

Verification of Recursive Data Types using Abstractions

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA

BY

Hung Tuan Pham

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Michael W. Whalen and Mats P.E. Heimdahl

March, 2014

© Hung Tuan Pham 2014
ALL RIGHTS RESERVED

Acknowledgements

I would like to thank all the people who contributed directly or indirectly to the work in this dissertation. First and foremost, I am deeply indebted to my advisors, Dr. Michael W. Whalen and Dr. Mats P.E. Heimdahl, for all the support and encouragement they gave me. This dissertation would not have been achievable without their guidance and constant feedback. I am grateful to Dr. Gopalan Nadathur and Dr. Marc Riedel for serving on my doctoral final exam committee and for their advice on this dissertation. I would like to thank Dr. Maria Gini for serving on my preliminary oral exam committee and for supporting my Doctoral Dissertation Fellowship application. I would like to express my very great appreciation to Dr. David Hardin, Dr. Konrad Slind, and Dr. Andrew Gacek at the Rockwell Collins Advanced Technology Center for many valuable discussions that have helped me understand my research area better.

I am also thankful to the following current and former members of the Crisys research group at the University of Minnesota for all the support they have given me during my time as a PhD student: Sanjai Rayadurgam, Matt Staats, Gregory Gay, Ian De Silva, Jason Biatek, Yashwant Vijayakumar, Chen Li, Edward Molnar, Anitha Murugesan, Dongjiang You, Lian Duan, Kevin Wendt, Kristin Mead, and Andreas Katis.

I appreciate the financial support from the 3M Science and Technology Fellowship in 2010-2014 and from the Doctoral Dissertation Fellowship at the University of Minnesota

in 2013-2014.

Most of all, I owe thanks to my family, who made all of this possible.

Dedication

To my dad Hung Huy Pham, my mom Thu Thi Nguyen, my sister Loan Quynh Pham,
and my wife Huyen Thi Ha.

Abstract

Reasoning about functions that operate over algebraic data types is an important problem for a large variety of applications. One application of particular interest is network applications that manipulate or reason about complex message structures, such as XML messages. In this dissertation, we present a decision procedure for reasoning about algebraic data types using abstractions that are provided by catamorphisms: fold functions that map instances of algebraic data types into values in a decidable domain. We show that the procedure is sound and complete for a class of monotonic catamorphisms. Our work extends a previous decision procedure that unrolls catamorphism functions until a solution is found.

We propose the categories of monotonic catamorphisms and associative-commutative catamorphisms, which we argue provide a better formal foundation than previous categorizations of catamorphisms. We use monotonic catamorphisms to address an incompleteness in the previous unrolling algorithm (and associated proof), and then use these notions to address two open problems from previous work: (1) we provide a bound on the number of unrollings necessary for completeness, showing that it is exponentially small with respect to formula size for associative-commutative catamorphisms, and (2) we demonstrate that associative-commutative catamorphisms can be combined within a formula whilst preserving completeness. Our combination results extend the set of problems that can be reasoned about using the catamorphism-based approach.

In addition, we generalize certain kinds of catamorphism functions to support additional parameters. This extension, called parameterized associative-commutative catamorphisms subsumes the associative-commutative class, 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.

We also describe an implementation of the approach, called RADA, which accepts formulas in an extended version of the SMT-Lib2 syntax. The procedure is quite general and is central to the reasoning infrastructure for Guardol, a domain-specific language for reasoning about network guards.

Contents

Acknowledgements	i
Dedication	iii
Abstract	iv
List of Tables	ix
List of Figures	x
1 Introduction	1
1.1 Dissertation Motivation	5
1.2 Contributions	9
2 Related Work	11
2.1 Verification Condition Generation	11
2.2 Verification Tools for Algebraic Data Types	14
2.3 Decision Procedures	16
2.3.1 Basic Concepts	16
2.3.2 Some Decision Procedures for Algebraic Data Types	17

3	Unrolling-based Decision Procedure and Monotonic Catamorphisms	31
3.1	Preliminaries	32
3.1.1	Catamorphisms	32
3.2	Properties of Trees and Shapes in the Parametric Logic	36
3.2.1	Properties of Trees	36
3.2.2	Properties of Tree Shapes	37
3.3	Unrolling-based Decision Procedure Revisited	38
3.3.1	Catamorphism Decision Procedure by Example	43
3.4	Monotonic Catamorphisms	47
3.4.1	Monotonic Catamorphisms	47
3.4.2	Examples of Monotonic Catamorphisms	49
3.5	Unrolling Decision Procedure - Proof of Correctness	53
3.5.1	Some Properties of Monotonic Catamorphisms	54
3.5.2	Proof of Correctness of the Unrolling-based Decision Procedure	55
4	Associative-Commutative (AC) Catamorphisms	64
4.1	Definition	65
4.2	The Monotonicity of AC Catamorphisms	68
4.3	Exponentially Small Upper Bound of the Number of Unrollings	72
4.4	Combining AC Catamorphisms	73
5	Parameterized Associative-Commutative (PAC) Catamorphisms	76
5.1	Parameterized Associative-Commutative Abstractions	78
5.2	Benefits of PAC Catamorphisms	80
5.2.1	Expressiveness	80
5.2.2	Usability	82
5.2.3	Efficiency	83

5.3	The Monotonicity of PAC Catamorphisms	84
5.4	AC Features in PAC Catamorphisms	91
5.5	The Relationship between Catamorphisms	93
6	Experimental Results	96
6.1	RADA: A Tool for Reasoning about Algebraic Data Types	96
6.1.1	Motivation	97
6.1.2	Tool Architecture	97
6.1.3	Implementation Improvement	100
6.2	Experimental Results	107
6.2.1	Experiments on AC catamorphisms	107
6.2.2	Experiments on PAC catamorphisms	109
6.2.3	Experiments on Guardol benchmarks	110
7	Conclusion and Discussion	112
	References	118
	Appendix A. Glossary and Acronyms	129
A.1	Glossary	129
A.2	Acronyms	129

List of Tables

2.1	Examples of some decidable first-order theories [1]	17
3.1	Sufficiently surjective catamorphisms in [2]	32
3.2	First few values of functions ns and \mathbb{C}	37
3.3	Examples of $\beta(t)$ with the <i>Multiset</i> catamorphism	48
4.1	Examples of monotonic but not associative-commutative catamorphisms	67
5.1	Some PAC catamorphisms that are not AC	84
6.1	The improvement in performance of RADA	100
6.2	Experimental results on benchmarks with AC catamorphisms	108
6.3	Experimental results on benchmarks with PAC catamorphisms	110
6.4	Experimental results on complex guard benchmarks from Guardol [3]	110
A.1	Acronyms	129

List of Figures

2.1	Syntax of the parametric logic	19
2.2	Semantics of the parametric logic	20
2.3	Syntax of DRYAD	29
3.1	An example of a tree	33
3.2	An example of a tree shape	34
3.3	Sketch of the unrolling-based decision procedure for algebraic data types	41
3.4	An example of how the decision procedure works	45
3.5	Examples of strict subtrees	51
3.6	Example of tree constructions in the proof of <i>Sortedness_{nodup}</i>	53
3.7	Construct a set of trees ST_{hp} from t_{term}	61
4.1	Example of converting a shape into a tree	68
4.2	Construct t_Q from $e_1, \dots, e_{ Q }$	69
4.3	Relationship between t'_0, t''_0 and $t'_{01}, t'_{02}, t''_{01}, t''_{02}$	71
5.1	Relationship between t'_0, t''_0 and $t'_{01}, t'_{02}, t''_{01}, t''_{02}$	88
5.2	The constructions of t'_{01} and t_{02}	90
5.3	Relationship between different types of catamorphisms	94
6.1	RADA architecture.	98
6.2	RADA grammar.	98

Chapter 1

Introduction

Software plays an important role in our lives, but the more complex software systems are, the more difficult it is to ensure correctness. According to a study by the U.S. Department of Commerce National Institute of Standards and Technology (NIST) in 2002, software errors cost the U.S. economy an estimated \$59.5 billion annually. Not only are these errors costly to fix, but software failures can even lead to loss of life, as in, for example, the Therac-25 radiation therapy machines, which massively overdosed six people because of software errors. Therefore, ensuring the correctness of software programs is vital.

Among methods to ensure software quality, *testing* and *formal verification* are the most prominent ones. The goal of testing is to reduce the frequency of failures by running a program under test with a set of test cases to gain some confidence that the program does not have defects. While easy to carry out and extremely popular in practice, software testing cannot guarantee that the program is error-free. Formal verification, on the other hand, aims to prove that the program has no defects. Therefore, formal verification is used to provide a rigorous guarantee of quality of critical systems. It can improve the quality of specifications and can be used in automatic test generation.

Nevertheless, formal verification cannot replace testing entirely - some testing will always be required to ensure that the software behaves as intended. While powerful and useful, formal verification is difficult to use and understand, and it is not always feasible to apply formal verification in verifying software systems.

There have been a number of formal verification techniques for determining program correctness proposed in the literature. Broadly speaking, we can categorize them into five main branches: refinement, Hoare logic, symbolic execution, induction, and next-state relation styles. The styles are not mutually exclusive since a verification technique can belong to multiple styles.

- *Refinement* gradually transforms a specification into a program through a series of correctness-preserving transformations. These transformations can be proved correct using a variety of different analysis techniques (e.g., the B-method [4] and Event-B [5]).
- *Hoare logic* [6] views a program as a *predicate transformer* that describes how a formula is transformed by the execution of a program. Starting from the postcondition of a function, it is possible to determine a *precondition* for the function that will satisfy the postcondition, given a set of *loop invariants*. Some examples of verifiers in this category include Dafny [7], JML tools [8], ESC/Java [9], and Microsoft Spec# [10].
- *Symbolic execution* [11] performs a symbolic path exploration of a program. It builds a formula characterizing the current path; A satisfying assignment to the variables in the formula describes the concrete values necessary for a program to execute this path. Symbolic execution has been used in numerous tools, e.g., Java PathFinder [12], a model checker for Java. The technique is also used in test case generators Pex [13] (for .NET), KLEE [14] (for C), and KLOVER [15] (for C++).

- *Induction* is a popular technique for reasoning about functional programming languages. In this approach, specifications are proved over recursive programs and data structures by defining an induction over a well-founded set, such as the natural numbers or the size of an algebraic data structure. The proof proceeds by demonstrating the specification on the base case and proving the recursive case by assuming the specification (the inductive hypothesis) over a “smaller” argument in the set. Practical theorem provers/program verifiers have been built to support inductive invariants (e.g. [16, 17, 18, 7]).
- *Next-state relation* systems reason about the set of next states from a current state. The technique is used in *model checking* [19, 20, 21, 22] to determine if a model of a system satisfies its specification by exhaustively examining all possible states of the system. If the model checking algorithm finds a state that violates a correctness property, it returns a counterexample to demonstrate an execution trace that leads to the violation. Model checking faces a state explosion problem, which can be handled by binary decision diagrams (BDDs) [23], partial order reduction [24], predicate abstraction [25], and bounded model checking [26]. Another approach is to use SAT/SMT encodings as in, for example, interpolation [27], PDR [28], and k-induction [29, 30].

One of the common ways to verify software programs today is to (1) transform the programs into *verification conditions* using some techniques such as refinements, Hoare logic, and induction, and then (2) discharge the verification conditions using analysis tools. The tools that generate verification conditions are called *verification condition generators*. Boogie [31] and Why3 [32] are two of the most powerful verification condition generators. A number of well-known verification systems have been constructed based on the verification condition generators, including Dafny [7], Chalice [33], Spec # [10],

HAVOC [34], VCC [35], and Krakatoa [36].

To discharge the verification conditions, we can use analysis tools such as *theorem provers* (e.g., HOL [37, 38], Coq [39]), which are powerful but require substantial expert knowledge, or *satisfiability modulo theories* (SMT) solvers, which are, on the other hand, fully automated and therefore preferred in industrial settings. In the last decade, SMT has become one of the most powerful techniques for verification. In general, an SMT solver is the combination of a SAT solver (i.e., a solver that, given a boolean formula, can return an assignment to the variables in the formula to make it evaluate to true or can conclude that such assignment does not exist) and other domain-specific theories (e.g., the theory of uninterpreted functions and the theory of recursive data types). SMT solvers can determine whether a formula is satisfiable or not and can return a model (counterexample) in case the formula is satisfiable. Notably, SMT solvers are usually not only expressive but also very efficient; in addition, a number of high quality SMT solvers have been built [40, 41, 42, 43, 44, 45, 46, 47, 48], thanks to the DPLL(T) algorithm proposed by Ganzinger et al. [49], a standard SMT-Lib2 format proposed by Barrett et al. [50], and the annual competition SMT-COMP [51].

The expressive power and scalability of SMT solvers are determined by the *decision procedures* [1, 52] embedded in them. A decision procedure is a theory-specific algorithm that, given a formula in the theory, can answer whether the formula is satisfiable or not. During the last few years, some notable advances have been made in the field of decision procedures. The *objective of this dissertation* is to propose a decision procedure for algebraic data types with abstractions that can be used to reason about recursive programs using SMT-based verification approaches.

1.1 Dissertation Motivation

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

Algebraic data types provide a significant challenge for decision procedures since they are recursive and usually unbounded in size. Early approaches focused on equalities and disequalities over the structure of elements of data types [53, 54]. While important, these structural properties are often not expressive enough to describe interesting properties involving the data stored in the data type. Instead, we often are interested in making statements both about the structure and contents of data within a data type. For example, one might want to express that all integers stored within a tree are positive or that the set of elements in a list does not contain a particular value.

Suter et al. described a parametric decision procedure for reasoning about algebraic data types using catamorphism (fold) functions [2]. In the procedure, catamorphisms describe the abstract views of the data type that can then be reasoned about in formulas. For example, suppose that we have a binary tree data type with functions to add and remove elements from the tree, as well as check whether an element was stored in the tree. Given a catamorphism *setOf* that computes the set of elements stored in the tree,

we could describe a specification for an ‘add’ function as:

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

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

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

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

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

¹The algorithm in [2] is in fact incomplete for inequalities over finite sets and for non-structural inequalities. We discuss this issue in detail in Section 2.3.2.

The first class of problems is especially important for *guards*, devices that mediate information sharing between security domains according to a specified policy. Typical guard operations include reading field values in a packet, changing fields in a packet, transforming a packet by adding new fields, dropping fields from a packet, constructing audit messages, and removing a packet from a stream. We have built automated reasoning tools (described in [3]) based on the decision procedure to support reasoning over guard applications.

Example 1.1. Suppose we have a catamorphism *remDirtyWords* that removes from an XML message *m* all the words that are in a given blacklist. In addition, suppose we want to verify the following idempotent property of the catamorphism: the result obtained after applying the catamorphism to a message twice is the same as the result obtained after applying the catamorphism to the message only once. We can write this property as a formula that can be decided by the decision procedure in [2] as follows:

$$remDirtyWords(m) = remDirtyWords(remDirtyWords(m))$$

We can also use the decision procedure [2] to verify some properties of recursively-defined data structures. For example, suppose we have a red-black tree *t* and we want to verify that after inserting an element *e* to the red-black tree, the new tree is still balanced and *e* must be in the new tree. Given a catamorphism *isBalanced* that maps a tree to **true** if it is balanced, the property can be formalized as follows:

$$t' = add(e, t) \Rightarrow isBalanced(t') \wedge e \in setOf(t')$$

which can also be effectively handled by the decision procedure in [2]. △

The procedure [2] was proved sound for all catamorphisms and complete for a class

of catamorphisms called *sufficiently surjective* catamorphisms, which we will describe in more detail in Section 3.1. Unfortunately, the algorithm in [2] was quite expensive to compute and required a specialized predicate called M_p to be defined separately for each catamorphism and proved correct w.r.t. the catamorphism using either a hand-proof or a theorem prover.

In [55], a generalized algorithm for the decision procedure was proposed, based on unrolling the catamorphism. This algorithm had three significant advantages over the algorithm in [2]: it was much less expensive to compute, it did not require the definition of M_p , and it was claimed to be complete for all sufficiently surjective catamorphisms. Unfortunately, the algorithm in [55] is in fact not complete for all sufficiently surjective catamorphisms.

In this dissertation, we propose an unrolling-based decision procedure to remove this incompleteness. We then address two open problems with the previous work [55]: (1) how many catamorphism unrollings are required in order to prove properties using the decision procedure? and (2) when is it possible to combine catamorphisms within a formula in a complete way? To address these issues, we introduce two further notions: *monotonic* catamorphisms, which describe an alternative formulation to the notion of *sufficiently surjective* catamorphisms for describing completeness, and *associative-commutative* (AC) catamorphisms, which can be combined within a formula while preserving completeness results. In addition, these catamorphisms have the property that they require a very small number of unrollings. This behavior explains some of the empirical success in applying catamorphism-based approaches on interesting examples from previous papers [55, 3].

In addition, we propose *parameterized associative-commutative* (PAC) catamorphisms, a generalized version of associative-commutative (AC) catamorphisms. PAC

catamorphisms have all the powerful features of AC catamorphisms: they are combinable and guarantee an exponentially small number of unrollings for our unrolling-based decision procedure. Furthermore, PAC catamorphisms are more general, computationally cheaper to reason about, and more expressive than AC ones.

We have implemented the decision procedure in an open-source tool called RADA (reasoning about algebraic data types), which has been used as a back-end verification tool in the Guardol system [3]. The successful uses of RADA in the Guardol project demonstrate that RADA not only could serve as a good research prototype tool but also holds great promise for being used in other real world applications.

1.2 Contributions

This dissertation has the following contributions:

- We provide a more useful mathematical foundation for describing a class of decidable catamorphisms, called *monotonic* catamorphisms. In addition, we propose an unrolling-based decision procedure for algebraic data types using monotonic catamorphisms and formally prove the completeness of the decision procedure.
- We define an important sub-class of monotonic catamorphisms called *associative-commutative* (AC) catamorphisms and demonstrate that an arbitrary number of these catamorphisms can be combined in a formula while preserving decidability. Another nice property of AC catamorphisms is that determining whether a catamorphism function is AC can be straightforwardly checked by an SMT solver, so we call these catamorphisms *detectable*. Finally, AC catamorphisms are guaranteed to require a small number of unrollings to solve.
- We generalize the syntax of catamorphisms to support additional parameters with

a class of catamorphisms called *parameterized associative-commutative* (PAC) catamorphisms, which allows parameters while preserving the good properties of AC catamorphisms.

- We describe an implementation of the approach, called RADA, which accepts formulas in an extended version of the SMT-Lib2 syntax [50], and demonstrate it on a range of examples.

Our **long-range goal** is to build a framework for automated reasoning about recursive programs and data structures that can be used to verify the correctness of industrial-scale software systems. While the proposed research might not completely be able to verify all different types of complex, real-world recursive programs, it does represent a step forward by enabling a wide range of non-trivial recursive programs and properties to be successfully and automatically verified when coupled with state-of-the-art theorem provers or SMT solvers.

The rest of this dissertation is organized as follows. Chapter 2 presents some related work that is closest to ours. In Chapter 3, we present the unrolling-based decision procedure and the idea of monotonic catamorphisms. Chapter 4 presents AC catamorphisms. Chapter 5 presents PAC catamorphisms. Experimental results for our approach are shown in Chapter 6. Finally, we conclude this dissertation in Chapter 7.

Chapter 2

Related Work

The most relevant work related to the research outlined in this dissertation fall in three broad categories: verification condition generation, verification tools for algebraic data types, and decision procedures for algebraic data types.

2.1 Verification Condition Generation

As discussed in Chapter 1, our work is related to the decision procedures used in SMT solvers. Therefore, our work relies on verification condition generators to create appropriate verification conditions that can be solved by our logic.

