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:

\[ \text{Multiset}(\text{Leaf}) = \emptyset \]
\[ \text{Multiset}(\text{Node}(t_L, e, t_R)) = \text{Multiset}(t_L) \uplus \{ e \} \uplus \text{Multiset}(t_R) \]

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 [11, 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 [11, 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., \( t = \text{Leaf} \)), two parameters for the recursive case (i.e., \( t = \text{Node} \)), 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 [3]) 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 \( \tau \) be a tree domain, in which each vertex can be a Leaf or a 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 \( \text{size}(t) \) as the total number of vertices in \( t \). Given a tree in the tree domain \( \tau \), 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 IV is a catamorphism that maps a tree in \( \tau \) 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 \( \text{DW} : \tau \rightarrow \text{int catamorphism} \):

\[
\text{DW}(\text{Leaf}) = 0 \\
\text{DW}(\text{Node}(t_L, e, t_R)) = \text{DW}(t_L) + (\text{int}(e) \cdot 1) + \text{DW}(t_R)
\]

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](image)

**Example 2 (Unrolling-based decision procedure):** For our guard example, suppose one of the VCs is: \( t = \text{Node}(t_L, e, t_R) \wedge \text{dirty}(e) \wedge \text{DW}(t) = 0 \). 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, \( \text{DW}(t) \) is treated as an uninterpreted function \( \text{UF}^\geq_0^\tau \rightarrow \text{int} \), which can return any value of type int (i.e., the codomain of \( \text{DW} \)) bigger or equal to 0 (i.e., the range of \( \text{DW} \)). The use of \( \text{UF}^\geq_1^\tau \) implies that for the first step we do not use control conditions. The formula becomes \( t = \text{Node}(t_L, e, t_R) \wedge \text{dirty}(e) \wedge \text{DW}(t) = 0 \wedge \text{DW}(t) = \text{UF}^\geq_0^\tau \) and is SAT. However, the SAT result is untrustworthy due to the presence of \( \text{UF}^\geq_1^\tau \); thus, we continue unrolling \( \text{DW}(t) \).

At unrolling depth 1, we allow \( \text{DW}(t) \) to be unrolled up to depth 1 and all the catamorphism applications at lower depths will be treated as uninterpreted functions. In particular, \( \text{UF}^\geq_1^\tau \) and \( \text{UF}^\geq_2^\tau \) are the uninterpreted functions for \( \text{DW}(t_L) \) and \( \text{DW}(t_R) \), respectively. The set of control conditions in this case is \( \{ \neg (t \neq \text{Leaf}) \} \). When we use the control conditions, \( \text{UF}^\geq_2^\tau \) and \( \text{UF}^\geq_3^\tau \) will not be used and the formula becomes \( t = \text{Node}(t_L, e, t_R) \wedge \text{dirty}(e) \wedge \text{DW}(t) = 0 \wedge \text{DW}(t) = 0 \wedge t = \text{Leaf} \), 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 \( \beta \)):** Given a catamorphism \( \alpha : \tau \rightarrow C \), we define \( \beta(t) : \tau \rightarrow \mathbb{N} \) as the cardinality of the inverse
function of \( \alpha(t) \):

\[
\beta(t) = |\alpha^{-1}(\alpha(t))|.
\]

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

A catamorphism \( \alpha \) 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 \( \alpha : \tau \rightarrow C \) is monotonic iff there exists a constant \( h_\alpha \in \mathbb{N}^+ \) such that:

\[
\forall t \in \tau : \text{height}(t) \geq h_\alpha \Rightarrow (\beta(t) = \infty \lor \exists t_0 \in \tau : \text{height}(t_0) = \text{height}(t) - 1 \land \beta(t_0) < \beta(t)).
\]

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

D. **Associative-commutative (AC) Catamorphisms**

AC catamorphisms are a powerful sub-class of monotonic catamorphisms. First, they can be detected by SMT solvers \[11, 13\] or theorem provers \[15\]. 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 \( \oplus : (C, C) \rightarrow C \) be an associative and commutative binary operator with an identity element \( id_\oplus \in C \) (i.e., \( \forall x \in C : x \oplus id_\oplus = id_\oplus \oplus x = x \)) and \( \delta : \mathcal{E} \rightarrow C \) be a function that maps an element value in \( \mathcal{E} \) into a value in \( C \). We define AC catamorphisms as follows:

