catamorphisms, which can be combined within a formula while preserving completeness results. These associative catamorphisms have the additional property that they require a very small number of unrollings to solve, and we demonstrate that this behavior explains some of the empirical success in applying catamorphism-based approaches on interesting examples from previous papers [8,28].

We have implemented the decision procedure in an open-source tool called RADA (reasoning about algebraic data types), which has been used as a back-end tool in the Guardol system [8]. The successful use of RADA in the Guardol project on large-scale guard programs demonstrates that the unrolling approach and the tools are sufficiently mature for use on interesting, real-world applications.

This paper offers the following contributions:

- We propose an unrolling-based decision procedure for algebraic data types with *generalized sufficiently surjective* catamorphisms.
- We provide a corrected proof of completeness for the decision procedure with generalized sufficiently surjective catamorphisms.
- We propose a new class of catamorphisms, called *monotonic* catamorphisms, and argue that it is a more intuitive notion than generalized sufficient surjectivity. We show that the number of unrollings needed for monotonic catamorphisms is linear.
- We also define an important subclass of monotonic catamorphisms called *associative* catamorphisms and show that an arbitrary number of these catamorphisms can be combined in a formula while preserving decidability. Another nice property of associative catamorphisms is that determining whether a catamorphism function is associative can be immediately checked by an SMT solver without performing unrolling, so we call these catamorphisms *detectable*. Finally, associative catamorphisms are guaranteed to require an exponentially small number of unrollings to solve.
- We describe an implementation of the approach, called RADA, which accepts formulas in an extended version of the SMT-LIB 2.0 syntax [3], and demonstrate it on a range of examples.

This paper is an expansion of previous work in [20,22]. It provides a complete and better organized exposition of the ideas from previous work, and includes substantial new material, including the new notion of generalized sufficient surjectivity, a set of revised, full proofs that work for both the class of sufficiently surjective catamorphisms in [27] and the new catamorphism classes in this paper, a demonstration of the relationship between monotonic and sufficiently surjective catamorphisms, new implementation techniques in RADA, and new experimental results.

The rest of this paper is organized as follows. Section 2 presents some related work that is closest to ours. In Sect. 3, we present the unrolling-based decision procedure and prove its completeness. Section 4 presents monotonic catamorphisms. Section 5 presents associative catamorphisms. The relationship between different types of catamorphisms is discussed in Sect. 6. Experimental results for our approach are shown in Sect. 7. We conclude this paper in Sect. 8.

2 Related Work

The most relevant work related to the research in this paper fall in two broad categories: verification tools and decision procedures for algebraic data types.

2.1 Verification Tools for Algebraic Data Types

We introduce in this paper a new verification tool called RADA to reason about algebraic data types with catamorphisms. RADA is described in detail in Sect. 7 and the algorithms behind it are presented in Sects. 3, 4, and 5. Besides RADA, there are some tools that support catamorphisms (as well as other functions) over algebraic data types. For example, Isabelle [16], PVS [18], and ACL2 [10] provide efficient support for both inductive reasoning and evaluation. Although very powerful and expressive, these tools usually need manual assistance and require substantial expert knowledge to construct a proof. On the contrary, RADA is fully automated and accepts input written in the popular SMT-LIB 2.0 format [3]; therefore, we believe that RADA is more suited for non-expert users.

In addition, there are a number of other tools built on top of SMT solvers that have support for data types. One of such tools is Dafny [13], which supports many imperative and object-oriented features; hence, Dafny can solve many verification problems that RADA cannot. On the other hand, Dafny does not have explicit support for catamorphisms, so for many problems it requires significantly more annotations than RADA. For example, RADA can, without any annotations other than the specification of correctness, demonstrate the correctness of insertion and deletion for red-black trees. From examining proofs of similarly complex data structures (such as the PriorityQueue) provided in the Dafny distribution, it is likely that these proofs would require significant annotations in Dafny.

Our work was inspired by the Leon system [4], which uses a semi-decision procedure to reason about catamorphisms [28]. While Leon uses Scala input, RADA offers a neutral input format, which is a superset of SMT-LIB 2.0. Also, Leon specifically uses Z3 [6] as its underlying SMT solver, whereas RADA is solver-independent: it currently supports both Z3 and CVC4. In fact, RADA can support any SMT solver that uses SMT-LIB 2.0 and that has support for algebraic data types and uninterpreted functions. RADA also guarantees the completeness of the results even when the input formulas have multiple catamorphisms for certain classes of catamorphisms such as PAC catamorphisms [21]; in this situation, it is unknown whether the decision procedure [28] used in Leon can ensure the completeness.[1] Recent work by the Leon group [23] broadens the class of formulas that can be solved by the tool towards arbitrary recursive functions, but it makes no claims on completeness.

2.2 Decision Procedures for Algebraic Data Types

The general approach of using abstractions to summarize algebraic data types has been used in the Jahob system [29,30] and in some procedures for algebraic data types [9,15,25,28]. However, it is often challenging to directly reason about the abstractions. One approach to overcome the difficulty (e.g., in [15,28]) is to approximate the behaviors of the abstractions using uninterpreted functions and then send the functions to SMT solvers [1,6] that have built-in support for uninterpreted functions and recursive data types.

Our approach extends the work by Suter et al. [27,28]. In [27], the authors propose a family of procedures for algebraic data types where catamorphisms are used to abstract tree terms. These procedures are claimed to be sound for all catamorphisms and complete with *sufficiently surjective* catamorphisms. Unfortunately, there are flaws in the completeness argument, and in fact the family of algorithms is incomplete for non-structural disequalities and catamorphisms over finite types. These incompletenesses and possible fixes to them are described in detail in [19]. An improved approach using a single unrolling-based decision

[1] The authors of [28] only claim completeness of the procedure when there is only one non-parametric catamorphism in the input formulas.

procedure is proposed in [28]. This approach is very similar to the algorithm that is proposed in this paper. Our approach addresses an incompleteness in the unrolling algorithm due to the use of uninterpreted functions without range restrictions that is described in Sect. 3.3. Another similar work 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. On the other hand, the class of data structures that [15] can work with is richer than that of our approach and can involve mutual references between elements (pointers).

Sato et al. [24] 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 refinement. Given a program with recursive data structures, they 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 [11]. Their method can work with higher-order functions while ours cannot. On the other hand, their method is incomplete and 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*.

Zhang et al. [31] define an approach for reasoning over datatypes with integer constraints related to the size of recursive data structures. This approach is much less general than ours: the size relation in [31] can be straightforwardly constructed as a monotonic integer catamorphism matching the shape of the datatype. On the other hand, the work in [31] presents a decision procedure for quantified formulas, whilst our approach only supports quantifier-free formulas.

3 Unrolling-Based Decision Procedure

Inspired by the decision procedures for algebraic data types by Suter et al. [27,28], in this section we present an unrolling-based decision procedure, the idea of generalized sufficient surjectivity, and proofs of soundness and completeness of the procedure for catamorphisms satisfying the condition.

3.1 Preliminaries

We describe the parametric logic used in the decision procedures for algebraic data types, which is also the logic used in our decision procedure. We also summarize the definition of catamorphisms and the idea of sufficient surjectivity from [27,28]. 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.

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	$::=$	$e \mid \mathsf{elem}(T) \mid \mathcal{T}_{\mathcal{E}}$	$\mathcal{E}$-terms
F_T	$::=$	$T = T \mid T \neq T$	Tree (dis)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	$::=$	$F_T \mid F_C \mid F_E \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi$	Formulas

Fig. 1 Syntax of the parametric logic

$$\begin{array}{rcl}
[\mathsf{Node}(T_1, e, T_2)] & = & \mathsf{Node}([T_1], [e]_{\mathcal{E}}, [T_2]) \\
[\mathsf{Leaf}] & = & \mathsf{Leaf} \\
[\mathsf{left}(\mathsf{Node}(T_1, e, T_2))] & = & [T_1] \\
[\mathsf{right}(\mathsf{Node}(T_1, e, T_2))] & = & [T_2] \\
[\mathsf{elem}(\mathsf{Node}(T_1, e, T_2))] & = & [e]_{\mathcal{E}} \\
[\alpha(t)] & = & \text{given by the catamorphism} \\
[T_1 = T_2] & = & [T_1] = [T_2] \\
[T_1 \neq T_2] & = & [T_1] \neq [T_2] \\
[E_1 = E_2] & = & [E_1]_{\mathcal{E}} = [E_2]_{\mathcal{E}} \\
[\mathcal{F}_{\mathcal{E}}] & = & [\mathcal{F}_{\mathcal{E}}]_{\mathcal{E}} \\
[C_1 = C_2] & = & [C_1]_{\mathcal{C}} = [C_2]_{\mathcal{C}} \\
[\mathcal{F}_{\mathcal{C}}] & = & [\mathcal{F}_{\mathcal{C}}]_{\mathcal{C}} \\
[\neg\phi] & = & \neg[\phi] \\
[\phi_1 \star \phi_2] & = & [\phi_1] \star [\phi_2] \text{ where } \star \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}
\end{array}$$

Fig. 2 Semantics of the parametric logic

3.1.1 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, a decidable element theory $\mathcal{L}_{\mathcal{E}}$ of values in an element domain $\mathcal{E}$ containing terms E, 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. Figure 1 shows the syntax of the logic instantiated for binary trees. Its semantics can be found in Fig. 2. The semantics refer to the catamorphism α as well as the semantics of elements $[\ ]_{\mathcal{E}}$ and collections $[\ ]_{\mathcal{C}}$. In a slight abuse of notation, we will also refer to terms in the union of C and E as CE terms (respectively, elements of the $\mathcal{CE}$ domain).

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 F_E 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 formulas in the parametric logic.

3.1.2 Catamorphisms

Given a tree in the parametric logic shown in Fig. 1, we can map the tree to a value in $\mathcal{C}$ using a *catamorphism*, which is a fold function of the following format:

Table 1 Sufficiently surjective catamorphisms in [27]

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)	(1 2)
		List(e) @ $\alpha(t_L)$ @ $\alpha(t_R)$ (pre-order)	(2 1)
		$\alpha(t_L)$ @ $\alpha(t_R)$ @ List(e) (post-order)	(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)	(1, 2, true)
		(min element, max element, true) (if tree sorted)	

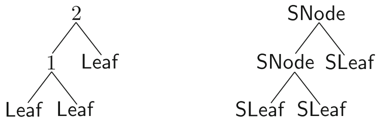


Fig. 3 An example of a tree and its shape

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

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

The catamorphisms defined in [27] are shown in Table 1. The first column contains catamorphism names.[2] 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 the tree in Fig. 3.

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 a tree, and true/false depending on whether the tree is sorted or not.

Infinitely surjective catamorphisms: Suter et al. [27] showed that many interesting catamorphisms are *infinitely surjective*. Intuitively, a catamorphism is infinitely surjective if the cardinality of its inverse function is infinite for all but a finite number of trees.

Definition 1 (*Infinitely surjective catamorphisms*) A catamorphism α is an infinitely surjective S-abstraction, where S is a finite set of trees, if and only if the inverse image $\alpha^{-1}\big(\alpha(t)\big)$ is finite for $t \in S$ and infinite for $t \notin S$.

Example 2 (*Infinitely surjective catamorphisms*) The *Set* catamorphism in Table 1 is an infinitely surjective {Leaf}-abstraction because:

[2] *SizeI*, which maps a tree to its number of *internal* nodes, was originally named *Size* in [27]. We rename the catamorphism to easily distinguish it from the function $size$, which returns the total number of *all* vertices in a tree, in this paper.

- $\left|Set^{-1}\big(Set(\mathsf{Leaf})\big)\right| = |Set^{-1}(\emptyset)| = 1$ (i.e., Leaf is the only tree in data type τ that can map to $\emptyset$ by the Set catamorphism). Hence, $Set^{-1}\big(Set(\mathsf{Leaf})\big)$ is finite.
- $\forall t \in \tau, t \neq \mathsf{Leaf} : \left|Set^{-1}\big(Set(t)\big)\right| = \infty$. The reason is that when t is not Leaf, we have $Set(t) \neq \emptyset$. Hence, there are an infinite number of trees that can map to $Set(t)$ by the catamorphism Set. For example, consider the tree in Fig. 3; let us call it t_0. We have $Set(t_0) = \{1, 2\}$; hence, $|Set^{-1}(\{1, 2\})| = \infty$ since there are an infinite number of trees in τ whose elements values are 1 and 2.

As a result, Set is infinitely surjective by Definition 1. △

Sufficiently surjective catamorphisms: The decision procedures by Suter et al. [27,28] were claimed to be complete if the catamorphism used in the procedures is *sufficiently surjective* [27]. Intuitively, a catamorphism is sufficiently surjective if the inverse of the catamorphism has sufficiently large cardinality for all but a finite number of tree shapes. In fact, the class of infinitely surjective catamorphisms is just a special case of sufficiently surjective catamorphisms [27].

To define the notion of sufficiently surjective catamorphisms, we have to define *tree shapes* first. The shape of a tree is obtained by removing all element values in the tree. Figure 3 shows an example of a tree and its shape.

Definition 2 (*Tree shapes*) The shape of a tree is defined by constant SLeaf and constructor SNode(_, _) as follows:

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

Definition 3 (*Sufficiently surjective catamorphisms* [27]) A catamorphism α is sufficiently surjective iff for each $p \in \mathbb{N}^+$, there exists, computable as a function of p,

- a finite set of shapes S_p
- a closed formula M_p in the union of the collection and element theories[3] such that for any collection element c, $M_p(c)$ implies $|\alpha^{-1}(c)| > p$

such that $M_p\big(\alpha(t)\big)$ or $shape(t) \in S_p$ for every tree term t.