Verification condition generators are tools that create verification conditions (mathematical logic formulas) from program's source code such that the validity of the formulas implies the correctness of the program. In other words, if we can verify that the verification conditions are valid, we can conclude that the program is correct; otherwise, there must be some errors in the program. The most popular verification condition generators are Boogie [31] and Why3 [32], which have been used in several powerful verification systems, including Dafny [7], Chalice [33], Spec # [10], HAVOC [34] (for C), VCC [35]

(for C), and Krakatoa [36] (for Java).

How do verification condition generators work? Our goal is to turn a program into a logical formula that reflects the correctness of the program. In addition to the source code of the program, we also need the specification of what the program is supposed to do; in particular, we need *preconditions* and *postconditions*. Given a program S , a precondition P describes what we can assume prior to the execution of the program, and a postcondition Q describes what must hold after the program terminates. The triple

$$\{P\} S \{Q\}$$

is known as the Hoare triple. Given the triple, we want to produce a logical formula called verification condition, which implies that if precondition P holds, every terminating execution of S establishes postcondition Q . To compute the verification condition, we generate a *weakest precondition*¹ $wp(S, Q)$ of S with respect to Q such that $wp(S, Q)$ ensures Q . As a result, if the actual precondition P implies the newly generated precondition $wp(S, Q)$, the program S must be correct. In other words, the verification condition for the program S is

$$P \Rightarrow wp(S, Q)$$

After the verification condition has been generated, it is passed to an automated theorem prover or SMT solver, which can then formally prove its correctness.

Example 2.1 (Verification condition generation). Consider the following Hoare triple

¹Regarding how weakest preconditions are generated, the reader is referred to [56].

(taken from the book by Kundu et al. [57]):

```

{true}
    a := x;
    if a < 0 then a := -a else skip;
    if z > 0 then z := z - 1 else skip;
{a ≥ 0}

```

The piece of code stores the absolute value of x in a . It also decreases the value of z by 1 if z is positive. The postcondition states that the value of a (i.e., the absolute value of x) must be non-negative regardless of what the original values of x and z are. The weakest precondition² in this case is:

$$\left(\begin{array}{l} x < 0 \Rightarrow \left((z > 0 \Rightarrow -x \geq 0) \wedge (\neg(z > 0) \Rightarrow -x \geq 0) \right) \\ \neg(x < 0) \Rightarrow \left((z > 0 \Rightarrow x \geq 0) \wedge (\neg(z > 0) \Rightarrow x \geq 0) \right) \end{array} \right) \wedge$$

As a result, the verification condition is:

$$\text{true} \Rightarrow \left(\begin{array}{l} \left(x < 0 \Rightarrow \left((z > 0 \Rightarrow -x \geq 0) \wedge (\neg(z > 0) \Rightarrow -x \geq 0) \right) \right) \wedge \\ \left(\neg(x < 0) \Rightarrow \left((z > 0 \Rightarrow x \geq 0) \wedge (\neg(z > 0) \Rightarrow x \geq 0) \right) \right) \end{array} \right)$$

which can be easily proved to be valid by a theorem prover or an SMT solver. \triangle

Verification condition generation in Guardol. We briefly discuss how verification conditions are generated for Guardol [3], a verification system for guards in which our work has been used as one of its back-end systems. First, a Guardol program is mapped into a formal AST in HOL4 [38], where the operational semantics of Guardol

²For the details of how this weakest precondition was computed, the reader is referred to [57].

are encoded. One can reason directly in HOL4 about programs using the operational semantics; unfortunately, such an approach has limited applicability, requiring expertise in the use of a higher order logic theorem prover. Instead, we would like to make use of the high automation offered by SMT systems. An obstacle: current SMT systems do not understand operational semantics.³ We surmount the problem in two steps. First, *decompilation into logic* [58] is used to deductively map properties of a program in an operational semantics to analogous properties over a mathematical function equivalent to the original program. Next, our decision procedure is used to reason about verification conditions involving algebraic data types generated from the previous step. This procedure necessarily has syntactic limitations, but it is able to handle a wide variety of interesting programs and their properties fully automatically.

2.2 Verification Tools for Algebraic Data Types

We introduce in this dissertation a new verification tool called RADA to reason about algebraic data types with catamorphisms. RADA is described in detail in Chapter 6 and the algorithms behind it are presented in Chapters 3, 4, and 5. Besides RADA, there are some tools that support catamorphisms (as well as other functions) over algebraic data types. For example, Isabelle [59], PVS [60], and ACL2 [16] provide efficient support for both inductive reasoning and evaluation. Although very powerful and expressive, these tools usually need manual assistance and require substantial expert knowledge to construct a proof. On the contrary, RADA is fully automated and accepts input written in the popular SMT-Lib 2.0 format [50]; therefore, we believe that RADA is more suited for non-expert users.

In addition, there are a number of other tools built on top of SMT solvers that have

³Reasons for this state of affairs: there is not one operational semantics because each programming language has a different semantics; moreover, the decision problem for such theories is undecidable.

support for data types. One of such tools is Dafny [7], which supports many imperative and object-oriented features; hence, Dafny can solve many verification problems that RADA cannot. On the other hand, Dafny does not have explicit support for catamorphisms, so for many problems it requires significantly more annotations than RADA. For example, RADA can, without any annotations other than the specification of correctness, demonstrate the correctness of insertion and deletion for red-black trees. From examining proofs of similarly complex data structures (such as the PriorityQueue) provided in the Dafny distribution, it is likely that these proofs would require significant annotations in Dafny.

Our work was inspired by the Leon system [61], which uses an unrolling-based semi-decision procedure to reason about catamorphisms [55]. While Leon uses Scala input, RADA offers a more neutral input format, which is a superset of SMT-Lib 2.0. In addition, Leon specifically uses Z3 [40] as its underlying SMT solver, whereas RADA is solver-independent: it currently supports both Z3 and CVC4. In fact, RADA can support any SMT solvers that use SMT-Lib 2.0 and that have supports for algebraic data types and uninterpreted functions. In addition, RADA guarantees the completeness of the results even when the input formulas have multiple catamorphisms with or without parameters for certain classes of catamorphisms such as PAC catamorphisms (presented in Chapter 5); in this situation, it is unknown whether the decision procedure [55] used in Leon can ensure the completeness or not because the authors [55] only claimed the completeness of the procedure when there is only one type of catamorphism in the input formulas and there are not any additional parameters to be defined within catamorphisms except the algebraic data types.

2.3 Decision Procedures

2.3.1 Basic Concepts

We define some basic concepts of decision procedures that are frequently used in this dissertation. For further introductory discussions, the interested reader is referred to the books [1, 52].

Definition 2.1 (Satisfiability and Validity). A formula is *satisfiable* if there exists an assignment of its variables to make the formula evaluate to **true**. A formula is *valid* if it is true with all assignments of its variables.

Definition 2.2 (Decision problem). The *decision problem* is that given a formula ϕ , decide whether ϕ is valid or not.

Definition 2.3 (Soundness). An algorithm is *sound* if whenever it concludes that a formula is valid, the formula is truly valid.

Definition 2.4 (Completeness). An algorithm is *complete* if it always terminates and given a valid formula, it can conclude that the formula is valid.

Definition 2.5 (Decision procedure). An algorithm is called a *decision procedure* for a logic if it is both sound and complete to solve the decision problem for any formula in the logic.

Definition 2.6 (Decidable logic). A logic is *decidable* if and only if there exists a decision procedure for it.

In the last few decades, researchers have developed decision procedures for various valuable domains, including propositional logic, equality logic and uninterpreted functions, linear arithmetic, bit vectors, arrays, pointer logic, and quantified formulae. Decision procedures can work well with not only one theory but also a combination of theories [62]. Table 2.1 shows some examples of these theories taken from the book [1].

Table 2.1: Examples of some decidable first-order theories [1]

Theory name	Example formula
Propositional logic	$x_1 \wedge (x_2 \vee \neg x_3)$
Equality	$y_1 = y_2 \wedge \neg(y_1 = y_3) \Rightarrow \neg(y_1 = y_3)$
Linear arithmetic	$(2z_1 + 3z_2 \leq 5) \vee (z_2 + 5z_2 - 10z_3 \geq 6)$
Bit vectors	$((a \gg b) \& c) < c$
Arrays	$(i = j \wedge a[j] = 1) \Rightarrow a[i] = 1$
Pointer logic	$p = q \wedge *p = 5 \Rightarrow *q = 5$
Combined theories	$(i \leq j \wedge a[j] = 1) \Rightarrow a[i] < 2$

2.3.2 Some Decision Procedures for Algebraic Data Types

Using abstractions to summarize recursively defined data structures is one of the popular ways to reason about algebraic data types. This idea has been used in the Jahob system [63, 64] and in some procedures for algebraic data types [65, 55, 66, 67]. However, it is often challenging to directly reason about the abstractions. One approach to overcome the difficulty, which is used in [55, 67], is to approximate the behaviors of the abstractions using uninterpreted functions and then send the functions to SMT solvers [40, 41] that have built-in support for uninterpreted functions and recursive data types (although recursive data types are not officially defined in the SMT-Lib 2.0 format [50]).

Our approach extends the work by Suter et al. [2, 55]. In [2], the authors propose a family of procedures for algebraic data types where catamorphisms are used to abstract tree terms. Their procedures were claimed to be complete with sufficiently surjective catamorphisms, which are closely related to the notion of monotonic catamorphisms we describe in Chapter 3, in which we show that all monotonic catamorphisms are sufficiently surjective and all sufficiently surjective catamorphisms described in [2] are monotonic. In the early phase of the Guardol project [3], we implemented the decision procedures [2] on top of OpenSMT [42] and found some flaws in the treatment of disequalities in the unification step of the procedures; fortunately, the flaws can be fixed

using the techniques in [53].

Our unrolling-based decision procedure discussed in Chapter 3 was inspired by the semi-decision procedure by Suter et al. [55]. Unfortunately, their work has a non-terminating issue involving the use of uninterpreted functions. Also, their method works with sufficiently surjective catamorphisms while ours is designed for monotonic catamorphisms.

One work that is close to ours is that of Madhusudan et al. [67], where a sound, incomplete, and automated method is proposed to achieve recursive proofs for inductive tree data-structures while still maintaining a balance between expressiveness and decidability. The method is based on DRYAD, a recursive extension of the first-order logic. DRYAD has some limitations: the element values in DRYAD must be of type `int` and only four classes of abstractions are allowed in DRYAD. In addition to the sound procedure, [67] shows a decidable fragment of verification conditions that can be expressed in STRAND_{dec} [68]. However, this decidable fragment does not allow us to reason about some important properties such as the height or size of a tree. However, the class of data structures that [67] can work with is richer than that of our approach.

Sato et al. [69] proposes a verification technique that has support for recursive data structures. The technique is based on higher-order model checking, predicate abstraction, and counterexample-guided abstraction refinements. Given a program with recursive data structures, they encode the structures as functions on lists, which are then encoded as functions on integers before sending the resulting program to the verification tool described in [70]. Their method can work with higher-order functions while ours cannot. On the other hand, their method cannot verify some properties of recursive data structures while ours can thanks to the use of catamorphisms. An example of such a property is as follows: after inserting an element to a binary tree, the set of all element values in the new tree must be a super set of that of the original tree.

In the remainder of this section, we describe in more detail some decision procedures that are closely related to ours and that can provide the background for our discussion in the subsequent chapters. They include the work by Suter et al. [2, 55] and the procedure proposed by Madhusudan [67].

Suter-Dotta-Kuncak Decision Procedure

We summarize the key steps of the decision procedure proposed by Suter et al. [2]. The input to the decision procedures is a formula ϕ of literals over elements of tree terms and abstractions produced by a catamorphism. The logic is *parametric* in the sense that we assume a data type τ to be reasoned about, an element theory \mathcal{E} containing element types and operations, a catamorphism α that is used to abstract the data type, and a decidable theory $\mathcal{L}_{\mathcal{C}}$ of values in a collection domain \mathcal{C} containing terms C generated by the catamorphism function. Figure 2.1 shows the syntax of the parametric logic instantiated for binary trees. Its semantics is in Figure 2.2.

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

Figure 2.1: 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 F_E describe equalities over collection terms C and element terms E , as well as other operations ($\mathcal{F}_{\mathcal{C}}$, $\mathcal{F}_{\mathcal{E}}$) allowed by the logic of collections

$[\text{Node}(T_1, e, T_2)]$	=	$\text{Node}([T_1], [e]_c, [T_2])$
$[\text{Leaf}]$	=	Leaf
$[\text{left}(\text{Node}(T_1, e, T_2))]$	=	$[T_1]$
$[\text{right}(\text{Node}(T_1, e, T_2))]$	=	$[T_2]$
$[\alpha(t)]$	=	given by the catamorphism
$[T_1 = T_2]$	=	$[T_1] = [T_2]$
$[T_1 \neq T_2]$	=	$[T_1] \neq [T_2]$
$[C_1 = C_2]$	=	$[C_1]_c = [C_2]_c$
$[\mathcal{F}_c]$	=	$[\mathcal{F}]_c$
$[\neg\phi]$	=	$\neg[\phi]$
$[\phi_1 \star \phi_2]$	=	$[\phi_1] \star [\phi_2]$ where $\star \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$

Figure 2.2: Semantics of the parametric logic

\mathcal{L}_C and elements \mathcal{L}_E . E defines terms in the element types \mathcal{E} contained within the branches of the data types. ϕ defines conjunctions of (restricted) formulas in the tree and collection theories. The ϕ terms are the ones solved by the decision procedures in [2]; these can be generalized to arbitrary propositional formulas (ψ) through the use of a DPLL solver [49] that manages the other operators within the formula. Although the logic and unrolling procedure is parametric with respect to data types, in the sequel we focus on binary trees to illustrate the concepts and proofs.

Purification. The original formula is separated into three distinct conjuncts $\phi_T \wedge \phi_B \wedge \phi_C$ where:

- ϕ_T only contains literals over tree terms.
- ϕ_C only contains literals over collection terms in \mathcal{L}_C .
- ϕ_B only contains literals of the form $c = \alpha(t)$ that connects the tree term t and its corresponding abstract collection value c by the catamorphism α .

Flattening of Tree Terms. The next step flattens the tree terms in the formula by

repeatedly applying the flattening rules below, where t_F is always a fresh variable.

$$T \doteq \text{Node}(T_1, E, T_2) \rightsquigarrow t_F = T_1 \wedge T \doteq \text{Node}(t_F, E, T_2)$$

$$T \doteq \text{Node}(t, E, T_2) \rightsquigarrow t_F = T_2 \wedge T \doteq \text{Node}(t, E, t_F)$$

$$T \doteq \text{left}(T_1) \rightsquigarrow t_F = T_1 \wedge T \doteq \text{left}(t_F)$$

$$T \doteq \text{right}(T_1) \rightsquigarrow t_F = T_1 \wedge T \doteq \text{right}(t_F)$$

$$T_1 = t \rightsquigarrow t = T_1$$

$$t \neq T_1 \rightsquigarrow t_F = T_1 \wedge t \neq t_F$$

It is straightforward that the process terminates. Eventually, a formula in which all tree terms are not inside constructors (**Node**) or selectors (**left** and **right**) is obtained.

Elimination of Selectors. This step eliminates the selectors **left** and **right** from the formula by the following rules:

$$t = \text{left}(t_1) \rightsquigarrow t_1 = \text{Node}(t_L, e, t_R) \wedge t = t_L$$

$$t = \text{right}(t_1) \rightsquigarrow t_1 = \text{Node}(t_L, e, t_R) \wedge t = t_R$$

In the rules, e, t_L , and t_R denote fresh variables.

Case splitting. To reduce the complexity of the original problem, a case splitting process is carried out in this stage as follows. For all pairs (t_i, t_j) of elements in the set $\{t_1, \dots, t_n, \text{Leaf}\}$, two new formulas are obtained by adding an equality $t_i = t_j$ and an disequality $t_i \neq t_j$ into the current formula respectively. Similarly, another case splitting phase is performed on the set (e_1, \dots, e_m) of \mathcal{E} -type elements. Obviously, if any new formula is satisfiable, the original formula is also satisfiable.

Unification. This step applies unification on the tree terms and element variables.

During the process, if there is any conflict between the tree terms or element variables, the formula is unsatisfiable. Otherwise, the next step is considered.

Normal Form after Unification. The formula is converted into the disjunctive normal form $N(\vec{T}(\vec{t}), \vec{t}) \wedge M(\vec{u}, \vec{t}, \vec{c}) \wedge F_E \wedge F_C$ where

- $N(\vec{T}(\vec{t}), \vec{t})$ contains conjuncts of the form $t_i \neq t_j$ and $t_i \neq T_j(\vec{t})$.
- $M(\vec{u}, \vec{t}, \vec{c})$ contains conjuncts of the form $c_i = \alpha(v_i)$.
- F_E contains conjuncts of the form $e_i = e_j$ and $e_i \neq e_j$.
- F_C contains conjuncts in the logic of collections \mathcal{L}_C .

Partial Evaluation of the Catamorphism. Let σ_S be the unification formula over tree terms. The formula $M(\vec{u}, \vec{t}, \vec{c})$ is partially evaluated by the following catamorphism-related rules:

$$\alpha(u) \rightsquigarrow \alpha(\sigma_S(u))$$

$$\alpha(\text{Node}(t_1, e, t_2)) \rightsquigarrow \text{combine}(\alpha(t_1), e, \alpha(t_2))$$

$$\alpha(\text{Leaf}) \rightsquigarrow \text{empty}$$

Note that after this transformation, α only applies to parameter variables. $M(\vec{u}, \vec{t}, \vec{c})$ is transformed into $M^1(\vec{t}, \vec{c}) \wedge F_C^1$.

Normal Form after Evaluating Catamorphism. The formula can now be expressed as $D \wedge E$, where

- $D = N(\vec{T}(\vec{t}), \vec{t}) \wedge M^1(\vec{t}, \vec{c})$
- $E = F_E \wedge F_C \wedge F_C^1$

Expressing Existence of Distinct Terms. If D is replaced by `true`, a sound procedure for proving unsatisfiability is obtained. The complete procedure is detailed in [2].

Invoking a Decision Procedure for Collections. At this point, the formula only has terms in the logic of collections $\mathcal{L}_{\mathcal{C}}$, which is assumed to be solvable by existing decision procedures.

Although the decision procedure is extremely powerful, it has several drawbacks. First, the complexity of the procedure is nondeterministic polynomial mainly due to the case splitting step, which aims to derive subproblems from the original formula by adding all possible equalities and disequalities between the tree terms and element variables. The completeness proof also results in a state explosion problem when instantiating every possible tree shape to match it with each t_i . As a result, the complexity prevents the work from being applied to analyze real-world programs. Second, the decision procedure has a subtle flaw in the unification algorithm when we consider tree disequalities. Fortunately, the flaw is local and we can fix it by using accurate unification rules introduced by Barret, Shikanian, and Tinelli [53].

Remark 2.1 (Problem with the treatment of disequalities in [2]). In the unification step in the decision procedure proposed by Suter et al. [2], the treatment of disequalities is flawed. When we have disequalities involving trees $t_i \neq t_j$, [2] reduced these as follows:

If for any disequality $t_i \neq t_j$ or $e_i \neq e_j$, we have that respectively $\sigma_S(t_i) = \sigma_S(t_j)$ or $\sigma_S(e_i) = \sigma_S(e_j)$, then our (sub)problem is unsatisfiable. Otherwise, the tree constraints are satisfiable and we move on to the constraints on the collection type \mathcal{C} .

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

type representing an enumeration:

```
type EnumType = { Foo | Bar | Baz };
```

then the following is unsatisfiable:

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

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

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

```
type EnumType = { Foo [arg : Bool] | Bar | Baz };
```

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

Finally, the decision procedure assumes that the only elements of \mathcal{E} are variables. This restriction is not realistic for the kinds of specifications that are created in Guardol [3]. For example, we could not analyze models that contained constants or arithmetic expressions. Semantically true disequalities like

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

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

It is not theoretically difficult to solve these problems, and a solution is found in, for example, [53]. Several additions to unification are required:

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

However, rather than improve this procedure, the authors described a second decision procedure, presented next.