**Definition 3 (AC catamorphisms)**: A catamorphism \( \alpha : \tau \rightarrow C \) is AC if:

\[
\alpha(t) = \begin{cases} 
\text{id}_\oplus & \text{if } t = \text{Leaf} \\
\alpha(t_L) \oplus \delta(e) \oplus \alpha(t_R) & \text{if } t = \text{Node}(t_L, e, t_R)
\end{cases}
\]

**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 \( \delta(e) = (\text{ite}(\text{dirty(e)}) \oplus 1)0 \) while in the Multiset catamorphism, the three factors are \( \ominus, \emptyset, \) 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 \( pr : \mathcal{E} \rightarrow \text{bool} \), a value \( c_{\text{leaf}} \in C \), a value \( c_{\text{pr}} \in C \), and a boolean value \( \text{rec} \), catamorphism \( \alpha : \tau \rightarrow C \) is PAC if:

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

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

**Signature.** Due to the generalization, the signature of PAC catamorphisms has four more elements than that of AC catamorphisms, including the value \( c_{\text{leaf}} \in C \) for the Leaf case, the value \( c_{\text{pr}} \in C \) for the recursive case when the predicate \( pr \)
**Definition 5 (PAC signature):** The signature of a PAC catamorphism \( \alpha \) is:

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

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

If \( \text{rec} = \text{false} \), the value of \( \alpha(t) \) may or may not depend on the structure of the tree. If there exists an element value \( e_t \) such that \( \text{pr}(e_t) = \text{true} \), the value of \( \alpha(t) \) is dependent on 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

\[
\alpha(t) = \begin{cases} 
  c_{\text{leaf}} & \text{if } t = \text{Leaf} \\
  (\alpha(t_L) + \delta(e) + \alpha(t_R)) & \text{if } t = \text{Node}(t_L, e, t_R) 
\end{cases}
\]

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 \( \alpha \) 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) \( \text{rec} = \text{true} \) or (2) \( \text{rec} = \text{false} \) and \( \exists \) element value \( e \in \mathcal{E} \) in \( t \) such that \( \text{pr}(e) = \text{true} \).

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 \( \alpha \) as defined in Definition [4] if we fix \( c_{\text{leaf}} \) to be \( id_{\oplus} \) and predicate \( \text{pr} \) to be false, \( \alpha \) becomes:

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

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 \( pr, c_{\text{leaf}}, c_{\text{pr}}, \text{rec} \), 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

\[
\text{NLeaves}(\text{Leaf}) = 1 \\
\text{NLeaves}(\text{Node}(t_L, e, t_R)) = \text{NLeaves}(t_L) + \text{NLeaves}(t_R)
\]

which maps a tree to its number of leaves. Because 1 is not an identity element of operator \( \oplus \), \( \text{NLeaves} \) 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 special 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 \( \text{isBad} : \mathcal{E} \rightarrow \text{bool} \) that determines whether an internal node is bad. We consider an internal node \( \text{Node}(\_ e \_ ) \) to be bad if \( \text{isBad}(e) = \text{true} \). Now consider a catamorphism called \( \text{NGN} : \tau \rightarrow \text{int} \) (number of good nodes), which maps a tree into the number of “good” internal nodes (1) that are not bad and (2) are not descendants of any bad nodes. We can define the catamorphism as follows:

\[
\text{NGN}(\text{Leaf}) = 0 \\
\text{NGN}(\text{Node}(t_L, e, t_R)) = \begin{cases} 
  \text{NGN}(t_L) + 1 + \text{NGN}(t_R) & \text{if } \neg \text{isBad}(e) \\
  0 & \text{if } \text{isBad}(e)
\end{cases}
\]

By Corollary 5 in [9], this catamorphism is not AC because the value of \( \text{NGN}(t) \) 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 Negative : \( \tau \rightarrow \text{bool} \), an AC catamorphism that maps a tree into true if all of its element values are negative:

\[
\begin{align*}
\text{Negative}(\text{Leaf}) &= \text{true} \\
\text{Negative}(\text{Node}(t_L, e, t_R)) &= \text{Negative}(t_L) \land (e < 0) \land \text{Negative}(t_R)
\end{align*}
\]