Example 3 (*Sufficiently surjective catamorphisms*) We showed in Example 2 that the Set catamorphism is infinitely surjective. Let us now show that the catamorphism is sufficiently surjective by Definition 3. Let $S_p = \{\mathsf{SLeaf}\}$ and $M_p(c) \equiv c \neq \emptyset$. For this M_p, the only base case to consider is the tree Leaf: either a tree is Leaf, whose shape is in S_p, or the catamorphism value returned is not the empty set, in which case M_p holds. Furthermore, $M_p(c)$ implies $|\alpha^{-1}(c)| = \infty$. △

Despite its name, sufficient surjectivity has no surjectivity requirement for the range of α. It only requires a "sufficiently large" number of trees for values satisfying the condition M_p. The $SizeI$ catamorphism is a good example of a sufficiently surjective catamorphism that is not surjective. In other words, there is no restriction for the range of a sufficiently surjective catamorphism. However, to ensure the completeness of the unrolling decision procedure, the range restriction must be taken into account. We will discuss this issue in Sect. 3.3.

[3] Note that Suter et. al in [27] describe M_p over the collection theory only, but that paper contains examples that involve both the collection and element theory (c.f., M_p for multiset catamorphisms). The addition of the element theory does not require modification to any of the proofs in our work or [27].

Table 1 describes all sufficiently surjective catamorphisms in [27].

The only catamorphism in [27] not in Table 1 is the *Mirror* catamorphism:

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

Since the cardinality of the inversion function of the catamorphism *Mirror* is always 1, the sufficiently surjective condition does not hold for this catamorphism.

3.2 Properties of Trees and Shapes in the Parametric Logic

We present some important properties of trees and shapes in the parametric logic (Sect. 3.1.1) which play important roles in the subsequent sections of this paper.

Properties of Trees We assume the standard definitions of height and size for trees in the parametric logic with $height(\mathsf{Leaf}) = 0$ and $size(\mathsf{Leaf}) = 1$. The following properties result directly from structural induction on trees in the parametric logic.

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 k internal nodes and $k + 1$ leaves.

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

$$\forall t \in \tau : size(t) \geq 2 \times height(t) + 1$$

Properties of Tree Shapes. We now show a special relationship between tree shapes and the well-known Catalan numbers [26], which, according to Koshy [12], can be computed as follows:

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

where $\mathbb{C}_n$ is the n^{th} Catalan number. Catalan numbers will be used to establish some properties of associative catamorphisms in Sect. 5.

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. Due to Property 2, the size of a shape is in $\bar{\mathbb{N}}$. Let $ns(s)$ be the $\underline{n}$umber of tree $\underline{s}$hapes of size $s \in \bar{\mathbb{N}}$.

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 $\mathbb{C}_{\frac{s-1}{2}}$ [12,26]. □

Lemma 2 *Function $ns : \bar{\mathbb{N}} \rightarrow \mathbb{N}^+$ is monotone:*

$$1 = ns(1) = ns(3) < ns(5) < ns(7) < ns(9) < \dots$$

Proof Clearly, $\mathbb{C}_1 = \mathbb{C}_0 = 1$. 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: $1 = \mathbb{C}_0 = \mathbb{C}_1 < \mathbb{C}_2 < \mathbb{C}_3 < \mathbb{C}_4 < \dots$, which completes the proof because of Lemma 1. □

3.3 Unrolling-Based Decision Procedure Revisited

This section presents our unrolling-based decision procedure, which was inspired by the work by Suter et al. [28]. First, let us define two notions that will be used frequently throughout the discussions in this section.

An uninterpreted function representing catamorphism applications. The evaluation of $\alpha(t_0)$ for some tree term $t_0 \in \tau$ might depend on the value of some $\alpha(t'_0)$ that we have no information to evaluate. In this case, our decision procedure treats $\alpha(t'_0)$ as an application of the uninterpreted function $U_\alpha(t'_0)$, where $U_\alpha : \tau \to \mathcal{C}$.

For example, suppose that only $\alpha(\mathsf{left}(t_0))$ needs to be considered as an uninterpreted function while evaluating $\alpha(t_0)$; we can compute $\alpha(t_0)$ as follows:

$$\alpha(t_0) = \begin{cases} \mathsf{empty} & \text{if } t_0 = \mathsf{Leaf} \\ \mathsf{combine}\big(U_\alpha\big(\mathsf{left}(t_0)\big), \mathsf{elem}(t_0), \alpha\big(\mathsf{right}(t_0)\big)\big) & \text{if } t_0 \neq \mathsf{Leaf} \end{cases}$$

Control conditions For each catamorphism application $\alpha(t)$, we use a control condition b_t to check whether the evaluation of $\alpha(t)$ depends on the uninterpreted function U_α or not. If b_t is true, we can evaluate $\alpha(t)$ without calling to the uninterpreted function U_α. For the $\alpha(t_0)$ example above, we have $b_{t_0} \equiv t_0 = \mathsf{Leaf}$.

The unrolling procedure proposed by Suter et al. [28] is restated in Algorithm 1, and our revised unrolling procedure is shown in Algorithm 2.

The input of both algorithms consists of

- a formula ϕ written in the parametric logic (described in Sect. 3.1.1) that consists of literals over elements of tree terms and tree abstractions generated by a catamorphism (i.e., a fold function that maps a recursively-defined data type to a value in a base domain). In other words, ϕ contains a recursive data type τ (a tree term as defined in the syntax), an element type $\mathcal{E}$ of the value stored in each tree node, a collection type $\mathcal{C}$ of tree abstractions in a decidable logic $\mathcal{L}_\mathcal{C}$, and a catamorphism $\alpha : \tau \to \mathcal{C}$ that maps an object in the data type τ to a value in the collection type $\mathcal{C}$.
- a program Π, which contains ϕ, the definitions of data type τ, and catamorphism α.

Algorithm 1: Unrolling decision procedure in [28] with *sufficiently surjective* catamorphisms

```
$\phi \leftarrow \phi[U_\alpha/\alpha]$
$(\phi, B, \_) \leftarrow unrollStep(\phi, \Pi, \emptyset)$
while true do
    switch $decide(\phi \wedge \bigwedge_{b \in B} b)$ do
        case SAT
            return "SAT"
        case UNSAT
            switch $decide(\phi)$ do
                case UNSAT
                    return "UNSAT"
                case SAT
                    $(\phi, B, \_) \leftarrow$
                      $unrollStep(\phi, \Pi, B)$
```

Algorithm 2: Unrolling decision proc. with *generalized sufficiently surjective* catamorphisms (Def. 5)

```
$\phi \leftarrow \phi[U_\alpha/\alpha]$
$(\phi, B, R) \leftarrow unrollStep(\phi, \Pi, \emptyset)$
while true do
    switch $decide(\phi \wedge \bigwedge_{b \in B} b)$ do
        case SAT
            return "SAT"
        case UNSAT
            switch $decide(\phi \wedge \bigwedge_{r \in R} r)$ do
                case UNSAT
                    return "UNSAT"
                case SAT
                    $(\phi, B, R) \leftarrow$
                      $unrollStep(\phi, \Pi, B)$
```

The decision procedure works on top of an SMT solver $\mathcal{S}$ that supports theories for τ, $\mathcal{E}$, $\mathcal{C}$, 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 U_α. We start by replacing all instances of the catamorphism α by instances of an uninterpreted function U_α using the substitution notation $\phi[U_\alpha/\alpha]$. The uninterpreted function can return any values in its codomain; thus, the presence of this uninterpreted function 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 if the determination of satisfiability is independent of the uninterpreted function at the "leaf" level of the unrolling. That is, if all the control conditions in B are true, the uninterpreted function U_α does not play any role in the satisfiability result. 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 logic plus some others (due to the uninterpreted function). The unrollings with control conditions represent an under-approximation: all models accepted by this model will be accepted by the logic with the catamorphism.

In addition, we observe that if a catamorphism instance is treated as an uninterpreted function, the uninterpreted function should only return values inside the *range*[4] of the catamorphism. In our decision procedure, a user-provided predicate R_α captures the range constraint of the catamorphism. R_α is applied to instances of $U_\alpha(t)$ to constrain the values of the uninterpreted function to the range of α.

Algorithm 2 determines the satisfiability of ϕ through repeated unrollings of α using the *unrollStep* function in Algorithm 3. 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 catamorphism one more time and returns a triple $(\phi_{i+1}, B_{i+1}, R_{i+1})$ containing the unrolled version ϕ_{i+1} of ϕ_i, a set of control conditions B_{i+1} for ϕ_{i+1}, and a set of range restrictions R_{i+1} for elements of the codomain of U_α corresponding to trees in the leaf-level of the unrolling.

[4] The codomain of a function $f : X \rightarrow Y$ is the set Y while the range of f is the actual set of all of the output the function can return in Y. For example, the codomain of *Height* when defined against SMT-LIB 2.0 [3] is Int while its range is the set of natural numbers.

Algorithm 3: Algorithm for *unrollStep*(ϕ, Π, B)

```
if $B = \emptyset$ then                                   /* Function called for the first time */
    $FN \leftarrow \{t \mid \alpha(t) \in \phi\}$                  /* Global set of frontier nodes */
    $B \leftarrow \{\mathsf{false}\}$
    $R \leftarrow \{R_\alpha(U_\alpha(t)) \mid t \in FN\}$
else
    $FN \leftarrow \bigcup\{\{\mathsf{left}(t), \mathsf{right}(t)\} \mid t \in FN\}$
    $B \leftarrow \emptyset$
    $R \leftarrow \emptyset$
    for $t \in FN$ do
        $B \leftarrow B \cup \{t = \mathsf{Leaf}\}$
        $\phi \leftarrow \phi \wedge (U_\alpha(t) =$
                $(\mathsf{ite}\ (t = \mathsf{Leaf})\ \mathsf{empty}_\Pi\ (\mathsf{combine}_\Pi(U_\alpha(\mathsf{left}(t)), \mathsf{elem}(t), U_\alpha(\mathsf{right}(t))))))$
        $R \leftarrow R \cup \{R_\alpha(U_\alpha(\mathsf{left}(t))), R_\alpha(U_\alpha(\mathsf{right}(t)))\}$
return $\phi, B, R$
```

The mechanism by which Algorithm 3 "unrolls" the catamorphism is actually by constraining the values returned by U_α. This is done with equality constraints that describe the structure of the catamorphism. Each time we unroll, we start from a set of recently unrolled "frontier" vertices FN that define the nodes at the current leaf-level of unrolling. FN is initialized when the function is called for the first time. We then extend the frontier by examining the left and right children of the frontier nodes and define the structure of α over the (previously unconstrained) left and right children of the current frontier of the unrolling process. The unrolling checks whether or not the tree in question is a Leaf; if so, its value is empty_Π; otherwise, its value is the result of applying the approximated catamorphism U_α to its children.

Function $decide(\varphi)$ in Algorithm 2 simply calls the solver $\mathcal{S}$ to check the satisfiability of φ and returns *SAT*/*UNSAT* accordingly. Algorithm 2 either terminates when ϕ is proved to be satisfiable without the use of the uninterpreted function (line 6) or ϕ is proved to be unsatisfiable when the presence of uninterpreted function cannot make the problem satisfiable (line 10).

Let us examine how satisfiability and unsatisfiability are determined in the procedure. In general, the algorithm keeps unrolling the catamorphism until we find a *SAT*/*UNSAT* result that we can trust. To do that, we need to consider several cases after each unrolling step is carried out. First, at line 5, ϕ is satisfiable and all the control conditions are true, which means the uninterpreted function is not involved in the satisfiable result. In this case, we have a complete tree model for the *SAT* result and we can conclude that the problem is satisfiable.

On the other hand, consider the case when $decide(\phi \wedge \bigwedge_{b \in B} b) = UNSAT$. The *UNSAT* may be due to the unsatisfiability of ϕ, or the set of control conditions, or both of them together. To understand the *UNSAT* case more deeply, we could try to check the satisfiability of ϕ alone. Note that checking ϕ alone would mean that the control conditions are not used; consequently, the values of the uninterpreted function could contribute to the *SAT*/*UNSAT* result. Therefore, we instead check ϕ with the range restrictions on the uninterpreted function in the satisfiability check (i.e., $decide(\phi \wedge \bigwedge_{r \in R} r)$ at line 8) to ensure that if a catamorphism instance is viewed as an uninterpreted function then the uninterpreted function only returns values inside the range of the catamorphism. If $decide(\phi \wedge \bigwedge_{r \in R} r) = UNSAT$ as at line 9, we can conclude that the problem is unsatisfiable because the presence of the uninterpreted function still cannot make the problem satisfiable as a whole. Finally, we need to consider the case

$decide(\phi \wedge \bigwedge_{r \in R} r) = SAT$ as at line 11. Since we already know that $decide(\phi \wedge \bigwedge_{b \in B} b) = UNSAT$, the only way to make $decide(\phi \wedge \bigwedge_{r \in R} r) = SAT$ is by using at least one value returned by the uninterpreted function, which also means that the *SAT* result is untrustworthy. Therefore, we need to keep unrolling at least one more time as denoted at line 12.