Suter-Köksal-Kuncak Decision Procedure

Based on the work in [2], Suter et al. [55] proposes a semi-decision procedure that exploits the ability to reason about recursive data types of Z3 [40], a state-of-the-art SMT solver. According to the authors [55], their algorithm is (1) sound for models and proofs and (2) terminating for all satisfiable formulas and for all sufficiently surjective catamorphisms.

The main algorithm of the paper is shown above. It maintains a formula φ and a set of boolean control literals B . The main idea of the algorithm is to keep unrolling

Algorithm 1: The decision procedure in [55]

```

1 begin
2    $(\varphi, B) \leftarrow \text{unroll}(\varphi, \emptyset)$ 
3   while true do
4     switch  $\text{decide}(\varphi \wedge B)$  do
5       case SAT
6         return SAT
7       case UNSAT
8         switch  $\text{decide}(\varphi)$  do
9           case UNSAT
10            return UNSAT
11          case SAT
12             $(\varphi, B) \leftarrow \text{unroll}(\varphi, B)$ 

```

the set of catamorphism applications until a desirable result is found. To keep track of the different branches of a catamorphism during the unrolling process, we use a set of control literals B . The function *decide* externally calls Z3 to check the satisfiability of the formula that is given as the parameter of *decide*. When Z3 checks the satisfiability, function invocations are treated as uninterpreted.

We can understand the *SAT/UNSAT* results that Z3 returns as follows. At line 6, the result is *SAT* because at this unrolling level, $\varphi \wedge B$ is satisfiable. If $\varphi \wedge B$ is unsatisfiable, the result will then depend on the satisfiability of φ . If φ is unsatisfiable, the result is *UNSAT* no matter what the value of B is; otherwise, B plays the main role in the unsatisfiability of $\varphi \wedge B$. Moreover, we cannot conclude anything at this point since B contains some uninterpreted functions that have not yet been unrolled. As a result, we need to perform the unrolling at least one more time at line 12 to observe what would happen next.

It is obvious that the semi-decision procedure in [55] is far less complicated than the previous work in [2]. However, the claim the authors made about its ability to serve as

a uniform procedure for the entire family is incorrect. The reason is that the result of the procedure depends on the value to which the uninterpreted functions are assigned, which can be arbitrary and inconsistent with the codomains of the corresponding catamorphisms. For example, the catamorphism that computes the height of a tree always returns a natural number, but it can be assigned to a negative value when it is evaluated as an uninterpreted function over the integer domain.

Remark 2.2 (Terminating problem in the unrolling procedure in [55]). In the satisfiability procedure for recursive programs in [55], the authors stated that their procedure is terminating for all sufficiently surjective abstractions. Unfortunately, this claim is incorrect. It is straightforward to create catamorphisms that satisfy the sufficient surjectivity definition that will not terminate using the procedure. For example, consider the following catamorphism, which sums the magnitudes of all the elements within the tree:

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

The catamorphism is sufficiently surjective under the construction in [2] with predicate $M_p(c) = c \geq 0$; for any value in the non-negative range of the type, there are an infinite number of trees that satisfy $\alpha^{-1}(c)$.

Suppose now that we attempt to prove the following predicate is unsatisfiable using the unrolling procedure in [55]:

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

for uninterpreted trees t_1 and t_2 . It will not terminate with the construction in [55]. The reason is that the uninterpreted function at the leaves of the unrolling can always choose

an arbitrarily large negative number to assign as the value of the catamorphism, thereby creating a satisfying assignment when evaluating the input formula without control conditions. Note that negative values are not in the range of *SumMagn*. The terminating problem may occur with any catamorphism where its range and its codomain are not the same, such as *SizeI* or *Height* catamorphisms.

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

Fixing this incompleteness issue is one of the important contributions of this dissertation; in Chapter 3, we will propose a decision procedure that is complete for a class of catamorphisms called *monotonic* catamorphisms.

Madhusudan-Qiu-Stefanescu Decision Procedure

The idea of using abstraction to reason about algebraic data types is also explored by Madhusudan, Qiu, and Stefanescu [67]. They present an approach to achieve recursive proofs for inductive tree data-structures while still maintaining a balance between expressiveness and decidability. The method is sound, incomplete, and automated. The authors show that the approach can be used to verify all popular tree-related algorithms including max-heaps, treaps, red-black trees, AVL trees, binomial heaps, and B-trees written in imperative programs.

Their work uses DRYAD, a quantifier-free extension of first-order logic, to express the program we desire to verify. The syntax of DRYAD is shown in Figure 2.3. Essentially, the logic is a combination of quantifier-free logic and recursive predicates/functions on trees. Each node in a tree contains an integer data-field. A node is in the *nil*

dir	\in	Dir	Direction
f	\in	DF	Data-field
p^*	$:$	$Loc \rightarrow \{true, false\}$	Recursively defined boolean function
i^*	$:$	$Loc \rightarrow Int$	Recursively defined integer function
si^*	$:$	$Loc \rightarrow \mathcal{S}(Int)$	Recursively defined set function
msi^*	$:$	$Loc \rightarrow \mathcal{MS}(Int)$	Recursively defined multiset function
x	\in	Loc Variables	Location variable
j	\in	Int Variables	Integer variable
q	\in	$Boolean$ Variables	Boolean variable
S	\in	$\mathcal{S}(Int)$ Variables	Set variable
MS	\in	$\mathcal{MS}(Int)$ Variables	Multiset variable
c	$:$	Int	Constant
lt	$::=$	$x \mid nil \mid lt.dir$	Loc term
it	$::=$	$c \mid j \mid lt.f \mid i^*(lt) \mid it_1 \pm it_2 \mid ite(\phi, it_1, it_2)$	Int term
sit	$::=$	$\emptyset \mid S \mid \{it\} \mid si^*(lt) \mid sit_1 \cup sit_2 \mid sit_1 \cap sit_2 \mid sit_1 \setminus sit_2 \mid ite(\phi, sit_1, sit_2)$	$\mathcal{S}(Int)$ term
$msit$	$::=$	$\emptyset_m \mid MS \mid \{it\}_m \mid msi^*(lt) \mid msit_1 \cup msit_2 \mid msit_1 \cap msit_2 \mid msit_1 \setminus msit_2 \mid ite(\phi, msit_1, msit_2)$	$\mathcal{MS}(Int)$ term
φ	$::=$	$true \mid q \mid p^*(lt) \mid lt_1 = lt_2 \mid it_1 \leq it_2 \mid sit_1 \subseteq sit_2 \mid msit_1 \subseteq msit_2 \mid sit_1 \leq sit_2 \mid msit_1 \leq msit_2 \mid it \in sit \mid it \in msit \mid \neg\varphi \mid \varphi_1 \vee \varphi_2$	Formula
$i^*(x)$	$\stackrel{def}{=}$	$ite(x = nil, i_{base}, i_{ind})$	Recursively-defined integer
$si^*(x)$	$\stackrel{def}{=}$	$ite(x = nil, si_{base}, si_{ind})$	Recursively-defined set-of-integers
$msi^*(x)$	$\stackrel{def}{=}$	$ite(x = nil, msi_{base}, msi_{ind})$	Recursively-defined multiset-of-integers
$p^*(x)$	$\stackrel{def}{=}$	$ite(x = nil, p_{base}, p_{ind})$	Recursively-defined predicate

Figure 2.3: Syntax of DRYAD

location if it is a leaf, or has Dir number of children otherwise. There are four types of recursive predicates/functions that are supported in the logic to help state complex heap properties. They include three functions i^* , si^* and msi^* that map a tree to an integer, to a set of integers, and to a multiset of integers, respectively; and a predicate p^* that maps a tree to a boolean value. A formula in DRYAD is a pair (Def, φ) , where Def is a set of definitions of recursive predicates/functions and φ is a formula.

Given a recursive imperative program with proof annotations written in DRYAD, one can describe the corresponding verification condition for the program. This process is done by executing the program symbolically over a footprint structure, which is denoted by a pair (SH, φ) , where SH is a symbolic heap and φ is a DRYAD formula. (SH, φ)

keeps evolving on basic blocks of code. After executing basic blocks, the verification condition is obtained by combining (SH, φ) with a pre-defined postcondition.

The postcondition can be soundly proved by replacing all recursive definitions in the generated verification condition with uninterpreted functions and then using state-of-the-art SMT solvers to check the validity of the obtained formula. In addition to the sound procedure, there is a decidable fragment of verification conditions that can be expressed in STRAND_{dec} [68] with some limitations including the following: (1) i^* is disallowed, (2) si^* and msi^* only support the union operation, and (3) p^* only supports conjunctions. Note that the limitations prevent us from reasoning about some important properties such as the height or cardinality of a tree.

The decision procedures proposed by Suter et al. [2, 55] and the work [67] have some common ideas. First, the notion of recursive predicates/functions is the same as catamorphisms [2, 55]. Although their work supports hierarchical recursive definitions, it only supports integers stored at each tree node. The second common idea of all the approaches is the use of uninterpreted functions to achieve sound results.

The completeness issue in Suter et al.'s algorithms [2, 55] as well as the completeness and catamorphism expressiveness issues in Madhusudan et al.'s work [67] motivate us to define both a modified procedure and a new mathematical foundation for the work, which we will present next in Chapter 3.

Chapter 3

Unrolling-based Decision Procedure and Monotonic Catamorphisms

Inspired by the decision procedures for algebraic data types by Suter et al. [2, 55] summarized in Section 2.3.2, we present in this chapter an unrolling-based decision procedure with the novel idea of *monotonic* catamorphisms. The logic used in our decision procedure is the same as the logic in the work by Suter et al. [2, 55], which has been summarized in Figure 2.1. First, we present in Section 3.1 some preliminaries about catamorphisms. We then discuss in Section 3.2 some properties of trees and shapes in the logic. In Section 3.3, we present in detail our unrolling-based decision procedure for algebraic data types. The decision procedure works with monotonic catamorphisms, which are discussed in Section 3.4. The correctness of the algorithm for these catamorphisms is shown in Section 3.5.

3.1 Preliminaries

We describe the definition of catamorphisms and the idea of sufficient surjectivity, which describes situations in which the decision procedures proposed by Suter et al. [2, 55] were claimed to be complete.

3.1.1 Catamorphisms

Given a tree in the parametric logic shown in Figure 2.1, we can map the tree into a value in \mathcal{C} using a *catamorphism*, which is a fold function of the following format:

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

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

Table 3.1: Sufficiently surjective catamorphisms in [2]

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

Catamorphisms from [2] are shown in Table 3.1. The first column contains catamorphism names¹. The next two columns define $\alpha(t)$ when t is a `Leaf` and when it is a `Node`,

¹*SizeI*, which maps a tree into its number of *internal* nodes, was originally named *Size* in [2]. We rename the catamorphism to easily distinguish between itself and function *size*, which returns the total number of *all* vertices in a tree, in this dissertation.

respectively. The last column shows examples of the application of each catamorphism to the tree in Figure 3.1.

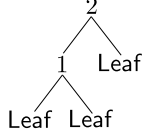


Figure 3.1: An example of a tree

In the *Min* catamorphism, min' is the same as the usual min function except that min' ignores `None` in the list of its arguments, which must contain at least one non-`None` value. The *Sortedness* catamorphism returns a triple containing the min and max element of a tree, and `true/false` depending on whether the tree is sorted or not.

Infinitely surjective catamorphisms: Suter et al. [2] showed that many interesting catamorphisms are *infinitely surjective*. Intuitively, a catamorphism is infinitely surjective if the cardinality of its inverse function becomes infinite if the tree argument of the catamorphism is not bounded by a fixed set of trees.

Definition 3.1 (Infinitely surjective catamorphisms). A catamorphism α is an infinitely surjective S -abstraction, where S is a set of trees, if and only if $\alpha^{-1}(\alpha(t))$ is finite for $t \in S$ and infinite for $t \notin S$.

Example 3.1 (Infinitely surjective catamorphisms). The *Set* catamorphism in Table 3.1 is an infinitely surjective `{Leaf}`-abstraction because:

- $\text{Set}^{-1}(\text{Set}(\text{Leaf})) = \text{Set}^{-1}(\emptyset) = 1$ (i.e., `Leaf` is the only tree in data type τ that can map to \emptyset by the *Set* catamorphism). Therefore, $\text{Set}^{-1}(\text{Set}(\text{Leaf}))$ is finite.
- $\forall t \in \tau, t \neq \text{Leaf} : |\text{Set}^{-1}(\text{Set}(t))| = \infty$. The reason is that when t is not `Leaf`, we have $\text{Set}(t) \neq \emptyset$. Hence, there are an infinite number of trees that can map to

$Set(t)$ by the catamorphism Set . For example, consider the tree in Figure 3.1; let us call it t_0 . We have $Set(t_0) = \{1, 2\}$; hence, $|Set^{-1}(\{1, 2\})| = \infty$ since there are an infinite number of trees in τ whose elements values are 1 and 2.

As a result, Set is infinitely surjective by Definition 3.1. \triangle

Sufficiently surjective catamorphisms: The decision procedures proposed by Suter et al. [2, 55] were claimed to be complete if the catamorphism used in the procedures is *sufficiently surjective* [2]. Intuitively, a catamorphism is sufficiently surjective if the inverse relation of the catamorphism has sufficiently large cardinality when tree shapes are larger than some finite bound. In fact, *the class of infinitely surjective catamorphisms is just a special case of sufficiently surjective catamorphisms* [2].

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

Definition 3.2 (Tree shapes). The shape of a tree is defined by constant `SLeaf` and constructor `SNode(-, -)` as follows:

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

Example 3.2 (Tree shape). Figure 3.2 shows the shape of the tree in Figure 3.1. \triangle

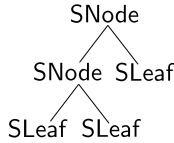


Figure 3.2: An example of a tree shape

Definition 3.3 (Sufficiently surjective catamorphisms). Catamorphism α is sufficiently surjective iff for each $p \in \mathbb{N}^+$, there exists, computable as a function of p ,

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

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

Example 3.3 (Sufficiently surjective catamorphisms). We demonstrated in Example 3.1 that the *Set* catamorphim is infinitely surjective. Let us now show that the catamorphim is sufficiently surjective by Definition 3.3. Let $S_p = \{\mathbf{SLeaf}\}$ and $M_p(c) \equiv c \neq \emptyset$. For this M_p , the only base case we need to consider is the tree **Leaf**: either a tree is **Leaf**, whose shape is in S_p , or the catamorphism value returned is not the empty set, in which predicate M_p holds. Furthermore, this M_p is sufficiently surjective: $c \neq \emptyset$ implies that $|\alpha^{-1}(c)| = \infty$ for the *Set* catamorphism. \triangle

Despite its name, sufficient surjectivity has no surjectivity requirement for the codomain of α . It only requires a “sufficiently large” number of trees for values satisfying the condition M_p . The *SumMagn* catamorphim in Remark 2.2 in Section 2.3.2 is a good example of a sufficiently surjective catamorphim but not surjective. In other words, there is no restriction for the the codomain of a sufficiently surjective catamorphim. However, to ensure the completeness of the unrolling decision procedure, the codomain restriction must be taken into account, as we already noted in Remark 2.2. We will discuss this issue more in Section 3.3.

Table 3.1 describes all sufficiently surjective catamorphisms in [2]. The only catamorphim in [2] not in Table 3.1 is the *Mirror* catamorphim:

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

The reason is that *Mirror* is not sufficiently surjective: since the cardinality of the inversion function of the catamorphism is always 1, the sufficiently surjective condition does not hold for this catamorphism.

3.2 Properties of Trees and Shapes in the Parametric Logic

We present some important properties of trees and shapes in the parametric logic (discussed in Section 2.3.2) which will play important roles in the subsequent sections of this dissertation.

3.2.1 Properties of Trees

Property 3.1 follows from the syntax of the parametric logic. Properties 3.2 and 3.3 are well-known properties of full binary trees [71, 72] (i.e., binary trees in which every internal node has exactly two children).

Property 3.1 (Type of tree). Any tree in the parametric logic is a full binary tree.

Property 3.2 (Size). The number of vertices in any tree in the parametric logic is odd.

Also, in a tree t of size $2k + 1$ ($k \in \mathbb{N}$), we have:

$$\begin{aligned} ni(t) &= k \\ nl(t) &= k + 1 \end{aligned}$$

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

Property 3.3 (Size vs. Height). In the parametric logic, the size of a tree of height

$h \in \mathbb{N}$ must be at least $2h + 1$:

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

3.2.2 Properties of Tree Shapes

In this section, we show a special relationship between tree shapes and the well-known Catalan numbers [73], which will be used to establish some properties of monotonic catamorphisms in Section 3.4 and AC catamorphisms in Chapter 4.

Define the size of the shape of a tree to be the size of the tree. Let $\bar{\mathbb{N}}$ be the set of odd natural numbers. Because of Property 3.2, the size of a shape is in $\bar{\mathbb{N}}$. Let $ns(s)$ be the number of tree shapes of size $s \in \bar{\mathbb{N}}$ and let \mathbb{C}_n , where $n \in \mathbb{N}$, be the n -th Catalan number [73].

Lemma 3.1. *The number of shapes of size $s \in \bar{\mathbb{N}}$ is the $\frac{s-1}{2}$ -th Catalan number:*

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

Proof. Property 3.1 implies that tree shapes are also full binary trees. The lemma follows since the number of full binary trees of size $s \in \bar{\mathbb{N}}$ is $\mathbb{C}_{\frac{s-1}{2}}$ [73, 74]. \square

Example 3.4 (Function ns and \mathbb{C}). Table 3.2 shows some first few values of function $ns : \bar{\mathbb{N}} \rightarrow \mathbb{N}^+$, which demonstrates the numbers of shapes of different sizes, and function $\mathbb{C} : \mathbb{N} \rightarrow \mathbb{N}^+$, which represents the Catalan numbers. \triangle

Table 3.2: First few values of functions ns and \mathbb{C}

n	0	1	2	3	4	5	6	7	8	9	10	11
$ns(2n + 1) = \mathbb{C}_n$	1	1	2	5	14	42	132	429	1430	4862	16796	58786

By using the expression $\mathbb{C}_n = \frac{1}{n+1} \binom{2n}{n}$ [73], we could easily compute the values that function $ns : \bar{\mathbb{N}} \rightarrow \mathbb{N}^+$ returns. This function satisfies the monotonic condition in Lemma 3.2.

Lemma 3.2. $1 = ns(1) = ns(3) < ns(5) < ns(7) < ns(9) < \dots$

Proof. According to Koshy [74], Catalan numbers can be computed as follows:

$$\begin{aligned} \mathbb{C}_0 &= 1 \\ \mathbb{C}_{n+1} &= \frac{2(2n+1)}{n+2} \mathbb{C}_n \quad (\text{where } n \in \mathbb{N}) \end{aligned}$$

Clearly, $\mathbb{C}_1 = 1$. When $n \geq 1$, we have:

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

Therefore, by induction on n , we obtain:

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

which completes the proof because of Lemma 3.1. □

3.3 Unrolling-based Decision Procedure Revisited

In this section, we restate the unrolling procedure proposed by Suter et al. [55], shown in Algorithms 2 (restating Algorithm 1), and propose a revised unrolling procedure, shown in Algorithm 3. The input of both algorithms consists of

- a formula ϕ written in a logic (described in Figures 2.1 and 2.2) that consists of literals over elements of tree terms and tree abstractions generated by a catamorphism (i.e., a fold function that maps a recursively-defined data type into a value in a base domain). In other words, ϕ contains a recursive data type τ , an element type \mathcal{E} of the value stored in each tree node, a collection type \mathcal{C} of tree abstractions in a decidable logic $\mathcal{L}_{\mathcal{C}}$, and a catamorphism $\alpha : \tau \rightarrow \mathcal{C}$ that maps an object in the data type τ into a value in the collection type \mathcal{C} .
- a program Π , which contains ϕ , the definitions of data type τ , and catamorphism α .

