Parameterized Abstractions for Reasoning about Algebraic Data Types

Tuan-Hung Pham
University of Minnesota, USA

Michael W. Whalen
University of Minnesota, USA

***Abstract*—Reasoning about algebraic data types is an important problem for a variety of proof tasks. Recently, decision procedures have been proposed for algebraic data types that create suitable abstractions of values in the types. A class of abstractions created from *catamorphism* functions has been shown to be theoretically applicable to a wide variety of reasoning tasks as well as efficient in practice. However, in previous work, the decidability of catamorphism functions involving parameters in addition to the data type argument has not been studied.**

In this paper, we generalize certain kinds of catamorphism functions to support additional parameters. This extension, called *parameterized associative-commutative catamorphisms* subsumes the associative-commutative class from earlier work, widens the set of functions that are known to be decidable, and makes several practically important functions (such as *forall*, *exists*, and *member*) over elements of algebraic data types straightforward to express.

I. Introduction

Reasoning about algebraic data types is important as they are a natural representation for recursively-defined data. In addition, they are a foundational concept for functional programming languages and provide a natural representation for everything from program syntax to XML messages. One prominent way to reason about algebraic data is to abstract the data into values in a decidable theory, as described in the work by Pham and Whalen [9], Suter et al. [12], [13], and Madhusudan et al. [8]. To support complete reasoning about algebraic types, the abstractions usually need to meet some requirements, such as the monotonicity [9] or the sufficient surjectivity [12], [13].

Recently, we proposed an unrolling-based decision procedure for algebraic data types [9]. In the decision procedure, algebraic data types are abstracted by *catamorphisms*, which are fold functions that recursively map the data types into values in a decidable domain. For example, we can map a binary tree into a multiset (bag) of its element values by the following *Multiset* catamorphism:

$$\begin{aligned} \mathit{Multiset}(\mathsf{Leaf}) &= \emptyset \\ \mathit{Multiset}\big(\mathsf{Node}(t_L, e, t_R)\big) &= \mathit{Multiset}(t_L) \uplus \{e\} \uplus \mathit{Multiset}(t_R) \end{aligned}$$

Our decision procedure works by successively unrolling the applications of catamorphisms, treats not-yet-unrolled catamorphism instances as uninterpreted functions, and then sends the resulting formula to SMT solvers [1], [3]. Experimental results with Guardol [4] show that the decision procedure can effectively handle complex verification conditions containing algebraic data types.

Among classes of catamorphisms that work with the decision procedure, associative-commutative (AC) catamorphisms [9] stand out as an important class for three reasons. First, they can be detected by state-of-the-art analysis tools such as SMT solvers [1], [3] or theorem provers [5]. Second, they are combinable within an input formula while preserving the completeness of the decision procedure. Third, they guarantee that the decision procedure in [9] terminates after an exponentially small number of unrollings.

This paper presents parameterized associative-commutative (PAC) catamorphisms, a generalized class of AC ones, and shows that they not only have all the aforementioned features of AC catamorphisms but are also more general, cheaper to computationally reason about, and more expressive than AC catamorphisms because of the parameterization in the format of PAC catamorphisms:

- *Expressiveness:* PAC catamorphisms are strictly more expressive than AC catamorphisms because they can account for both element values and the structure of data type instances, whereas AC catamorphisms can account only for element values.
- *Usability:* PAC catamorphisms provide a more general way to abstract the content of algebraic data types. In particular, some higher-order functions such as *Forall*, *Exists*, and *Member* can be expressed as PAC catamorphisms while AC catamorphisms can only be first-order. In addition, by parameterizing the behaviors of catamorphisms, all AC catamorphisms proposed in our previous work [9] can be augmented. For example, consider the *Multiset* catamorphism mentioned before. By parameterizing the *Multiset* catamorphism, it is possible to ignore element values that are in a user-provided blacklist, or ignore subtrees that contain elements in the blacklist. Those behaviors of the augmented *Multiset* catamorphism are not supported by the construction of AC catamorphisms.
- *Efficiency:* Unlike AC catamorphisms, PAC catamorphisms have support for pruning some computational branches, leading to more efficient analysis.

In addition to the data type (e.g., tree t in the *Multiset* catamorphism), PAC catamorphisms support four more parameters, including one parameter for the base case of the data type (i.e., $t = \mathsf{Leaf}$), two parameters for the recursive case (i.e., $t = \mathsf{Node}$), and a predicate that serves as a filter for the recursive case. To the best of our knowledge, this is the first work that discusses the decidability of parameterized abstractions for algebraic data types.

The rest of the paper is organized as follows. Section II presents some preliminaries. Section III proposes PAC catamorphisms, whose benefits are demonstrated with concrete examples in Section IV. Section V shows that PAC catamorphisms preserve all powerful properties of AC catamorphisms. Experimental results are discussed in Section VI. Next, we present related work in Section VII. Finally, we conclude the paper in Section VIII.

II. Catamorphism Decision Procedure by Example

As an example of how the procedure in [9] can be used, let us consider a guard application (such as those in [4]) that needs to determine if an HTML message may be 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. We would like 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 if a word is "dirty".

A. Catamorphisms

Let τ be a tree domain, in which each vertex can be a Leaf or a $\mathsf{Node}(t_L, e, t_R)$, where $t_L, t_R \in \tau$ and e is an *element value* in an element theory $\mathcal{E}$. We denote $size(t)$ as the total number of vertices in t. Given a tree in the tree domain τ, we can map the tree to a value in a *decidable* domain $\mathcal{C}$ by catamorphisms (aka fold functions), which recursively traverse the tree and combine its element values.

Example 1 (Catamorphisms): Function *Multiset* in Section I is a catamorphism that maps a tree in τ to a multiset of all the element values stored in the tree. In this case, $\mathcal{C}$ is the multiset domain. In our dirty-word example, the tree elements are strings and we can map a tree to an integer representing the number of dirty words in the tree by the following $DW : \tau \rightarrow \mathsf{int}$ catamorphism:

$$DW(\mathsf{Leaf}) = 0$$
$$DW\big(\mathsf{Node}(t_L, e, t_R)\big) = DW(t_L) + \big(\mathsf{ite}(dirty(e))\ 1\ 0\big) + DW(t_R)$$

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

B. Unrolling-based Decision Procedure

These formulas can be discharged using an unrolling decision procedure shown in Fig. 1. The procedure uses an SMT solver that supports theories for $\tau, \mathcal{E}, \mathcal{C}$, and uninterpreted functions. The only part of the formulas that is not inherently supported by the solver is the application of the catamorphism. Hence, the main idea of the 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 tree leaves as calls to uninterpreted functions. The algorithm successively overapproximates and underapproximates the satisfiability of the original program using a set of "control conditions". If we use these conditions (i.e., these conditions are true), the satisfying assignment does not use any uninterpreted function values, so we have a complete finite model and hence *SAT* results are accurate. If we do not use these conditions (i.e., at least one of them is false), the uninterpreted functions are allowed to contribute to the *SAT*/*UNSAT* result. If the solver returns *UNSAT* in this case, the original problem must be *UNSAT* since assigning any values to the uninterpreted functions still cannot make the problem *SAT*. The details of the procedure are in [9].