Similarly, we can define the AC catamorphism Positive : \( \tau \rightarrow \text{bool} \) as follows:

\[
\begin{align*}
\text{Positive}(\text{Leaf}) &= \text{true} \\
\text{Positive}(\text{Node}(t_L, e, t_R)) &= \text{Positive}(t_L) \land (e > 0) \land \text{Positive}(t_R)
\end{align*}
\]

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 \( pr_u : \mathcal{E} \rightarrow \text{bool} \), these catamorphisms (as well as many others) can be defined by a single parametric catamorphism Forall : \( \tau \rightarrow \text{bool} \):

\[
\begin{align*}
\text{Forall}(\text{Leaf}) &= \text{true} \\
\text{Forall}(\text{Node}(t_L, e, t_R)) &= \text{Forall}(t_L) \land pr_u(e) \land \text{Forall}(t_R)
\end{align*}
\]

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 higher-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 \( \alpha_{\text{AC}} \) and a PAC catamorphism \( \alpha_{\text{PAC}} \), for every tree \( t \in \tau \) that the two catamorphisms accept as input, \( \alpha_{\text{PAC}}(t) \) requires less or equal number of recursive calls to compute its value than \( \alpha_{\text{AC}}(t) \).

**Proof:** For AC catamorphisms, when \( t = \text{Node}(t_L, e, t_R) \), \( \alpha_{\text{AC}}(t) \) is always computed in terms of \( \alpha_{\text{AC}}(t_L) \) and \( \alpha_{\text{AC}}(t_R) \), which in turn will be computed in terms of
\( \alpha_{AC} \) (their sub-trees). Hence, to compute \( \alpha_{AC}(t) \), the total number of function calls we need to make to \( \alpha_{AC} \) is equal to \( \text{size}(t) \). For PAC catamorphisms, on the other hand, when \( t = \text{Node}(t_L, e, t_R) \), \( \alpha_{PAC}(t) \) might or might not need to call \( \alpha_{PAC}(t_L) \) and \( \alpha_{PAC}(t_R) \), depending on the value of \( \text{pr}(e) \). Thus, the total number of function calls to \( \alpha_{PAC} \) to compute \( \alpha_{PAC}(t) \) is at most \( \text{size}(t) \).

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

\[
\text{Forall} = \text{true} \\
\text{Forall} = \begin{cases} 
\text{Forall}(t_L) \land \text{Forall}(t_R) & \text{if } \text{pr}_u(e) \\
\text{false} & \text{if } \neg \text{pr}_u(e)
\end{cases}
\]

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 catamorph 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 domain collection \( \mathcal{C} \). We can use SMT solvers [11] 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 \textit{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 catamorphism \( (1 \leq i \leq m) \) is \( \text{sig}(\alpha_i) = (\mathcal{C}_i, \mathcal{E}_i, \oplus, \delta_i, \text{leaf}_i, \text{pr}_i, \text{rec}_i) \). Catamorphism \( \alpha \) with signature \( \text{sig}(\alpha) = (\mathcal{C}, \mathcal{E}, \oplus, \delta, \text{leaf}, \text{pr}, \text{rec}) \) is a combination of \( \alpha_1, \ldots, \alpha_m \) if

- \( \mathcal{C} \) is the domain of \( m \)-tuples, where the \( i \)-th element of each tuple is in \( \mathcal{C}_i \).
- \( \oplus : (\mathcal{C}, \mathcal{C}) \rightarrow \mathcal{C} \) is defined as follows, given \( \langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_m \rangle \in \mathcal{C} \):
  \[ \langle x_1, \ldots, x_m \rangle \oplus \langle y_1, \ldots, y_m \rangle = \langle x_1 \oplus y_1, \ldots, x_m \oplus y_m \rangle \]
- \( \delta : \mathcal{E} \rightarrow \mathcal{C} \) is defined as follows:
  \[ \delta(e) = \langle \delta_1(e), \delta_2(e), \ldots, \delta_m(e) \rangle \]
- \( \text{leaf} : \mathcal{C} \) is defined as follows:
  \[ \text{leaf} = \langle \text{leaf}_1, \text{leaf}_2, \ldots, \text{leaf}_m \rangle \]
- \( \text{pr} : \mathcal{C} \) is defined as follows:
  \[ \text{pr} = \langle \text{pr}_1, \text{pr}_2, \ldots, \text{pr}_m \rangle \]