Algorithm 2: Unrolling decision procedure in [55] with *sufficiently surjective catamorphisms*

```

1  $(\phi, B) \leftarrow \text{unrollStep}(\phi, \Pi, \emptyset)$ 
2 while true do
3   switch  $\text{decide}(\phi \wedge \bigwedge_{b \in B} b)$  do
4     case SAT
5       return "SAT"
6     case UNSAT
7       switch  $\text{decide}(\phi)$  do
8         case UNSAT
9           return "UNSAT"
10        case SAT
11           $(\phi, B) \leftarrow \text{unrollStep}(\phi, \Pi, B)$ 

```

Algorithm 3: Revised unrolling procedure with *monotonic catamorphisms*

```

1  $(\phi, B) \leftarrow \text{unrollStep}(\phi, \Pi, \emptyset)$ 
2 while true do
3   switch  $\text{decide}(\phi \wedge \bigwedge_{b \in B} b)$  do
4     case SAT
5       return "SAT"
6     case UNSAT
7       switch  $\text{decide}(\phi \wedge R_{\alpha})$  do
8         case UNSAT
9           return "UNSAT"
10        case SAT
11           $(\phi, B) \leftarrow \text{unrollStep}(\phi, \Pi, B)$ 

```

The decision procedure works on top of an SMT solver \mathcal{S} that supports theories for $\tau, \mathcal{E}, \mathcal{C}$, and uninterpreted functions. Note that the only part of the parametric logic that is not inherently supported by \mathcal{S} is the applications of the catamorphism. The main

idea of the decision procedure is to approximate the behavior of the catamorphism by repeatedly unrolling it and treating the calls to the not-yet-unrolled catamorphism instances at the leaves as calls to an uninterpreted function. However, the uninterpreted function can return any values in its codomain; thus, the presence of these uninterpreted functions can make *SAT* results untrustworthy. To address this issue, each time the catamorphism is unrolled, a set of boolean control conditions B is created to determine if the determination of satisfiability is independent of the uninterpreted functions at the bottom level. That is, if all the control conditions in B are true, the list of uninterpreted functions does not play any role in the satisfiability result.

In other words, 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*.

In addition, we observe that if a catamorphism instance is treated as an uninterpreted function, the uninterpreted function should only return values inside the *range* of the catamorphism; therefore, in our decision procedure, R_α captures the range of catamorphism α and it is included in the satisfiability check whenever the determination of satisfiability may require the use of such uninterpreted functions.

The unrollings without control conditions represent an over-approximation of the formula with the semantics of the program with respect to the parametric logic, in that it accepts all models accepted by the parametric logic plus some others (due to the

uninterpreted functions). The unrollings with control conditions represent an under-approximation: all models accepted by this model will be accepted by the parametric logic with the catamorphism.

The algorithm determines the satisfiability of ϕ through repeated unrolling α using the *unrollStep* function. Given a formula ϕ_i generated from the original ϕ after unrolling the catamorphism i times and the set of control conditions B_i of ϕ_i , function *unrollStep*(ϕ_i, Π, B_i) unrolls the catamorphism one more time and returns a pair (ϕ_{i+1}, B_{i+1}) containing the unrolled version ϕ_{i+1} of ϕ_i and a set of control conditions B_{i+1} for ϕ_{i+1} . Function *decide*(φ) simply calls SMT solver \mathcal{S} to check the satisfiability of φ and returns *SAT/UNSAT* accordingly. The algorithm either terminates when ϕ is proved to be satisfiable without the use of uninterpreted functions (line 5) or ϕ is proved to be unsatisfiable when assigning any values to uninterpreted functions still cannot make the problem satisfiable (line 9). Figure 3.3 shows the unrolling idea used in the decision procedure.

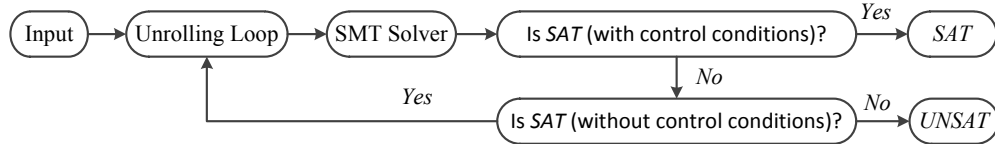


Figure 3.3: Sketch of the unrolling-based decision procedure for algebraic data types

Let us examine how satisfiability and unsatisfiability are determined in the algorithm. In general, the algorithm keeps unrolling the catamorphism until we find a *SAT/UNSAT* result that we can trust. To do that, we need to consider several cases after each unrolling step is carried out. First, at line 4, ϕ is satisfiable and all the control conditions are true, which means uninterpreted functions are not involved in the satisfiable result. In this case, we have a complete tree model for the *SAT* result and we can conclude that the problem is satisfiable.

On the other hand, let us consider the case when $decide(\phi \wedge \bigwedge_{b \in B} b) = UNSAT$. The *UNSAT* may be due to the unsatisfiability of ϕ , or the set of control conditions, or both of them together. To understand the *UNSAT* more deeply, we can try to check the satisfiability of ϕ alone. Note that checking ϕ alone also means that the control conditions are not used; consequently, the values of uninterpreted functions may contribute to the *SAT/UNSAT* result. Therefore, we include R_α in the satisfiability check (i.e., $decide(\phi \wedge R_\alpha)$ at line 7) to ensure that if a catamorphism instance is viewed as an uninterpreted function then the uninterpreted function only returns values inside the range of the catamorphism. If $decide(\phi \wedge R_\alpha) = UNSAT$ as at line 8, we can conclude that the problem is unsatisfiable because assigning any values in the range of the catamorphism to the uninterpreted functions still cannot make the problem satisfiable as a whole. Finally, we need to consider the case $decide(\phi \wedge R_\alpha) = SAT$ as at line 10. Since we already know that $decide(\phi \wedge \bigwedge_{b \in B} b) = UNSAT$, the only way to make $decide(\phi \wedge R_\alpha)$ be *SAT* is by calling to at least one uninterpreted function, which also means that the *SAT* result is untrustworthy. Therefore, we need to keep unrolling at least one more time as denoted at line 11.

The central problem of Algorithm 2, as described in Remark 2.2 in Section 2.3.2, is that its termination is not guaranteed. For example, non-termination can occur if the uninterpreted function U_α representing α can return values outside the range of α . Consider an unsatisfiable input problem: $SizeI(t) < 0$, for an uninterpreted tree t when $SizeI$ is defined over the integers in an SMT solver. Although $SizeI$ is sufficiently surjective [2], Algorithm 2 will not terminate since each uninterpreted function at the leaves of the unrolling can always choose an arbitrarily large negative number to assign as the value of the catamorphism, thereby creating a satisfying assignment when evaluating the input formula without control conditions. These negative values are outside the

range of *SizeI*. Broadly speaking, this termination problem can occur for any catamorphism that is not surjective. Unless an underlying solver supports predicate subtyping, such catamorphisms are easily constructed. In fact, *SizeI* or *Height* catamorphisms are not surjective when defined against SMT-Lib 2.0 [50].

To address the non-termination issue, we need to constrain the assignments to uninterpreted functions $U_\alpha(t)$ representing $\alpha(t)$ to return only values inside the range of α . Let R_α be a condition that, together with $U_\alpha(t)$, represents the range of α . The collection of values that $U_\alpha(t)$ can return can be constrained by R_α . In Algorithm 3, the user-provided range R_α is included in the *decide* function to make sure that any values that uninterpreted function $U_\alpha(t)$ returns could be mapped to some “real” tree $t' \in \tau$ such that $\alpha(t') = U_\alpha(t)$:

$$\forall c \in \mathcal{C} : (c = U_\alpha(t) \wedge R_\alpha(c)) \Rightarrow (\exists t' \in \tau : \alpha(t') = c) \quad (3.1)$$

Formula (3.1) defines a correctness condition for R_α . Unfortunately, it is difficult to prove this without the aid of a theorem prover. On the other hand, it is straightforward to determine whether R_α is a sound approximation of the range of R (that is, all values in the range of R are accepted by R_α) using induction and an SMT solver. To do so, we simply unroll α a single time over an uninterpreted tree t . We assume R_α is true for the uninterpreted functions in the unrolling but that R_α is false over the unrolling. If an SMT solver can prove that the formula is *UNSAT*, then R_α soundly approximates the range; this unrolling encodes both the base and inductive case.

3.3.1 Catamorphism Decision Procedure by Example

As an example of how the procedure in Algorithm 3 can be used, let us consider a guard application (such as those in [3]) that needs to determine whether 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 whether a word is “dirty”.

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

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

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

Example 3.5 (Unrolling-based decision procedure). For our guard example, suppose one of the verification conditions is:

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

The formula is *UNSAT* because t has at least one dirty word (e in this case), so its number of dirty words cannot be 0. Figure 3.4 shows how the procedure works for this

example.

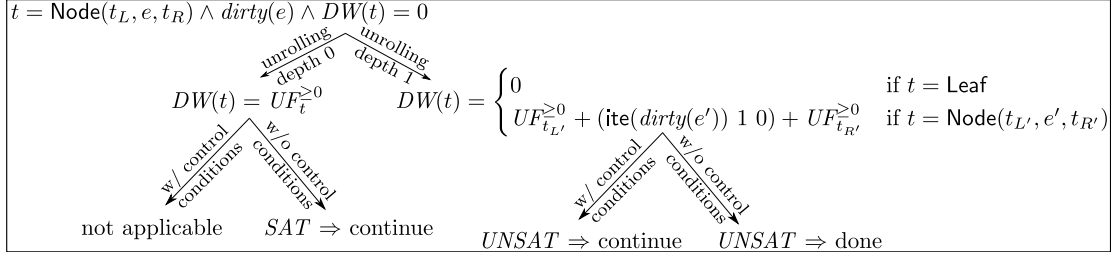


Figure 3.4: An example of how the decision procedure works

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

$$t = \text{Node}(t_L, e, t_R) \wedge \text{dirty}(e) \wedge DW(t) = 0 \wedge DW(t) = UF_t^{\geq 0}$$

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

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

control conditions, the formula becomes

$$t = \text{Node}(t_L, e, t_R) \wedge \text{dirty}(e) \wedge DW(t) = 0 \wedge \neg(t = \text{Leaf}) \wedge \left(DW(t) = 0 \wedge t = \text{Leaf} \vee \right. \\ \left. DW(t) = UF_{t_L}^{\geq 0} + (\text{ite}(\text{dirty}(e')) \ 1 \ 0) + UF_{t_R}^{\geq 0} \wedge t = \text{Node}(t_L, e', t_R) \right)$$

which is equivalent to

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

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.

Without control conditions, the formula becomes

$$t = \text{Node}(t_L, e, t_R) \wedge \text{dirty}(e) \wedge DW(t) = 0 \wedge \left(DW(t) = 0 \wedge t = \text{Leaf} \vee \right. \\ \left. DW(t) = UF_{t_L}^{\geq 0} + (\text{ite}(\text{dirty}(e')) \ 1 \ 0) + UF_{t_R}^{\geq 0} \wedge t = \text{Node}(t_L, e', t_R) \right)$$

which, after eliminating the **Leaf** case (because t must be a **Node**) and unifying $\text{Node}(t_L, e, t_R)$ with $\text{Node}(t_L, e', t_R)$ (because both of them are equal to t), simplifies to

$$t = \text{Node}(t_L, e, t_R) \wedge \text{dirty}(e) \wedge DW(t) = 0 \wedge DW(t) = UF_{t_L}^{\geq 0} + (\text{ite}(\text{dirty}(e)) \ 1 \ 0) + UF_{t_R}^{\geq 0}$$

which, after evaluating the *ite* statement, is equivalent to

$$t = \text{Node}(t_L, e, t_R) \wedge \text{dirty}(e) \wedge DW(t) = 0 \wedge DW(t) = UF_{t_L}^{\geq 0} + 1 + UF_{t_R}^{\geq 0}$$

which is *UNSAT* because $UF_{t_L}^{\geq 0} + 1 + UF_{t_R}^{\geq 0} > 0$. However, getting *UNSAT* without control conditions guarantees that the original formula is *UNSAT*; thus, the process

terminates here.

△

3.4 Monotonic Catamorphisms

We now propose *monotonic* catamorphisms and prove that Algorithm 3 is complete for this class, provided that R_α accurately characterizes the range of α . We show that this class is a subset of sufficiently surjective catamorphisms, but general enough to include all catamorphisms described in [2, 55] and all those that we have run into in industrial experience. Monotonic catamorphisms admit a termination argument in terms of the number of unrollings, which is an open problem in [55]. Moreover, a subclass of monotonic catamorphisms, *associative-commutative* (AC) catamorphisms can be combined while preserving completeness of the formula, addressing another open problem in [55].

3.4.1 Monotonic Catamorphisms

We will prove that our decision procedure is 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 3.4 (Function β). Given a catamorphism $\alpha : \tau \rightarrow \mathcal{C}$, 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 3.6 (Function β). Intuitively, $\beta(t)$ is the number of distinct trees that can map to $\alpha(t)$ via catamorphism α . The value of $\beta(t)$, where $t \in \tau$, clearly depends on α . For example, if α is the *Set* catamorphism, $\beta(\text{Leaf}) = 1$; also, $\forall t \in \tau, t \neq \text{Leaf} : \beta(t) = \infty$

since there are an infinite number of trees that have the same set of element values.

Table 3.3: Examples of $\beta(t)$ with the *Multiset* catamorphism


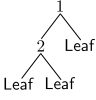
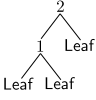
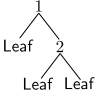
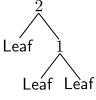
t					
$\alpha(t)$	$\{1\}$	$\{1, 2\}$	$\{1, 2\}$	$\{1, 2\}$	$\{1, 2\}$
$\beta(t)$	1	4	4	4	4

Table 3.3 shows some examples of $\beta(t)$ with the *Multiset* catamorphism. The value of $\beta(\text{Node}(\text{Leaf}, 1, \text{Leaf}))$ is 1 because it is the only tree in the parametric logic that can map to $\{1\}$ by catamorphism *Multiset*. The last four trees are distinct and they are all the trees that can map to the multiset $\{1, 2\}$ via catamorphism *Multiset*. Therefore,

$$\begin{aligned}
 \beta(\text{Node}(\text{Node}(\text{Leaf}, 2, \text{Leaf}), 1, \text{Leaf})) &= \beta(\text{Node}(\text{Node}(\text{Leaf}, 1, \text{Leaf}), 2, \text{Leaf})) \\
 &= \beta(\text{Node}(\text{Leaf}, 1, \text{Node}(\text{Leaf}, 2, \text{Leaf}))) \\
 &= \beta(\text{Node}(\text{Leaf}, 2, \text{Node}(\text{Leaf}, 1, \text{Leaf}))) \\
 &= 4
 \end{aligned}$$

Similarly, if α is catamorphism *DW* in Section 3.3.1, we have $\forall t \in \tau : \beta(t) = \infty$. Δ

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

Definition 3.5 (Monotonic catamorphisms). A catamorphism $\alpha : \tau \rightarrow \mathcal{C}$ is monotonic

iff there exists a constant $h_\alpha \in \mathbb{N}^+$ such that:

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

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

Example 3.7 (Monotonic catamorphisms). Catamorphism *DW* in Section 3.3.1 is monotonic with $h_\alpha = 1$ and *Multiset* is monotonic with $h_\alpha = 2$. An example of a non-monotonic catamorphism is *Mirror* in [2]:

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

Because $\forall t \in \tau : \beta_{\text{Mirror}}(t) = 1$, the catamorphism is not monotonic. We will discuss in detail some examples of monotonic catamorphisms in Section 3.4.2. \triangle

3.4.2 Examples of Monotonic Catamorphisms

This section proves that all sufficiently surjective catamorphisms introduced by Suter et al. [2] are monotonic. These catamorphisms are listed in Table 3.1. Note that the *Sortedness* catamorphism can be defined to allow or not allow duplicate elements [2]; we define *Sortedness_{dup}* and *Sortedness_{nodup}* for the *Sortedness* catamorphism where duplications are allowed and disallowed, respectively.

The monotonicity of *Set*, *SizeI*, *Height*, *Some*, *Min*, and *Sortedness_{dup}* catamorphisms is easily proved by showing the relationship between infinitely surjective abstractions (see Definition 3.1) and monotonic catamorphisms.

Lemma 3.3. *Infinitely surjective abstractions are monotonic.*

Proof. According to Definition 3.1, α is infinitely surjective S -abstraction, where S is a set of trees, if and only if $\beta(t)$ is finite for $t \in S$ and infinite for $t \notin S$. Therefore, α is monotonic with $h_\alpha = \max\{\text{height}(t) \mid t \in S\} + 1$. \square

Theorem 3.4. *Set, SizeI, Height, Some, Min, and Sortedness_{dup} are monotonic.*

Proof. [2] showed that *Set*, *SizeI*, *Height*, and *Sortedness_{dup}* are infinitely surjective abstractions. Also, *Some* and *Min* have the properties of infinitely surjective $\{\text{Leaf}\}$ -abstractions. Therefore, the theorem follows from Lemma 3.3. \square

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

Definition 3.6 (Strict subtree). Given two trees t_1 and t_2 in the tree domain τ , tree t_1 is a *top-down* subtree of tree t_2 , denoted by $t_1 \preceq t_2$, iff:

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

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

$$t_1 \preceq t_2 \wedge \text{size}(t_1) < \text{size}(t_2)$$

Similarly, we define the notion \succ of strict supertrees as the inverse of \prec . Both \prec and \succ are transitive, i.e., $t_1 \prec t_2 \wedge t_2 \prec t_3 \Rightarrow t_1 \prec t_3$ and $t_1 \succ t_2 \wedge t_2 \succ t_3 \Rightarrow t_1 \succ t_3$. Figure 3.5 shows some examples of strict subtrees.

Next, we state Lemma 3.5 before proving that *Multiset*, *List*, and *Sortedness_{nodup}* catamorphisms are monotonic. The proof of Lemma 3.5 is omitted since it is obvious.

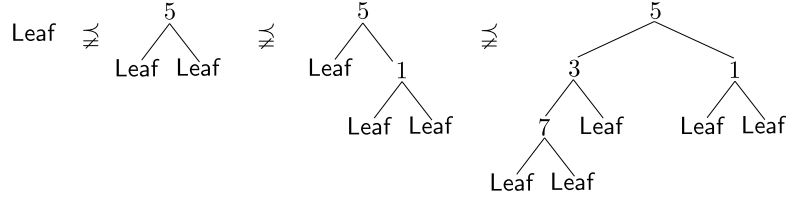


Figure 3.5: Examples of strict subtrees

Lemma 3.5. *For all $h \in \mathbb{N}^+$, any tree of height h must be a strict supertree of at least one tree of height $h - 1$.*

Theorem 3.6. *List catamorphisms are monotonic.*

Proof. Let α be a *List* catamorphism. Let $h_\alpha = 2$. For any tree t such that $\text{height}(t) \geq h_\alpha$, there are exactly $ns(\text{size}(t))$ distinct trees that can map to $\alpha(t)$. Thus

$$\beta(t) = ns(\text{size}(t))$$

Due to Lemma 3.5, there exists t_0 such that $t_0 \not\preceq t \wedge \text{height}(t_0) = \text{height}(t) - 1$, which leads to $\text{size}(t_0) < \text{size}(t)$. From Property 3.3, $\text{height}(t) \geq h_\alpha = 2$ implies $\text{size}(t) \geq 5$. From Lemma 3.2, $ns(\text{size}(t_0)) < ns(\text{size}(t))$, which means $\beta(t_0) < \beta(t)$. \square

Theorem 3.7. *Multiset catamorphisms are monotonic.*

Proof. Let α be a *Multiset* catamorphism. Let $h_\alpha = 2$. Consider any tree t such that $\text{height}(t) \geq h_\alpha$. Let $P(t)$ be the number of distinct permutations of multiset $\alpha(t)$. P is monotonic since $\forall t_1, t_2$ where $t_1 \not\preceq t_2$, $\alpha(t_1)$ is a sub-multiset of $\alpha(t_2)$, which implies $P(t_1) \leq P(t_2)$.