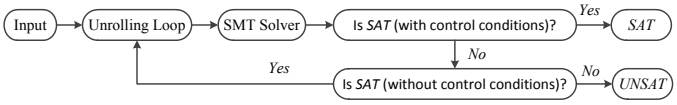

Fig. 1: Sketch of the unrolling-based decision procedure for algebraic data types

Example 2 (Unrolling-based decision procedure): For our guard example, suppose one of the VCs is: $t = \mathsf{Node}(t_L, e, t_R) \wedge dirty(e) \wedge DW(t) = 0$. The formula is *UNSAT* because t has at least one dirty word (e in this case), so its number of dirty words cannot be 0. Fig. 2 shows how the procedure works for this example.

At unrolling depth 0, $DW(t)$ is treated as an uninterpreted function $UF_t^{\geq 0} : \mathsf{int}$, which 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 $UF_t^{\geq 0}$ implies that for the first step we do not use control conditions. The formula becomes $t = \mathsf{Node}(t_L, e, t_R) \wedge dirty(e) \wedge DW(t) = 0 \wedge DW(t) = UF_t^{\geq 0}$ and is *SAT*. However, the *SAT* result is untrustworthy due to the presence of $UF_t^{\geq 0}$; 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 uninterpreted functions. In particular, $UF_{t_{L'}}^{\geq 0}$ and $UF_{t_{R'}}^{\geq 0}$ are the uninterpreted functions for $DW(t_{L'})$ and $DW(t_{R'})$, respectively. The set of control conditions in this case is $\{\neg(t \neq \mathsf{Leaf})\}$. When we use the control conditions, $UF_{t_{L'}}^{\geq 0}$ and $UF_{t_{R'}}^{\geq 0}$ will not be used and the formula becomes $t = \mathsf{Node}(t_L, e, t_R) \wedge dirty(e) \wedge DW(t) = 0 \wedge DW(t) = 0 \wedge t = \mathsf{Leaf}$, 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. Similarly, without control conditions, we still get *UNSAT*. However, getting *UNSAT* without control conditions guarantees that the original formula is *UNSAT*; thus, the process terminates here. △

C. Monotonic Catamorphisms

Our decision procedure in [9] has been proven to be sound with all types of catamorphisms and complete with *monotonic* catamorphisms. First, let us define the notion of the cardinality of the inverse function of catamorphisms.

Definition 1 (Function β): Given a catamorphism $\alpha : \tau \rightarrow \mathcal{C}$, we define $\beta(t) : \tau \rightarrow \mathbb{N}$ as the cardinality of the inverse

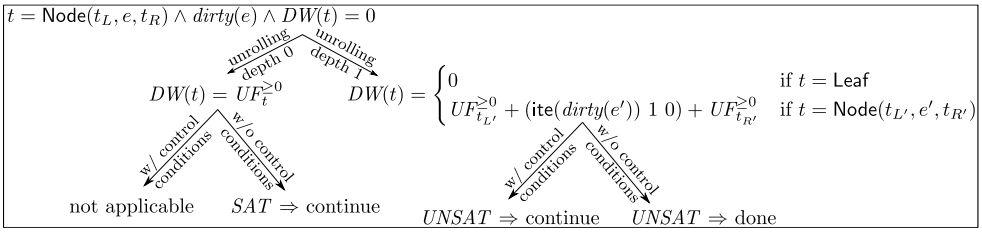

Fig. 2: An example of the decision procedure

function of $\alpha(t)$:

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

Example 3 (Function β): If α is *Multiset*, we have $\beta\big(\mathsf{Node}(\mathsf{Leaf}, 1, \mathsf{Leaf})\big) = 1$ because it is the only tree that can map to $\{1\}$ by *Multiset*. On the other hand, $\beta\big(\mathsf{Node}(\mathsf{Node}(\mathsf{Leaf}, 1, \mathsf{Leaf}), 2, \mathsf{Leaf})\big) = 4$ since there are 4 trees that can map to multiset $\{1, 2\}$. Similarly, if α is *DW*, we have $\forall t \in \tau : \beta(t) = \infty$. △

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 terms 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 2 (Monotonic catamorphisms): Catamorphism $\alpha : \tau \to \mathcal{C}$ is monotonic iff there exists a constant $h_\alpha \in \mathbb{N}^+$ such that:

$$\begin{aligned} &\forall t \in \tau : height(t) \geq h_\alpha \\ &\quad \Rightarrow \big(\beta(t) = \infty \vee \\ &\qquad \exists t_0 \in \tau : height(t_0) = height(t) - 1 \wedge \beta(t_0) < \beta(t)\big) \end{aligned}$$

Example 4 (Monotonic catamorphisms): *DW* and *Multiset* are monotonic with $h_\alpha = 1$ and $h_\alpha = 2$, respectively. Other examples of monotonic catamorphisms are *SizeI*, *Height*, *List*, *Sortedness*, *Min*, *Max*, etc. [9]. An example of a non-monotonic catamorphism is *Mirror* in [12] since $\forall t \in \tau : \beta_{\mathit{Mirror}}(t) = 1$. △

D. Associative-commutative (AC) Catamorphisms

AC catamorphisms are a powerful sub-class of monotonic catamorphisms. First, they can be detected by SMT solvers [1], [3] or theorem provers [5]. Second, they can be arbitrarily combined within an input formula while preserving the completeness of the decision procedure in [9]. Third, they allow the procedure to terminate after a small number of unrollings. Let $\oplus : (\mathcal{C}, \mathcal{C}) \to \mathcal{C}$ be an associative and commutative binary operator with an identity element $id_\oplus \in \mathcal{C}$ (i.e., $\forall x \in \mathcal{C} : x \oplus id_\oplus = id_\oplus \oplus x = x$) and $\delta : \mathcal{E} \to \mathcal{C}$ be a function that maps an element value in $\mathcal{E}$ into a value in $\mathcal{C}$. We define AC catamorphisms as follows:

Definition 3 (AC catamorphisms): A catamorphism $\alpha : \tau \to \mathcal{C}$ is AC if

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

Example 5 (AC catamorphisms): The *DW* and *Multiset* catamorphisms are AC. In the *DW* catamorphism, the operator $\oplus$ is $+$, the identity element $id_\oplus$ is 0, and the mapping function is $\delta(e) = \big(\mathsf{ite}(\mathit{dirty}(e))\ 1\ 0\big)$ while in the *Multiset* catamorphism, the three factors are $\uplus, \emptyset$, and $\delta(e) = \{e\}$, respectively. △

III. Parameterized Associative-Commutative Catamorphisms

We present parameterized associative-commutative (PAC) catamorphisms, a generalized version of AC catamorphisms with four more parameters, which offer some more important features compared with AC catamorphisms (Section IV). Although more general, PAC catamorphisms are still monotonic (Appendix A) and they preserve all the powerful characteristics of AC catamorphisms (Section V).