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. For example, consider an unsatisfiable formula: $SizeI(t) < 0$ when *SizeI* is defined over the integers in an SMT solver. Although *SizeI* is sufficiently surjective [27], 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*. Broadly speaking, this termination problem can occur for any catamorphism that is not surjective. Unless an underlying solver supports predicate subtyping, such catamorphisms are easily constructed. In fact, *SizeI* and *Height* catamorphisms are not surjective when defined against SMT-LIB 2.0 [3]. 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 *SizeI* catamorphism is an example of a sufficiently surjective catamorphism that is not surjective.

To address the non-termination issue, we need to constrain the assignments to the uninterpreted function U_α representing α to return only values from the range of α. A user-provided predicate R_α is used as a recognizer for the range of α to make sure that any values that uninterpreted function U_α returns can actually be returned by α:

$$\forall c \in \mathcal{C} : R_\alpha(c) \Leftrightarrow (\exists t \in \tau : \alpha(t) = c) \tag{1}$$

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 an overapproximation of the range of α (that is, all values in the range of α are accepted by R_α) using an inductive argument that can be checked using an SMT solver. To do so, we check the following formula:

$$\exists c_1, c_2 \in \mathcal{C}, e \in \mathcal{E} : (\neg R_\alpha(\mathsf{empty}_\Pi)) \vee (R_\alpha(c_1) \wedge R_\alpha(c_2) \wedge \neg R_\alpha(\mathsf{combine}_\Pi(c_1, e, c_2)))$$

This formula, which can be directly analyzed by an SMT solver, checks whether R_α is true for leaf-level trees (checking empty) and for non-leaf trees (using an inductive argument over $\mathsf{combine}$). If the solver proves that the formula is *UNSAT*, then R_α overapproximates the range of α.

This check ensures that the unrolling algorithm is sound (we don't 'miss' any possible catamorphism values) but not that it is complete. For example, the predicate $R_\alpha(c) = \mathsf{true}$ recognizes the entire codomain, $\mathcal{C}$, and leads to the incompletenesses mentioned earlier for the *SizeI* and *Height* catamorphisms. In our approach, it is the user's responsibility to provide an accurate range recognizer predicate.

3.3.1 Catamorphism Decision Procedure by Example

As an example of how the procedure in Algorithm 2 can be used, let us consider a guard application (such as those in [8]) that needs to determine whether an HTML message may be

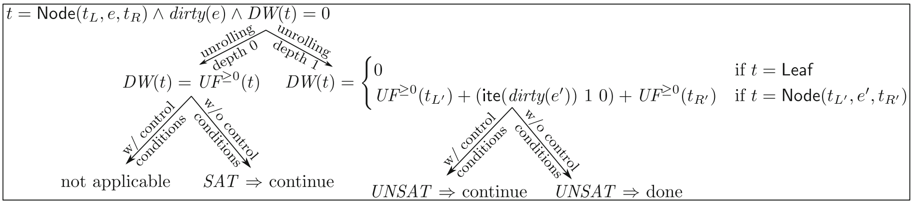

$t = \mathsf{Node}(t_L, e, t_R) \land dirty(e) \land DW(t) = 0$
unrolling depth 0
unrolling depth 1
$DW(t) = UF^{\geq 0}(t)$
$DW(t) = \begin{cases} 0 & \text{if } t = \mathsf{Leaf} \\ UF^{\geq 0}(t_{L'}) + (\mathsf{ite}(dirty(e'))\ 1\ 0) + UF^{\geq 0}(t_{R'}) & \text{if } t = \mathsf{Node}(t_{L'}, e', t_{R'}) \end{cases}$
w/ control conditions
w/o control conditions
w/ control conditions
w/o control conditions
not applicable
$SAT \Rightarrow$ continue
$UNSAT \Rightarrow$ continue
$UNSAT \Rightarrow$ done

Fig. 4 An example of how the decision procedure works

sent across a trusted to untrusted network boundary. One aspect of this determination may involve checking whether the message contains a significant number of "dirty words"; if so, it should be rejected. Our goal is to ensure that this guard application works correctly.

We can check the correctness of this program by splitting the analysis into two parts. A verification condition generator (VCG) generates a set of formulas to be proved about the program and a back end solver attempts to discharge the formulas. In the case of the guard application, these back end formulas involve tree terms representing the HTML message, a catamorphism representing the number of dirty words in the tree, and equalities and inequalities involving string constants and uninterpreted functions for determining whether a word is "dirty".

In our dirty-word example, the tree elements are strings and we can map a tree to the number of its dirty words by the following $DW : \tau \to \mathsf{int}$ catamorphism:

$$DW(t) = \begin{cases} 0 & \text{if } t = \mathsf{Leaf} \\ DW(t_L) + \big(\mathsf{ite}\ dirty(e)\ 1\ 0\big) + DW(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \end{cases}$$

where $\mathcal{E}$ is string and $\mathcal{C}$ is int. We use ite to denote an if-then-else statement.

For our guard example, suppose one of the verification conditions is:

$$t = \mathsf{Node}(t_L, e, t_R) \land dirty(e) \land DW(t) = 0$$

which is *UNSAT*: since t has at least one dirty word (i.e., value e), its number of dirty words cannot be 0. Fig. 4 shows how the procedure works in this case.

At unrolling depth 0, $DW(t)$ is treated as an uninterpreted function $UF^{\geq 0} : \tau \to \mathsf{int}$, which, given a tree, can return any value of type int (i.e., the codomain of DW) bigger or equal to 0 (i.e., the range of DW). The use of $UF^{\geq 0}(t)$ implies that for the first step we do not use control conditions. The formula becomes

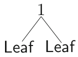
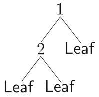
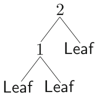
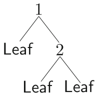
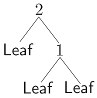
$$t = \mathsf{Node}(t_L, e, t_R) \land dirty(e) \land DW(t) = 0 \land DW(t) = UF^{\geq 0}(t)$$

and is *SAT*. However, the *SAT* result is untrustworthy due to the presence of $UF^{\geq 0}(t)$; thus, we continue unrolling $DW(t)$.

At unrolling depth 1, we allow $DW(t)$ to be unrolled up to depth 1 and all the catamorphism applications at lower depths will be treated as instances of the uninterpreted function. In particular, $UF^{\geq 0}(t_{L'})$ and $UF^{\geq 0}(t_{R'})$ are the values of the uninterpreted function for $DW(t_{L'})$ and $DW(t_{R'})$, respectively. The set of control conditions in this case is $\{t = \mathsf{Leaf}\}$; if we use the set of control conditions (i.e., all control conditions in the set hold), the values of $UF^{\geq 0}(t_{L'})$ and $UF^{\geq 0}(t_{R'})$ will not be used. Hence, in the case of using the control conditions, the formula becomes:

$$t = \mathsf{Node}(t_L, e, t_R) \land dirty(e) \land DW(t) = 0 \land (t = \mathsf{Leaf}) \land \Big(DW(t) = 0 \land t = \mathsf{Leaf} \lor$$
$$DW(t) = UF^{\geq 0}(t_{L'}) + \big(\mathsf{ite}\ dirty(e')\ 1\ 0\big) + UF^{\geq 0}(t_{R'}) \land t = \mathsf{Node}(t_{L'}, e', t_{R'})\Big)$$

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

which is equivalent to

$$t = \mathsf{Node}(t_L, e, t_R) \wedge dirty(e) \wedge DW(t) = 0 \wedge t = \mathsf{Leaf}$$

which is *UNSAT* since t cannot be Node and Leaf at the same time. Since we get *UNSAT* with control conditions, we continue the process without using control conditions. Without control conditions, the formula becomes

$$t = \mathsf{Node}(t_L, e, t_R) \wedge dirty(e) \wedge DW(t) = 0 \wedge \Big(DW(t) = 0 \wedge t = \mathsf{Leaf} \vee$$
$$DW(t) = UF^{\geq 0}(t_{L'}) + \big(\mathsf{ite}\ dirty(e')\ 1\ 0\big) + UF^{\geq 0}(t_{R'}) \wedge t = \mathsf{Node}(t_{L'}, e', t_{R'})\Big)$$

which after eliminating the Leaf case (since t must be a Node) and unifying $\mathsf{Node}(t_L, e, t_R)$ with $\mathsf{Node}(t_{L'}, e', t_{R'})$ (since they are equal to t), simplifies to

$$t = \mathsf{Node}(t_L, e, t_R) \wedge dirty(e) \wedge DW(t) = 0 \wedge$$
$$DW(t) = UF^{\geq 0}(t_{L'}) + \big(\mathsf{ite}\ dirty(e)\ 1\ 0\big) + UF^{\geq 0}(t_{R'})$$

which after evaluating the ite expression, is equivalent to

$$t = \mathsf{Node}(t_L, e, t_R) \wedge dirty(e) \wedge DW(t) = 0 \wedge DW(t) = UF^{\geq 0}(t_{L'}) + 1 + UF^{\geq 0}(t_{R'})$$

which is *UNSAT* because $UF^{\geq 0}(t_{L'}) + 1 + UF^{\geq 0}(t_{R'}) > 0$. Getting *UNSAT* without control conditions guarantees that the original formula is *UNSAT*.

We have another example of the procedure in Example 9 in Sect. 7.

3.4 Correctness of the Unrolling Decision Procedure

We now prove the correctness of the unrolling decision procedure in Algorithm 2. First, let us define the notion of the cardinality of the inverse function of catamorphisms.

Definition 4 (Function β) Given a catamorphism $\alpha : \tau \to \mathcal{C}$, we define $\beta : \tau \to \mathbb{N} \cup \{\infty\}$ as the cardinality of the inverse function of $\alpha(t)$:

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

Example 4 (*Function* β) Intuitively, $\beta(t)$ is the number of distinct trees that can map to $\alpha(t)$ via catamorphism α. The value of $\beta(t)$ clearly depends on α. For example, for the *Set* catamorphism, $\beta_{Set}(\mathsf{Leaf}) = 1$; also, $\forall t \in \tau, t \neq \mathsf{Leaf} : \beta_{Set}(t) = \infty$ since there are an infinite number of trees that have the same set of element values. For the *DW*catamorphism in Sect. 3.3.1, we have $\forall t \in \tau : \beta_{DW}(t) = \infty$.

Table 2 shows some examples of $\beta(t)$ with the *Multiset* catamorphism. The only tree that can map to $\{1\}$ by catamorphism *Multiset* is $\mathsf{Node}(\mathsf{Leaf}, 1, \mathsf{Leaf})$. The last four trees are all the trees that can map to the multiset $\{1, 2\}$. △

Throughout this section we assume $\alpha : \tau \rightarrow \mathcal{C}$ is a catamorphism defined by empty and $\mathsf{combine}$ with β as defined in Definition 4. We will prove that our decision procedure is complete if α satisfies the *generalized sufficient surjectivity* condition defined in Definition 5.

Definition 5 (*Generalized sufficient surjectivity*) Given a catamorphism α and the corresponding function β from Definition 4, we say that α is a generalized sufficiently surjective (GSS) catamorphism if it satisfies the following condition. For every number $p \in \mathbb{N}$, there exists some height $h_p \in \mathbb{N}$, computable as a function of p, such that for every tree t with $\mathit{height}(t) \geq h_p$ we have $\beta(t) > p$.

Corollary 1 *Sufficiently surjective catamorphisms are GSS.*

Proof Let α be a sufficiently surjective catamorphism. By Definition 3, there exists a finite set of shapes S_p such that for every tree t, $\mathit{shape}(t) \in S_p$ or $\beta(t) > p$. Taking $h_p = 1 + \max\{\mathit{height}(t) \mid t \in S_p\}$ ensures that for all trees t, if $\mathit{height}(t) \geq h_p$ then $\beta(t) > p$, as needed. □

We claim that our unrolling-based decision procedure with GSS catamorphisms is (1) sound for proofs: if the procedure returns UNSAT, then there are no models, (2) sound for models: every model returned by the procedure makes the formula true, (3) terminating for satisfiable formulas, and (4) terminating for unsatisfiable formulas. We do not present the proofs for the first three properties, which can be adapted with minor changes from similar proofs in [28]. Rather, we focus on proving that Algorithm 2 is terminating for unsatisfiable formulas.

In order to reason about our unrolling-based decision procedure we define a related mathematical notion of unrolling called n-level approximation. We show that for large enough values of n the n-level approximation of α can be used in place of α while preserving key satisfiability constraints. Finally, we show that our unrolling-based decision procedure correctly uses uninterpreted functions to model n-level approximations of α.

Definition 6 (*n-level approximation*) We say that α_0 is a 0-level approximation of α iff $\forall t \in \tau : \alpha_0(t) \in \mathit{range}(\alpha)$. For $n > 0$ we say α_n is an n-level approximation of α iff

$$\begin{aligned}\alpha_n(\mathsf{Leaf}) &= \mathsf{empty}\\ \alpha_n(\mathsf{Node}(t_L, e, t_R)) &= \mathsf{combine}(\alpha_{n-1}(t_L), e, \alpha_{n-1}(t_R))\end{aligned}$$

where α_{n-1} is an $(n-1)$-level approximation of α.

Lemma 3 *If α_n is an n-level approximation of α then it is also m-level approximation for all $0 \leq m < n$.*

Proof When $n = 0$ the result is vacuously true. For $n > 0$, induction on n shows that α_n is an $(n-1)$-level approximation of α. The result then follows by induction on $n - m$. □

Intuitively, an n-level approximation α_n of α always returns values in the range of α. In particular, α_n agrees with α on short trees, and on taller trees α_n provides values that α also provides for taller trees. These intuitions are formalized in the next series of lemmas.

Lemma 4 *Let α_n be an n-level approximation of α. Then $range(\alpha_n) \subseteq range(\alpha)$. Equivalently, for every t there exists t' such that $\alpha_n(t) = \alpha(t')$.*

Proof Straightforward induction on n. □

Lemma 5 *Let α_n be an n-level approximation of α. If $height(t) < n$ then $\alpha_n(t) = \alpha(t)$.*

Proof Straightforward induction on n. □

Lemma 6 *Let α_n be an n-level approximation of α. If $height(t) \geq n$ then there exists t' such that $\alpha_n(t) = \alpha(t')$ and $height(t') \geq n$.*

Proof Induction on n. When $n = 0$ the result follows from the definition of α_0 and $range(\alpha)$. In the inductive step let t be a tree such that $height(t) \geq n$. Since $n > 0$ we have $t = \mathsf{Node}(t_L, e, t_R)$ for some t_L, e, and t_R. By the definition of α_n we have $\alpha_n(t) = \mathsf{combine}(\alpha_{n-1}(t_L), e, \alpha_{n-1}(t_R))$ where α_{n-1} is an $(n-1)$-level approximation of α. By the definition of height we have $height(t_L) \geq n - 1$ or $height(t_R) \geq n - 1$. Without loss of generality, let us assume $height(t_L) \geq n - 1$ (the argument for t_R is symmetric). Then by the inductive hypothesis there exists t'_L with $\alpha_{n-1}(t_L) = \alpha(t'_L)$ and $height(t'_L) \geq n - 1$. By Lemma 4 there exists t'_R with $\alpha_{n-1}(t_R) = \alpha(t'_R)$. Letting $t' = \mathsf{Node}(t'_L, e, t'_R)$ we have

$$\alpha_n(t) = \mathsf{combine}(\alpha_{n-1}(t_L), e, \alpha_{n-1}(t_R)) = \mathsf{combine}(\alpha(t'_L), e, \alpha(t'_R)) = \alpha(t')$$

Also, since $height(t'_L) \geq n - 1$ we have $height(t') \geq n$. □

Definition 7 We say a formula ϕ in the parametric logic is in standard form if it has the form $\phi_t \wedge \phi_{ce}$ where ϕ_t is a conjunction of disequalities between distinct tree variables and ϕ_{ce} is a formula in the $\mathcal{CE}$ theory where α is applied only to tree variables.

In the following lemma we write $\phi[\alpha_n/\alpha]$ to denote the formula ϕ with all occurrences of α replaced with α_n.

Lemma 7 *Let ϕ be a formula in the standard form and let α be a GSS catamorphism. Then there exists an n such that if $\phi[\alpha_n/\alpha]$ is satisfiable for some n-level approximation α_n of α then ϕ is satisfiable.*

Proof Let ϕ have the form $\phi_t \wedge \phi_{ce}$ from Definition 7. Let p be the number of disequalities in ϕ_t. By Definition 5 there exists a height h_p such that for any tree t of height greater than or equal to h_p we have $\beta(t) > p$. Let $n = h_p$.

Let α_n be an n-level approximation of α such that $\phi[\alpha_n/\alpha]$ is satisfiable. Let $\mathcal{M}$ be a model of $\phi[\alpha_n/\alpha]$. We will construct $\mathcal{M}'$, a modified version of $\mathcal{M}$ with different values for the tree variables, such that $\mathcal{M}'$ satisfies ϕ. In particular, for each tree variable x in ϕ we will construct $\mathcal{M}'$ such that $\alpha_n(\mathcal{M}(x)) = \alpha(\mathcal{M}'(x))$ and such that $\mathcal{M}'$ is a model for the disequalities in ϕ_t. This will ensure that $\mathcal{M}'$ is a model of ϕ_{ce} since α is only applied to tree variables in ϕ_{ce}, and thus $\mathcal{M}'$ will be a model for ϕ. We construct $\mathcal{M}'$ by considering each tree variable in turn.

Let x be a tree variable in ϕ. Let $T = \mathcal{M}(x)$ be the concrete value of x in the model $\mathcal{M}$. If $height(T) < n$ then $\alpha_n(T) = \alpha(T)$ by Lemma 5. In that case we can take $\mathcal{M}'(x) = \mathcal{M}(x)$. Otherwise, $height(T) \geq n$ and by Lemma 6 there exists a T' such that $height(T') \geq n$ and $\alpha_n(T) = \alpha(T')$. By Definition 5 we have $\beta(T') > p$. That is, we have more than p distinct trees T'' such that $\alpha(T'') = \alpha(T') = \alpha_n(T)$. Since we have p disequalities in ϕ_t there are at most p forbidden values for T'' in order to satisfy ϕ_t. Thus we can always make a selection for x in $\mathcal{M}'$ such that the p disequalities are still satisfied. □

Lemma 8 *A formula ϕ in the parametric logic can be translated to an equisatisfiable formula $\phi_1 \vee \cdots \vee \phi_k$ such that each ϕ_i is in standard form.*

Proof We prove this lemma by providing a series of translation steps from ϕ to a disjunction of formulas in standard form. Each step of the translation will produce an equisatisfiable formula which is closer to standard form.

To simplify the presentation of the translation steps, we write $\phi[e]$ to indicate that e is an expression that appears in ϕ, and we then write $\phi[e']$ to denote ϕ with all occurrences of e replaced by e'.

Many of these translation steps will introduce new variables. We will write such variables as $\bar{x}$, with a line over them, to emphasizes they they are fresh variables.

Step 1 (DNF) Convert ϕ to disjunctive normal form. It then suffices to show that each conjunctive clause can be converted into a disjunction of standard form formulas.

Step 2 (Eliminate Selectors) Given a conjunctive clause ϕ we eliminate all selectors $\mathsf{left}(t)$, $\mathsf{elem}(t)$, and $\mathsf{right}(t)$ by repeatedly applying the following conversions.

$$\begin{aligned}
\phi[\mathsf{left}(t)] &\rightsquigarrow t = \mathsf{Node}(\bar{t}_L, \bar{e}, \bar{t}_R) \wedge \phi[\bar{t}_L] \\
\phi[\mathsf{elem}(t)] &\rightsquigarrow t = \mathsf{Node}(\bar{t}_L, \bar{e}, \bar{t}_R) \wedge \phi[\bar{e}] \\
\phi[\mathsf{right}(t)] &\rightsquigarrow t = \mathsf{Node}(\bar{t}_L, \bar{e}, \bar{t}_R) \wedge \phi[\bar{t}_R]
\end{aligned}$$

This results in a conjunctive clause with no selectors.

Step 3 (Tree Unification) Given a conjunctive clause ϕ with no tree selectors, we now eliminate all equalities between tree terms. Such tree equalities can only appear as top level conjuncts in the clause. Let $\phi = \phi_{eq} \wedge \phi'$ where ϕ_{eq} contains all of the tree equalities. We eliminate the equalities by doing first-order term unification with the modification that constraints between terms in the element theory are left unsolved. If unification fails then we can replace ϕ with $\perp$ since the clause is unsatisfiable. If unification succeeds it returns a most general unifier σ and a conjunction of element theory equalities E. Then this step produces $\phi'\sigma \wedge E$ which is a conjunctive clause with no tree selectors or tree equalities.

Step 4 (Reduce Disequalities) Given a conjunctive clause with no tree selectors or tree equalities, we now reduce all tree term disequalities to disequalities between distinct tree variables. We do this by repeatedly applying the following transformations. In these translations t_v stands for a tree variable. To save space, we treat $t_1 \neq t_2$ and $t_2 \neq t_1$ as equivalent for these translations.

$$\begin{aligned}
\mathsf{Leaf} \neq \mathsf{Leaf} \wedge \phi &\rightsquigarrow \perp \\
\mathsf{Leaf} \neq \mathsf{Node}(t_L, e, t_R) \wedge \phi &\rightsquigarrow \phi \\
\mathsf{Node}(t_L, e, t_R) \neq \mathsf{Node}(t'_L, e', t'_R) \wedge \phi &\rightsquigarrow ((t_L \neq t'_L) \wedge \phi) \vee \\
&\qquad ((e \neq e') \wedge \phi) \vee \\
&\qquad ((t_R \neq t'_R) \wedge \phi) \\
t_v \neq \mathsf{Leaf} \wedge \phi[t_v] &\rightsquigarrow \phi[\mathsf{Node}(\bar{t}_L, \bar{e}, \bar{t}_R)] \\
t_v \neq \mathsf{Node}(t'_L, e', t'_R) \wedge \phi[t_v] &\rightsquigarrow \phi[\mathsf{Leaf}] \vee \\
&\qquad (\bar{t}_L \neq t'_L \wedge \phi[\mathsf{Node}(\bar{t}_L, \bar{e}, \bar{t}_R)]) \vee \\
&\qquad (\bar{e} \neq e' \wedge \phi[\mathsf{Node}(\bar{t}_L, \bar{e}, \bar{t}_R)]) \vee \\
&\qquad (\bar{t}_R \neq t'_R \wedge \phi[\mathsf{Node}(\bar{t}_L, \bar{e}, \bar{t}_R)]) \\
t_v \neq t_v \wedge \phi &\rightsquigarrow \perp
\end{aligned}$$

The termination of these transformations is obvious since the term size of the leading disequality is always getting smaller. Note that some transformations may produce a disjunction of conjunctive clauses. This is not a problem since the initial conjunctive clause that we focus on is already part of a top-level disjunction.

After these transformations we will have a disjunction of conjunctive clauses where each clause has no tree selectors or tree equalities, and all tree disequalities are between distinct tree variables.

Step 5 (Partial Evaluation of α) Given a conjunctive clause ϕ where each clause has no tree selectors or tree equalities, and all tree disequalities are between distinct tree variables, we now partially evaluate α. We do this by repeatedly applying the following transformations.

$$\begin{aligned} \phi[\alpha(\mathsf{Leaf})] &\rightsquigarrow \phi[\mathsf{empty}] \\ \phi[\alpha(\mathsf{Node}(t_L, e, t_R))] &\rightsquigarrow \phi[\mathsf{combine}(\alpha(t_L), e, \alpha(t_R))] \end{aligned}$$

After these transformations we will have a conjunctive clause where there are no tree selectors or tree equalities, all tree disequalities are between distinct tree variables, and α is applied only to tree variables. Thus the clause is in standard form. Therefore the original formula will be transformed into a disjunction of standard form clauses. □

Lemma 9 *Given a formula ϕ in the parametric logic and a GSS catamorphism α, there exists an n such that if $\phi[\alpha_n/\alpha]$ is satisfiable for some n-level approximation α_n of α then ϕ is satisfiable.*

Proof By Lemma 8 we can translate ϕ to the equisatisfiable formula $\phi_1 \vee \cdots \vee \phi_k$ where each ϕ_i is in standard form. By Lemma 7, for each ϕ_i we have an n_i such that if $\phi_i[\alpha_{n_i}/\alpha]$ is satisfiable for some n_i-level approximation α_{n_i} of α then ϕ_i is satisfiable. Let $n = \max\{n_1, \ldots, n_k\}$. By Lemma 3 we have for each ϕ_i that if $\phi_i[\alpha_n/\alpha]$ is satisfiable for some n-level approximation α_n of α then ϕ_i is satisfiable. Thus if $\phi[\alpha_n/\alpha]$ is satisfiable for some n-level approximation α_n of α then ϕ is satisfiable. □

Theorem 1 *Given a formula ϕ in the parametric logic and a GSS catamorphism α, there exists an n such that if ϕ is unsatisfiable then $\phi[\alpha_n/\alpha]$ is unsatisfiable for all n-level approximations α_n of α.*

Proof This is the contrapositive of Lemma 9. □

Theorem 2 *Given an unsatisfiable formula ϕ in the parametric logic, a GSS catamorphism α, and a correct range predicate R_α, Algorithm 2 is terminating.*

Proof Let ϕ be an unsatisfiable formula. By Theorem 1, there exists an n such that $\phi[\alpha_n/\alpha]$ is unsatisfiable for all n-level approximations α_n of α.

In Algorithm 2, α is initially replaced by an uninterpreted function U_α which is unrolled during the algorithm. Consider the n^{th} unrolling together with the range restrictions and call the resulting formula $\phi_n = \phi[U_\alpha/\alpha] \wedge C_n$. Note that C_0 is the initial range constraints on U_α without any unrolling. We will show that ϕ_n is unsatisfiable and thus the algorithm will terminate with UNSAT within the first n unrollings.

Suppose, towards contradiction, that ϕ_n is satisfiable. Let $\mathcal{M}$ be a model of ϕ_n. It is not necessarily true that $\mathcal{M}(U_\alpha)$ is an n-level approximation of α since it may, for example, return any value for inputs to which U_α is not applied in ϕ_n. However, for the values to which U_α is applied in $\phi[U_\alpha/\alpha]$ it acts like an n-level approximation of α due to the constraints imposed by C_n and by the correctness of R_α. Thus, we can construct a new model $\mathcal{M}'$ which differs from $\mathcal{M}$ only in the value of U_α and such that: (1) $\mathcal{M}(U_\alpha)$ and $\mathcal{M}'(U_\alpha)$ agree on all values to which U_α is applied in $\phi[U_\alpha/\alpha]$ and (2) $\mathcal{M}'(U_\alpha)$ is an n-level approximation of α.

By construction, $\mathcal{M}'$ satisfies $\phi[U_\alpha/\alpha]$. Therefore $\mathcal{M}'$ satisfies $\phi[\mathcal{M}'(U_\alpha)/\alpha]$ which contradicts the fact that $\phi[\alpha_n/\alpha]$ is unsatisfiable for all n-level approximations α_n of α. Thus ϕ_n must be unsatisfiable. □

This proof implies that Algorithm 2 terminates for unsatisfiable formulas after a bounded number of unrollings based on ϕ and α. If we compute this bound explicitly, then we can terminate the algorithm early with SAT after the computed number of unrollings. However, if we are interested in complete tree models (in which all variables are assigned concrete values), we still need to keep unrolling until we reach line 6 in Algorithm 2.

The bound on the number of unrollings needed to check unsatisfiability depends on two factors. First, the structure of ϕ gives rise to a number of tree variable disequalities in the conversion of Lemma 8. Second, the unrolling bound depends on the relationship between p and h_p in Definition 5. In Sect. 4, we show that the unrolling bound is linear (Theorem 3) in the number of disequalities for a class of catamorphisms called *monotonic* catamorphisms. Later, in Sect. 5, we show that this bound can be made logarithmic (Lemma 15) in the number of disequalities for a special, but common, form of catamorphisms called *associative* catamorphisms.

In practice, computing the exact bound on the number of unrollings is impractical. The conversion process described in Lemma 8 is focused on correctness rather than efficiency. Instead, it is much more efficient to do only the unrolling of α and leave all other formula manipulation to an underlying SMT solver. Even from this perspective, the bounds we establish on the number of unrollings are still useful to explain why the procedure is so efficient is practice.

4 Monotonic Catamorphisms

We now propose *monotonic* catamorphisms and prove that Algorithm 2 is complete for this class by showing that monotonic catamorphisms satisfy the GSS condition. We also show that this class is a subset of sufficiently surjective catamorphisms, but general enough to include all catamorphisms described in [27,28] 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 [28]. Moreover, a subclass of monotonic catamorphisms, *associative* catamorphisms can be combined while preserving completeness of the formula, addressing another open problem in [28].

4.1 Monotonic Catamorphisms

A catamorphism α is *monotonic* if for every "high enough" tree $t \in \tau$, either $\beta(t) = \infty$ or there exists a tree $t_0 \in \tau$ such that t_0 is smaller than t and $\beta(t_0) < \beta(t)$. Intuitively, this condition ensures that the more number of unrollings we have, the more candidates SMT solvers can assign to tree variables to satisfy all the constraints involving catamorphisms. Eventually, the number of tree candidates will be large enough to satisfy all the constraints involving tree equalities and disequalities among tree terms, leading to the completeness of the procedure.

Definition 8 (*Monotonic catamorphisms*) A catamorphism $\alpha : \tau \rightarrow \mathcal{C}$ is monotonic iff there exists a constant $h_\alpha \in \mathbb{N}$ 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 $h'_\alpha \geq h_\alpha$. In our decision procedure, we assume that if α is monotonic, the range of α can be expressed precisely as a predicate R_α.

Example 5 (*Monotonic catamorphisms*) Catamorphism *DW* in Sect. 3.3.1 is monotonic with $h_\alpha = 1$ and *Multiset* is monotonic with $h_\alpha = 2$. An example of a non-monotonic catamorphism is *Mirror* in [27]:

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

Because $\forall t \in \tau : \beta_{Mirror}(t) = 1$, the catamorphism is not monotonic. We will discuss in detail some examples of monotonic catamorphisms in Sect. 4.3. △

4.2 Some Properties of Monotonic Catamorphisms

Definition 9 ($\mathcal{M}_\beta$) $\mathcal{M}_\beta(h)$ is the minimum of $\beta(t)$ of all trees t of height h:

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

Corollary 2 *$\mathcal{M}_\beta(h)$ is always greater or equal to 1.*

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

Lemma 10 *(*Monotonic Property of $\mathcal{M}_\beta$*) Function $\mathcal{M}_\beta : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ satisfies the following monotonic property:*

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

Proof Consider any $h \in \mathbb{N}$ such that $h \geq h_\alpha$. There are two cases to consider: $\mathcal{M}_\beta(h) = \infty$ and $\mathcal{M}_\beta(h) < \infty$.

Case 1 [$\mathcal{M}_\beta(h) = \infty$]: From Definition 9, every tree t_h of height h has $\beta(t_h) = \infty$ because $\mathcal{M}_\beta(h) = \infty$. Hence, every tree t_{h+1} of height $h + 1$ has $\beta(t_{h+1}) = \infty$ from Definition 8. Thus, $\mathcal{M}_\beta(h + 1) = \infty$.

Case 2 [$\mathcal{M}_\beta(h) < \infty$]: Let t_{h+1} be any tree of height $h + 1$. From Definition 8, 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 9, $\mathcal{M}_\beta(h) < \beta(t_{h+1})$.

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 9. □

Corollary 3 *For any natural number $p > 0$, $\mathcal{M}_\beta(h_\alpha + p) > p$.*

Proof By induction on p based on Lemma 10 and Corollary 2. □

Theorem 3 *Monotonic catamorphisms are GSS (Definition 5).*

Proof Let α be a monotonic catamorphism with h_α as in Definition 8. Let $h_p = h_\alpha + p$. From Corollary 3, $\mathcal{M}_\beta(h_p) > p$. Based on Lemma 10, we can show by induction on h that $\forall h \geq h_p : \mathcal{M}_\beta(h) > p$. By Definition 9, for every tree t with $height(t) \geq h_p$ we have $\beta(t) > p$. Therefore α is GSS. The proof of Theorem 3 shows that monotonic catamorphisms admit a linear bound on the number of unrollings needed to establish unsatisfiability in our procedure. □

4.3 Examples of Monotonic Catamorphisms

This section proves that all sufficiently surjective catamorphisms introduced by Suter et al. [27] are monotonic. These catamorphisms are listed in Table 1. Note that the *Sortedness* catamorphism can be defined to allow or not allow duplicate elements [27]; 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 (see Definition 1) and monotonic catamorphisms.

Lemma 11 *Infinitely surjective abstractions are monotonic.*

Proof According to Definition 1, 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 4 *Set, SizeI, Height, Some, Min, and $Sortedness_{dup}$ are monotonic.*

Proof [27] showed that *Set*, *SizeI*, *Height*, and $Sortedness_{dup}$ are infinitely surjective abstractions. Also, *Some* and *Min* have the properties of infinitely surjective {Leaf}-abstractions. Therefore, the theorem follows from Lemma 11. □

It is more challenging to prove that *Multiset*, *List*, and $Sortedness_{nodup}$ catamorphisms are monotonic since they are not infinitely surjective abstractions. In Sect. 5 we will introduce the notion of associative catamorphisms which includes *Multiset*, $List_{inorder}$, and $Sortedness_{nodup}$, and prove in Theorem 9 that all associative catamorphisms are monotonic. For now, we conclude this section by showing that the all *List* catamorphisms are monotonic.

Theorem 5 *List catamorphisms are monotonic.*

Proof Let α be a *List* catamorphism. For any tree t there are exactly $ns\big(size(t)\big)$ distinct trees that can map to $\alpha(t)$. This is true because: (1) the length of the list $\alpha(t)$ is the number of internal nodes of t, (2) there are exactly $ns\big(size(t)\big)$ tree shapes with the same number of internal nodes as t, and (3) the order of elements in $\alpha(t)$ gives rise to exactly one tree with the same catamorphism value $\alpha(t)$ for each tree shape. Thus, $\beta(t) = ns\big(size(t)\big)$.

Let $h_\alpha = 2$ and let t be an arbitrary tree with $height(t) \geq 2$. Then there exists t_0 such that $height(t_0) = height(t) - 1$ and $size(t_0) < size(t)$. By Property 3, $height(t) \geq h_\alpha = 2$ implies $size(t) \geq 5$. By Lemma 2, $ns\big(size(t_0)\big) < ns\big(size(t)\big)$, which means $\beta(t_0) < \beta(t)$. Therefore α is monotonic. □

4.4 Monotonic Catamorphisms are Sufficiently Surjective

In this section, we demonstrate that monotonic catamorphisms are a strict subset of sufficiently surjective catamorphisms. To this end, we prove that all monotonic catamorphisms are sufficiently surjective, and then provide a witness catamorphism to show that there exists a sufficiently surjective catamorphism that is not monotonic. Although this indicates that monotonic catamorphisms are less general, the constructed catamorphism is somewhat esoteric and we have not found any practical application in which a catamorphism is sufficiently surjective but not monotonic.

To demonstrate that monotonic catamorphisms are sufficiently surjective, we describe a predicate M_h (for each h) that is generic for any monotonic catamorphism. We show that $M_h(\alpha(t))$ holds for any tree with $height(t) \geq h$ in Lemma 12. We then show that if $M_h(c)$ holds then there exists a tree t with $height(t) \geq h$ and $\alpha(t) = c$ in Lemma 13. The proof of sufficient surjectivity follows directly from these lemmas.

Definition 10 (*M_h for arbitrary catamorphism α*) Let α be defined by empty and $\mathsf{combine}$. Let R_α be a predicate which recognizes exactly the range of α. We define M_h recursively as follows:

$$\begin{aligned} M_0(c) &= R_\alpha(c) \\ M_{h+1}(c) &= \exists c_L, e, c_R : c = \mathsf{combine}(c_L, e, c_R) \wedge \\ &\qquad ((M_h(c_L) \wedge R_\alpha(c_R)) \vee (R_\alpha(c_L) \wedge M_h(c_R)) \end{aligned}$$

Note that $M_h(c)$ may have nested existential quantifiers, but these can always be moved out to the top-level.

Lemma 12 *If $height(t) \geq h$ then $M_h(\alpha(t))$ holds.*

Proof Induction on h. In the base case, $h = 0$. Given an arbitrary tree t, we have $M_0(\alpha(t)) = R_\alpha(\alpha(t))$ which holds by the definition of R_α.

In the inductive case assume the formula holds for a fixed h, and let t be an arbitrary tree with $height(t) \geq h + 1$. Let $t = \mathsf{Node}(t_L, e, t_R)$ where either $height(t_L) = height(t) - 1$ or $height(t_R) = height(t) - 1$. Without loss of generality assume that $height(t_L) = height(t) - 1$; the other case is symmetric. Then we have $height(t_L) \geq h$ and so by the inductive hypothesis $M_h(\alpha(t_L))$ holds. We also know $R_\alpha(\alpha(t_R))$ by the definition of R_α. Therefore $M_{h+1}(\alpha(t))$ holds. □

Lemma 13 *If $M_h(c)$ holds, there exists t such that $height(t) \geq h$ and $c = \alpha(t)$.*

Proof Induction on h. In the base case, $h = 0$. We know $M_0(c)$ holds and so $R_\alpha(c)$ holds. By the definition of R_α there exists a tree t such that $c = \alpha(t)$. Trivially, $height(t) \geq 0$.

In the inductive case assume the formula holds for a fixed h, and assume $M_{h+1}(c)$ holds. Expanding the definition of M_{h+1} we know that there exists c_L, e, and c_R such that $c = \mathsf{combine}(c_L, e, c_R)$ and either $M_h(c_L) \wedge R_\alpha(c_R)$ or $R_\alpha(c_L) \wedge M_h(c_R)$ holds. Without loss of generality assume the former; the latter case is symmetric. Given $M_h(c_L)$ holds the inductive hypothesis gives us a tree t_L with $height(t_L) \geq h$ and $c_L = \alpha(t_L)$. Given $R_\alpha(c_R)$ we know there exists t_R with $c_R = \alpha(t_R)$ by the definition of R_α. Let $t = \mathsf{Node}(t_L, e, t_R)$. Since $height(t_L) \geq h$ we have $height(t) \geq h + 1$. Moreover,

$$\alpha(t) = \alpha(\mathsf{Node}(t_L, e, t_R)) = \mathsf{combine}(\alpha(t_L), e, \alpha(t_R)) = \mathsf{combine}(c_L, e, c_R) = c$$

Thus t has the required properties. □

Theorem 6 *Monotonic catamorphisms are sufficiently surjective.*

Proof Let α be a monotonic catamorphism, and let h_p be as given in Theorem 3. We define $S_p = \{shape(t) \mid height(t) < h_p\}$, and show that for each tree t either $shape(t) \in S_p$ or that $M_{h_p}\big(\alpha(t)\big)$ holds for the tree. We partition trees by height. If a tree is shorter in height than h_p then it is captured by S_p. Otherwise, by Lemma 12, $M_{h_p}(\alpha(t))$ holds.

Now we need to satisfy the constraints on S_p and M_{h_p}. First, we note that the set S_p is clearly finite. Second, we show that $M_{h_p}(c)$ implies $|\alpha^{-1}(c)| > p$. If $M_{h_p}(c)$ holds, then by Lemma 13 there exists a tree t of height greater than or equal to h_p such that $c = \alpha(t)$, and then as in Theorem 3, $|\alpha^{-1}(c)| > p$. □

We next demonstrate that there are sufficiently surjective catamorphisms that are not monotonic. Consider the following "almost identity" catamorphism id^* that maps a tree of unit elements (i.e., null) to a pair of its height and another unit-element tree:

$$id^*(t) = \begin{cases} \langle 0, \mathsf{Leaf}\rangle & \text{if } t = \mathsf{Leaf} \\ \langle h, tt\rangle & \text{if } t = \mathsf{Node}(t_L, (), t_R) \end{cases}$$

where

$$h = 1 + \max\{id^*(t_L).\mathit{first}, id^*(t_R).\mathit{first}\}$$

$$tt = \begin{cases} \mathsf{Node}(id^*(t_L).second, (), id^*(t_R).second) & \text{if } h \; is \; odd \\ \mathsf{Node}(id^*(t_L).second, (), \mathsf{Leaf}) & \text{if } h \text{ is even} \end{cases}$$

If the height of the tree is odd, then it returns a tree with the same top-level structure as the input tree. If the height is even, it returns a tree whose left side is the result of the catamorphism and whose right side is Leaf.