\textbf{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 \textit{forall}, \textit{exists}, and \textit{member}. Each of the first 10 benchmarks in Table II only involves one catamorphism. Catamorphisms \textit{NLeaves}, \textit{Forall} and \textit{NGN} have been introduced in Section IV. Catamorphism \textit{Exists} maps a tree into true if the tree contains at least one element value that satisfies a user-provided predicate \( \text{pr}_u \) while catamorphism \textit{Member} maps a tree into true if the tree contains a user-provided value \( x \). The last two examples consist of the combination of \textit{NGN} and a slightly modified version of the catamorphism to demonstrate the combinability of PAC catamorphisms as discussed in Section V.
### TABLE II: Experimental results with PAC catamorphisms

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Result</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall01</td>
<td>sat</td>
<td>0.352</td>
</tr>
<tr>
<td>forall02</td>
<td>unsat</td>
<td>0.246</td>
</tr>
<tr>
<td>exists01</td>
<td>sat</td>
<td>0.046</td>
</tr>
<tr>
<td>exists02</td>
<td>unsat</td>
<td>0.048</td>
</tr>
<tr>
<td>member01</td>
<td>sat</td>
<td>0.167</td>
</tr>
<tr>
<td>member02</td>
<td>unsat</td>
<td>0.257</td>
</tr>
<tr>
<td>nleaves01</td>
<td>sat</td>
<td>0.332</td>
</tr>
<tr>
<td>nleaves02</td>
<td>unsat</td>
<td>0.161</td>
</tr>
<tr>
<td>ngn01</td>
<td>sat</td>
<td>0.428</td>
</tr>
<tr>
<td>ngn02</td>
<td>unsat</td>
<td>0.113</td>
</tr>
</tbody>
</table>

**Single PAC catamorphisms**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Result</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>member01</td>
<td>sat</td>
<td>0.167</td>
</tr>
<tr>
<td>member02</td>
<td>unsat</td>
<td>0.257</td>
</tr>
<tr>
<td>nleaves01</td>
<td>sat</td>
<td>0.332</td>
</tr>
<tr>
<td>nleaves02</td>
<td>unsat</td>
<td>0.161</td>
</tr>
<tr>
<td>ngn01</td>
<td>sat</td>
<td>0.428</td>
</tr>
<tr>
<td>ngn02</td>
<td>unsat</td>
<td>0.113</td>
</tr>
</tbody>
</table>

**Combination of PAC catamorphisms**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Result</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ngn_ngn01</td>
<td>sat</td>
<td>0.556</td>
</tr>
<tr>
<td>ngn_ngn02</td>
<td>unsat</td>
<td>0.157</td>
</tr>
</tbody>
</table>

### TABLE III: Experimental results on Guardol benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Result</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email_Guard_Correct_All</td>
<td>17 unsats</td>
<td>≈ 0.009/obligation</td>
</tr>
<tr>
<td>RBTree.Black_Property</td>
<td>12 unsats</td>
<td>≈ 2.142/obligation</td>
</tr>
<tr>
<td>RBTree.Red_Property</td>
<td>12 unsats</td>
<td>≈ 0.163/obligation</td>
</tr>
<tr>
<td>array_checksum.SumListAdd</td>
<td>2 unsats</td>
<td>≈ 0.028/obligation</td>
</tr>
<tr>
<td>array_checksum.SumListAdd_Alt</td>
<td>13 unsats</td>
<td>≈ 0.012/obligation</td>
</tr>
</tbody>
</table>

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 [13], [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 declare, a decidable fragment of DRYAD that can be embedded into the decidable logic STRAND [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 [9] 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 $R_\alpha$, 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

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 $pr : E \rightarrow \text{bool}$ is satisfiable if $\exists e \in E : pr(e) = true$.

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

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

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

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

Corollary 3: Given a PAC catamorphism $\alpha$ with $rec = false$ 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 $pr(e_{t'}) = true$.