Definition 4 (PAC Catamorphisms): Given a predicate $\mathsf{pr} : \mathcal{E} \to \mathsf{bool}$, a value $c_{\mathsf{leaf}} \in \mathcal{C}$, a value $c_{\mathsf{pr}} \in \mathcal{C}$, and a boolean value rec, catamorphism[1] $\alpha : \tau \to \mathcal{C}$ is PAC if:

$$\alpha(t) = \begin{cases} c_{\mathsf{leaf}} & \text{if } t = \mathsf{Leaf} \\ \alpha(t_L) \oplus \delta(e) \oplus \alpha(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \wedge \neg \mathsf{pr}(e) \\ \alpha(t_L) \oplus c_{\mathsf{pr}} \oplus \alpha(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \wedge \mathsf{pr}(e) \wedge \mathsf{rec} \\ c_{\mathsf{pr}} & \text{if } t = \mathsf{Node}(t_L, e, t_R) \wedge \mathsf{pr}(e) \wedge \neg \mathsf{rec} \end{cases}$$

There are three differences in presentation between PAC and AC catamorphisms. First, Leaf is mapped to a parametric value c_{leaf} instead of $id_\oplus$, an identity element of $\oplus$. Next, element value e at each node in PAC catamorphisms is either mapped to $\delta(e)$ or c_{pr} depending on whether $\mathsf{pr}(e)$ is true or false, respectively, instead of only being mapped to $\delta(e)$ as in AC catamorphisms. Third, PAC catamorphisms have an extra parameter rec to determine in the case $\mathsf{pr}(e) = \mathsf{true}$ whether $\alpha(t)$ should be computed as $\alpha(t_L) \oplus c_{\mathsf{pr}} \oplus \alpha(t_R)$ or just as c_{pr}.

Signature. Due to the generalization, the signature of PAC catamorphisms has four more elements than that of AC catamorphisms, including the value $c_{\mathsf{leaf}} \in \mathcal{C}$ for the Leaf case, the value $c_{\mathsf{pr}} \in \mathcal{C}$ for the recursive case when the predicate pr

[1]Strictly speaking, a PAC catamorphism should be in the form $\alpha(t, \mathsf{pr}, c_{\mathsf{leaf}}, c_{\mathsf{pr}}, \mathsf{rec})$. However, since the last four parameters are unchanged during the argument passing process (i.e., $\alpha(t, \mathsf{pr}, c_{\mathsf{leaf}}, c_{\mathsf{pr}}, \mathsf{rec})$ is computed in terms of $\alpha(t_L, \mathsf{pr}, c_{\mathsf{leaf}}, c_{\mathsf{pr}}, \mathsf{rec})$ and $\alpha(t_R, \mathsf{pr}, c_{\mathsf{leaf}}, c_{\mathsf{pr}}, \mathsf{rec})$), we do not explicitly write the four parameters for brevity.

does not hold, the definition of the predicate pr itself, and the boolean value rec to determine how the catamorphism behaves when predicate pr holds.

Definition 5 (PAC signature): The signature of a PAC catamorphism α is:

$$sig(\alpha) = \langle \mathcal{C}, \mathcal{E}, \oplus, \delta, c_{\mathsf{leaf}}, c_{\mathsf{pr}}, \mathsf{pr}, \mathsf{rec} \rangle$$

Values. If $\mathsf{rec} = \mathsf{true}$, because of the associative and commutative operator $\oplus$, the value of a PAC catamorphism α for any tree t has an important property: it is independent of the structure of the tree.

If $\mathsf{rec} = \mathsf{false}$, the value of $\alpha(t)$ may or may not depend on the structure of the tree. If there exists an element value $e_t \in t$ such that $\mathsf{pr}(e_t) = \mathsf{true}$, the value of $\alpha(t)$ is dependent of the structure of the tree because the computation of $\alpha(t)$ ignores some parts of t, depending on the location of element value e_t. Otherwise, the value of $\alpha(t)$ is independent of the structure of the tree and simplifies to

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

whose value is, due to the associative and commutative operator $\oplus$, independent of the locations of element values.

Corollary 1 (Values of PAC catamorphisms): The value of $\alpha(t)$, where α is a PAC catamorphism, only depends on the values of elements in t and does not depend on the relative positions of the element values iff (1) $\mathsf{rec} = \mathsf{true}$ or (2) $\mathsf{rec} = \mathsf{false}$ and $\nexists$ element value $e \in \mathcal{E}$ in t: $\mathsf{pr}(e) = \mathsf{true}$.

To ensure the completeness of decision procedure, PAC catamorphisms must be monotonic [9]. For the sake of space, we present the proof of monotonicity of PAC catamorphisms in Appendix A.

IV. Benefits of PAC Catamorphisms

We demonstrate the advantages of PAC catamorphisms over AC catamorphisms in terms of expressiveness, usability, and efficiency with some concrete examples.

A. Expressiveness

Theorem 1: PAC catamorphisms are more expressive than AC catamorphisms.

Proof: Given a PAC catamorphism α as defined in Definition 4, if we fix c_{leaf} to be $id_{\oplus}$ and predicate pr to be false, α becomes:

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

which is an AC catamorphism by Definition 3. Therefore, AC catamorphisms are a special case of PAC catamorphisms. If we vary the values of parameters pr, c_{leaf}, c_{pr}, and rec, we will get some PAC catamorphisms that are not AC. ■

Let us give some examples to demonstrate that some PAC catamorphisms are not AC. First, consider the catamorphism

$$\begin{aligned} \mathit{NLeaves}(\mathsf{Leaf}) &= 1 \\ \mathit{NLeaves}\big(\mathsf{Node}(t_L, \_, t_R)\big) &= \mathit{NLeaves}(t_L) + \mathit{NLeaves}(t_R) \end{aligned}$$

which maps a tree to its number of leaves. Because 1 is not an identity element of operator $+$, *NLeaves* is not AC. However, it is still PAC. In other words, while AC catamorphisms only allow an identity of the operator to be used for Leaf nodes, *PAC catamorphisms do not have this restriction*.

Also, *PAC catamorphisms support predicates that can be defined over element values* while AC catamorphisms do not. For example, suppose we have a predicate $\mathit{isBad} : \mathcal{E} \to \mathsf{bool}$ that determines whether an internal node is bad. We consider an internal node $\mathsf{Node}(\_, e, \_)$ to be bad if $\mathit{isBad}(e) = \mathsf{true}$. Now consider a catamorphism called $\mathit{NGN} : \tau \to \mathsf{int}$ (number of good nodes), which maps a tree into the number of "good" internal nodes that (1) are not bad and (2) are not descendants of any bad nodes. We can define the catamorphism as follows:

$$\begin{aligned} \mathit{NGN}(\mathsf{Leaf}) &= 0 \\ \mathit{NGN}\big(\mathsf{Node}(t_L, e, t_R)\big) &= \begin{cases} \mathit{NGN}(t_L) + 1 + \mathit{NGN}(t_R) & \text{if } \neg \mathit{isBad}(e) \\ 0 & \text{if } \mathit{isBad}(e) \end{cases} \end{aligned}$$

By Corollary 5 in [9], this catamorphism is not AC because the value of $\mathit{NGN}(t)$ clearly depends on the locations of the element values of t: if we swap two element values in the tree, good nodes can turn bad and vice versa. However, we can still define this catamorphism as a PAC catamorphism.

B. Usability

In [9], we discussed $\mathit{Negative} : \tau \to \mathsf{bool}$, an AC catamorphism that maps a tree into true if all of its element values are negative:

$$\begin{aligned} \mathit{Negative}(\mathsf{Leaf}) &= \mathsf{true} \\ \mathit{Negative}\big(\mathsf{Node}(t_L, e, t_R)\big) &= \mathit{Negative}(t_L) \wedge (e < 0) \wedge \mathit{Negative}(t_R) \end{aligned}$$

Similarly, we can define the AC catamorphism $\mathit{Positive} : \tau \to \mathsf{bool}$ as follows:

$$\begin{aligned} \mathit{Positive}(\mathsf{Leaf}) &= \mathsf{true} \\ \mathit{Positive}\big(\mathsf{Node}(t_L, e, t_R)\big) &= \mathit{Positive}(t_L) \wedge (e > 0) \wedge \mathit{Positive}(t_R) \end{aligned}$$

We can observe that the two AC catamorphisms express properties expected to hold over all elements of the tree. If we can provide a predicate $pr_u : \mathcal{E} \to \mathsf{bool}$, then these catamorphisms (as well as many others) can be defined by a single parametric catamorphism $\mathit{Forall} : \tau \to \mathsf{bool}$:

$$\begin{aligned} \mathit{Forall}(\mathsf{Leaf}) &= \mathsf{true} \\ \mathit{Forall}\big(\mathsf{Node}(t_L, e, t_R)\big) &= \mathit{Forall}(t_L) \wedge pr_u(e) \wedge \mathit{Forall}(t_R) \end{aligned}$$

Obviously, *Forall*, a PAC catamorphism, provides a more compact and general abstraction than AC catamorphisms such as *Positive* and *Negative*. Thus, it is *possible to define high-order functions* such as *Forall*, *Exists*, and *Member* with PAC catamorphisms while we cannot do this with AC abstractions.

C. Efficiency

Theorem 2: Given an AC catamorphism α_{AC} and a PAC catamorphism α_{PAC}, for every tree $t \in \tau$ that the two catamorphisms accept as input, $\alpha_{PAC}(t)$ requires less or equal number of recursive calls to compute its value than $\alpha_{AC}(t)$.

Proof: For AC catamorphisms, when $t = \mathsf{Node}(t_L, e, t_R)$, $\alpha_{AC}(t)$ is always computed in terms of $\alpha_{AC}(t_L)$ and $\alpha_{AC}(t_R)$, which in turn will be computed in terms of

TABLE I: Some PAC catamorphisms that are not AC

Name	$\mathcal{C}$	$\oplus$	$\delta(e)$	c_{leaf}	c_{pr}	pr	rec
Forall	bool	$\wedge$	$pr_u(e)$	true	false	$\neg pr_u$	true/false
Exists	bool	$\vee$	$pr_u(e)$	false	true	pr_u	true/false
Member	bool	$\vee$	$(e = x)$	false	true	$(e = x)$	true/false
NGN	int	$+$	1	0	0	*isBad*	false
NLeaves	int	$+$	0	1		false	true/false

α_{AC}(their sub-trees). Hence, to compute $\alpha_{AC}(t)$, the total number of function calls we need to make to α_{AC} is equal to $size(t)$. For PAC catamorphisms, on the other hand, when $t = \mathsf{Node}(t_L, e, t_R)$, $\alpha_{PAC}(t)$ might or might not need to call $\alpha_{PAC}(t_L)$ and $\alpha_{PAC}(t_R)$, depending on the value of $\mathsf{pr}(e)$. Thus, the total number of function calls to α_{PAC} to compute $\alpha_{PAC}(t)$ is at most $size(t)$. ■

Take the *Forall* catamorphism as an example. Although compact, it is not optimal in terms of computation: if $t = \mathsf{Node}(t_L, e, t_R)$, the values of *Forall*$(t_L)$ and *Forall*(t_R) are computed regardless what $pr_u(e)$ is. However, if $pr_u(e) = \mathsf{false}$, we can conclude that *Forall*$(t) = \mathsf{false}$ without computing *Forall*(t_L) and *Forall*(t_R). Based on this observation, we can rewrite the catamorphism as follows:

$$\textit{Forall}(\mathsf{Leaf}) = \mathsf{true}$$

$$\textit{Forall}\big(\mathsf{Node}(t_L, e, t_R)\big) = \begin{cases} \textit{Forall}(t_L) \wedge \textit{Forall}(t_R) & \text{if } pr_u(e) \\ \mathsf{false} & \text{if } \neg pr_u(e) \end{cases}$$

which is PAC but not AC. Since AC catamorphisms cannot prune recursive computations while PAC catamorphisms can, PAC catamorphisms can be more efficient than AC ones.

Table I shows the full definitions of all PAC catamorphisms discussed in Section IV. They are some PAC catamorphisms that cannot be naturally expressed in an AC way. Note that from Theorem 1, every AC catamorphism is PAC.

V. AC Features in PAC Catamorphisms

AC catamorphisms have some powerful properties: they are detectable, combinable, and only require an exponentially small number of unrollings for the decision procedure in [9]. This section shows that PAC catamorphisms still have all the properties of AC catamorphisms.

Detection. Like AC catamorphisms, PAC catamorphisms can be detected. A catamorphism written in the format in Definition 4 is PAC if $\oplus$ is an associative and commutative operator over the collection domain $\mathcal{C}$. We can use SMT solvers [1], [3] or theorem provers [5] to check this property of operator $\oplus$.

Exponentially Small Upper Bound of the Number of Unrollings. Since PAC catamorphisms are monotonic (proved in Appendix A), they can be used in the decision procedure in [9]. Like AC catamorphisms, PAC catamorphisms guarantee that the number of unrollings is *exponentially small* compared with the size of the input formula, which is represented by the maximum number of inequalities between tree terms in the input formula. The proof of the exponentially small number of unrollings is nearly the same as that in [9]; the only difference is that we use Lemma 2 in Appendix A to generalize the result for PAC catamorphisms instead of Lemma 8 in [9], which only works for AC catamorphisms.

Combining PAC Catamorphisms. One of the most powerful properties of PAC catamorphisms is that they can be combinable. Let $\alpha_1, \ldots, \alpha_m$ be m PAC catamorphisms, where the signature of the i-th catamorphim ($1 \le i \le m$) is $sig(\alpha_i) = \langle \mathcal{C}_i, \mathcal{E}, \oplus_i, \delta_i, c_{\mathsf{leaf}\,i}, c_{\mathsf{pr}\,i}, \mathsf{pr}, \mathsf{rec} \rangle$. Catamorphism α with signature $sig(\alpha) = \langle \mathcal{C}, \mathcal{E}, \oplus, \delta, c_{\mathsf{leaf}}, c_{\mathsf{pr}}, \mathsf{pr}, \mathsf{rec} \rangle$ is a combination of $\alpha_1, \ldots, \alpha_m$ if

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

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

- $\delta : \mathcal{E} \to \mathcal{C}$ is defined as follows:

  $$\delta(e) = \big\langle \delta_1(e), \delta_2(e), \ldots, \delta_m(e) \big\rangle$$

- $c_{\mathsf{leaf}} : \mathcal{C}$ is defined as follows:

  $$c_{\mathsf{leaf}} = \langle c_{\mathsf{leaf}\,1}, c_{\mathsf{leaf}\,2}, \ldots, c_{\mathsf{leaf}\,m} \rangle$$

- $c_{\mathsf{pr}} : \mathcal{C}$ is defined as follows:

  $$c_{\mathsf{pr}} = \langle c_{\mathsf{pr}\,1}, c_{\mathsf{pr}\,2}, \ldots, c_{\mathsf{pr}\,m} \rangle$$

Theorem 3: A combination of PACs is PAC (Proof is in Appendix B).

VI. Experimental Results

We have implemented support for PAC catamorphisms in RADA [10], an implementation of our unrolling-based decision procedure [9] for algebraic data types. We have also evaluated the tool with a collection of benchmark examples. Each example contains verification conditions related to parameterized catamorphisms and has 60–115 lines of code written in a format similar to SMT-Lib 2.0 [2]. The results are very promising: all of the benchmarks were automatically verified by RADA in a short amount of time.

Table II consists of 12 benchmarks involving PAC catamorphisms; some of them represent important higher-order functions such as *forall*, *exists*, and *member*. Each of the first 10 benchmarks in Table II only involves one catamorphism. Catamorphisms *NLeaves*, *Forall* and *NGN* have been introduced in Section IV. Catamorphism *Exists* maps a tree into true if the tree contains at least one element value that satisfies a user-provided predicate pr_u while catamorphism *Member* maps a tree into true if the tree contains a user-provided value x. The last two examples consist of the combination of *NGN* and a slightly modified version of the catamorphism to demonstrate the combinability of PAC catamorphisms as discussed in Section V.

In addition to PAC catamorphisms, we have also experimented RADA with some examples in Table III containing general non-PAC parameterized catamorphisms automatically generated from the Guardol verification system [4]. They consist of verification conditions to prove some interesting properties of red black trees and the checksums of trees of

TABLE II: Experimental results with PAC catamorphisms

	Benchmark	Result	Time (s)
Single PAC catamorphisms	forall01	sat	0.352
	forall02	unsat	0.246
	exists01	sat	0.046
	exists02	unsat	0.048
	member01	sat	0.167
	member02	unsat	0.257
	nleaves01	sat	0.332
	nleaves02	unsat	0.161
	ngn01	sat	0.428
	ngn02	unsat	0.113
Combination of PAC catamorphisms	ngn_ngn01	sat	0.556
	ngn_ngn02	unsat	0.157

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.

TABLE III: Experimental results on Guardol benchmarks

Benchmark	Result	Time (s)
Email_Guard_Correct_All	17 unsats	$\approx$ 0.009/obligation
RBTree.Black_Property	12 unsats	$\approx$ 2.142/obligation
RBTree.Red_Property	12 unsats	$\approx$ 0.163/obligation
array_checksum.SumListAdd	2 unsats	$\approx$ 0.028/obligation
array_checksum.SumListAdd_Alt	13 unsats	$\approx$ 0.012/obligation

All benchmarks were run on a Ubuntu machine using an Intel Core I5 running at 2.8 GHz with 4GB RAM. All the running time was measured when Z3 was used as the reasoning engine of the tool. RADA and all the benchmarks are available at http://crisys.cs.umn.edu/rada.

VII. Related Work

The idea of using abstractions to reason about algebraic data types has been explored by the Jahob [14], [15] and Leon systems [13]. In the decision procedures proposed by Suter et al. [12], [13], algebraic data types are abstracted by sufficiently surjective catamorphisms, which are closely related to the monotonicity construction in [9]. Sufficiently surjective catamorphisms are difficult to automatically detect and it is not known whether sufficiently surjective catamorphisms can be combined in a decidable way.

Madhusudan et al. [8] proposes DRYAD, a logic to reason about inductive tree data structures abstracted by recursive abstractions. However, the collection of abstractions supported by this work is more limited than ours. In particular, they only support four types of abstractions: from a tree to an integer, to a set of integers, to a multiset of integers, or to a boolean value. The abstractions used in DRYAD_{dec}, a decidable fragment of DRYAD that can be embedded into the decidable logic STRAND_{dec} [7], are even more limited. However, the class of data structures that [8] can work with is richer than that of our approach.

Sato et al. [11] introduces a model checker that has support for recursive data structures. Unlike ours, the element type in their work must be int. In their approach, recursive data structures are first encoded as functions on lists, and then encoded as functions on integers before the verification tool in [6] is used. Their method cannot verify some properties of recursive data structures, such as the properties of red-black trees, while ours can thanks to the use of catamorphisms.

VIII. Conclusion

This paper presents parameterized associative-commutative (PAC) catamorphisms, a generalized version of associative-commutative (AC) catamorphisms [9]. We have shown that PAC catamorphisms have all the powerful features of AC catamorphisms: they are automatically detectable, combinable, and guarantee an exponentially small number of unrollings for the unrolling-based decision procedure in [9]. Furthermore, we have demonstrated that PAC catamorphisms are more general, computationally optimal, and expressive than AC ones.

One of the challenges we would like to work on in the future is to ensure the completeness of the decision procedure in [9] by accurately capturing the ranges of PAC catamorphisms. This is not a problem for surjective catamorphisms such that *Forall*, *Exist*, or *Member*. However, for non-surjective catamorphisms such as *NGN*, we need to encode their ranges by a predicate R_α as discussed in [9].

Acknowledgements. The first author was sponsored in part by a University of Minnesota Doctoral Dissertation Fellowship 2013-2014 and a 3M Fellowship 2010-2014. This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

References

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[2] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT*, 2010.

[3] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[4] D. Hardin, K. Slind, M. Whalen, and T.-H. Pham. The Guardol Language and Verification System. In *TACAS*, pages 18–32, 2012.

[5] M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Springer, 2000.

[6] N. Kobayashi, R. Sato, and H. Unno. Predicate Abstraction and CEGAR for Higher-Order Model Checking. In *PLDI*, pages 222–233, 2011.

[7] P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. In *POPL*, pages 611–622, 2011.

[8] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive Proofs for Inductive Tree Data-Structures. In *POPL*, pages 123–136, 2012.

[9] T.-H. Pham and M. W. Whalen. An Improved Unrolling-Based Decision Procedure for Algebraic Data Types. In *VSTTE*, 2013.

[10] T.-H. Pham and M. W. Whalen. RADA: A Tool for Reasoning about Algebraic Data Types with Abstractions. In *ESEC/FSE*, 2013.

[11] R. Sato, H. Unno, and N. Kobayashi. Towards a Scalable Software Model Checker for Higher-Order Programs. In *PEPM*, 2013.

[12] P. Suter, M. Dotta, and V. Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *POPL*, pages 199–210, 2010.

[13] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability Modulo Recursive Programs. In *SAS*, pages 298–315, 2011.

[14] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *PLDI*, pages 349–361, 2008.

[15] K. Zee, V. Kuncak, and M. C. Rinard. An Integrated Proof Language for Imperative Programs. In *PLDI*, pages 338–351, 2009.

APPENDIX A
THE MONOTONICITY OF PAC CATAMORPHISMS

To work with our unrolling-based decision procedure for algebraic data types in [9], PAC catamorphisms must be monotonic (see Definition 2 in Section II). In this Appendix, we prove the monotonicity of PAC catamorphisms. First, let us introduce some new supporting lemmas and corollaries.

Definition 6 (Satisfiable Predicate): Predicate $\mathsf{pr} : \mathcal{E} \rightarrow \mathsf{bool}$ is satisfiable if $\exists e \in \mathcal{E} : \mathsf{pr}(e) = \mathsf{true}$.

Lemma 1: Given a PAC catamorphism α with $\mathsf{rec} = \mathsf{false}$, if pr is satisfiable, then $|\alpha^{-1}(c_{\mathsf{pr}})| = \infty$.

Proof: Since pr is satisfiable, from Definition 6, there exists $e_0 \in \mathcal{E}$ such that $\mathsf{pr}(e_0) = \mathsf{true}$. Also, there are an infinite number of trees such that the element values in their roots are e_0. Furthermore, α maps each of these trees to c_{pr} because $\mathsf{pr}(e_0) = \mathsf{true}$ and $\mathsf{rec} = \mathsf{false}$. Hence, $|\alpha^{-1}(c_{\mathsf{pr}})| = \infty$. ■

Corollary 2: Given a PAC catamorphism α with $\mathsf{rec} = \mathsf{false}$ and a tree $t \in \tau$, if there exists an element value $e_t \in t$ such that $\mathsf{pr}(e_t) = \mathsf{true}$, then $\beta(t) = \infty$.

Proof: Let t_{e_t} be the tree rooted at e_t in t. Since $\mathsf{pr}(e_t) = \mathsf{true}$ and $\mathsf{rec} = \mathsf{false}$, we have $\alpha(t_{e_t}) = c_{\mathsf{pr}}$ by Definition 4. By Lemma 1, $|\alpha^{-1}(c_{\mathsf{pr}})| = \infty$. In other words, $|\alpha^{-1}\big(\alpha(t_{e_t})\big)| = \beta(t_{e_t}) = \infty$. Thus, we have $\beta(t) = \infty$ by Lemma 6 in [9]. ■

Corollary 3: Given a PAC catamorphism α with $\mathsf{rec} = \mathsf{false}$ and $t \in \tau$, either

- $\beta(t) = \infty$, or
- $\beta(t) < \infty$ and for all tree t' in the collection of $\beta(t)$ trees that can map to $\alpha(t)$, there does not exist any element value $e_{t'}$ in t' such that $\mathsf{pr}(e_{t'}) = \mathsf{true}$.

Proof: This corollary follows from Corollary 2. ■

The proof of monotonicity of PAC catamorphisms involves some properties of tree shapes and strict subtrees [9], which are defined as follows.

Definition 7 (Tree shapes): The shape of a tree is defined by constant SLeaf and constructor $\mathsf{SNode}(\_,\_)$ as follows:

$$\begin{aligned} \mathit{shape}(\mathsf{Leaf}) &= \mathsf{SLeaf} \\ \mathit{shape}\big(\mathsf{Node}(t_L, \_, t_R)\big) &= \mathsf{SNode}\big(\mathit{shape}(t_L), \mathit{shape}(t_R)\big) \end{aligned}$$

We also denote $ns(s)$ as the number of shapes of size s.

Definition 8 (Strict subtrees): Given two trees t_1 and t_2 in the tree domain τ, tree t_1 is a subtree of tree t_2, denoted by $t_1 \preceq t_2$, iff:

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

Tree t_1 is a strict subtree of t_2, denoted by $t_1 \npreceq t_2$, iff $t_1 \preceq t_2 \wedge \mathit{size}(t_1) < \mathit{size}(t_2)$.

We now prove a lemma about the relationship between $\beta(t)$ and $ns(s)$, which plays an important role in proving the monotonicity of PAC catamorphisms.

Lemma 2: If α is a PAC catamorphism then $\forall t \in \tau : \beta(t) \geq ns\big(\mathit{size}(t)\big)$.

Proof: Let t be any tree in τ. If $\mathsf{rec} = \mathsf{true}$, from Corollary 1, the value of $\alpha(t)$ does not depend on the relative locations of elements values in t. The proof of the lemma in this case is similar to that of Lemma 8 in [9] with minor changes.

If $\mathsf{rec} = \mathsf{false}$, the value of $\beta(t)$ can either be infinity or not. If $\beta(t) = \infty$, the lemma follows immediately. If $\beta(t) < \infty$, from Corollary 3, there does not exist any element value e_t in t such that $\mathsf{pr}(e_t) = \mathsf{true}$. Hence, from Corollary 1, the computation of $\alpha(t)$ does not depend on the relative locations of any element values in t and we can use a similar proof as in that of Lemma 8 in [9]. ■

Now, let us prove that PAC catamorphisms are monotonic. We split the proof into two separate cases: the first one is for the case of PAC catamorphisms with $\mathsf{rec} = \mathsf{true}$ and the other one is for PAC catamorphisms with $\mathsf{rec} = \mathsf{false}$.

Lemma 3: PAC catamorphisms with $\mathsf{rec} = \mathsf{true}$ are monotonic.

Proof: Let α be a PAC catamorphism with $\mathsf{rec} = \mathsf{true}$. Let $h_\alpha = 4$. Consider any tree $t \in \tau$ such that $\mathit{height}(t) \geq h_\alpha = 4$. If $\beta(t) = \infty$, the monotonic condition for t in Definition 2 holds.

On the other hand, suppose $\beta(t) < \infty$. By Lemma 4 in [9], $\exists t_0 \in \tau : t_0 \npreceq t \wedge \mathit{height}(t_0) = \mathit{height}(t) - 1 \geq 3$. Let Q be the set of internal nodes that are in t but not in t_0. Q is not empty since $t_0 \npreceq t$. Let $e_1, \ldots, e_{|Q|}$ be the elements stored in $|Q|$ nodes in Q. We define a new mapping function as follows:

$$\delta'(e) = \begin{cases} \delta(e) & \text{if } \mathsf{pr}(e) = \mathsf{false} \\ c_{\mathsf{pr}} & \text{if } \mathsf{pr}(e) = \mathsf{true} \end{cases}$$

and the value of $\alpha(t)$ can be computed as follows:

$$\begin{aligned} \alpha(t) = \alpha(t_0) \ \oplus \ \delta'(e_1) \ \oplus \ \delta'(e_2) \ \oplus \ \ldots \ \oplus \ \delta'(e_{|Q|}) \\ \oplus \ \underbrace{c_{\mathsf{leaf}} \ \oplus \ \ldots \ \oplus \ c_{\mathsf{leaf}}}_{|Q| \text{ occurrences of } c_{\mathsf{leaf}}} \end{aligned} \tag{1}$$

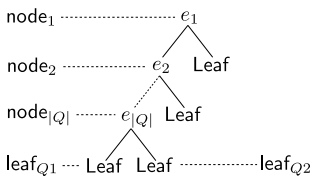

Fig. 3: Construct t_Q

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

By Property 3 in [9], $\mathit{height}(t_0) \geq 3$ implies $\mathit{size}(t_0) \geq 7$. By Lemma 2 in [9], $ns\big(\mathit{size}(t_0)\big) \geq ns(7) > 2$. By Lemma 2, $\beta(t_0) \geq ns\big(\mathit{size}(t_0)\big) > 2$. Since there is at most one Leaf tree in the set of $\beta(t_0)$ trees that can map to $\alpha(t_0)$, there are

at least $\beta(t_0)-1$ bigger-than-Leaf trees that can map to $\alpha(t_0)$. Since $\beta(t_0) > 2$, the number of such bigger-than-Leaf trees is at least 2. Let t'_0 and t''_0 be any two of them. That is, t'_0 and t''_0 are two different bigger-than-Leaf trees and

$$\alpha(t'_0) = \alpha(t''_0) = \alpha(t_0) \tag{2}$$

Note also that all bigger-than-Leaf trees in τ, including t'_0 and t''_0, have at least two leaves at their lowest depths.

Consider t'_0. Let leaf'_1 and leaf'_2 be any pair of distinct leaves at the lowest depth of t'_0. Let t'_{01} and t'_{02} be the trees obtained by replacing leaf'_1 and leaf'_2 in t'_0 with t_Q, respectively. Since $t_Q \neq \mathsf{Leaf}$, we have $t'_{01} \neq t'_{02}$. We have

$$\begin{aligned} & \alpha(t'_{01}) = \alpha(t'_{02}) \\ = \ & \alpha(t'_0) \ \oplus \ \delta'(e_1) \ \oplus \ \delta'(e_2) \ \oplus \ \ldots \ \oplus \ \delta'(e_{|Q|}) \\ & \quad \oplus \underbrace{c_{\mathsf{leaf}} \ \oplus \ \ldots \ \oplus \ c_{\mathsf{leaf}}}_{|Q| \text{ occurrences of } c_{\mathsf{leaf}}} \\ = \ & \alpha(t) \qquad \text{[From Equations (1) and (2)]} \end{aligned}$$

Hence, *from any bigger-than-*Leaf *tree that can map to* $\alpha(t_0)$*, we can generate at least 2 distinct trees that can map to* $\alpha(t)$.

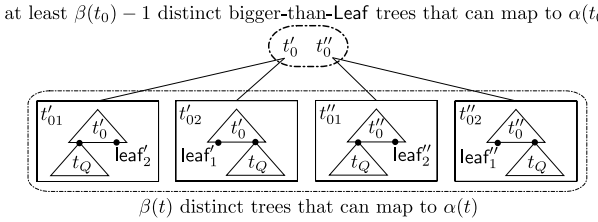

Fig. 4: Relationship between t'_0, t''_0 and $t'_{01}, t'_{02}, t''_{01}, t''_{02}$

Consider t''_0. We construct two different trees t''_{01} and t''_{02} from t''_0 and t_Q such that $\alpha(t''_{01}) = \alpha(t''_{02}) = \alpha(t)$ using the same method as before. Since $t'_0 \neq t''_0$, four trees $t'_{01}, t'_{02}, t''_{01}$, and t''_{02} are mutually different. Fig. 4 shows their relationship.

Moreover, t'_0 and t''_0 are any pair of different bigger-than-Leaf trees that can map to $\alpha(t_0)$. Thus, *from the set of at least* $\beta(t_0)-1$ *distinct bigger-than-*Leaf *trees that can map to* $\alpha(t_0)$*, we can generate at least* $2 \times \big(\beta(t_0)-1\big)$ *distinct trees that can map to* $\alpha(t)$. Hence, $\beta(t) \geq 2 \times \big(\beta(t_0)-1\big)$, which leads to $\beta(t) > \beta(t_0)$ since $\beta(t_0) > 2$. As a result, α is monotonic based on Definition 2. ■

Lemma 4: PAC catamorphisms with $\mathsf{rec} = \mathsf{false}$ are monotonic.

Proof: Let α be a PAC catamorphism with $\mathsf{rec} = \mathsf{false}$. The proof outline is as follows:

1) If pr is unsatisfiable, catamorphism α is also a PAC catamorphism with $\mathsf{rec} = \mathsf{true}$. Thus, α is monotonic from Lemma 3.
2) On the other hand, if pr is satisfiable, consider any tree $t \in \tau$ of height at least $h_\alpha = 2$. There are two sub-cases as follows.
   a) If $\exists e_t \in t : \mathsf{pr}(e_t) = \mathsf{true}$, we show that $\beta(t) = \infty$, which implies the monotonicity of α by Definition 2.
   b) If $\nexists e_t \in t : \mathsf{pr}(e_t) = \mathsf{true}$, we show that $\exists t_0 \in \tau$ such that $height(t_0) = height(t)-1$ and $\beta(t_0) < \beta(t)$. Hence, α is monotonic by Definition 2.

We now present the proof in detail. If predicate pr is unsatisfiable, the definition of the PAC catamorphism α can be rewritten as follows:

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

which can easily be mapped to a special case of the definition of a PAC catamorphism with $\mathsf{rec} = \mathsf{true}$, which is monotonic by Lemma 3. Thus, α is monotonic.

On the other hand, consider the case when predicate pr is satisfiable. We will prove that α is monotonic with $h_\alpha = 2$. Let $t \in \tau$ be any tree of height at least 2. There are two sub-cases to consider:

Sub-case 1: [There exists an element value e_t in t such that $\mathsf{pr}(e_t) = \mathsf{true}$]. From Corollary 2, $\beta(t) = \infty$. Therefore, the monotonic condition holds for t.

Sub-case 2: [There does not exist any element values in t to make pr hold]. From Lemma 4 in [9], there exists $t_0 \in \tau$ such that $t_0 \precneqq t$ and $height(t_0) = height(t) - 1 \geq 1$. Our goal is to prove that either $\beta(t) = \infty$ or $\beta(t_0) < \beta(t)$.

Let Q be the collection of internal nodes that are in t but not in t_0. Q is not empty since $t_0 \precneqq t$. Let $e_1, e_2, \ldots, e_{|Q|}$ be all the element values in Q. By construction, every element value in t_0 and Q must be in the collection of element values in t. The condition in this sub-case implies that there does not exist any element values in t, t_0, and Q that can make pr hold. Therefore, we have

$$\begin{aligned} \alpha(t) = \alpha(t_0) \ & \oplus \ \delta(e_1) \ \oplus \ \delta(e_2) \ \oplus \ \ldots \ \oplus \ \delta(e_{|Q|}) \\ & \oplus \underbrace{c_{\mathsf{leaf}} \ \oplus \ \ldots \ \oplus \ c_{\mathsf{leaf}}}_{|Q| \text{ occurrences of } c_{\mathsf{leaf}}} \end{aligned} \tag{3}$$

Let $t'_0 \in \tau$ be any tree in the collection of $\beta(t_0)$ trees that can map to $\alpha(t_0)$ via catamorphism α. Note that t_0 is also in this collection. Hence, we have

$$\alpha(t'_0) = \alpha(t_0) \tag{4}$$

Next, we construct a tree t_Q from $e_1, \ldots, e_{|Q|}$ as in Fig. 3. Given t_Q, by replacing leaf_{Q1} with t'_0, we obtain a distinct tree t'_{01} such that:

$$\begin{aligned} \alpha(t'_{01}) = \alpha(t'_0) \ & \oplus \ \delta(e_1) \ \oplus \ \delta(e_2) \ \oplus \ \ldots \ \oplus \ \delta(e_{|Q|}) \\ & \oplus \underbrace{c_{\mathsf{leaf}} \ \oplus \ \ldots \ \oplus \ c_{\mathsf{leaf}}}_{|Q| \text{ occurrences of } c_{\mathsf{leaf}}} \end{aligned} \tag{5}$$

From Equations (3), (4), and (5), we have: $\alpha(t) = \alpha(t'_{01})$. Thus, from each tree t'_0 in the set of $\beta(t_0)$ distinct trees that can map to $\alpha(t_0)$, we can generate a distinct tree t'_{01} that can map to $\alpha(t)$. Hence, from $\beta(t_0)$ distinct trees that can map to $\alpha(t_0)$, we can generate at least $\beta(t_0)$ distinct trees that can map to $\alpha(t)$.

Let $B^{\mathsf{leaf}_{Q1}}$ be the set of $\beta(t_0)$ distinct trees that can map to $\alpha(t)$ generated by the substitutions of leaf_{Q1} in t_Q as discussed before. Obviously, leaf_{Q2} exists in all the trees in $B^{\mathsf{leaf}_{Q1}}$ since leaf_{Q2} is untouched during the substitution process.

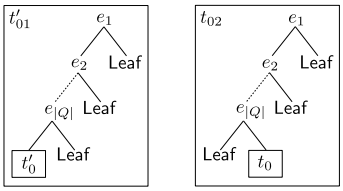

Fig. 5: The constructions of t'_{01} and t_{02}.

Next, we show that there exists at least another tree that can map to $\alpha(t)$ but is not in $B^{\mathsf{leaf}_{Q1}}$. Given t_Q, we now replace leaf_{Q2} with t_0 to obtain a tree t_{02}. The constructions of t'_{01} and t_{02} are shown in Fig. 5. We have:

$$\begin{aligned}\alpha(t_{02}) = \alpha(t_0) \ \oplus \ \delta(e_1) \ \oplus \ \delta(e_2) \ \oplus \ \ldots \ \oplus \ \delta(e_{|Q|}) \\ \oplus \ \underbrace{c_{\mathsf{leaf}} \ \oplus \ \ldots \ \oplus \ c_{\mathsf{leaf}}}_{|Q| \text{ occurrences of } c_{\mathsf{leaf}}} \\ = \alpha(t) \qquad [\text{ From Equation (3) }]\end{aligned}$$

Thus, t_{02} is also a tree that can map to $\alpha(t)$. Since $height(t_0) \geq 1$, t_0 must not be a Leaf tree. Therefore, by replacing leaf_{Q2} in t_Q with t_0 to obtain t_{02}, leaf_{Q2} must not be in t_{02}. Moreover, since leaf_{Q2} is in all the trees in $B^{\mathsf{leaf}Q1}$, tree t_{02} is different from all the trees in $B^{\mathsf{leaf}_{Q1}}$. Thus, there are at least $\beta(t_0)+1$ distinct trees that can map to $\alpha(t)$, including t_{02} and those in $B^{\mathsf{leaf}_{Q1}}$. In other words,

$$\begin{aligned}\beta(t_0) + 1 \leq \beta(t) \\ \therefore \qquad \beta(t_0) < \beta(t)\end{aligned}$$

Therefore, if $\beta(t_0)$ is infinite, $\beta(t)$ must also be infinite; otherwise, if $\beta(t_0)$ is finite, we have $\beta(t_0) < \beta(t)$. Hence, the monotonic condition holds for t by Definition 2. ■

Theorem 4: PAC catamorphisms are monotonic.

Proof: The theorem follows from Lemmas 3 and 4. ■

APPENDIX B
PROOF OF THEOREM 3

Proof: Let α be a combination of m PAC catamorphisms $\alpha_1, \ldots, \alpha_m$. By construction, it is straightforward that α is written in the format of a PAC catamorphism in Definition 4. We prove α is really a PAC catamorphism by showing that $\oplus$ is an associative and commutative operator.

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

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

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

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

■