Definition 11 Let $st(h)$ be the set of all trees with unit element of height less or equal to h. We can construct $st(h)$ as follows:

$$st(h) = \begin{cases} \mathsf{Leaf} & \text{if } h = 0 \\ \{\mathsf{Node}(l, (), r) \mid l, r \in st(h-1)\} \cup \{\mathsf{Leaf}\} & \text{if } h > 0 \end{cases}$$

Definition 12 Let $count_{st}(h) = |st(h)|$, i.e., the size of the set of all trees of unit element of height less than or equal to h.

Corollary 4

$$count_{st}(h) = \begin{cases} 1 & \textit{if } h = 0 \\ \big(count_{st}(h-1)\big)^2 + 1 & \textit{if } h > 0 \end{cases}$$

Proof Induction on h. The base case is trivial. For the inductive case, we have

$$\begin{aligned} count_{st}(h+1) &= |st(h+1)| \\ &= \big|\{\mathsf{Node}(l, (), r) \mid l, r \in st(h)\}\big| + 1 \\ &= |st(h)|^2 + 1 \\ &= \big(count_{st}(h)\big)^2 + 1 \end{aligned}$$

Thus the equation holds for all h. □

Note that the number of trees grows quickly; e.g., the values for $h \in 0.5$ are $1, 2, 5, 26, 677, 458330$.

Theorem 7 *The id^* catamorphism is sufficiently surjective.*

Proof For a given natural number p, we choose S_p and M_p as follows:

- $S_p = \{t \mid height(t) \leq p + 5\}$, and
- $M_p(\langle h, t\rangle) = h > p + 5 \wedge R_\alpha(h, t)$

where:

$$R_\alpha(h, t) = (height(t) = h \wedge shape(t, h))$$

and:

$$shape(t, k) = \begin{cases} \mathsf{true} & \text{if } t = \mathsf{Leaf} \\ shape(t_L, k-1) \wedge shape(t_R, k-1) & \text{if } k \text{ is odd and } t = \mathsf{Node}(t_L, (), t_R) \\ shape(t_L, k-1) \wedge t_R = \mathsf{Leaf} & \text{if } k \text{ is even and } t = \mathsf{Node}(t_L, (), t_R) \end{cases}$$

The R_α function is the (computable) recognizer of $\langle h, t\rangle$ pairs in the range of α. It is obvious that S_p is finite, and that either $t \in S_p$ or $M_p(\alpha(t))$.

Next, we must show that if $M_p(\langle h, t\rangle)$ then $|\alpha^{-1}(\langle h, t\rangle)| > p$. We note that if h is even, then t has a right-hand subtree that is Leaf, and there are $count_{st}(h-1)$ such subtrees that can be mapped to Leaf via the catamorphism. Similarly, if h is odd, then one of t's children will have a right-hand subtree that is Leaf (note that $h > 5$, thus there exists such a subtree), so there are at least $count_{st}(h-2)$ such subtrees. Finally, we note that for all values $k > 5$, $count_{st}(k-1) > count_{st}(k-2) > k$, so if $h > p+5$, $|\alpha^{-1}(\langle h, t\rangle)| \geq count_{st}(h-2) > h > p+5 > p$. □

Theorem 8 *The id^* catamorphism is not monotonic.*

Proof We will prove this theorem by contradiction. Suppose id^* was monotonic for some height h_α. First, we note that for all trees $t \in \tau$ where element type is unit, $\beta(t)$ is finite, bounded by the finite number of trees of height $height(t)$.

We choose an arbitrary odd height h_0 such that $h_0 \geq h_\alpha$. Let t_{min,h_0} be the tree of height h_0 such that $\beta(t_{min,h_0}) = \mathcal{M}_\beta(h_0)$, which means:

$$\forall t \in \tau, height(t) = h_0 : \beta(t) \geq \beta(t_{min,h_0})$$

We can extend the tree t_{min,h_0} to a new tree $t_{bad,h_0+1} = \mathsf{Node}(t_{min,h_0}, (), \mathsf{Leaf})$, which has (even) height $h_0 + 1$. We can construct a bijection from every tree $t' \in \alpha^{-1}\big(\alpha(t_{bad,h_0+1})\big)$ to a tree in $\alpha^{-1}\big(\alpha(t_{min,h_0})\big)$ by extracting the left subtree of t' (by construction, every right subtree of a tree in $\alpha^{-1}\big(\alpha(t_{bad,h_0+1})\big)$ is Leaf). Therefore, $\beta(t_{bad,h_0+1}) = \beta(t_{min,h_0})$. Due to the construction of t_{min,h_0}, we have:

$$\forall t \in \tau, height(t) = h_0 : \beta(t) \geq \beta(t_{bad,h_0+1})$$

which implies that we cannot find any tree of height h_0 to satisfy the condition for monotonic catamorphisms in Definition 8 for tree t_{bad,h_0+1} of height $h_0 + 1$. □

4.5 A Note on Combining Monotonic Catamorphisms

One might ask if it is possible to have multiple monotonic catamorphisms in the input formula while still maintaining the completeness of the decision procedure. In general, when we combine multiple monotonic catamorphisms, the resulting catamorphism might not be monotonic or even GSS; therefore, the completeness of the decision procedure is not guaranteed. For example, consider the monotonic catamorphisms $List_{preorder}$ and *Sortedness* (their definitions are in Table 1). For any $h \in \mathbb{N}^+$ we can construct a right skewed tree tree t of height h as follows:

$$\mathsf{Node}\Big(\mathsf{Leaf}, 1, \mathsf{Node}\big(\mathsf{Leaf}, 2, \mathsf{Node}(\mathsf{Leaf}, 3, \ldots \mathsf{Node}(\mathsf{Leaf}, h, \mathsf{Leaf}))\big)\Big)$$

The values of $List_{preorder}(t)$ and $Sortedness(t)$ are as follows:

- $List_{preorder}(t) = (1\ 2\ \ldots\ h)$ – i.e., the element values are $1, 2, \ldots, h$.
- $Sortedness(t) = (1, h, \mathsf{true})$ – i.e., min $= 1$, max $= h$, and t is sorted.

Let α be the combination of these catamorphisms and let β be defined as in Definition 4. We have $\beta(t) = 1$ as t is the only tree that can map to the values of $List_{preorder}(t)$ and $Sortedness(t)$ above. Thus, α cannot be monotonic or even GSS.

Although the combinability is not a feature that monotonic catamorphisms can guarantee, Sect. 5 presents a subclass of monotonic catamorphisms, called associative catamorphisms, that supports the combination of catamorphisms in our procedure.

5 Associative Catamorphisms

We have presented an unrolling-based decision procedure that is guaranteed to be both sound and complete with GSS catamorphisms (and therefore also with sufficiently surjective and monotonic catamorphisms). When it comes to catamorphisms, there are many interesting open problems, for example: when is it possible to combine catamorphisms in a complete way, or how computationally expensive is it to solve catamorphism problems? This section attempts to characterize a useful class of "combinable" GSS catamorphisms that maintain completeness under composition.

We name this class *associative* catamorphisms due to the associative properties of the operator used in the catamorphisms. Associative catamorphisms have some very powerful important properties: they are detectable,[5] combinable, and impose an exponentially small upper bound on the number of unrollings. Many catamorphisms presented so far are in fact associative.

Definition 13 (*Associative catamorphism*) A catamorphism $\alpha : \tau \to \mathcal{C}$ is associative if

$$\alpha(\mathsf{Node}(t_L, e, t_R)) = \alpha(t_L) \oplus \delta(e) \oplus \alpha(t_R)$$

where $\oplus : (\mathcal{C}, \mathcal{C}) \to \mathcal{C}$ is an associative binary operator. Here, $\delta : \mathcal{E} \to \mathcal{C}$ is a function that maps[6] an element value in $\mathcal{E}$ to a corresponding value in $\mathcal{C}$.

Associative catamorphisms are detectable. A catamorphism, written in the format in Definition 13, is associative if the $\oplus$ operator is associative. This condition can be easily proved by SMT solvers [1,6] or theorem provers such as ACL2 [10]. Also, because of the associative operator $\oplus$, the value of an associative catamorphism for a tree is independent of the shape of the tree.

We present associative catamorphisms syntactically in Definition 13. They can also be described semantically by requiring that α is preserved by tree rotations:

$$\alpha\big(\mathsf{Node}(t_1, e_1, \mathsf{Node}(t_2, e_2, t_3))\big) = \alpha\big(\mathsf{Node}(\mathsf{Node}(t_1, e_1, t_2), e_2, t_3))\big)$$

[5] *detectable* in this context means that it is possible to determine whether or not a catamorphism is an associative catamorphism using an SMT solver.

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

This is still detectable by checking the satisfiability the corresponding query:

$$R_\alpha(c_1) \wedge R_\alpha(c_2) \wedge R_\alpha(c_3) \wedge$$
$$\mathsf{combine}(c_1, e_1, \mathsf{combine}(c_2, e_2, c_3)) \neq \mathsf{combine}(\mathsf{combine}(c_1, e_1, c_2), e_2, c_3)$$

This semantic definition of associativity is broader than the purely syntactic one (because it does not depend on the associative binary operator $\oplus$), but is less intuitive. We work with the syntactic definition in this section, but the main results over to the semantic definition as well.

Corollary 5 *(*Values of associative catamorphisms*) The value of $\alpha(t)$, where α is an associative catamorphism, only depends on the ordering and values of elements in t. In particular, $\alpha(t)$ does not depend on the shape of the tree:*

$$\alpha(t) = \alpha(\mathit{Leaf}) \oplus \delta(e_1) \oplus \alpha(\mathit{Leaf}) \oplus \delta(e_2) \oplus \cdots \oplus \alpha(\mathit{Leaf}) \oplus \alpha(e_n) \oplus \alpha(\mathit{Leaf})$$

where $e_1, e_2, \ldots, e_n$ is the in-order listing of the elements of the nodes of t. When $t = \mathit{Leaf}$, we simply have $\alpha(t) = \alpha(\mathit{Leaf})$.

Proof Straightforward induction on the structure of t. □

Example 6 (*Associative catamorphisms*) In Table 1, *Height*, *Some*, *List*$_{preorder}$ and *List*$_{postorder}$ are not associative because their values depend on the shape of the tree.

The other catamorphisms in Table 1 are associative, including *Set*, *Multiset*, *SizeI*, *List*$_{inorder}$, *Min*, and *Sortedness* (both with and without duplicates). The *DW* catamorphism in Sect. 3.3.1 is also associative, where the operator $\oplus$ is $+$ and the mapping function is $\delta(e) = (\mathsf{ite}\ \mathit{dirty}(e)\ 1\ 0)$. For *Multiset*, the two components are $\uplus$ and $\delta(e) = \{e\}$.

Furthermore, we can define associative catamorphisms based on associative operators such as $+$, $\cap$, max, $\vee$, $\wedge$, etc. We can also use a user-defined function as the operator in an associative catamorphism. For example, the catamorphism *Leftmost* which finds the leftmost element value in a tree is associative where $\delta(e) = \mathsf{Some}(e)$, $\alpha(\mathsf{Leaf}) = \mathsf{None}$, and $\oplus$ is defined by

$$\begin{aligned}
\mathsf{None} \oplus \mathsf{None} &= \mathsf{None} \\
\mathsf{Some}(e) \oplus \mathsf{None} &= \mathsf{Some}(e) \\
\mathsf{None} \oplus \mathsf{Some}(e) &= \mathsf{Some}(e) \\
\mathsf{Some}(e_L) \oplus \mathsf{Some}(e_R) &= \mathsf{Some}(e_L)
\end{aligned}$$

The symmetrically defined *Rightmost* catamorphism is also associative.

We do not require that $\alpha(\mathsf{Leaf})$ is an identity for the operator $\oplus$, though it often is in practice. An example where it is not is the *Size* catamorphism which computes the total size of a tree (rather than just the number of internal nodes computed by *SizeI*). In this case we have $\delta(e) = 1$, $\alpha(\mathsf{Leaf}) = 1$, and operator $\oplus$ is $+$. △

5.1 The Monotonicity of Associative Catamorphisms

This section shows that associative catamorphisms are monotonic, and therefore sufficiently surjective and GSS.

Theorem 9 *Associative catamorphisms are monotonic.*

Proof Let $h_\alpha = 2$. Let t be a tree with $height(t) \geq 2$. If $\beta(t) = \infty$ then we are done. Otherwise, suppose $\beta(t) < \infty$. We want to show that there exists a tree t_0 such that $height(t_0) = height(t) - 1$ and $\beta(t_0) < \beta(t)$. Since $height(t) \geq 2$ we can write $t = \mathsf{Node}(t_L, e, t_R)$ where either $height(t_L) = height(t) - 1$ or $height(t_R) = height(t) - 1$. Without loss of generality assume $height(t_R) = height(t) - 1$; the argument is symmetric for the other case. We will show that $\beta(t_R) < \beta(t)$ so that t_R satisfies the conditions required for t_0.