For each permutation of multiset $\alpha(t)$, there are $ns(\text{size}(t))$ distinct trees corresponding to this permutation and those trees are all mapped to $\alpha(t)$. Thus

$$\beta(t) = P(t) \times ns(\text{size}(t))$$

From Lemma 3.5, there exists t_0 such that $t_0 \not\preceq t$ and $height(t_0) = height(t) - 1$, which leads to $size(t_0) < size(t)$. From Property 3.3, $height(t) \geq h_\alpha = 2$ implies $size(t) \geq 5$. From Lemma 3.2, $ns(size(t_0)) < ns(size(t))$. Since P is monotonic and $t_0 \not\preceq t$, $P(t_0) \leq P(t)$. Thus, $ns(size(t_0)) \times P(t_0) < ns(size(t)) \times P(t)$, which causes $\beta(t_0) < \beta(t)$. \square

Theorem 3.8. *Sortedness_{nodup} catamorphisms over integer trees are monotonic.*

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

On the other hand, consider the case when t is sorted. We have $\alpha(t) = (a, b, \text{true})$, where $a = \min(t)$ and $b = \max(t)$. Since there are only a finite number of sorted trees whose elements are distinct and in the finite range $[a, b]$, we have $\beta(t) < \infty$.

From Lemma 3.5, there exists t_0 such that $t_0 \not\preceq t$ and $height(t_0) = height(t) - 1$. Note that t_0 does not need to be sorted. Since $height(t_0) = height(t) - 1 \geq 1$, tree t_0 has at least one internal node; in other words, $ni(t_0) \geq 1$. Because $t_0 \not\preceq t$, we have $ni(t_0) < ni(t)$. Because duplications are not allowed in t , $ni(t) \leq b - a + 1$. Therefore,

$$1 \leq ni(t_0) < b - a + 1 \tag{3.2}$$

From t_0 we construct a tree t'_0 such that $height(t'_0) = height(t) - 1$ and $\beta(t'_0) < \beta(t)$ as follows. Note that if such t'_0 exists, the catamorphism is monotonic by Definition 3.5.

1. Initially, t'_0 is set to be t_0 .
2. Let seq be the sequence of internal nodes obtained from the in-order traversal of t'_0 , excluding all leaves. There are exactly $ni(t_0)$ internal nodes in seq .

3. For the i -th internal node in seq , where $1 \leq i \leq ni(t_0)$, we reset the value of its element to $a + i - 1$.

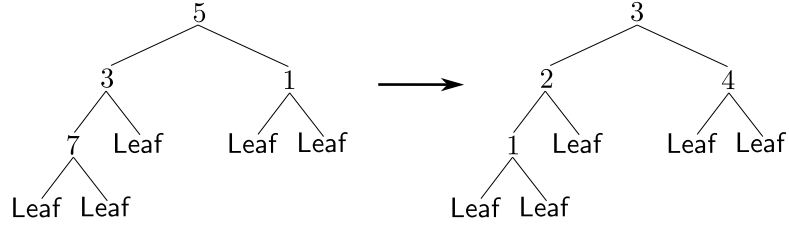


Figure 3.6: Example of tree constructions in the proof of $Sortedness_{nodup}$

After this process², t'_0 is sorted, $height(t'_0) = height(t_0) = height(t) - 1$, $\min(t'_0) = a$, $\max(t'_0) = a + ni(t_0) - 1$ and the range of the element values in t'_0 is in $[a, a + ni(t_0) - 1]$. From Equation (3.2), this range is a strict sub-range of $[a, b]$. As a result, $|\alpha^{-1}((a, a + ni(t_0) - 1, \text{true}))| < |\alpha^{-1}((a, b, \text{true}))|$ since the bigger range for elements we have, the more number of sorted trees with distinct elements we can construct from the range. Consequently, $\beta(t'_0) < \beta(t)$. \square

3.5 Unrolling Decision Procedure - Proof of Correctness

We now prove the correctness of the unrolling decision procedure in Algorithm 3. We start with some properties of monotonic catamorphisms in Section 3.5.1 and then discuss the main proofs in Section 3.5.2. In this section, p stands for the number of disequalities between tree terms in the input formula.

²Figure 3.6 shows an example of t_0 (i.e., the tree on the left) and how we construct t'_0 (i.e., the tree on the right) given t_0 . In this example, seq contains the nodes with values $7 \rightarrow 3 \rightarrow 5 \rightarrow 1$. The minimum value in the sequence is 1. Hence, we reset the sequence in t'_0 to $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. From a tree t_0 with range $[1, 7]$, we have constructed a tree t'_0 of the same height and same size but with minimal range (i.e., $[1, 4]$).

3.5.1 Some Properties of Monotonic Catamorphisms

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

Definition 3.7 (\mathcal{M}_β). $\mathcal{M}_\beta(h)$ is the minimum value of $\beta(t)$ of all trees t of height h :

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

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

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

Lemma 3.9 (Monotonic Property of \mathcal{M}_β). *Function $\mathcal{M}_\beta : \mathbb{N} \rightarrow \mathbb{N}$ satisfies the following monotonic property:*

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

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

Case 1 [$\mathcal{M}_\beta(h) = \infty$]: From Definition 3.7, every tree t_h of height h has $\beta(t_h) = \infty$ because $\mathcal{M}_\beta(h) = \infty$. Hence, every tree t_{h+1} of height $h+1$ has $\beta(t_{h+1}) = \infty$ from Definition 3.5. Thus, $\mathcal{M}_\beta(h+1) = \infty$.

Case 2 [$\mathcal{M}_\beta(h) < \infty$]: Let t_{h+1} be any tree of height $h+1$. From Definition 3.5, there are two sub-cases as follows.

- Sub-case 1 [$\beta(t_{h+1}) = \infty$]: Because $\mathcal{M}_\beta(h) < \infty$, we have $\mathcal{M}_\beta(h) < \beta(t_{h+1})$.
- Sub-case 2 [there exists t_h of height h such that $\beta(t_h) < \beta(t_{h+1})$]: From Definition 3.7, $\mathcal{M}_\beta(h) < \beta(t_{h+1})$.

In both sub-cases, we have $\mathcal{M}_\beta(h) < \beta(t_{h+1})$. Since t_{h+1} can be any tree of height $h + 1$, we have $\mathcal{M}_\beta(h) < \mathcal{M}_\beta(h + 1)$ from Definition 3.7. \square

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

Proof. By induction on h based on Lemma 3.9 and Corollary 1. \square

Theorem 3.10. *For every number $p \in \mathbb{N}^+$, there exists some height $h_p \geq h_\alpha$, computable as a function of p , such that for every height $h \geq h_p$ and for every tree t_h of height h , we have $\beta(t_h) > p$.*

Proof. Let $h_p = h_\alpha + p$. From Corollary 2, $\mathcal{M}_\beta(h_p) > p$. Based on Lemma 3.9, we could show by induction on h that $\forall h \geq h_p : \mathcal{M}_\beta(h) > p$. Hence, this theorem follows from Definition 3.7. \square

Lemma 3.11. *For all number $p \in \mathbb{N}^+$ and for all tree $t \in \tau$, we have:*

$$\beta(t) > p \Rightarrow \beta(\mathbf{Node}(-, -, t)) > p \wedge \beta(\mathbf{Node}(t, -, -)) > p$$

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

3.5.2 Proof of Correctness of the Unrolling-based Decision Procedure

We claim that our unrolling-based decision procedure with monotonic catamorphisms is (1) sound for proofs, (2) sound for models, (3) terminating for satisfiable formulas, and (4) terminating for unsatisfiable formulas. We do not present the proofs for the

first three properties, which can be adapted with minor changes from similar proofs in [55]. Rather, we focus on proving that Algorithm 3 is terminating for unsatisfiable formulas. As defined in Figure 2.1, the logic is described over only conjunctions, but this can easily be generalized to arbitrary formulas using $DPLL(T)$ [49]. The structure of the proof in this case is the same. The outline of the proof is as follows:

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

Now, let ϕ_{in} be an input formula of Algorithm 3. Without loss of generality, we purify the formula ϕ_{in} (as in [2]) and then perform tree unification (as in [53]) on the resulting

formula. If there is a clash during the unification process, ϕ_{in} must be unsatisfiable; otherwise, we obtain a substitution function $\sigma = \{t_{var}^1 \mapsto T_1, \dots, t_{var}^m \mapsto T_m\}$ where each tree variable t_{var}^i , where $1 \leq i \leq m$, does not appear anywhere in tree terms T_1, \dots, T_m . Following [2], the remaining variables (which unify only with themselves and occur only at the leaves of tree terms) we label *parametric variables*.

We substitute for tree variables and obtain a formula $\phi_P = \phi_t \wedge \phi_c \wedge \phi_e \wedge \phi_b$ that is equisatisfiable with ϕ_{in} , where

- ϕ_t contains disequalities over tree terms (tree equalities have been removed through unification),
- ϕ_c contains formulas in the collections theory,
- ϕ_e contains formulas in the element theory, and
- ϕ_b is a set of formulas of the form $c = \alpha(t)$, where c is a variable in the collections theory and t is a tree term.

We observe that given substitution function σ and any model for ϕ_P , it is straightforward to create a model for ϕ_{in} .

Given ϕ_P , Algorithm 3 produces formulas $\phi_{\mathfrak{D}}$ which are the same as ϕ_P except that each term $c = \alpha(t)$ in ϕ_b is replaced by $c = \alpha_{\mathfrak{D}}(t)$, where $\alpha_{\mathfrak{D}}$ is the catamorphism unrolled \mathfrak{D} times.

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

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

We next define a lemma that states that assignments to catamorphisms are *compatible* with all formula constraints other than structural disequalities between trees. We define ϕ_P^* to be the formula obtained by removing all the tree disequality constraints ϕ_t in ϕ_P .

Lemma 3.12. *Given a formula ϕ_P^* with monotonic catamorphism α and correct range predicate R_α , after $\mathfrak{D} \geq \text{depth}_{\phi_{in}}^{\max} + 1$ unrollings, if $\phi_{\mathfrak{D}}$ has a model, then ϕ_P^* also has a model.*

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

- If $\text{height}(t_d) < \text{depth}_{\phi_{in}}^{\max} + 1$, t_d belongs to a set of complete trees generated by Algorithm 3. In this case, $\alpha_{\mathfrak{D}}(t_d) = \alpha(t_d)$, so we can set $t^* = t_d$.
- If $\text{height}(t_d) \geq \text{depth}_{\phi_{in}}^{\max} + 1$, then the calculation of $\alpha_{\mathfrak{D}}(t_d)$ involves at least one uninterpreted function $U_\alpha(t_{uif})$ over some subtree³ t_{uif} of t_d , and it is possible that $\alpha(t_{uif}) \neq U_\alpha(t_{uif})$. Since we have unrolled the catamorphism $\text{depth}_{\phi_{in}}^{\max} + 1$ times, tree t_{uif} corresponds to a subtree of a parametric variable in the original problem. The only constraints on the structure of such subtrees are those on the value of $U_\alpha(t_{uif})$; we call them uninterpreted-constrained subtrees. Furthermore, by the condition of R_α , there exists a concrete subtree t' such that $U_\alpha(t_{uif}) = \alpha(t')$.

We construct t^* by replacing all such uninterpreted-constrained subtrees t_{uif} with

³We use the standard definition of subtrees in the proof of Lemma 3.12. That is, t_1 is a subtree of t_2 if and only if the root of t_1 is a vertex of t_2 . Note that this definition is different from the notion of *top-down* subtrees in Definition 3.6.

concrete subtrees t' such that $U_\alpha(t_{uif}) = \alpha(t')$. The structure of t^* matches the structure of t_d up to the level of the deepest tree term in ϕ_P^* (whose depth cannot be deeper than $depth_{\phi_{in}}^{\max}$), so all other formula constraints on t_d are preserved.

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

Theorem 3.13. *Given a formula ϕ_{in} with monotonic catamorphism α and correct range predicate R_α , after $\mathfrak{D} = depth_{\phi_{in}}^{\max} + 1 + h_p$ unrollings, if $\phi_{\mathfrak{D}}$ has a model, then ϕ_{in} also has a model.*

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

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

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

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

From Lemma 3.12, as long as we can construct a concrete tree $t' \in \tau$ such that $\alpha(t') = U_\alpha(t_{uif})$ for each t_{uif} where $U_\alpha(t_{uif})$ is in $\mathcal{M}_\mathfrak{D}$ while still maintaining the disequalities between tree terms in $\phi_\mathfrak{D}$, we can have a corresponding model \mathcal{M} for ϕ_P . There are two cases for each tree term t_{term} in $\phi_\mathfrak{D}$ as follows.

Case 1 [$height(t_{term}) < depth_{\phi_{in}}^{\max} + 1 + h_p$]: Tree t_{term} is complete because uninterpreted functions $U_\alpha(t_{uif})$ can only appear at depth $depth_{\phi_{in}}^{\max} + 1 + h_p$. For every tree term $t'_{term} \in \phi_\mathfrak{D}$ such that the disequality $t_{term} \neq t'_{term}$ is in $\phi_\mathfrak{D}$, there are two sub-cases to consider:

- Sub-case 1 [$height(t'_{term}) < depth_{\phi_{in}}^{\max} + 1 + h_p$]: In this case, tree term t'_{term} is also complete; hence, the two tree terms t_{term} and t'_{term} must be different since $\phi_\mathfrak{D}$ has a model $\mathcal{M}_\mathfrak{D}$ for them.
- Sub-case 2 [$height(t'_{term}) \geq depth_{\phi_{in}}^{\max} + 1 + h_p$]: The two tree terms t_{term} and t'_{term} must also be different since their heights are different. In particular, $height(t'_{term}) \geq depth_{\phi_{in}}^{\max} + 1 + h_p > height(t_{term})$.

Hence, the disequality constraints in $\phi_\mathfrak{D}$ between t_{term} and other tree terms are satisfied.

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

Next, we construct a set ST_{h_p} of trees of heights at least h_p as follows. Initially, ST_{h_p} is empty. Starting from each vertex u at depth $depth_{\phi_{in}}^{\max} + 1 + h_p$ of t_{term} , we go bottom-up h_p steps to reach a node u_{h_p} , then add t_{h_p} to ST_{h_p} if $t_{h_p} \notin ST_{h_p}$ where t_{h_p}

is the tree rooted at u_{h_p} . After this process, ST_{h_p} is not empty since $height(t_{term}) \geq depth_{\phi_{in}}^{\max} + 1 + h_p$. We visualize this step in Figure 3.7.

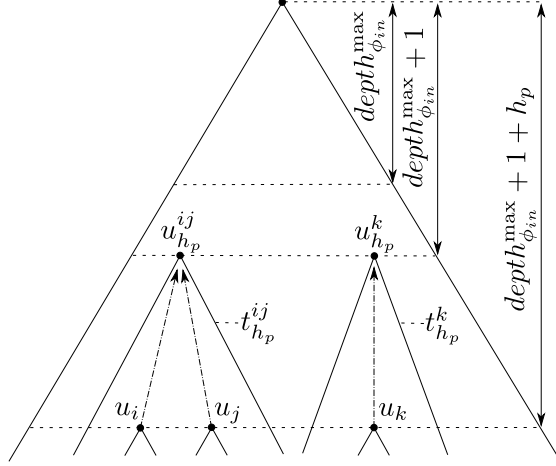


Figure 3.7: Construct a set of trees ST_{h_p} from t_{term}

For each tree $t_{h_p} \in ST_{h_p}$, the relative location of its root u_{h_p} in t_{term} is at depth $depth_{\phi_{in}}^{\max} + 1$. Therefore, there is no disequality constraint between t_{h_p} and any original tree terms in $\phi_{\mathcal{D}}$, which cannot locate at a depth bigger than $depth_{\phi_{in}}^{\max}$. Hence, the choice of t_{h_p} only depends on $\alpha(t_{h_p})$. Because $height(t_{h_p}) \geq h_p$, we have $\beta(t_{h_p}) > p$ from Theorem 3.10. From Lemma 3.11, we have $\beta(t_{term}) > p$.

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

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

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

Proof. This is the immediate contrapositive of Theorem 3.13. Suppose ϕ_{in} does not have a model. In this case, $\phi_{\mathfrak{D}}$ also does not have a model and the SMT solver \mathcal{S} will return *UNSAT*. \square

This proof implies that Algorithm 3 terminates after no more than $depth_{\phi_{in}}^{\max} + 1 + h_p$ number of unrollings for unsatisfiable formulas. If the number of unrollings exceeds $depth_{\phi_{in}}^{\max} + 1 + h_p$, we conclude that ϕ_{in} is satisfiable; note that if we are interested in complete tree models, we still need to keep unrolling until we reach line 5 in Algorithm 3.

Corollary 4. *Monotonic catamorphisms are sufficiently surjective.*

Proof. Recall the definition of sufficient surjectivity in Definition 3.3. We define $\alpha_{h_p}(t)$ as the unrolling of catamorphism α to depth h_p for tree t , and $CC(\alpha_{h_p}(t))$ as the set of control conditions for the unrolling.

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

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

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

We note that the set S_p is clearly finite.

Finally, we show that $M_p(c)$ implies $|\alpha^{-1}(c)| > p$. Given a catamorphism value c , by virtue of the restriction that at least one control condition $b \in CC(\alpha_{h_p}(t))$ is false and R_α , we can map that value to some tree t such that the height of t is greater than h_p . If c satisfies $M_p(c)$, then by Theorem 3.10, $|\alpha^{-1}(c)| > p$. \square

Chapter 4

Associative-Commutative (AC) Catamorphisms

In the previous chapter, we have presented an unrolling-based decision procedure that is guaranteed to be both sound and complete with monotonic catamorphisms. When it comes to catamorphisms, there are many interesting open problems, for example: when is it possible to combine catamorphisms in a complete way, or how computationally expensive is it to solve catamorphism problems? This chapter attempts to characterize a useful class of “combinable” monotonic catamorphisms that maintain completeness under composition.

We name this class *associative-commutative* (AC) catamorphisms due to the associative and commutative properties of the operators used in the catamorphisms. AC catamorphisms have some very powerful important properties: they are detectable¹, combinable, and impose an exponentially small upper bound of the number of unrollings. Many catamorphisms that have been presented so far in this dissertation are

¹*detectable* in this context means that it is possible to determine whether or not a catamorphism is an AC catamorphism using an SMT solver.

in fact AC.

4.1 Definition

Definition 4.1 (AC catamorphism). Catamorphism $\alpha : \tau \rightarrow \mathcal{C}$ is AC if

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

where $\oplus : (\mathcal{C}, \mathcal{C}) \rightarrow \mathcal{C}$ 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 $\delta : \mathcal{E} \rightarrow \mathcal{C}$ is a function that maps² an element value in \mathcal{E} into a corresponding value in \mathcal{C} .

Like catamorphisms defined in [2, 55], AC catamorphisms are fold functions mapping the content of a tree in the parametric logic into a value in a collection domain where a decision procedure is assumed to be available. There are two main differences in the presentation between AC catamorphisms and those in [2, 55]. First, the combine function is replaced by an associative, commutative operator \oplus and function δ . Second, Leaf is mapped to the identity value of operator \oplus instead of the empty value of \mathcal{C} (though the two quantities are usually the same in practice).

Detection: Unlike sufficiently surjective catamorphisms, AC catamorphisms are detectable. A catamorphism, written in the format in Definition 4.1, is AC if the following conditions hold:

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

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

These conditions can be easily proved by SMT solvers [41, 40] or theorem provers such as ACL2 [16].