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 $SNode\langle \_ , \_ \rangle$ as follows:

\[ shape(Leaf) = SLeaf \]

\[ shape(Node(t_L, \_ , t_R)) = SNode(shape(t_L), shape(t_R)) \]

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 $\tau$, tree $t_1$ is a subtree of tree $t_2$, denoted by $t_1 \preceq t_2$, iff:

\[ t_1 = Leaf \lor t_1 = Node(t_{1L}, e, t_{1R}) \land t_2 = Node(t_{2L}, e, t_{2R}) \land t_{1L} \preceq t_{2L} \land t_{1R} \preceq t_{2R} \]

Tree $t_1$ is a strict subtree of $t_2$, denoted by $t_1 \preceq t_2$, iff $t_1 \preceq t_2 \land size(t_1) < size(t_2)$.

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 $\alpha$ is a PAC catamorphism then $\forall t \in \tau : \beta(t) \geq ns(size(t))$.

Proof: Let $t$ be any tree in $\tau$. If $rec = true$, 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 $rec = false$, 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 2 there does not exist any element value $e_t$ in $t$ such that $pr(e_t) = true$. 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 $rec = true$ and the other one is for PAC catamorphisms with $rec = false$.

Lemma 3: PAC catamorphisms with $rec = true$ are monotonic.

Proof: Let $\alpha$ be a PAC catamorphism with $rec = true$. Let $h_{a} = 4$. Consider any tree $t \in \tau$ such that $height(t) \geq h_{a} = 4$. 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], $3t_0 = \exists t : t_0 \preceq t \land height(t_0) = height(t) - 1 \geq 3$. Let $Q$ be the set of internal nodes that are in $t$ but not in $t_0$. $Q$ is not empty since $t_0 \preceq t$. Let $e_1, \ldots, e_{|Q|}$ be the elements stored in $|Q|$ nodes in $Q$. We define a new mapping function as follows:

\[
\delta'(e) = \begin{cases} 
\delta(e) & \text{if } pr(e) = false \\
\ e_{pr} & \text{if } pr(e) = true
\end{cases}
\]

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

\[
\alpha(t) = \alpha(t_0) + \delta'(e_1) + \delta'(e_2) + \ldots + \delta'(e_{|Q|})
\]

(1)

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: $node_{|Q|} = Node(Leaf, e_{|Q|}, Leaf)$ and $node_{j} = Node(node_{j+1}, e_j, Leaf)$, where $Q \geq j \geq 1$. Let $leaf_{Q1}$ and $leaf_{Q2}$ be the two leaves of $node_{|Q|}$.

By Property 3 in [9], $height(t_0) \geq 3$ implies $size(t_0) \geq 7$. By Lemma 2 in [9], $ns(size(t_0)) \geq ns(7) > 2$. By Lemma 2 $\beta(t_0) \geq ns(size(t_0)) > 2$. 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

$$\alpha(t'_0) = \alpha(t''_0) = \alpha(t_0)$$

(2)

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

Consider $t'_0$. Let leaf $\ell'_1$ and leaf $\ell'_2$ 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 $\ell'_1$ and leaf $\ell'_2$ in $t'_0$ with $t_Q$, respectively. Since $t_Q \neq \text{Leaf}$, we have $t'_{01} \neq t'_{02}$. We have

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

$$= \alpha(t'_0) + \delta'(|e_{t_Q}|)$$

(3)

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 $\alpha(t)$.

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 $t'_{01}, t'_{02}, t''_{01}, t''_{02}$ 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 $2 \times (\beta(t_0) - 1)$ distinct trees that can map to $\alpha(t)$. Hence, $\beta(t) \geq 2 \times (\beta(t_0) - 1)$, which leads to $\beta(t) > \beta(t_0)$ since $\beta(t_0) > 2$. As a result, $\alpha$ is monotonic based on Definition 2.

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

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

1) If $\text{pr}$ is unsatisfiable, catamorphism $\alpha$ is also a PAC catamorphism with $\text{rec} = \text{true}$. Thus, $\alpha$ is monotonic from Lemma 3.

2) On the other hand, if $\text{pr}$ is satisfiable, consider any tree $t \in \tau$ of height at least $h_{\alpha} = 2$. There are two sub-cases as follows.

a) If $\exists e \in t : \text{pr}(e) = \text{true}$, we show that $\beta(t) = \infty$, which implies the monotonicity of $\alpha$ by Definition 2.

b) If $\forall e \in t : \text{pr}(e) = \text{true}$, we show that $\exists t_0 \in \tau$ such that $\text{height}(t_0) = \text{height}(t) - 1$ and $\beta(t_0) < \beta(t)$. Hence, $\alpha$ is monotonic by Definition 2.