There are $\beta(t_R)$ trees that map to $\alpha(t_R)$. For each such tree t'_R the tree $t' = \mathsf{Node}(t_L, e, t'_R)$ maps to $\alpha(t)$. The distinctness of each t'_R ensures that each t' is also distinct. Now all we need to find is one additional tree which maps to $\alpha(t)$ but is not one of the t' above. Since $height(t) \geq 2$, we know $height(t_R) \geq 1$ and can write $t_R = \mathsf{Node}(t_{RL}, e_R, t_{RR})$. Consider the rotated tree $t_{new} = \mathsf{Node}(\mathsf{Node}(t_L, e, t_{RL}), e_R, t_{RR})$. This tree is distinct from the t' trees above since the left branches are distinct: $\mathsf{Node}(t_L, e, t_{LR}) \neq t_L$. Moreover, since $\oplus$ is associative we have

$$\begin{aligned}
\alpha(t_{new}) &= \big(\alpha(t_L) \oplus \delta(e) \oplus \alpha(t_{RL})\big) \oplus \delta(e_R) \oplus \alpha(t_{RR}) \\
&= \alpha(t_L) \oplus \delta(e) \oplus \big(\alpha(t_{RL}) \oplus \delta(e_R) \oplus \alpha(t_{RR})\big) \\
&= \alpha(t_L) \oplus \delta(e) \oplus \alpha(t_R) \\
&= \alpha(t)
\end{aligned}$$

Thus $\beta(t_R) < \beta(t)$, and therefore α is monotonic. □

Since associative catamorphisms are monotonic, they are also GSS by Theorem 3, meaning that associative catamorphisms can be used in our unrolling decision procedure.

Remark 1 Thus far we have used binary trees as our inductive datatype τ. Our results so far have been generic for any inductive datatypes, but Theorem 9 is not. In particular, the theorem does not hold when τ is the list datatype. Over the list datatype the catamorphism $Multiset$ is associative, but not monotonic since $\beta_{Multiset}(\{0, 0, \ldots, 0\}) = 1$. The proof of Theorem 9 fails since there is no rotate operation on lists as there is on trees. Similarly, the unrolling bounds in the next section do not necessarily hold for list-like datatypes.

5.2 Exponentially Small Upper Bound on the Number of Unrollings

In the proof of Theorem 3, we showed that monotonic catamorphisms admit a linear bound on the number of unrollings needed to establish unsatisfiability in our procedure. However, even for monotonic catamorphisms, the number of unrollings may be large for a large input formula with many tree disequalities, leading to a high complexity for the algorithm. This section shows that for associative catamorphisms, the bound can be made *exponentially small*.

Lemma 14 *If α is an associative catamorphism then $\forall t \in \tau : \beta(t) \geq ns\big(size(t)\big)$*

Proof Let $t \in \tau$ be an arbitrary tree. By the definition of ns there are $ns\big(size(t)\big)$ tree shapes with the same size as t and therefore the same number of internal nodes. Each tree shape gives rise to a tree which has the same in-order listing of elements as t. Since the shapes are distinct, the trees will be distinct. By Corollary 5, the value of $\alpha(t)$ depends only on the in-order listing of the element values. Thus all such trees map to the same collection value. Therefore, $\beta(t) \geq ns\big(size(t)\big)$. □

Lemma 15 *If α is associative 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 14. Hence, $\beta(t_h) \geq ns(2h+1)$ from Property 3 and Lemma 2. From Lemma 1, $\beta(t_h) \geq \mathbb{C}_h$. Therefore, $\mathcal{M}_\beta(h) \geq \mathbb{C}_h$ by Definition 9. □

Let $h_p = \min\{h \mid \mathbb{C}_h > p\}$ so that $\mathbb{C}_{h_p} > p$. From Lemma 15, $\mathcal{M}_\beta(h_p) \geq \mathbb{C}_{h_p} > p$. Thus h_p satisfies the GSS condition for α. Moreover, 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 $p = 10,000$, we can choose $h_p = 10$ since $\mathbb{C}_{10} = 16,796 > 10,000$. Similarly, when $p = 50,000$, we can choose $h_p = 11$ since $\mathbb{C}_{11} = 58,786$.

5.3 Combining Associative Catamorphisms

Let $\alpha_1, \ldots, \alpha_m$ be m associative catamorphisms where α_i is given by the collection domain $\mathcal{C}_i$, the operator $\oplus_i$, and the function δ_i. We construct the catamorphism α componentwise from the α_i as follows:

- $\mathcal{C}$ is the domain of m-tuples, where the i^{th} element of each tuple is in $\mathcal{C}_i$.
- $\oplus : (\mathcal{C}, \mathcal{C}) \rightarrow \mathcal{C}$ is defined as

$$\langle x_1, \ldots, x_m \rangle \oplus \langle y_1, \ldots, y_m \rangle = \langle x_1 \oplus_1 y_1, \ldots, x_m \oplus_m y_m \rangle$$

- $\delta : \mathcal{E} \rightarrow \mathcal{C}$ is defined as $\delta(e) = \big\langle \delta_1(e), \ldots, \delta_m(e) \big\rangle$
- α is defined as in Definition 13.

Example 7 (*Combine associative catamorphisms*) Consider *Set* and *SizeI* catamorphisms in Table 1, which are associative. When we combine the two associative catamorphisms (assuming *Set* is used before *SizeI*), we get a new catamorphism *SetSizeI* that maps a tree to a pair of values: the former is the set of all the elements in the tree and the latter is the number of internal nodes in the tree. For example, if we apply *SetSizeI* to the tree in Fig. 3, we get $\langle \{1, 2\}, 2 \rangle$. △

Remark 2 Every catamorphism obtained from the combination of associative catamorphisms is also associative.

Proof The associativity of the componentwise $\oplus$ follows directly from the associativity of the $\oplus_i$ operators. □

Note that while it is easy to combine associative catamorphisms, it might be challenging to compute the range predicate R_α of the combination of those associative catamorphisms. For example, consider *Min* and *Sum*, two simple surjective associative catamorphisms whose ranges are trivially equal to their codomains. The range of their combination is:

$$\begin{aligned} & Min(t) = \mathsf{None} \ \wedge \ Sum(t) = 0 \\ \vee \ & Min(t) < 0 \\ \vee \ & Min(t) \geq 0 \ \wedge \ \big(Sum(t) = Min(t) \vee Sum(t) \geq 2 \times Min(t)\big) \end{aligned}$$

As with individual catamorphisms, it is the user's responsibility to create an appropriate R_α predicate.

6 The Relationship Between Catamorphisms

We have summarized two types of catamorphisms previously proposed by Suter et al. [27], namely infinitely surjective and sufficiently surjective catamorphisms in Definitions 1 and 3,

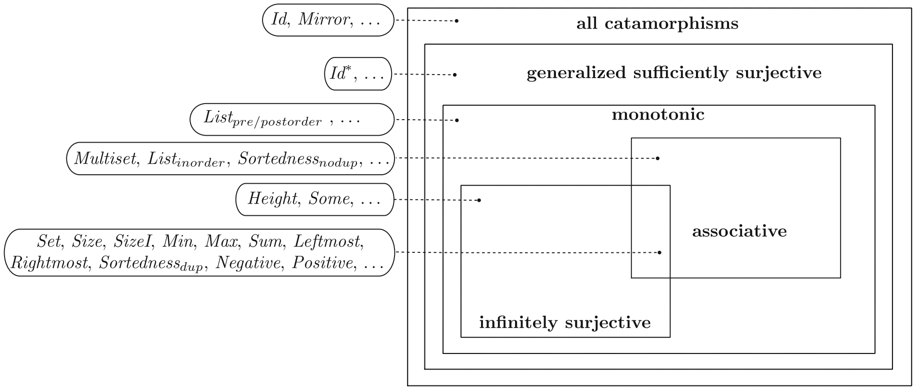

Id, Mirror, ...
Id*, ...
List$_{pre/postorder}$, ...
Multiset, List$_{inorder}$, Sortedness$_{nodup}$, ...
Height, Some, ...
Set, Size, SizeI, Min, Max, Sum, Leftmost, Rightmost, Sortedness$_{dup}$, Negative, Positive, ...
all catamorphisms
generalized sufficiently surjective
monotonic
associative
infinitely surjective

Fig. 5 Relationship between different types of catamorphisms

respectively. We have also proposed three different classes of catamorphisms: GSS (Definition 5), monotonic (Definition 8), and associative (Definition 13). This section discusses how these classes of catamorphisms are related to each other and how they fit into the big picture, depicted in Fig. 5 with some catamorphism examples.

Between sufficiently surjective and GSS catamorphisms: We have shown that all sufficiently surjective catamorphisms are GSS (Corollary 1). We have not demonstrated that the inclusion is strict as this would require reasoning about what is and is not representable in the definition of M_p. Frankly, we believe there is no need for sufficient surjectivity given the notion of GSS.

Between monotonic and sufficiently surjective catamorphisms: All monotonic catamorphisms are sufficiently surjective (Theorem 6). This shows that although the definition of monotonic catamorphisms from this paper and the idea of sufficiently surjective catamorphisms from Suter et al. [27] may look different from each other, they are actually closely related. Moreover, monotonic catamorphisms provide linear bounds in our decision procedure.

Between infinitely surjective and monotonic catamorphisms: All infinitely surjective catamorphisms are monotonic (Lemma 11). Thus, infinitely surjective catamorphisms are not just a sub-class of sufficiently surjective catamorphisms as presented in Suter et al. [27], they are also a sub-class of monotonic catamorphisms.

Between associative and monotonic catamorphisms: All associative catamorphisms are monotonic (Theorem 9). Moreover, associative catamorphisms provide exponentially small bounds in our decision procedure.

Between infinitely surjective and associative catamorphisms: The set of infinitely surjective catamorphisms and that of associative catamorphisms are intersecting, as shown in Fig. 5 with some catamorphism examples.

7 Implementation and Experimental Results

We introduce RADA[7], an open source tool to reason about algebraic data types with abstractions that is conformant with the SMT-LIB 2.0 format [3]. The algorithms behind RADA were

[7] http://crisys.cs.umn.edu/rada/.

$\langle command \rangle_1$	::=	(**declare-datatypes** () ($\langle datatype \rangle^+$))
$\langle datatype \rangle$	::=	($\langle symbol \rangle$ $\langle datatype\_branch \rangle^+$)
$\langle datatype\_branch \rangle$	::=	($\langle symbol \rangle$ $\langle datatype\_branch\_para \rangle^*$)
$\langle datatype\_branch\_para \rangle$	::=	($\langle symbol \rangle$ $\langle sort \rangle$)
$\langle command \rangle_2$	::=	(**define-catamorphism** $\langle catamorphism \rangle$)
$\langle catamorphism \rangle$	::=	($\langle symbol \rangle$ ($\langle sort \rangle$) $\langle sort \rangle$ $\langle term \rangle$ [**:post-cond** $\langle term \rangle$])
$\langle selector\_application \rangle$	::=	$\langle symbol \rangle$ $\langle symbol \rangle$
$\langle tester\_application \rangle$	::=	**is-**$\langle symbol \rangle$ $\langle symbol \rangle$

Fig. 6 RADA grammar

described in previous sections. It can function as a back-end for reasoning about recursive programs that manipulate algebraic data types. RADA was designed to be host-language and solver-independent, and it can use either CVC4 or Z3 as its underlying SMT solver. RADA has also been successfully integrated into the Guardol system [8], replacing our implementation of the Suter-Dotta-Kuncak decision procedure [27] on top of OpenSMT [5]. Experiments show that our tool is reliable, fast, and works seamlessly across multiple platforms, including Windows, Unix, and Mac OS. We have used RADA in the Guardol project for reasoning about functional implementations of complex data structures and to reason about *guard applications* that determine whether XML messages should be allowed to cross network security domains. How RADA was integrated into Guardol is presented in [8].

The overall architecture of RADA follows closely the decision procedure described in Sect. 3.3. We use CVC4 [1] and Z3 [6] as the underlying SMT solvers in RADA due to their powerful abilities to reason about recursive data types. The grammar of RADA in Fig. 6 is based on the SMT-LIB 2.0 [3] format with some new syntax for selectors, testers, data type declarations, and catamorphism declarations. Note that although selectors, testers, and data type declarations are not defined in SMT-LIB 2.0, all of them are currently supported by both CVC4 and Z3; thus, only catamorphism declarations are not understood by these solvers. **:post-cond**, which is used to declare R_α, is optional since we do not need to specify R_α when α is surjective (e.g., SumTree in Example 8).