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

Definition 4.2 (Signature of AC catamorphisms). The signature of an AC catamorphism α is defined as follows:

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

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

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

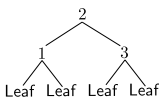
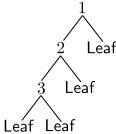
$$\alpha(t) = \begin{cases} id_{\oplus} & \text{if } t = \text{Leaf} \\ \delta(e_1) \oplus \delta(e_2) \oplus \dots \oplus \delta(e_{ni(t)}) & \text{otherwise} \end{cases}$$

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

Example 4.1 (AC catamorphisms). In Table 3.1, *Height*, *List*, *Some*, and *Sortedness* are not AC because their values depend on the positions of tree elements. Table 4.1

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

Table 4.1: Examples of monotonic but not associative-commutative catamorphisms

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

Other catamorphisms in Table 3.1, including *Set*, *Multiset*, *SizeI*, and *Min* are AC. The *DW* catamorphism in Section 3.3.1 is also AC. In the *DW* catamorphism, the operator \oplus is $+$, the identity element id_{\oplus} is 0, and the mapping function is $\delta(e) = (\text{ite}(\text{dirty}(e))\ 1\ 0)$. In the *Multiset* catamorphism, the three factors are \uplus , \emptyset , and $\delta(e) = \{e\}$, respectively.

Furthermore, we could define other AC catamorphisms based on well-known associative and commutative operators such as $+$, \cap , \max , \vee , \wedge , etc. We could also use user-defined functions as the operators in AC catamorphisms. For example, the following user-defined operator is an AC operator:

$$\text{SumMagnOp}(e_1, e_2) = |e_1| + |e_2|$$

in this case, we will need the help of an additional analysis tool to verify that all conditions for AC catamorphisms are met. △

4.2 The Monotonicity of AC Catamorphisms

AC catamorphisms are not only automatically detectable but also monotonic. Thus, they can be used in Algorithm 3.

Lemma 4.1. *If α is an AC catamorphism then*

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

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

Property 3.2 implies that any shape of size $size(t)$ must have exactly $ni(t)$ SNode(s) and $nl(t)$ SLeaf(s). Let $sh_1, \dots, sh_{ns(size(t))}$ be all shapes of size $size(t)$. From sh_i , where $1 \leq i \leq ns(size(t))$, we construct a tree t_i by converting every SLeaf in sh_i into a Leaf and converting the j -th SNode in sh_i into a structurally corresponding Node with element value L_j , where $1 \leq j \leq ni(t)$. Figure 4.1 shows an example of the conversion process.

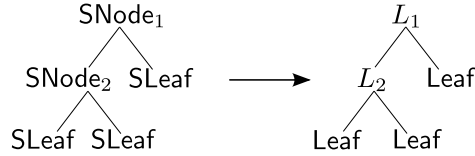


Figure 4.1: Example of converting a shape into a tree

After this process, $t_1, \dots, t_{ns(size(t))}$ are mutually different because their shapes $sh_1, \dots, sh_{ns(size(t))}$ are distinct. From Corollary 5, we obtain

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

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

Theorem 4.2. *AC catamorphisms are monotonic.*

Proof. Let α be an AC catamorphism. Let $h_\alpha = 4$. Consider any tree t such that $height(t) \geq h_\alpha = 4$. If $\beta(t) = \infty$, the monotonic condition for t in Definition 3.5 holds.

Suppose on the other hand that $\beta(t) < \infty$. Due to Lemma 3.5, there exists t_0 such that $t_0 \not\preceq t \wedge height(t_0) = height(t) - 1 \geq 3$, which implies $size(t_0) \geq 7$ due to Property 3.3. Due to Lemma 3.2, $ns(size(t_0)) \geq ns(7) > 2$. From Lemma 4.1,

$$\beta(t_0) > 2 \tag{4.1}$$

which is mathematically equivalent to

$$\beta(t_0) < 2 \times (\beta(t_0) - 1) \tag{4.2}$$

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

$$\alpha(t) = \alpha(t_0) \oplus \delta(e_1) \oplus \delta(e_2) \oplus \dots \oplus \delta(e_{|Q|}) \tag{4.3}$$

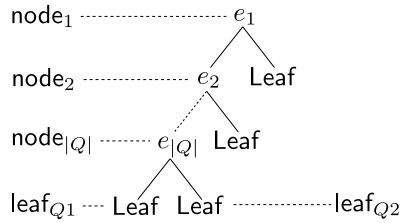


Figure 4.2: Construct t_Q from $e_1, \dots, e_{|Q|}$

Next, we construct a tree t_Q from $e_1, \dots, e_{|Q|}$. The construction of t_Q is shown in Figure 4.2. 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: $node_{|Q|} = \text{Node}(\text{Leaf}, e_{|Q|}, \text{Leaf})$ and

$\text{node}_j = \text{Node}(\text{node}_{j+1}, e_j, \text{Leaf})$, where $Q > j \geq 1$. Let leaf_{Q1} and leaf_{Q2} be the left and right leaves of $\text{node}_{|Q|}$, respectively.

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

$$\beta(t_0) - 1 \leq \text{num}_{t_0} \leq \beta(t_0) \quad (4.4)$$

From Equations (4.1) and (4.4), we have $\text{num}_{t_0} \geq 2$, which means there are at least 2 distinct bigger-than-Leaf trees that can map to $\alpha(t_0)$. Let t'_0 and t''_0 be any two of them. That is, t'_0 and t''_0 are two different bigger-than-Leaf trees and

$$\alpha(t'_0) = \alpha(t''_0) = \alpha(t_0) \quad (4.5)$$

Let us consider t'_0 . Let t'_{01} and t'_{02} be the trees obtained by replacing leaf_{Q1} and leaf_{Q2} in t_Q with t'_0 , respectively. Since $t'_0 \neq \text{Leaf}$, we have $t'_{01} \neq t'_{02}$. From Corollary 5, we have

$$\alpha(t'_{01}) = \alpha(t'_{02}) = \alpha(t'_0) \oplus \delta(e_1) \oplus \delta(e_2) \oplus \dots \oplus \delta(e_{|Q|})$$

From Equation (5.1), Equation (5.2), and the above equation, we obtain

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

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

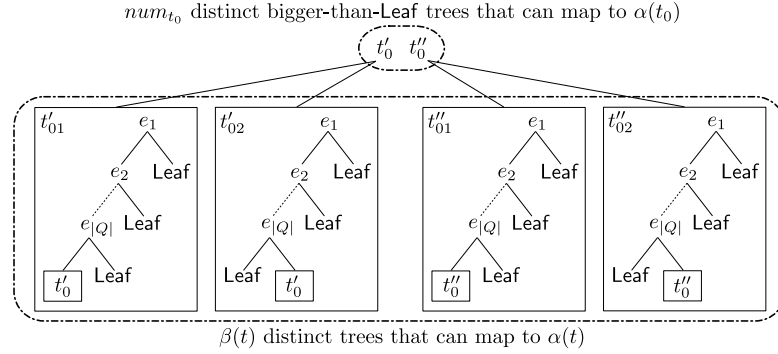


Figure 4.3: Relationship between t'_0, t''_0 and $t'_{01}, t'_{02}, t''_{01}, t''_{02}$

Let us consider t''_0 . We construct t''_{01} and t''_{02} from t''_0 and t_Q such that $\alpha(t''_{01}) = \alpha(t''_{02}) = \alpha(t)$ using the same method as above. It is clear that $t''_{01} \neq t'_{01}$ since $t''_0 \neq t'_0$. Also, $t''_{02} \neq t'_{01}$ since t''_{02} has leaf_{Q1} but does not have leaf_{Q2} while t'_{01} has leaf_{Q2} but does not have leaf_{Q1} . Similarly, we can show that $t''_{01} \neq t'_{02}$ and $t''_{02} \neq t'_{02}$. Thus, four trees $t'_{01}, t'_{02}, t''_{01}$, and t''_{02} obtained from t'_0 and t''_0 are mutually different. The relationship between them is shown in Figure 4.3.

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

$$\begin{aligned}
 2 \times num_{t_0} &\leq \beta(t) \\
 \therefore 2 \times (\beta(t_0) - 1) &\leq \beta(t) \quad [\text{From Equation (4.4)}] \\
 \therefore \beta(t_0) &< \beta(t) \quad [\text{From Equation (4.2)}]
 \end{aligned}$$

As a result, α is monotonic based on Definition 3.5. □

4.3 Exponentially Small Upper Bound of the Number of Unrollings

In the proof of Theorem 3.10, we use $h_p = h_\alpha + p$ to guarantee that the algorithm terminates after unrolling no more than $depth_{\phi_{in}}^{\max} + 1 + h_p$ times. The upper bound implies that the number of unrollings may be large when p is large, leading to a high complexity for the algorithm with monotonic catamorphisms.

In this section, we demonstrate that in the case of AC catamorphisms, we could choose a different value for h_p such that not only the termination of the algorithm is guaranteed with h_p , but also the growth of h_p is *exponentially small* compared with that of p . Recall from the proof of Theorem 3.10 that as long as we can choose $h_p \geq h_\alpha$ such that $\mathcal{M}_\beta(h_p) > p$, Theorem 3.10 will follow. We will define such h_p after proving the following important lemma.

Lemma 4.3. *If α is AC then $\forall h \in \mathbb{N} : \mathcal{M}_\beta(h) \geq \mathbb{C}_h$.*

Proof. Let $t_h \in \tau$ be any tree of height h . We have:

$$\begin{aligned} \beta(t_h) &\geq ns(\text{size}(t_h)) && \text{[From Lemma 4.1]} \\ \therefore \beta(t_h) &\geq ns(2h + 1) && \text{[From Property 3.3 and Lemma 3.2]} \\ \therefore \beta(t_h) &\geq \mathbb{C}_h && \text{[From Lemma 3.1]} \end{aligned}$$

As a result, $\mathcal{M}_\beta(h) \geq \mathbb{C}_h$ from Definition 3.7. □

Let $h_p = \max\{h_\alpha, \min\{h \mid \mathbb{C}_h > p\}\}$. By construction, $h_p \geq h_\alpha$ and $\mathbb{C}_{h_p} > p$. From Lemma 4.3, $\mathcal{M}_\beta(h_p) \geq \mathbb{C}_{h_p} > p$. Thus, Theorem 3.10 follows.

Moreover, the growth³ of \mathbb{C}_n is exponential [75]. Thus, h_p is exponentially smaller than p since $\mathbb{C}_{h_p} > p$. For example, when $p = 10^4$, we can choose $h_p = 10$ since

³One can show that $\mathbb{C}_n \sim \frac{4^n}{\sqrt{\pi n^3}}$ by using Stirling's approximations for $n!$ [75].

$\mathbb{C}_{10} > 10^4$ as shown in Table 3.2. Similarly, when $p = 5 \times 10^4$, we can choose $h_p = 11$. In the example, we assume that $h_\alpha \leq 10$, which is true for all catamorphisms in this dissertation.

4.4 Combining AC Catamorphisms

Let $\alpha_1, \dots, \alpha_m$ be m AC catamorphisms where the signature of the i -th catamorphism ($1 \leq i \leq m$) is $\text{sig}(\alpha_i) = \langle \mathcal{C}_i, \mathcal{E}, \oplus_i, \delta_i \rangle$. Catamorphism α with signature $\text{sig}(\alpha) = \langle \mathcal{C}, \mathcal{E}, \oplus, \delta \rangle$ is a combination of $\alpha_1, \dots, \alpha_m$ 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 $\langle x_1, \dots, x_m \rangle, \langle y_1, \dots, y_m \rangle \in \mathcal{C}$:

$$\text{id}_\oplus = \langle \text{id}_{\oplus_1}, \text{id}_{\oplus_2}, \dots, \text{id}_{\oplus_m} \rangle$$

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

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

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

- α is defined as in Definition 4.1.

Example 4.2 (Combine AC catamorphisms). Consider *Set* and *SizeI* catamorphisms in Table 3.1, which are AC as demonstrated in Example 4.1. When we combine the two AC catamorphisms, assuming *Set* is used before *SizeI*, we get a new catamorphism *SetSizeI* that can map a tree to a pair of values: the former is the set of all the elements in the tree and the latter is an integer indicating the number of internal nodes in the tree. For example, if we apply *SetSizeI* to the tree in Figure 3.1, we get $\langle \{1, 2\}, 2 \rangle$. \triangle

Theorem 4.4. *Every catamorphism obtained from the combination of AC catamorphisms is also AC.*

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

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

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

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

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

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

of $\oplus_1, \dots, \oplus_m$, respectively:

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

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

Chapter 5

Parameterized

Associative-Commutative (PAC)

Catamorphisms

From the base of monotonic and AC catamorphisms, there are many useful directions to extend the work. One of the most challenging open questions is that: can we generalize the idea of catamorphisms to support parameters? As we can observe from the previous chapters as well as from the literature [2, 55, 67], parameters (except the data type arguments) have not been formally supported by catamorphisms when it comes to abstracting algebraic data types. In other words, the decidability of catamorphism functions involving parameters in addition to the data type argument has not been studied.

In this chapter¹, we generalize AC catamorphisms to support additional parameters. This extension, called *parameterized associative-commutative* (PAC) catamorphisms subsumes the associative-commutative class, 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. We show that PAC catamorphisms 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 Chapter 4 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.

¹Technically, we can combine this chapter with Chapter 4, but we would prefer to keep them separate for the sake of clarity. In addition, we believe that AC catamorphisms can be an important foundation for many future work; thus, we decided to devote a whole chapter for the crucial class of 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 this chapter is organized as follows. Section 5.1 proposes PAC catamorphisms, whose benefits are demonstrated with concrete examples in Section 5.2. To ensure the completeness of decision procedure, PAC catamorphisms must be monotonic, which is proved in Section 5.3. Section 5.4 shows that PAC catamorphisms preserve all powerful properties of AC catamorphisms. Section 5.5 summarizes the relationship between different types of catamorphisms.

5.1 Parameterized Associative-Commutative Abstractions

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 5.2). Although more general, PAC catamorphisms are still monotonic (Section 5.3) and they preserve all the powerful characteristics of AC catamorphisms (Section 5.4).

Definition 5.1 (Parameterized Associative-Commutative (PAC) Catamorphisms). Given a predicate $\text{pr} : \mathcal{E} \rightarrow \text{bool}$, a value $c_{\text{leaf}} \in \mathcal{C}$, a value $c_{\text{pr}} \in \mathcal{C}$, and a boolean value rec ,

catamorphism² $\alpha : \tau \rightarrow \mathcal{C}$ is PAC if:

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

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

Signature. Due to the generalization, the signature of PAC catamorphisms has four more elements than that of AC catamorphisms, including the value $c_{\text{leaf}} \in \mathcal{C}$ for the `Leaf` case, the value $c_{\text{pr}} \in \mathcal{C}$ 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 5.2 (PAC signature). The signature of a PAC catamorphism α is:

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

²Strictly speaking, a PAC catamorphism should be in the form $\alpha(t, \text{pr}, c_{\text{leaf}}, c_{\text{pr}}, \text{rec})$. However, since the last four parameters are unchanged during the argument passing process (i.e., $\alpha(t, \text{pr}, c_{\text{leaf}}, c_{\text{pr}}, \text{rec})$ is computed in terms of $\alpha(t_L, \text{pr}, c_{\text{leaf}}, c_{\text{pr}}, \text{rec})$ and $\alpha(t_R, \text{pr}, c_{\text{leaf}}, c_{\text{pr}}, \text{rec})$), we do not explicitly write the four parameters for brevity.

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

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

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

Corollary 6 (Values of PAC catamorphisms). *The value of $\alpha(t)$, where α is a PAC catamorphism, only depends on the values of elements in t and does not depend on the relative positions of the element values iff (1) $\text{rec} = \text{true}$ or (2) $\text{rec} = \text{false}$ and \nexists element value $e \in \mathcal{E}$ in t : $\text{pr}(e) = \text{true}$.*

5.2 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.

5.2.1 Expressiveness

Theorem 5.1. *PAC catamorphisms are more expressive than AC catamorphisms.*

Proof. Given a PAC catamorphism α as defined in Definition 5.1, if we fix c_{leaf} to be id_{\oplus} and predicate pr to be **false**, α becomes:

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

which is an AC catamorphism by Definition 4.1. Therefore, AC catamorphisms are a special case of PAC catamorphisms. If we vary the values of parameters pr , c_{leaf} , c_{pr} , and rec , we will get some PAC catamorphisms that are not AC. \square

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

$$NLeaves(t) = \begin{cases} 1 & \text{if } t = \text{Leaf} \\ NLeaves(t_L) + NLeaves(t_R) & \text{if } t = \text{Node}(t_L, -, t_R) \end{cases}$$

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

Also, *PAC catamorphisms support predicates that can be defined over element values* while AC catamorphisms do not. For example, suppose we have a predicate $isBad : \mathcal{E} \rightarrow \text{bool}$ that determines whether an internal node is bad. We consider an internal node $\text{Node}(-, e, -)$ to be bad if $isBad(e) = \text{true}$. Now consider a catamorphism called $NGN : \tau \rightarrow \text{int}$ (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:

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

By Corollary 5, 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, we can still define this catamorphism as a PAC catamorphism.

5.2.2 Usability

Let us consider $Negative : \tau \rightarrow \text{bool}$, an AC catamorphism that maps a tree into **true** if all of its element values are negative:

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

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

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

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

catamorphism $Forall : \tau \rightarrow \text{bool}$:

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

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

5.2.3 Efficiency

Theorem 5.2. *Given an AC catamorphism α_{AC} and a PAC catamorphism α_{PAC} , for every tree $t \in \tau$ that the two catamorphisms accept as input, $\alpha_{PAC}(t)$ always requires less or equal number of recursive calls to compute its value than $\alpha_{AC}(t)$.*

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

Take the $Forall$ 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 $Forall(t_L)$ and $Forall(t_R)$ are computed regardless what $pr_u(e)$ is. However, if $pr_u(e) = \text{false}$, we can conclude

that $Forall(t) = \text{false}$ without computing $Forall(t_L)$ and $Forall(t_R)$. Based on this observation, we can rewrite the catamorphism as follows:

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

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 catamorphisms.

Table 5.1: Some PAC catamorphisms that are not AC

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

Table 5.1 shows the full definitions of all PAC catamorphisms discussed in Section 5.2. They are some PAC catamorphisms that cannot be naturally expressed in an AC way. Note that from Theorem 5.1, every AC catamorphism is PAC.

5.3 The Monotonicity of PAC Catamorphisms

To work with our unrolling-based decision procedure for algebraic data types in Section 3.3, PAC catamorphisms must be monotonic (see Definition 3.5). In this section, we prove the monotonicity of PAC catamorphisms. First, let us introduce some new supporting lemmas and corollaries.

Definition 5.3 (Satisfiable Predicate). Predicate $\text{pr} : \mathcal{E} \rightarrow \text{bool}$ is satisfiable if $\exists e \in \mathcal{E} : \text{pr}(e) = \text{true}$.

Lemma 5.3. *Given a PAC catamorphism α with $\text{rec} = \text{false}$, if pr is satisfiable, then $|\alpha^{-1}(c_{\text{pr}})| = \infty$.*

Proof. Since pr is satisfiable, from Definition 5.3, there exists $e_0 \in \mathcal{E}$ such that $\text{pr}(e_0) = \text{true}$. 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 $\text{pr}(e_0) = \text{true}$ and $\text{rec} = \text{false}$. Hence, $|\alpha^{-1}(c_{\text{pr}})| = \infty$. \square

Corollary 7. *Given a PAC catamorphism α with $\text{rec} = \text{false}$ and a tree $t \in \tau$, if there exists an element value $e_t \in t$ such that $\text{pr}(e_t) = \text{true}$, then $\beta(t) = \infty$.*

Proof. Let t_{e_t} be the tree rooted at e_t in t . Since $\text{pr}(e_t) = \text{true}$ and $\text{rec} = \text{false}$, we have $\alpha(t_{e_t}) = c_{\text{pr}}$ by Definition 5.1. By Lemma 5.3, $|\alpha^{-1}(c_{\text{pr}})| = \infty$. In other words, $|\alpha^{-1}(\alpha(t_{e_t}))| = \beta(t_{e_t}) = \infty$. Thus, we have $\beta(t) = \infty$ by Lemma 3.11. \square

Corollary 8. *Given a PAC catamorphism α with $\text{rec} = \text{false}$ 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 $\text{pr}(e_{t'}) = \text{true}$.

Proof. This corollary follows immediately from Corollary 7. \square

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 5.4. *If α is a PAC catamorphism then $\forall t \in \tau : \beta(t) \geq ns(\text{size}(t))$.*

Proof. Let t be any tree in τ . If $\text{rec} = \text{true}$, from Corollary 6, 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 4.2 with minor changes.

If $\text{rec} = \text{false}$, 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 8, there does not exist any element value e_t in t such that $\text{pr}(e_t) = \text{true}$. Hence, from Corollary 6, 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 4.1. \square

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 $\text{rec} = \text{true}$ and the other one is for PAC catamorphisms with $\text{rec} = \text{false}$. The proof of monotonicity of PAC catamorphisms involves some properties of tree shapes (see Definition 3.2) and strict subtrees (see Definition 3.6) defined before.

Lemma 5.5. *PAC catamorphisms with $\text{rec} = \text{true}$ are monotonic.*

Proof. Let α be a PAC catamorphism with $\text{rec} = \text{true}$. Let $h_\alpha = 4$. Consider any tree $t \in \tau$ such that $\text{height}(t) \geq h_\alpha = 4$. If $\beta(t) = \infty$, the monotonic condition for t in Definition 3.5 holds.

On the other hand, suppose $\beta(t) < \infty$. By Lemma 3.5, $\exists t_0 \in \tau : t_0 \not\preceq t \wedge \text{height}(t_0) = \text{height}(t) - 1 \geq 3$. Let Q be the set of internal nodes that are in t but not in t_0 . Q is not empty since $t_0 \not\preceq t$. Let $e_1, \dots, e_{|Q|}$ be the elements stored in $|Q|$ nodes in Q . We define a new mapping function as follows:

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

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

$$\alpha(t) = \alpha(t_0) \oplus \delta'(e_1) \oplus \delta'(e_2) \oplus \dots \oplus \delta'(e_{|Q|}) \oplus \underbrace{c_{\text{leaf}} \oplus \dots \oplus c_{\text{leaf}}}_{|Q| \text{ occurrences of } c_{\text{leaf}}} \quad (5.1)$$

Next, we construct a tree t_Q from $e_1, \dots, e_{|Q|}$ as in Figure 4.2. By Property 3.3, $height(t_0) \geq 3$ implies $size(t_0) \geq 7$. By Lemma 3.2, $ns(size(t_0)) \geq ns(7) > 2$. By Lemma 5.4, $\beta(t_0) \geq ns(size(t_0)) > 2$. 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

$$\alpha(t'_0) = \alpha(t''_0) = \alpha(t_0) \quad (5.2)$$

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 $t_Q \neq \text{Leaf}$, we have $t'_{01} \neq t'_{02}$. We have

$$\begin{aligned} & \alpha(t'_{01}) = \alpha(t'_{02}) \\ &= \alpha(t'_0) \oplus \delta'(e_1) \oplus \delta'(e_2) \oplus \dots \oplus \delta'(e_{|Q|}) \oplus \underbrace{c_{\text{leaf}} \oplus \dots \oplus c_{\text{leaf}}}_{|Q| \text{ occurrences of } c_{\text{leaf}}} \\ &= \alpha(t) \quad [\text{From Equations (5.1) and (5.2)}] \end{aligned}$$

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 $\alpha(t)$.*

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. Figure 5.1 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*

at least $\beta(t_0) - 1$ distinct bigger-than-Leaf trees that can map to $\alpha(t_0)$

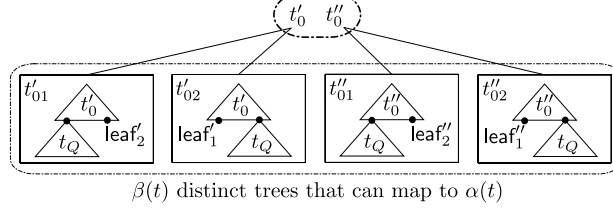


Figure 5.1: Relationship between t'_0, t''_0 and $t'_{01}, t'_{02}, t''_{01}, t''_{02}$

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, α is monotonic based on Definition 3.5. \square

Lemma 5.6. *PAC catamorphisms with $\text{rec} = \text{false}$ are monotonic.*

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

1. If pr is unsatisfiable, catamorphism α is also a PAC catamorphism with $\text{rec} = \text{true}$. Thus, α is monotonic from Lemma 5.5.
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 $\exists e_t \in t : \text{pr}(e_t) = \text{true}$, we show that $\beta(t) = \infty$, which implies the monotonicity of α by Definition 3.5.
 - (b) If $\nexists e_t \in t : \text{pr}(e_t) = \text{true}$, 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, α is monotonic by Definition 3.5.

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

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

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 5.5. Thus, α is monotonic.

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

Sub-case 1: [There exists an element value e_t in t such that $\text{pr}(e_t) = \text{true}$]. From Corollary 7, $\beta(t) = \infty$. 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 3.5, there exists $t_0 \in \tau$ such that $t_0 \not\preceq t$ and $\text{height}(t_0) = \text{height}(t) - 1 \geq 1$. 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 $t_0 \not\preceq t$. Let $e_1, e_2, \dots, e_{|Q|}$ 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

$$\alpha(t) = \alpha(t_0) \oplus \delta(e_1) \oplus \delta(e_2) \oplus \dots \oplus \delta(e_{|Q|}) \oplus \underbrace{c_{\text{leaf}} \oplus \dots \oplus c_{\text{leaf}}}_{|Q| \text{ occurrences of } c_{\text{leaf}}} \quad (5.3)$$

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

$$\alpha(t'_0) = \alpha(t_0) \quad (5.4)$$

Next, we construct a tree t_Q from $e_1, \dots, e_{|Q|}$ as in Figure 4.2. Given t_Q , by replacing

leaf_{Q_1} with t'_0 , we obtain a distinct tree t'_{01} such that:

$$\alpha(t'_{01}) = \alpha(t'_0) \oplus \delta(e_1) \oplus \delta(e_2) \oplus \dots \oplus \delta(e_{|Q|}) \oplus \underbrace{c_{\text{leaf}} \oplus \dots \oplus c_{\text{leaf}}}_{|Q| \text{ occurrences of } c_{\text{leaf}}} \quad (5.5)$$

From Equations (5.3), (5.4), and (5.5), we have: $\alpha(t) = \alpha(t'_{01})$. Thus, from each tree t'_0 in the set of $\beta(t_0)$ distinct trees that can map to $\alpha(t_0)$, we can generate a distinct tree t'_{01} that can map to $\alpha(t)$. Hence, from $\beta(t_0)$ distinct trees that can map to $\alpha(t_0)$, we can generate at least $\beta(t_0)$ distinct trees that can map to $\alpha(t)$.

Let $B^{\text{leaf}_{Q_1}}$ 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 $B^{\text{leaf}_{Q_1}}$ since leaf_{Q_2} is untouched during the substitution process.

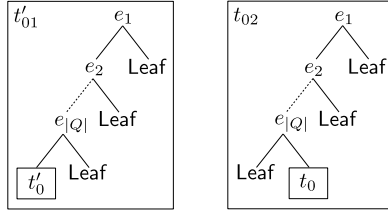


Figure 5.2: The constructions of t'_{01} and t_{02} .

Next, we show that there exists at least another tree that can map to $\alpha(t)$ but is not in $B^{\text{leaf}_{Q_1}}$. Given t_Q , we now replace leaf_{Q_2} with t_0 to obtain a tree t_{02} . The constructions of t'_{01} and t_{02} are shown in Figure 5.2. We have:

$$\begin{aligned} \alpha(t_{02}) &= \alpha(t_0) \oplus \delta(e_1) \oplus \delta(e_2) \oplus \dots \oplus \delta(e_{|Q|}) \oplus \underbrace{c_{\text{leaf}} \oplus \dots \oplus c_{\text{leaf}}}_{|Q| \text{ occurrences of } c_{\text{leaf}}} \\ &= \alpha(t) \quad [\text{From Equation (5.3)}] \end{aligned}$$

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 leaf_{Q_2} in t_Q with t_0 to obtain t_{02} , leaf_{Q_2} must not be

in t_{02} . Moreover, since leaf_{Q2} is in all the trees in $B^{\text{leaf}_{Q1}}$, tree t_{02} is different from all the trees in $B^{\text{leaf}_{Q1}}$. Thus, there are at least $\beta(t_0) + 1$ distinct trees that can map to $\alpha(t)$, including t_{02} and those in $B^{\text{leaf}_{Q1}}$. In other words,

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

Therefore, if $\beta(t_0)$ is infinite, $\beta(t)$ must also be infinite; otherwise, if $\beta(t_0)$ is finite, we have $\beta(t_0) < \beta(t)$. Hence, the monotonic condition holds for t by Definition 3.5. \square

Theorem 5.7. *PAC catamorphisms are monotonic.*

Proof. The theorem follows from Lemmas 5.5 and 5.6. \square

5.4 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 Section 3.3. 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 5.1 is PAC if \oplus is an associative and commutative operator over the collection domain \mathcal{C} . We can use SMT solvers [41, 40] or theorem provers [16] to check this property of operator \oplus .

Exponentially Small Upper Bound of the Number of Unrollings. Since PAC catamorphisms are monotonic (Theorem 5.7), they can be used in the decision procedure in Section 3.3. 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 Section 4.3; the only difference is that we use Lemma 5.4 to generalize the result for PAC catamorphisms instead of Lemma 4.1, 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 $\alpha_1, \dots, \alpha_m$ be m PAC catamorphisms, where the signature of the i -th catamorphism ($1 \leq i \leq m$) is

$$\text{sig}(\alpha_i) = \langle \mathcal{C}_i, \mathcal{E}, \oplus_i, \delta_i, c_{\text{leaf}i}, c_{\text{pr}i}, \text{pr}, \text{rec} \rangle$$

Catamorphism α with signature $\text{sig}(\alpha) = \langle \mathcal{C}, \mathcal{E}, \oplus, \delta, c_{\text{leaf}}, c_{\text{pr}}, \text{pr}, \text{rec} \rangle$ is a combination of $\alpha_1, \dots, \alpha_m$ 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 $\langle x_1, \dots, x_m \rangle, \langle y_1, \dots, y_m \rangle \in \mathcal{C}$:

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

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

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

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

$$c_{\text{leaf}} = \langle c_{\text{leaf}1}, c_{\text{leaf}2}, \dots, c_{\text{leaf}m} \rangle$$

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

$$c_{\text{pr}} = \langle c_{\text{pr}_1}, c_{\text{pr}_2}, \dots, c_{\text{pr}_m} \rangle$$

Theorem 5.8. *A combination of PAC catamorphisms is PAC.*

Proof. Let α be a combination of m PAC catamorphisms $\alpha_1, \dots, \alpha_m$. By construction, it is straightforward that α is written in the format of a PAC catamorphism in Definition 5.1. We prove α is really a PAC catamorphism by showing that \oplus is an associative and commutative operator. The structure of the proof is now the same as that in the proof of Theorem 4.4. \square

5.5 The Relationship between Catamorphisms

We have summarized two types of catamorphisms previously proposed by Suter et al. [2], including infinitely surjective and sufficiently surjective catamorphisms in Definitions 3.1 and 3.3, respectively. We have also proposed different classes of catamorphisms, including monotonic (Definition 3.5), AC (Definition 4.1), and PAC catamorphisms (Definition 5.1) that are used in our decision procedure. This section discusses how these classes of catamorphisms are related to each other and how they fit into the big picture. Their relationship is shown in Figure 5.3 with some catamorphism examples. Let us consider each pair of catamorphism classes in more detail as follows:

- *Between monotonic and sufficiently surjective catamorphisms:* Corollary 4 shows that all monotonic catamorphisms are sufficiently surjective. This shows that although the definition of monotonic catamorphisms from this dissertation and the idea of sufficiently surjective catamorphisms from Suter et al. [2] may look different from each other, they are actually closely related. We have also observed that even

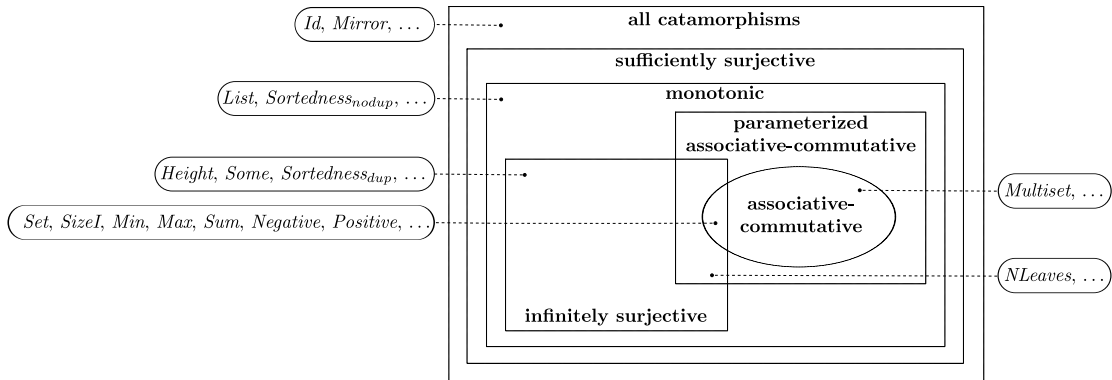


Figure 5.3: Relationship between different types of catamorphisms

though the set of sufficiently surjective catamorphisms is theoretically a super-set of that of monotonic catamorphisms (Corollary 4), in practice, however, we are not aware of any sufficiently surjective catamorphisms that are not monotonic (at the time of writing).

- *Between infinitely surjective and monotonic catamorphisms:* All infinitely surjective catamorphisms are monotonic, as proved in Lemma 3.3. Therefore, infinitely surjective catamorphisms are not just a sub-class of sufficiently surjective catamorphisms as presented in Suter et al. [2], they are also a sub-class of monotonic catamorphisms. As a result, we can use infinitely surjective catamorphisms with our decision procedure.
- *Between AC and PAC catamorphisms:* All AC catamorphisms are PAC, as shown in Theorem 5.1. The difference between them is that PAC catamorphisms have more parameters and hence more general, but also more complicated than AC catamorphisms.
- *Between PAC and monotonic catamorphisms:* All PAC catamorphisms are monotonic, as proved in Theorem 5.7. Consequently, both AC and PAC catamorphisms

are supported by our decision procedure.

- *Between infinitely surjective and AC/PAC catamorphisms:* The set of infinitely surjective catamorphisms and that of AC/PAC catamorphisms are intersecting, as shown in Figure 5.3 with some catamorphism examples.

Summary. We have presented parameterized associative-commutative (PAC) catamorphisms, a generalized version of associative-commutative (AC) catamorphisms. 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. Furthermore, we have demonstrated that PAC catamorphisms are more general, computationally optimal, and expressive than AC ones.

Chapter 6

Experimental Results

6.1 RADA: A Tool for Reasoning about Algebraic Data Types

In this chapter, we present RADA, a portable, scalable open source tool for reasoning about formulas containing algebraic data types using catamorphism (fold) functions. It can function as a back-end for reasoning about recursive programs that manipulate algebraic types. The algorithms behind RADA were described in previous chapters. RADA operates by successively unrolling catamorphisms and uses either CVC4 [41] and Z3 [40] as reasoning engines. We have used RADA for reasoning about functional implementations of complex data structures and to reason about *guard applications* that determine whether XML messages should be allowed to cross network security domains. Promising experimental results demonstrate that RADA can be used in several practical contexts.

6.1.1 Motivation

Tools have been created to reason about algebraic data types, such as the Leon verification system [55, 61] that works on top of Z3 and reasons over functions containing complex algebraic data structures written in Scala. However, these tools tend to be tightly integrated with the host language that they reason over: the Leon verification system is tightly integrated with Scala. For broader applicability, we would like to have a language-agnostic tool to perform this reasoning.

We introduce RADA¹, an open source tool to reason about algebraic data types with abstractions that is conformant with the SMT-Lib 2.0 format [50]. RADA was designed to be host-language and solver-independent and it can use either CVC4 or Z3 as its underlying SMT solver. RADA has also been tested on all major platforms. RADA has also been successfully integrated into the Guardol system [3], replacing our implementation of the Suter-Dotta-Kuncak decision procedure [2] on top of OpenSMT [42] in the Guardol system [3]. Experiments show that our tool is reliable, fast, and works seamlessly across multiple platforms, including Windows, Unix, and Mac OS.

6.1.2 Tool Architecture

Figure 6.1 shows the overall architecture of RADA, which follows closely the decision procedure described in Section 3.3. We use CVC4 [41] and Z3 [40] as the underlying SMT solvers in RADA because of their powerful abilities to reason about recursive data types. The grammar of RADA in Figure 6.2 is based on the SMT-Lib 2.0 [50] format with some new syntax for selectors, testers, data type declarations, and catamorphism declarations.

Note that although selectors, testers, and data type declarations are not defined in SMT-Lib 2.0, all of them are currently supported by both CVC4 and Z3; therefore, only

¹<http://crisis.cs.umn.edu/rada/>.

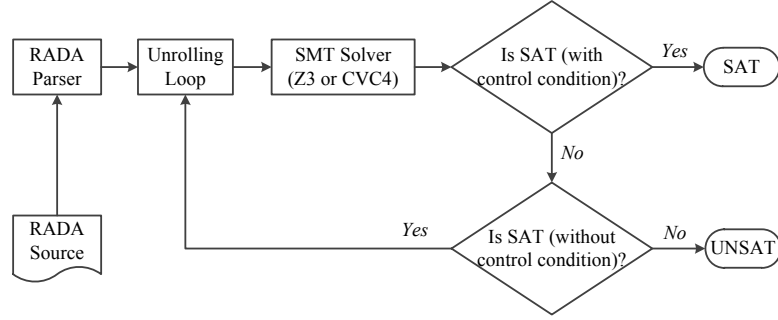


Figure 6.1: RADA architecture.

$\langle command \rangle_1$	$::=$	$(\text{declare-datatypes } () \langle datatype \rangle^+)$
$\langle datatype \rangle$	$::=$	$(\langle symbol \rangle \langle datatype_branch \rangle^+)$
$\langle datatype_branch \rangle$	$::=$	$(\langle symbol \rangle \langle datatype_branch_para \rangle^*)$
$\langle datatype_branch_para \rangle$	$::=$	$(\langle symbol \rangle \langle sort \rangle)$
$\langle command \rangle_2$	$::=$	$(\text{define-catamorphism } \langle catamorphism \rangle)$
$\langle catamorphism \rangle$	$::=$	$(\langle symbol \rangle (\langle sort \rangle) \langle sort \rangle \langle term \rangle$ $[\text{:post-cond } \langle term \rangle])$
$\langle selector_application \rangle$	$::=$	$\langle symbol \rangle \langle symbol \rangle$
$\langle tester_application \rangle$	$::=$	$\text{is-} \langle symbol \rangle \langle symbol \rangle$

Figure 6.2: RADA grammar.

catamorphism declarations are not understood by these solvers. **:post-cond**, which is used to declare R_α , is optional because we do not need to specify R_α when α is a surjective function (e.g., catamorphism `SumTree` discussed in Example 6.1).

Example 6.1 (RADA syntax). Let us consider an example to illustrate the syntax used in RADA. Suppose we have a data type `RealTree` that represents a binary tree of real numbers. Each node of the tree can be either a `Leaf` or a `Node(left : RealTree, elem : Real, right : RealTree)`. To abstract a `RealTree`, we could use a function `SumTree : RealTree → Real` that maps the tree into a number showing the sum of all the elements stored in the tree.

A `RealTree`, which can be a leaf or a root node with two subtrees and a number

stored in the node, can be written in RADA syntax as follows:

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

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

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

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

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

```
(assert (= (SumTree t1) 5.0))

(check-sat)
```

As expected, RADA returns `sat` for the above example. △

6.1.3 Implementation Improvement

RADA was first published at ESEC/FSE 2013 [76]. Since then, we have been working on improving the performance of the tool. Compared with the version published at ESEC/FSE 2013, the current version² has been dramatically improved in terms of speed. Table 6.1 shows the running times of the two versions of RADA on some complex Guardol benchmarks³. The running times were measured on a Ubuntu machine using an Intel Core I5 running at 2.8 GHz with 4GB RAM. For brevity, we denote the ESEC/FSE version of RADA by $RADA_{FSE}$ and the current version by $RADA_{current}$.

Table 6.1: The improvement in performance of RADA

Benchmark name	# obligations	Time (s)	
		$RADA_{FSE}$	$RADA_{current}$
Email_Guard_Correct_All.rada	17	0.527s	0.153s
RBTree.Black_Property.rada	12	141.024s	25.704s
RBTree.Red_Property.rada	12	7.968s	1.956s
array_checksum.SumListAdd_Alt.rada	13	0.325s	0.156s

As we can observe from Table 6.1, $RADA_{current}$ is multiple times faster than $RADA_{FSE}$. The main techniques among other things that we used to achieve this improvement are: (1) solve proof obligations in parallel, (2) reuse the definitions of catamorphism bodies when unrolling, and (3) solve each proof obligation incrementally.

²At the time of writing (February 2014).

³Full experimental results will be presented later in Section 6.2.

Solve proof obligations in parallel. We can write multiple proof obligations in RADA; each of them can be put in a push-pop pair as in SMT-Lib 2.0 [50]. For instance,

```
(push)
// Obligation A
(pop)
```

```
(push)
// Obligation B
(pop)
```

In $RADA_{FSE}$, proof obligations are handled sequentially. In other words, proof obligation B can only be considered after proof obligation A has been completely discharged. On the contrary, $RADA_{current}$ discharges proof obligations in parallel. It supports a thread pool of a configurable size of proof obligations. All the proof obligations in the pool are solved concurrently and all the remaining proof obligations are put in a waiting list. As soon as a proof obligation in the thread pool is discharged, the pool adds a new proof obligation from the waiting list to the pool (if any).

Reuse the definitions of catamorphism bodies when unrolling. In $RADA_{FSE}$, when we have a catamorphism application, e.g., `SumTree (Node(t2 5.0 t3))` with the `SumTree` catamorphism and tree terms `t2` and `t3` in Example 6.1, the catamorphism application is assigned to the corresponding definition of the catamorphism body with the given parameter. In this case, it will be as follows:

```
;; Method (1)
(assert (= (SumTree (Node(t2 5.0 t3)))
           (ite (is-Leaf (Node(t2 5.0 t3)))
```

```

0.0
(+ (SumTree (left (Node(t2 5.0 t3))))
  (elem (Node(t2 5.0 t3)))
  (SumTree (right (Node(t2 5.0 t3))))))

```

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

```

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

```

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

```

;; Method (2)
(assert (= (SumTree (Node(t2 5.0 t3)))
  (SumTree_GeneratedCatDefineFun ((Node(t2 5.0 t3))))))

```

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

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

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

```
;; Method (3)
(assert (SumTree_GeneratedUnrollDefineFun ((Node(t2 5.0 t3)))))
```

As we can observe, the performance of RADA is significantly improved by implementing method (3) as in $RADA_{current}$ instead of method (1) as in $RADA_{FSE}$. When we need to unroll a catamorphism application, we just need to call the corresponding user-defined function with suitable parameters instead of expanding tree terms repeatedly.

Solve each proof obligation incrementally. We observe that in our unrolling-based decision procedure, we need two calls to an SMT solver (i.e., two *decide* calls in Algorithm 3) at each unrolling step to determine whether we have found a trustworthy *SAT/UNSAT* answer or not. One of the reasons why $RADA_{FSE}$ is not very fast was because all the calls to the SMT solver were handled independently. That is, for each call to the SMT solver, $RADA_{FSE}$ initializes an instance of the SMT solver, checks the satisfiability of the current SMT query by the SMT solver instance, and then closes the instance of the SMT solver, forgetting all the information the SMT solver instance has collected. There are two issues with this approach: (1) $RADA_{FSE}$ does not take advantage of what the SMT solver instance has learned from the previous SMT query, and (2) $RADA_{FSE}$ pays a price in terms of performance for initializing and closing the SMT solver instance all the time.

In contrast, $RADA_{current}$ handles both issues, making the version much faster than

$RADA_{FSE}$. First, $RADA_{current}$ solves each proof obligation incrementally, i.e., the information collected from the SMT queries is reused over time. Second, there is only one instance of the SMT solver for each proof obligation we want to solve; in other words, $RADA_{current}$ creates an instance of the SMT solver when we start solving the proof obligation and only closes the SMT solver instance after the obligation has been completely discharged.

Example 6.2 (Incremental solving with RADA). To demonstrate how we solve each proof obligation incrementally in $RADA_{current}$, let us present step by step the way $RADA_{current}$ solves the `RealTree` example in Example 6.1. First, $RADA_{current}$ sends to an SMT solver the information about the `RealTree` data type:

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

Next, $RADA_{current}$ declares an uninterpreted function called `SumTree`, which represents the `SumTree` catamorphism in Example 6.1. Note that the SMT solver views `SumTree` as an uninterpreted function: the solver does not know what content of the function is; it only knows that `SumTree` takes as input a `RealTree` and returns a `Real` value as the output.

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

$RADA_{current}$ then feeds to the SMT solver the original problem we want to solve:

```

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

```

Additionally, $RADA_{current}$ creates two user-defined functions as previously discussed as a preprocessing step:

```

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

(define-fun SumTree_GeneratedUnrollDefineFun ((foo RealTree)) Bool (
  = (SumTree foo) (SumTree_GeneratedCatDefineFun foo))

```

$RADA_{current}$ then tries to check the satisfiability of the problem without unrolling any catamorphism application:

```

(check-sat)

```

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

are located, so that we can remove the control conditions later by a corresponding `pop` statement.

```

; Unrolling step
(assert (SumTree_GeneratedUnrollDefineFun t1))
(push)

; Assertions for control conditions
(assert (not (not (is-Leaf t1))))
(check-sat)

```

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

```

; Remove the control conditions
(pop)
(check-sat)

```

However, when checking the satisfiability without control conditions, we get `sat` from the SMT solver again. According to our unrolling decision procedure in Algorithm 3, we have to try another unrolling step; thus, $RADA_{current}$ sends the following to the SMT solver:

```

; Unrolling step
(assert (SumTree_GeneratedUnrollDefineFun (left t1)))
(assert (SumTree_GeneratedUnrollDefineFun (right t1)))
(push)

```

```

; Assertions for control conditions
(assert (not (not (is-Leaf (left t1)))))
(assert (not (not (is-Leaf (right t1)))))
(check-sat)

```

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

6.2 Experimental Results

We have implemented our decision procedure in RADA and evaluated the tool with a collection of benchmark guard examples listed in Tables 6.2, 6.3, and 6.4. The results are very promising: all of the benchmark examples were automatically verified by RADA in a short amount of time.

6.2.1 Experiments on AC catamorphisms

Table 6.2 contains simple manually created benchmarks involving AC catamorphisms. The first set consists of examples related to *Sum*, an AC catamorphism that computes the sum of all element values in a tree. The second set contains combinations of AC catamorphisms that are used to verify some interesting properties such as (1) there does not exist a tree with at least one element value that is both positive and negative and (2) the minimum value in a tree cannot be bigger than the maximum value in the tree. The definitions of the AC catamorphisms used in the benchmarks are as follows.

Table 6.2: Experimental results on benchmarks with AC catamorphisms

	Benchmark	Result	Time (s)
Single AC catamorphisms	sumtree01	sat	0.039
	sumtree02	sat	0.037
	sumtree03	sat	0.083
	sumtree04	unsat	0.034
	sumtree05	sat	0.041
	sumtree06	sat	0.038
	sumtree07	sat	0.031
	sumtree08	unsat	0.035
	sumtree09	unsat	0.033
	sumtree10	sat	0.032
	sumtree11	sat	0.043
	sumtree12	unsat	0.033
	sumtree13	sat	0.025
	sumtree14	unsat	0.044
Combination of AC catamorphisms	min_max01	unsat	0.057
	min_max02	unsat	0.738
	min_max_sum01	unsat	1.165
	min_max_sum02	sat	0.149
	min_max_sum03	sat	0.357
	min_max_sum04	sat	0.373
	min_size_sum01	unsat	0.873
	min_size_sum02	sat	0.114
	negative_positive01	unsat	0.038
	negative_positive02	unsat	0.136

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

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

- *Max* is defined in a similar way to *Min* in Table 3.1.
- *Negative* maps a tree to **true** if all the element values in the tree are negative and **false** otherwise. We assume that \mathcal{E} is a numeric type and *Leaf* is both positive and

negative.

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

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

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

6.2.2 Experiments on PAC catamorphisms

Table 6.3 consists of 12 benchmarks involving PAC catamorphisms; some of them represent important higher-order functions such as *forall*, *exists*, and *member*.

- Each of the first 10 benchmarks in Table 6.3 only involves one catamorphism. Catamorphisms *NLeaves*, *Forall* and *NGN* have been introduced in Section 5.2. Catamorphism *Exists* maps a tree into true if the tree contains at least one element value that satisfies a user-provided predicate pr_u while catamorphism *Member* maps a tree into true if the tree contains a user-provided value x .
- The last two examples consist of the combination of *NGN* and a slightly modified version of the catamorphism to demonstrate the combinability of PAC catamorphisms as discussed in Section 5.4.

Table 6.3: Experimental results on benchmarks with PAC catamorphisms

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

6.2.3 Experiments on Guardol benchmarks

In addition to AC and PAC catamorphisms, we have also experimented RADA with some examples in Table 6.4 containing general non-PAC parameterized catamorphisms automatically generated from the Guardol verification system [3]. They consist of verification conditions to prove some interesting properties of red black trees and the checksums of trees of arrays. These examples are complex: each of them contains multiple verification conditions, some data types, and a number of mutually related parameterized catamorphisms. For example, the Email Guard benchmark has 8 mutually recursive data types, 6 catamorphisms, and 17 complex obligations.

Table 6.4: Experimental results on complex guard benchmarks from Guardol [3]

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

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, its source code, and all the benchmarks in this dissertation are available at <http://crisys.cs.umn.edu/rada/>.

Chapter 7

Conclusion and Discussion

In this dissertation, we have proposed an unrolling-based decision procedure for algebraic data types with monotonic catamorphisms. Like sufficiently surjective catamorphisms, monotonic catamorphisms are fold functions that map algebraic data types into values in a decidable domain. We have shown that all sufficiently surjective catamorphisms known in the literature to date [2] are also monotonic. We have established an upper bound of the number of unrollings with monotonic catamorphisms. Furthermore, we have pointed out a sub-class of monotonic catamorphisms, namely associative-commutative (AC) catamorphisms, which are proved to be detectable, combinable, and guarantee an exponentially small maximum number of unrollings thanks to their close relationship with Catalan numbers. Our combination results extend the set of problems that can easily be reasoned about using the catamorphism-based approach.

In addition, we have proposed parameterized associative-commutative (PAC) catamorphisms, a generalized version of associative-commutative (AC) catamorphisms and 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. Furthermore, we

have shown that PAC catamorphisms are more general, computationally optimal, and expressive than AC ones.

We have also presented RADA, an open source tool to reason about inductive data types. RADA fully supports all types of catamorphisms discussed in this dissertation as well as other general user-defined abstraction functions. The tool was designed to be simple, efficient, portable, and easy to use. The successful uses of RADA in the Guardol project [3] demonstrate that RADA not only could serve as a good research prototype tool but also holds great promise for being used in other real world applications.

Future Work and Discussion

There are a number of areas in which the unrolling-based decision procedure and the RADA system can be improved or extended. Beyond the practical need to investigate and improve the scaling aspects of our algorithms, there are a few interesting research directions.

Capture catamorphism ranges. One of the challenges we would like to work on in the future is to ensure the completeness of the decision procedure by accurately capturing the ranges of monotonic catamorphisms. Without an accurate range, the decision procedure loses completeness. This is not a problem for surjective catamorphisms such as *Min* or *Sum*, to name a few, because the ranges of surjective catamorphisms are equal to their codomains. However, for a non-surjective catamorphism such as *Height*, we need to encode its range by a predicate R_α as discussed in Algorithm 3:

$$Height(t) \geq 0$$

because without the range constraint, the underlying SMT solvers can assign a negative value to $Height(t)$ when $Height(t)$ is treated as an uninterpreted function.

This issue can easily occur when we combine multiple AC catamorphisms because a catamorphism results from the combination of multiple other ones can have a very complicated range. For example, consider Min and Sum , two simple surjective AC catamorphisms. The range constraint of their combination is as follows:

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

which is not trivial to come up with. When we have more AC catamorphisms, it is even harder to compute the range. A practical way to handle this situation is to use a sound approximation of the range and refine it as needed using induction and an SMT solver, as previously presented in Section 3.3.

Improve the termination conditions. The termination argument of our decision procedure is based on $\mathfrak{D} = depth_{\phi_{in}}^{\max} + 1 + h_p$ (as in Theorem 3.13), a value that captures the maximum depth of unrollings needed for a conclusion of satisfiability to be drawn. It would not be difficult to naively over-approximate the maximum depth \mathfrak{D} based on the number of distinct tree terms in the formula; however, computing \mathfrak{D} correctly would require implementing an accurate tree unification/disunification as in Barrett et al. [53] because we need to compute the maximum depth of any tree term in the formula (for computing $depth_{\phi_{in}}^{\max}$) and the number of disequalities among tree terms (for computing h_p). This has not been supported in RADA.

Given an input formula and a maximum depth \mathfrak{D} , if the catamorphism in the formula is monotonic and we have an accurate range R_α of the catamorphism, our decision procedure can terminate after no more than \mathfrak{D} number of unrollings. The reason is that if we exceed \mathfrak{D} , we can immediately conclude that the input formula is satisfiable (Corollary 3). However, if we do not know if our catamorphism is monotonic or if R_α is correct or not, we can still process as follows. First, we check if the satisfying assignment after exceeding \mathfrak{D} is valid or not. If it is valid, we have a satisfiable formula; otherwise, either the catamorphism is not monotonic or R_α is incorrect. If R_α is incorrect, which is a property we can check using induction and an SMT solver, we can refine R_α . On the contrary, the catamorphism is not monotonic and we need to revise it.

The above observation suggests that there is a strong relationship between the maximum depth \mathfrak{D} , the monotonicity of the catamorphism, and the range of the catamorphism. In the future, we would like to investigate the relationship further to improve the termination conditions of our decision procedure.

Support strings in RADA. Our verification strategy has focused on reasoning about unbounded data, as this is the central difficulty in reasoning about guard applications. However, there are many directions in which this work can and should be extended. The most significant omission is the ability to reason about string operations. In the current Guardol tool suite we treat strings as uninterpreted types and string operations as uninterpreted functions and so cannot reason in a complete way about string-manipulating guards. For example, we would be unable to prove:

```
assert((string_concat "hello " "world") = "hello world") ;
```

Instead, the tools would produce a counterexample in which `string_concat` yields an arbitrary value. Fortunately, there are several decision and semi-decision procedures [77, 78] for strings that could be integrated into RADA to address this problem. In addition,

some SMT solvers such as CVC4 have recently started supporting string constraints; therefore, we believe we could integrate the ability to handle strings of SMT solvers into RADA in the near future.

Some problems not solvable by our decision procedure. We present here some typical examples of the problems not solvable by our work.

- *Problems not solvable by SMT solvers after abstraction:* The key idea of our decision procedure is that we use catamorphisms to abstract away algebraic data types to obtain a problem in decidable theories that SMT solvers can handle. If the obtained problem after the abstraction process cannot be solved by SMT solvers, the original problem also cannot be solvable by our decision procedure.
- *Problems involving non-monotonic catamorphisms.* For non-monotonic catamorphisms (e.g., the *Id* catamorphism that maps a tree into itself), the completeness of our decision procedure is not guaranteed. The reason is that when non-monotonic catamorphisms are unrolled, the number of tree candidates to satisfy all the disequalities among tree terms is not increased, making it impossible to satisfy all the constraints after a certain unrolling steps. On the other hand, theorem solvers such as ACL2 [16] are very efficient for those types of problems as they have a huge collection of heuristics for algebraic data types and for performing proofs by induction automatically. Dafny [7] with its strong support for user-defined lemmas also has the ability to work with more general recursive functions than ours.
- *Problems involving data types other than inductive tree-like data structures.* Our catamorphism-based decision procedure has been designed for inductive tree-like data structures because our catamorphisms require base and inductive cases. As a result, our work cannot handle more general data structures, such as those that

contain cycles. Those cyclic data structures, e.g., circular lists and doubly linked lists, can be reasoned about by other work such as the sound and terminating procedure by Chin et al. [79], which also allows user-defined recursive predicates to reduce the verification conditions to standard theories. Although their work does not guarantee completeness, the logic fragment that they can handle is expressive.

Although there are many avenues for future work and possible improvements in algorithms for reasoning about complex data, our contributions represent a solid step towards our goal of advancing the field of formal verification, broadening the set of problems that can be solved formally with completeness guarantees. Together with other increasingly efficient heuristic methods, we believe that formal verification can play a central role in verifying the correctness of industrial-scale software systems.

References

- [1] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2008.
- [2] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 199–210, 2010.
- [3] David Hardin, Konrad Slind, Michael Whalen, and Tuan-Hung Pham. The Guardol Language and Verification System. In *Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 18–32, 2012.
- [4] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
- [5] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
- [6] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969.

- [7] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
- [8] Lilian Burdy, Yoonsik Cheon, David R Cok, Michael D Ernst, Joseph R Kiniry, Gary T Leavens, K Rustan M Leino, and Erik Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
- [9] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.
- [10] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.
- [11] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.
- [12] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engg.*, 10(2):203–232, April 2003.
- [13] Nikolai Tillmann and Jonathan De Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

- [14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.
- [15] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 609–615, Berlin, Heidelberg, 2011. Springer-Verlag.
- [16] M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Springer, 2000.
- [17] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, Berlin, Heidelberg, 2007.
- [18] Dino Distefano and Matthew J. Parkinson J. jStar: Towards Practical Verification for Java. *SIGPLAN Not.*, 43(10):213–226, October 2008.
- [19] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.
- [20] Jean-Pierre Queille and Joseph Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.
- [21] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

- [22] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
- [23] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.
- [24] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
- [25] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification, CAV '97*, pages 72–83, London, UK, UK, 1997. Springer-Verlag.
- [26] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS '99*, pages 193–207, London, UK, UK, 1999. Springer-Verlag.
- [27] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *15th International Conference of Computer Aided Verification, CAV 2003*, pages 1–13, 2003.
- [28] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient Implementation of Property Directed Reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD '11*, pages 125–134, Austin, TX, 2011. FMCAD Inc.
- [29] Mary Sheeran, Satnam Singh, and Gunnar Stålmarmark. Checking Safety Properties Using Induction and a SAT-Solver. In *Proceedings of the Third International*

- Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.
- [30] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software Verification Using k-Induction. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 351–368, Berlin, Heidelberg, 2011. Springer-Verlag.
- [31] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of the 4th international conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.
- [32] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, August 2011.
- [33] K. Rustan Leino, Peter Müller, and Jan Smans. Foundations of Security Analysis and Design V. chapter Verification of Concurrent Programs with Chalice, pages 195–222. Springer-Verlag, Berlin, Heidelberg, 2009.
- [34] Shuvendu K. Lahiri and Shaz Qadeer. