Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 13-018

On Parameterized Abstractions in Unrolling-Based Decision Procedure for Algebraic Data Types

Tuan-hung Pham and Michael W. Whalen

June 03, 2013

On Parameterized Abstractions in Unrolling-Based Decision Procedure for Algebraic Data Types

Tuan-Hung Pham and Michael Whalen

University of Minnesota
Minneapolis, MN 55455, USA

Abstract. Reasoning about algebraic data types is an important problem for a variety of proof tasks. Recently, a variety of decision procedures have been proposed for algebraic data types involving creating 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 demonstrated.

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* and *exists*) over elements of algebraic data types straightforward to express.

1 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 [10], Suter et al. [12,13], and Madhusudan et al. [9]. To support complete reasoning about algebraic types, the abstractions usually need to meet some requirements, such as the monotonicity [10] or the sufficient surjectivity [12,13].

Recently, we proposed an unrolling-based decision procedure for algebraic data types [10]. 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 set of its element values by the following *Set* catamorphism:

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

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 [2,3]. Experimental results with Guardol [5] 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 [10] stand out as an important class for three reasons. First, they are automatically detectable by state-of-the-art analysis tools such as SMT solvers [2,3] or theorem provers [6]. Second, they are combinable within an input formula while preserving the completeness of the decision procedure. Third, they guarantee that the decision procedure in [10] terminates after an exponentially small number of unrollings.

In this paper, we present parameterized associative-commutative (PAC) catamorphisms, a generalized class of AC ones, and show 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, as 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 [10] can be strengthened. For example, consider the *Set* catamorphism, which maps a tree into a set of all of its element values. By parameterizing the *Set* 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 strengthened *Set* catamorphism are not inherently supported by the construction of AC catamorphisms.
- *Efficiency:* In terms of efficiency, some AC catamorphisms can be written as PAC catamorphisms that prune some computational branches, leading to more efficient analysis.

The rest of the paper is organized as follows. Section 2 presents some preliminaries, including the notation of AC catamorphisms proposed in [10]. Section 3 studies some examples to demonstrate the need of PAC catamorphisms, which are then presented in detail in Section 4. Experimental results are summarized in Section 5. Next, we discuss related work in Section 6. Finally, we conclude the paper with directions for future work in Section 7.

2 Preliminaries

This section briefly summarizes the notation used for algebraic data types and catamorphisms that will be discussed in this paper.

Parametric Logic. The input to the decision procedure in [10] is a formula ϕ of literals over elements of tree terms and abstractions produced by a catamorphism. The logic is *parametric* in the sense that we assume a data type τ to be reasoned about, an element theory $\mathcal{E}$ containing element types and operations, a catamorphism α that is used to abstract the data type, and a decidable theory $\mathcal{L}_\mathcal{C}$ of values in a collection domain $\mathcal{C}$ containing terms C generated by the catamorphism function. Fig. 1 shows the syntax of the parametric logic instantiated for binary trees. Its semantics can be found in [12].

T	::=	$t \mid \mathsf{Leaf} \mid \mathsf{Node}(T, E, T) \mid \mathsf{left}(T) \mid \mathsf{right}(T)$	Tree terms
C	::=	$c \mid \alpha(T) \mid \mathcal{T}_\mathcal{C}$	$\mathcal{C}$-terms
E	::=	variables of type $\mathcal{E} \mid \mathsf{elem}(T) \mid \mathcal{T}_\mathcal{E}$	Expression
F_T	::=	$T = T \mid T \neq T$	Tree (in)equations
F_C	::=	$C = C \mid \mathcal{F}_\mathcal{C}$	Formula of $\mathcal{L}_\mathcal{C}$
F_E	::=	$E = E \mid \mathcal{F}_\mathcal{E}$	Formula of $\mathcal{L}_\mathcal{E}$
ϕ	::=	$\bigwedge F_T \wedge \bigwedge F_C \wedge \bigwedge F_E$	Conjunctions

Fig. 1. Syntax of the parametric logic

The syntax of the logic ranges over data type terms T and $\mathcal{C}$-terms of a decidable collection theory $\mathcal{L}_\mathcal{C}$. $\mathcal{T}_\mathcal{C}$ and $\mathcal{F}_\mathcal{C}$ are arbitrary terms and formulas in $\mathcal{L}_\mathcal{C}$, as are $\mathcal{T}_\mathcal{E}$ and $\mathcal{F}_\mathcal{E}$ in $\mathcal{L}_\mathcal{E}$. Tree formulas F_T describe equalities and disequalities over tree terms. Collection formulas F_C and element formulas E_C describe equalities over collection terms C and element terms E, as well as other operations ($\mathcal{F}_\mathcal{C}$, $\mathcal{F}_\mathcal{E}$) allowed by the logic of collections $\mathcal{L}_\mathcal{C}$ and elements $\mathcal{L}_\mathcal{E}$. E defines terms in the element types $\mathcal{E}$ contained within the branches of the data types. ϕ defines conjunctions of (restricted) formulas in the tree and collection theories. The ϕ terms solved by the decision procedure can be generalized to arbitrary propositional formulas through the use of a DPLL solver [4] that manages the other Boolean operators within the formula.

Catamorphisms. 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 the values stored in the tree in some recursive ways. For example, given a binary tree of integer values, we can map the tree to a number representing the summation of all the element values in the tree by the following $Sum : \tau \rightarrow \mathsf{int}$ catamorphism:

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

In this case, the type $\mathcal{E}$ of element values in the tree is int and the target type $\mathcal{C}$ in a decidable domain is also int. We can also use other catamorphisms to map the tree to values in other domains. For instance, if we use the following $\mathit{Multiset} : \tau \to \mathsf{multiset}$ catamorphism

$$\mathit{Multiset}(t) = \begin{cases} \emptyset & \text{if } t = \mathsf{Leaf} \\ \mathit{Multiset}(t_L) \uplus \{e\} \uplus \mathit{Multiset}(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \end{cases}$$

we can map a tree to a multiset (bag) of all the integer numbers stored in the tree. In this case, $\mathcal{C}$ is the $\mathsf{multiset}$ domain.

Monotonic catamorphisms. Our decision procedure in [10] has been proven to be sound and complete if the catamorphisms used in the procedure are monotonic. The formal definition of monotonic catamorphisms is below.

Definition 1 (Monotonic catamorphisms). *A catamorphism $\alpha : \tau \to \mathcal{C}$ is monotonic iff there exists a constant $h_\alpha \in \mathbb{N}^+$ such that:*

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

where $\beta(t)$ is defined[1] *as the number of trees that can map to $\alpha(t)$.*

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

In addition to the *Sum* and *Multiset* catamorphisms, some other examples of monotonic catamorphisms are *SizeI*, *Height*, *List*, *Sortedness*, *Min*, *Max*, etc. Full descriptions of these catamorphisms are in [10].

Associative-commutative (AC) catamorphisms. We also proposed in [10] AC catamorphisms, a sub-class of monotonic catamorphisms, that have some powerful properties. First, they can be automatically detectable by SMT solvers [2,3] or theorem provers [6]. Second, they can be arbitrarily combined within an input formula while preserving the completeness of the decision procedure in [10]. Third, they allow our decision procedure in [10] to terminate after an exponentially small number of unrollings.

Definition 2 (AC catamorphisms). *A catamorphism $\alpha : \tau \to \mathcal{C}$ is AC if*

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

where $\oplus : (\mathcal{C}, \mathcal{C}) \to \mathcal{C}$ is an associative and commutative binary operator with an identity element $id_\oplus \in \mathcal{C}$ (i.e., $\forall x \in \mathcal{C} : x \ \oplus \ id_\oplus = id_\oplus \ \oplus \ x = x$) and $\delta : \mathcal{E} \to \mathcal{C}$ is a function that maps[2] *an element value in $\mathcal{E}$ into a corresponding value in $\mathcal{C}$.*

[1] As $\beta(t)$ will be frequently referred to in the remainder of the paper, we give some examples of $\beta(t)$ in Appendix A.1.

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

Both the *Sum* and *Multiset* catamorphisms mentioned before are good examples of AC catamorphisms. In the *Sum* catamorphism, the operator $\oplus$ is $+$, the identity element $id_{\oplus}$ is 0, and the mapping function is $\delta(e) = e$ while in the *Multiset* catamorphism, the three factors are $\uplus, \emptyset$, and $\delta(e) = \{e\}$, respectively.

3 Motivating Examples

This section presents some motivating examples to demonstrate the advantages of PAC catamorphisms in terms of expressiveness, usability, and efficiency.

Expressiveness. We will show in Section 4 that all AC catamorphisms are PAC. Here, we demonstrate that some PAC catamorphisms are not AC. Let us assume that we have a predicate $isBad : \mathcal{E} \to \mathsf{bool}$ that determines whether an internal node is bad or not. We consider an internal node $\mathsf{Node}(\_, e, \_)$ to be bad if $isBad(e) = \mathsf{true}$. Now let us consider a catamorphism called $NGN : \tau \to \mathsf{int}$ (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:

$$NGN(t) = \begin{cases} 0 & \text{if } t = \mathsf{Leaf} \\ NGN(t_L) + 1 + NGN(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \wedge isBad(e) = \mathsf{false} \\ 0 & \text{if } t = \mathsf{Node}(t_L, e, t_R) \wedge isBad(e) = \mathsf{true} \end{cases}$$

By Corollary 5 in [10], this catamorphism is not AC because the value of $NGN(t)$ 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, this catamorphism can still be defined as a PAC catamorphism. Hence, PAC catamorphisms are more expressive than AC catamorphisms.

Usability. In [10], we discussed $Negative : \tau \to \mathsf{bool}$, an AC catamorphism that maps a tree into true if all of its element values are negative:

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

We also presented $Positive : \tau \to \mathsf{bool}$, an AC catamorphism that maps a tree into true if all of its element values are positive:

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

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 $Forall : \tau \to \mathsf{bool}$:

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

Obviously, *Forall*, a PAC catamorphism, provides a more compact and general abstraction than AC catamorphisms such as *Positive* and *Negative*.

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

$$\mathit{Forall}(t) = \begin{cases} \mathsf{true} & \text{if } t = \mathsf{Leaf} \\ \mathit{Forall}(t_L) \wedge \mathsf{true} \wedge \mathit{Forall}(t_R) & \text{if } t = \mathsf{Node}(t_L, e, t_R) \wedge pr_u(e) = \mathsf{true} \\ \mathsf{false} & \text{if } t = \mathsf{Node}(t_L, e, t_R) \wedge pr_u(e) = \mathsf{false} \end{cases}$$

Because AC catamorphisms do not have means to prune recursive computations, PAC catamorphisms can be more efficient than AC catamorphisms.

4 Parameterized Associative-Commutative Abstractions

In [10], we proposed AC catamorphisms, a sub-class of monotonic catamorphisms, that have some powerful properties: they are detectable, combinable, and only require an exponentially small number of unrollings for the decision procedure in [10]. This section presents a generalized version of AC catamorphisms and show that the new version still has all the properties of AC catamorphisms.

4.1 Definitions

Definition 3 (Parameterized Associative-Commutative (PAC) Catamorphism). *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* $\alpha : \tau \to \mathcal{C}$ *is PAC if:*

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

where $\oplus$ *is an associative and commutative operator over* $\mathcal{C}$*.*

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 $\oplus$. Second, 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 additional parameter rec, which determines 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}.

Clearly, the definition of PAC catamorphisms is more general than that of AC ones: if we set c_{leaf} to be $id_{\oplus}$ and predicate pr to always be false, a PAC catamorphism α in Definition 3 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, according to Definition 2, an AC catamorphism.

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 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 4 (Signature of PAC catamorphisms). *The signature of a PAC catamorphism α is defined as follows:*

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

Examples of PAC catamorphisms: Obviously, every AC catamorphism in [10] can be converted into a corresponding PAC catamorphism. In Table 1, we introduce some other PAC catamorphisms that cannot be naturally expressed by an AC way. They include *Forall*, *Exists*, *Member*, and *NGN* catamorphisms. We have briefly discussed *Forall* and *NGN* in Section 3.

Table 1. Some PAC catamorphisms

Name	$\mathcal{C}$	$\oplus$	$\delta(e)$	c_{leaf}	c_{pr}	pr	rec	Value of the catamorphism
Forall	bool	$\wedge$	$pr_u(e)$	true	false	$\neg pr_u$	true/false	true if all element values satisfy predicate pr_u
Exists	bool	$\vee$	$pr_u(e)$	false	true	pr_u	true/false	true if $\exists$ an element value satisfies predicate pr_u
Member	bool	$\vee$	$(e = x)$	false	true	$(e = x)$	true/false	true if x is a member of the tree
NGN	int	$+$	1	0	0	$isBad$	false	number of good nodes

Detection: Like AC catamorphisms, PAC catamorphisms are automatically detectable. A catamorphism written in the format in Definition 3 is PAC if $\oplus$ is an associative and commutative operator over the collection domain $\mathcal{C}$. We can simply use SMT solvers [2,3] or theorem provers [6] to check the associativity and commutativity of operator $\oplus$.

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 $\mathsf{rec} = \mathsf{false}$, on the other hand, the value of $\alpha(t)$ might or might 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, if there does not exist any element value $e_t \in t$ such that $\mathsf{pr}(e_t) = \mathsf{true}$, the value of $\alpha(t)$ is independent of the structure of the tree because the computation of $\alpha(t)$ now reduces 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*

- $\mathsf{rec} = \mathsf{true}$*, or*
- $\mathsf{rec} = \mathsf{false}$ *and there does not exist any element value in t that can make predicate* pr *hold.*

4.2 PAC Catamorphisms are Monotonic

To work with our unrolling-based decision procedure for algebraic data types in [10], PAC catamorphisms must be monotonic. In this section, we prove the monotonicity[3] of PAC catamorphisms. To prove that, we need to introduce some new supporting lemmas and corollaries.

Definition 5 (Satisfiability of Predicate). *Predicate* $\mathsf{pr} : \mathcal{E} \to \mathsf{bool}$ *is satisfiable if and only if*

$$\exists e \in \mathcal{E} : \mathsf{pr}(e) = \mathsf{true}$$

Lemma 1. *Given a PAC catamorphism α with* $\mathsf{rec} = \mathsf{false}$*, if predicate* pr *is satisfiable, then*

$$|\alpha^{-1}(c_{\mathsf{pr}})| = \infty$$

Proof. Since pr is satisfiable, from Definition 5, 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 given any tree $t \in \tau$, if there exists an element value e_t in t such that* $\mathsf{pr}(e_t) = \mathsf{true}$*, we have $\beta(t) = \infty$.*

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 3. 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 [10]. □

[3] The definition of monotonicity is in Definition 1 in Section 2.

Corollary 3. *Given a PAC catamorphism α with* $\mathsf{rec} = \mathsf{false}$ *and given any tree $t \in \tau$, we have 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 immediately from Corollary 2. □

Lemma 2. *If α is a PAC catamorphism then*

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

where $size(t)$ is the total number of vertices in tree t and $ns(s)$ is the number of shapes[4] *of size s.*

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 [10] 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. On the other hand, consider the case when $\beta(t) < \infty$. 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 [10]. □

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 $height(t) \geq h_\alpha = 4$. If $\beta(t) = \infty$, the monotonic condition for t in Definition 1 holds.

Suppose on the other hand that $\beta(t) < \infty$. By Lemma 4 in [10], there exists t_0 such that $t_0 \precneqq t \wedge height(t_0) = height(t) - 1 \geq 3$, where $\precneqq$ is the notion of strict subtrees in [10]. 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 $e_1, \ldots, e_{|Q|}$ be the elements stored in $|Q|$ nodes in Q. We define a new mapping function $\delta' : \mathcal{E} \to \mathcal{C}$ 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:

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

[4] In [10], we showed that $ns(s)$ is, intuitively, the number of full binary trees of size s.

Next, we construct a tree t_Q from $e_1, \ldots, e_{|Q|}$. The construction of t_Q is shown in Fig. 2. Let node_i, where $1 \leq i \leq |Q|$, be the node corresponding to e_i in t_Q. We build t_Q in a bottom-up fashion as follows: $\mathsf{node}_{|Q|} = \mathsf{Node}(\mathsf{Leaf}, e_{|Q|}, \mathsf{Leaf})$ and $\mathsf{node}_j = \mathsf{Node}(\mathsf{node}_{j+1}, e_j, \mathsf{Leaf})$, where $Q > j \geq 1$. Let leaf_{Q1} and leaf_{Q2} be the left and right leaves of $\mathsf{node}_{|Q|}$, respectively.

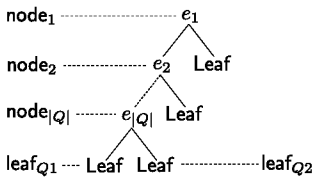


Fig. 2. Construct t_Q

By Property 3 in [10], $height(t_0) \geq 3$ implies $size(t_0) \geq 7$. By Lemma 2 in [10], $ns\big(size(t_0)\big) \geq ns(7) > 2$. By Lemma 2, $\beta(t_0) \geq ns\big(size(t_0)\big) > 2$. 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

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

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

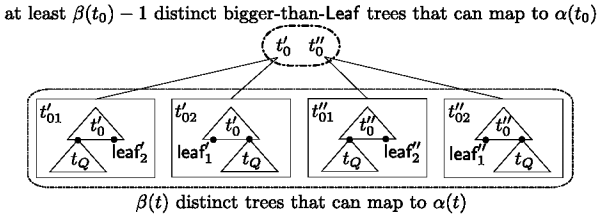


Fig. 3. Relationship between t'_0, t''_0 and $t'_{01}, t'_{02}, t''_{01}, t''_{02}$

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. Their relationship is shown in Fig. 3.

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,

$$\beta(t) \geq 2 \times \big(\beta(t_0) - 1\big) > \beta(t_0) \qquad [\text{Since } \beta(t_0) > 2]$$

As a result, α is monotonic based on Definition 1. □

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.
   (b) If $\nexists e_t \in t : \mathsf{pr}(e_t) = \mathsf{true}$, we show that there exists $t_0 \in \tau$ such that $height(t_0) = height(t) - 1$ and $\beta(t_0) < \beta(t)$. As a result, the monotonic condition for α holds.

Now, let us 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 according to Lemma 3. Therefore, α must be 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 [10], 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

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

Next, we construct a tree t_Q from $e_1, \ldots, e_{|Q|}$ as in Fig. 2. Given t_Q, by replacing leaf_{Q1} with t'_0, we obtain a distinct tree t'_{01} such that:

$$\alpha(t'_{01}) = \alpha(t'_0) \oplus \delta(e_1) \oplus \delta(e_2) \oplus \ldots \oplus \delta(e_{|Q|}) \oplus \underbrace{c_{\mathsf{leaf}} \oplus \ldots \oplus c_{\mathsf{leaf}}}_{|Q| \text{ occurrences of } c_{\mathsf{leaf}}} \quad (4)$$

From Equations (3) and (4), 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.

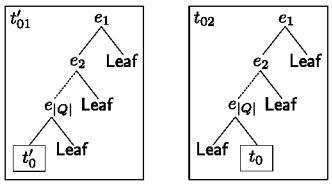


Fig. 4. The constructions of t'_{01} and t_{02}.

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. 4. We have:

$$\alpha(t_{02}) = \alpha(t_0) \oplus \delta(e_1) \oplus \delta(e_2) \oplus \ldots \oplus \delta(e_{|Q|}) \oplus \underbrace{c_{\mathsf{leaf}} \oplus \ldots \oplus c_{\mathsf{leaf}}}_{|Q| \text{ occurrences of } c_{\mathsf{leaf}}}$$

$$= \alpha(t) \qquad [\text{ From Equation (3) }]$$

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,

$$\beta(t_0) + 1 \leq \beta(t)$$
$$\therefore \qquad \beta(t_0) < \beta(t)$$

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. □

Theorem 1. *PAC catamorphisms are monotonic.*

Proof. The theorem follows from Lemmas 3 and 4. □

Remark 1. Theorem 1 is a generalized version of Theorem 7 in [10].

4.3 Exponentially Small Upper Bound of the Number of Unrollings

Since PAC catamorphisms are monotonic, they can be used in the decision procedure in [10]. 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 [10]; the only difference is that we use Lemma 2 in this paper to generalize the result for PAC catamorphisms instead of Lemma 8 in [10], which only works for AC catamorphisms.

4.4 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 ($1 \leq i \leq m$) 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 $\langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_m \rangle \in \mathcal{C}$:

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

- $\delta : \mathcal{E} \to \mathcal{C}$ is defined as follows: $\delta(e) = \langle \delta_1(e), \delta_2(e), \ldots, \delta_m(e) \rangle$
- $c_{\mathsf{leaf}} : \mathcal{C}$ 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 2. *Every catamorphism obtained from the combination of PAC catamorphims is also PAC.*

Proof. Provided in Appendix A.2. □

Remark 2. Theorem 2 is a generalized version of Theorem 8 in [10].

5 Experimental Results

We have implemented support for PAC catamorphisms in RADA, an implementation of our unrolling-based decision procedure for algebraic data types in [10]. We have also evaluated the tool with a collection of 10 benchmark examples listed in Table 2. Each example contains verification conditions related to PAC catamorphisms and has 80–110 lines of code written in a format similar to SMT-Lib 2.0 [1]. The results are very promising: all of the benchmark examples were automatically verified by RADA in a short amount of time.

The PAC catamorphisms used in the benchmarks are *Forall*, *Exists*, *Member*, and *NGN*, which have been introduced in Table 1. Each example in the first 8 benchmarks in Table 2 only involves one catamorphism. On the contrary, the last

Table 2. Experimental results

Benchmark	Catamorphism	Result	# unrollings [10]	Time (s)
forall01	*Forall*	sat	7	1.250
forall02		unsat	6	0.870
exists01	*Exists*	sat	1	0.256
exists02		unsat	1	0.285
member01	*Member*	sat	5	0.634
member02		unsat	6	0.797
ngn01	*NGN*	sat	7	1.173
ngn02		unsat	4	0.482
ngn_ngn01	Combination of two	sat	7	1.381
ngn_ngn02	PAC catamorphisms	unsat	4	0.646

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 4.4.

All benchmarks were run on a Ubuntu machine using an Intel Core I5 running at 2.8 GHz with 4GB RAM. RADA, all the benchmarks in this paper, and other Guardol [5] benchmarks are available at `http://crisys.cs.umn.edu/rada`.

6 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 [10]. Unlike monotonic catamorphisms, sufficiently surjective catamorphisms are difficult to automatically detect and in general cannot be composed in a decidable way.

Madhusudan et al. [9] 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} [8], are even more limited.

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 [7] is used. However, their method can work with high-order functions while ours cannot.

7 Conclusion

This paper presents parameterized associative-commutative (PAC) catamorphisms, a generalized version of associative-commutative (AC) catamorphisms

[10]. 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 [10]. 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 [10] 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 [10].

References

1. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT*, 2010.
2. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.
3. Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.
4. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *CAV*, pages 175–188, 2004.
5. David Hardin, Konrad Slind, Michael Whalen, and Tuan-Hung Pham. The Guardol Language and Verification System. In *TACAS*, pages 18–32, 2012.
6. M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Springer, 2000.
7. Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate Abstraction and CEGAR for Higher-Order Model Checking. In *PLDI*, pages 222–233, 2011.
8. P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable Logics Combining Heap Structures and Data. In *POPL*, pages 611–622, 2011.
9. Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive Proofs for Inductive Tree Data-Structures. In *POPL*, pages 123–136, 2012.
10. Tuan-Hung Pham and Michael Whalen. An Improved Unrolling-Based Decision Procedure for Algebraic Data Types. In *VSTTE*, 2013.
11. Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. Towards a Scalable Software Model Checker for Higher-Order Programs. In *PEPM*, pages 53–62, 2013.
12. Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *POPL*, pages 199–210, 2010.
13. Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability Modulo Recursive Programs. In *SAS*, pages 298–315, 2011.
14. Karen Zee, Viktor Kuncak, and Martin Rinard. Full Functional Verification of Linked Data Structures. In *PLDI*, pages 349–361, 2008.
15. Karen Zee, Viktor Kuncak, and Martin C. Rinard. An Integrated Proof Language for Imperative Programs. In *PLDI*, pages 338–351, 2009.

A Appendix

A.1 Examples of $\beta(t)$

We always have $\beta(\mathsf{Leaf}) = 1$, regardless of what catamorphism α is. The value of $\beta(t_{\neq \mathsf{Leaf}})$, where $t_{\neq \mathsf{Leaf}} \neq \mathsf{Leaf}$, depends on the definition of α. For example, if α is the *Set* catamorphism, $\beta(t_{\neq \mathsf{Leaf}}) = \infty$ since there are an infinite number of trees that have the same set of element values.

Table 3. Examples of $\beta(t)$ with the *Multiset* catamorphism

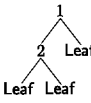
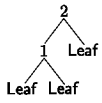
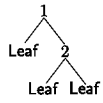
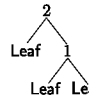
t					
$\alpha(t)$	$\{1\}$	$\{1,2\}$	$\{1,2\}$	$\{1,2\}$	$\{1,2\}$
$\beta(t)$	1	4	4	4	4

Table 3 shows some examples of $\beta(t)$ with the *Multiset* catamorphism. The value of β(first tree) is 1 because it is the only tree that can map to $\{1\}$ by catamorphism *Multiset*. The last four trees are distinct and they are all the trees that can map to the same multiset $\{1,2\}$; hence, their $\beta(t)$ are equal to 4.

A.2 Proof of Theorem 2

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 3. 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:

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

□