Example 8 (*RADA syntax*) Let us consider an example to illustrate the syntax used in RADA. Suppose we have a data type RealTree that contains real numbers:

```
(declare-datatypes ()
  ((RealTree
     (Leaf)
     (Node (left RealTree) (elem Real) (right RealTree)))))
```

Next, a RealTree can be abstracted into a real number representing the sum of all elements in the tree by catamorphism SumTree, which can be defined as follows:

```
(define-catamorphism SumTree ((t RealTree)) Real
  (ite (is-Leaf t)
       0.0
       (+ (SumTree (left t))
          (elem t)
          (SumTree (right t)))))
```

where is-Leaf is a tester that checks if a RealTree is a leaf node and left t, elem t, and right t are selectors that select the corresponding data type branches in a RealTree named t. Given the definitions of data type RealTree and catamorphism SumTree, one may want to check some properties of a RealTree, for example:

```
(declare-fun t1 () RealTree)
(declare-fun t2 () RealTree)
(declare-fun t3 () RealTree)
(assert (= t1 (Node t2 5.0 t3)))
(assert (= (SumTree t1) 5.0))
(check-sat)
```

As expected, RADA returns *SAT* for the above example. △

Since RADA was first published [22], we have been working on improving the performance of the tool. Compared with the version in [22], the current version of RADA is multiple times faster thanks to the following implementation techniques.
Technique 1: *Solve proof obligations in parallel.* Multiple proof obligations can be written in RADA within push-pop pairs (as in SMT-LIB 2.0[3]). For instance,

```
(push) Obligation_A (pop)       (push) Obligation_B (pop)
```

We preprocess the original SMT file. If the file has parallelizable obligations, we split it into multiple separate files (each file has only one obligation). RADA discharges proof obligations in parallel. It supports a thread pool of a configurable size of proof obligations. All the proof obligations in the pool are solved concurrently and all the remaining proof obligations are put in a waiting list. As soon as a proof obligation in the thread pool is discharged, the pool adds a new proof obligation from the waiting list to the pool (if any).
Technique 2: *Reuse the definitions of catamorphism bodies when unrolling.* In general, when we have a catamorphism application, e.g., SumTree (Node t2 5.0 t3) with the SumTree catamorphism and tree terms t2 and t3 in Example 8, the catamorphism application is assigned to the corresponding definition of the catamorphism body with the given parameter. In this case, it will be as follows:

```
(assert (= (SumTree (Node t2 5.0 t3))
           (ite (is-Leaf (Node t2 5.0 t3))
                0.0
                (+ (SumTree (left (Node t2 5.0 t3)))
                   (elem (Node t2 5.0 t3))
                   (SumTree (right (Node t2 5.0 t3)))))))
```

However, as the unrolling procedure progresses, the tree parameters will keep getting bigger (because they are unrolled) and the catamorphism applications will appear frequently in the SMT query. This leads to the following issue: the definitions of catamorphism bodies appear again and again. To address this issue, it is desirable to be able to reuse the definitions of catamorphism bodies. To do that, RADA creates a user-defined function for each catamorphism body, for example with the SumTree catamorphism:

```
    (define-fun SumTree_GeneratedCatDefineFun ((t RealTree))
Real
      (ite (is-Leaf t)
          0.0
          (+ (SumTree (left t))
             (elem t)
             (SumTree (right t)))))
```

and whenever we want to calculate a catamorphism application, we just need to call the corresponding user-defined function we just created:

```
(assert (= (SumTree (Node t2 5.0 t3))
           (SumTree_GeneratedCatDefineFun
           (Node t2 5.0 t3))))
```

We can also parameterize the above equality assertion by creating another user-defined function for it as follows:

```
(define-fun SumTree_GeneratedUnrollDefineFun
  ((t RealTree)) Bool
  (= (SumTree t) (SumTree_GeneratedCatDefineFun t)))
```

and now all what we need to do is use the short, newly created function:

```
   (assert (SumTree_GeneratedUnrollDefineFun
(Node t2 5.0 t3)))
```

In other words, when we need to unroll a catamorphism application, we just need to call the corresponding function with suitable parameters instead of expanding tree terms repeatedly. *Technique 3*: *Solve each proof obligation incrementally.* We observe that in our decision procedure, we need two calls to an SMT solver (i.e., two *decide* calls in Algorithm 2) at each unrolling step to determine if we have found a trustworthy *SAT*/*UNSAT* answer. There are two issues if the calls to the SMT solver are handled independently: (1) we would not take advantage of what the SMT solver instance has learned from the previous SMT query, and (2) we would pay a performance price for initializing and closing the SMT solver instance each time.

RADA addresses those issues as follows. First, RADA solves each proof obligation incrementally, i.e., the information collected from the SMT queries is reused over time. Second, there is only one instance of the SMT solver for each proof obligation we want to solve; in other words, RADA creates an instance of the SMT solver when we start solving the proof obligation and only closes the SMT solver instance after the obligation has been completely discharged. We show below an example of incremental solving with RADA.

Example 9 (*Example of incremental solving with RADA*) Let us present step by step how RADA solves the RealTree example in Example 8. First, RADA sends to an SMT solver the declaration of the RealTree data type, which is the declare-datatypes statement in Example 8.

Next, RADA declares an uninterpreted function called SumTree, which represents the SumTree catamorphism in Example 8. Note that the SMT solver views SumTree as an uninterpreted function: the solver does not know what content of the function is; it only knows that SumTree takes as input a RealTree and returns a Real value as the output.

```
(declare-fun SumTree (RealTree) Real)
```

RADA then feeds to the SMT solver the original problem we want to solve:

```
(declare-fun t1 () RealTree)
(declare-fun t2 () RealTree)
(declare-fun t3 () RealTree)
(assert (= t1 (Node t2 5.0 t3)))
(assert (= (SumTree t1) 5.0))
```

Additionally, RADA creates two user-defined functions as previously discussed as a preprocessing step:

```
(define-fun SumTree_GeneratedCatDefineFun ((t RealTree))
  Real
  (ite (is-Leaf t)
       0.0
       (+ (SumTree (left t))
          (elem t)
          (SumTree (right t)))))

(define-fun SumTree_GeneratedUnrollDefineFun
   ((t RealTree)) Bool
   (= (SumTree t) (SumTree_GeneratedCatDefineFun t)))
```

RADA then tries to check the satisfiability of the problem without unrolling any catamorphism application:

```
(check-sat)
```

The SMT solver will return *SAT*. In this case, we are using the uninterpreted function; hence, the *SAT* result is untrustworthy. Therefore, we have to continue the process by unrolling the catamorphism application SumTree t1. We also add a push statement and then add the control conditions to the problems before checking its satisfiability. Note that the push statement is used here to mark the position in which the control conditions are located, so that we can remove the control conditions later by a corresponding pop statement.

```
(assert (SumTree_GeneratedUnrollDefineFun t1))     [Unrolling step]
(push)
(assert (is-Leaf t1))    [Assertions for control conditions]
(check-sat)
```

The SMT solver will return *UNSAT*, which means using the control conditions might be too restrictive and we have to remove the control conditions by using a pop statement and try again:

```
(pop)                          [Remove the control conditions]
(check-sat)
```

However, when checking the satisfiability without control conditions, we get *SAT* from the SMT solver again. Based on our decision procedure in Algorithm 2, we have to try another unrolling step; thus, RADA sends the following to the solver:

```
(assert (SumTree_GeneratedUnrollDefineFun (left t1)))   [Unrolling step]
(assert (SumTree_GeneratedUnrollDefineFun (right t1)))
(push)
(assert (is-Leaf (left t1)))   [Assertions for control conditions]
(assert (is-Leaf (right t1)))
(check-sat)
```

This time the SMT solver still returns *SAT*. However, we are using control conditions and getting *SAT*, which means the *SAT* result is trustworthy. Thus, RADA returns *SAT* as the answer to the original problem. This example has shown how we can use only one SMT solver instance to solve the problem incrementally. △

Table 3 Experimental results

Type	Benchmark	Result	Time (s)
Single associative catamorphisms	sumtree(01|02|03|05|06|07|10|11|13)	sat	0.025–0.083
	sumtree(04|08|09|12|14)	unsat	0.033–0.044
Combination of associative catamorphisms	min_max(01|02)	unsat	0.057–0.738
	min_max_sum01	unsat	1.165
	min_max_sum(02|03|04)	sat	0.149 – 0.373
	min_size_sum01	unsat	0.873
	min_size_sum02	sat	0.114
	negative_positive(01|02)	unsat	0.038 – 0.136
Guardol	Email_Guard_Correct_All	17 unsats	≈ 0.009/obligation
	RBTree.Black_Property	12 unsats	≈ 2.142/obligation
	RBTree.Red_Property	12 unsats	≈ 0.163/obligation
	array_checksum.SumListAdd	2 unsats	≈ 0.028/obligation
	array_checksum.SumListAdd_Alt	13 unsats	≈ 0.012/obligation

7.1 Experimental Results

We have implemented our decision procedure in RADA and evaluated the tool with a collection of benchmark guard examples listed in Table 3. All of the benchmark examples were automatically verified by RADA in a short amount of time.

Experiments on associative catamorphisms. The first set of benchmarks consists of examples related to *Sum*, an associative catamorphism that computes the sum of all element values in a tree. The second set contains combinations of associative 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 cannot be bigger than the maximum value in the tree. The definitions of the associative catamorphisms used in the benchmarks are as follows: *Sum* is defined as in Example 8, *Max* is defined in a similar way to *Min* in Table 1, and *Negative* and *Positive* are defined as in [21].

Experiments on Guardol benchmarks. In addition to associative catamorphisms, we have also evaluated RADA on some examples in the last set of benchmark containing general catamorphisms that have been automatically generated from the Guardol verification system [8]. They consist of verification conditions to prove some interesting properties of red black trees and the checksums of trees of arrays. These examples are complex: each of them contains multiple verification conditions, some data types, and a number of mutually related parameterized catamorphisms. For example, the Email Guard benchmark has 8 mutually recursive data types, 6 catamorphisms, and 17 complex obligations.

All benchmarks were run on a Ubuntu machine (Intel Core I5, 2.8 GHz, 4GB RAM). All running time was measured when Z3 was used as the underlying SMT solver.

8 Conclusion and Discussion

In this paper, we have proposed an unrolling-based decision procedure for algebraic data types with a new idea of generalized sufficiently surjective catamorphisms. We have also presented a class of generalized sufficiently surjective catamorphisms called monotonic catamorphisms and have shown that all sufficiently surjective catamorphisms known in the literature to date [27] are also monotonic. We have established a linear upper bound on the number of unrollings needed to establish unsatisfiability with monotonic catamorphisms. Furthermore, we have pointed out a sub-class of monotonic catamorphisms, namely associative catamorphisms, which are proved to be detectable, combinable, and guarantee an exponentially small unrolling bound 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.

We have also presented RADA, an open source tool to reason about inductive data types. RADA fully supports all types of catamorphisms discussed in this paper as well as other general user-defined abstraction functions. The tool was designed to be simple, efficient, portable, and easy to use. The successful uses of RADA in the Guardol project [8] demonstrate that RADA not only could serve as a good research prototype tool but also holds great promise for being used in other real world applications.

Compliance with Ethical Standards

Conflict of Interest We declare that we have no conflict of interest.

Research Involving Human Participants and/or Animals We declare that this research does not involve human participants and/or animals.

Informed Consent We declare that no informed consent is needed since this research does not involve human participants.

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, pp. 171–177 (2011)
2. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for satisfiability in the theory of recursive data types. Electron. Notes Theor. Comput. Sci. **174**(8), 23–37 (2007)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: SMT (2010)
4. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: verification by translation to recursive functions. In: SCALA, pp. 1:1–1:10 (2013)
5. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: TACAS, pp. 150–153 (2010)
6. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS, pp. 337–340 (2008)
7. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press, Cambridge (2009)
8. Hardin, D., Slind, K., Whalen, M., Pham, T.H.: The guardol language and verification system. In: TACAS, pp. 18–32 (2012)
9. Jacobs, S., Kuncak, V.: Towards Complete Reasoning about Axiomatic Specifications. In: VMCAI, pp. 278–293 (2011)
10. Kaufmann, M., Manolios, P., Moore, J.: Computer-Aided Reasoning: ACL2 Case Studies. Springer, Heidelberg (2000)
11. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: PLDI, pp. 222–233 (2011)
12. Koshy, T.: Catalan Numbers with Applications. Oxford University Press, Oxford (2009)
13. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR, pp. 348–370 (2010)
14. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL, pp. 611–622 (2011)

15. Madhusudan, P., Qiu, X., Stefanescu, A.: Recursive proofs for inductive tree data-structures. In: POPL, pp. 123–136 (2012)
16. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Berlin (2002)
17. Oppen, D.C.: Reasoning About Recursively Defined Data Structures. J. ACM **27**(3), 403–411 (1980)
18. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: CADE, pp. 748–752 (1992)
19. Pham, T.H.: Verification of recursive data types using abstractions. Ph.D. thesis, University of Minnesota (2014)
20. Pham, T.H., Whalen, M.: An improved unrolling-based decision procedure for algebraic data types. In: VSTTE (2013)
21. Pham, T.H., Whalen, M.W.: Parameterized abstractions for reasoning about algebraic data types. In: CFV (2013). Available at http://www-users.cs.umn.edu/~hung/papers/cfv13
22. Pham, T.H., Whalen, M.W.: RADA: A tool for reasoning about algebraic data types with abstractions. In: ESEC/SIGSOFT FSE, pp. 611–614 (2013)
23. Reynolds, A., Kuncak, V., Induction for SMT Solvers. In: VMCAI, (2015)
24. Sato, R., Unno, H., Kobayashi, N.: Towards a Scalable Software Model Checker for Higher-Order Programs. In: PEPM, pp. 53–62 (2013)
25. Sofronie-Stokkermans, V.: Locality results for certain extensions of theories with bridging functions. In: CADE, pp. 67–83 (2009)
26. Stanley, R.P.: Enumerative Combinatorics, vol. 2. Cambridge University Press, Cambridge (2001)
27. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL, pp. 199–210 (2010)
28. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: SAS (2011)
29. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: PLDI, pp. 349–361 (2008)
30. Zee, K., Kuncak, V., Rinard, M.C.: An integrated proof language for imperative programs. In: PLDI, pp. 338–351 (2009)
31. Zhang, T., Sipma, H.B., Manna, Z.: Decision procedures for term algebras with integer constraints. In: Information and Computation, pp. 152–167 (2004)