We now present the proof in detail. If predicate $\text{pr}$ is unsatisfiable, the definition of the PAC catamorphism $\alpha$ can be rewritten as follows:

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

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

On the other hand, consider the case when predicate $\text{pr}$ is satisfiable. We will prove that $\alpha$ 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 $\text{pr}(e_t) = \text{true}$. From Corollary 2, $\beta(t) = \infty$. Therefore, the monotonic condition holds for $t$.]

Let $Q$ be the collection of internal nodes that are in $t$ but not in $t_0$. $Q$ is not empty since $t_0 \not\subseteq t$. 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 $\text{pr}$ hold. Therefore, we have

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

$$+ \text{leaf} + \ldots + \text{leaf}$$

(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 $\alpha$. Note that $t_0$ is also in this collection. Hence, we have

$$\alpha(t'_0) = \alpha(t_0)$$

(4)

Next, we construct a tree $t_Q$ from $e_1, e_2, \ldots, e_{|Q|}$ as in Fig. 3. Given $t_Q$, by replacing leaf $Q_1$ with $t'_0$, we obtain a distinct tree $t'_{01}$ such that:

$$\alpha(t_{01}) = \alpha(t'_0) + \delta(e_1) + \delta(e_2) + \ldots + \delta(e_{|Q|})$$

$$+ \text{leaf} + \ldots + \text{leaf}$$

(5)

From Equations (3), (4), and (5), we have: $\alpha(t) = \alpha(t_{01})$. Thus, from each tree $t'_{01}$ 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 $\text{leaf}_{Q_2}$ be the set of $\beta(t_0)$ distinct trees that can map to $\alpha(t)$ generated by the substitutions of leaf $Q_1$ in $t_Q$ as discussed before. Obviously, leaf $Q_2$ exists in all the trees in $\text{leaf}_{Q_2}$ since leaf $Q_2$ is untouched during the substitution process.
Next, we show that there exists at least another tree that can map to \( \alpha(t) \) but is not in \( B_{\text{leaf}Q_1} \). Given \( t_Q \), we now replace \( \text{leaf}_{Q_2} \) with \( t_0 \) to obtain a tree \( t_{02} \). The constructions of \( t'_{01} \) and \( t_{02} \) are shown in Fig. 5. We have:

\[
\alpha(t_{02}) = \alpha(t_0) \oplus \delta(e_1) \oplus \delta(e_2) \oplus \ldots \oplus \delta(e_{|Q|})
\]

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

Thus, \( t_{02} \) is also a tree that can map to \( \alpha(t) \). Since \( \text{height}(t_0) \geq 1 \), \( t_0 \) must not be a Leaf tree. Therefore, by replacing \( \text{leaf}_{Q_2} \) in \( t_Q \) with \( t_0 \) to obtain \( t_{02} \), \( \text{leaf}_{Q_2} \) must not be in \( t_{02} \). Moreover, since \( \text{leaf}_{Q_2} \) is in all the trees in \( B_{\text{leaf}Q_1} \), tree \( t_{02} \) is different from all the trees in \( B_{\text{leaf}Q_1} \). Thus, there are at least \( \beta(t_0) + 1 \) distinct trees that can map to \( \alpha(t) \), including \( t_{02} \) and those in \( B_{\text{leaf}Q_1} \). In other words,

\[
\beta(t_0) + 1 \leq \beta(t)
\]

\[
\therefore \quad \beta(t_0) < \beta(t)
\]

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

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

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

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

\[
\langle \langle x_1, \ldots, x_m \rangle \oplus \langle y_1, \ldots, y_m \rangle \rangle \oplus \langle z_1, \ldots, z_m \rangle \\
= \langle x_1 \oplus_1 y_1, \ldots, x_m \oplus_m y_m \rangle \oplus \langle z_1, \ldots, z_m \rangle \\
= \langle x_1, \ldots, x_m \rangle \oplus \langle y_1 \oplus_1 z_1, \ldots, y_m \oplus_m z_m \rangle \\
= \langle x_1, \ldots, x_m \rangle \oplus \langle y_1, \ldots, y_m \rangle \oplus \langle z_1, \ldots, z_m \rangle
\]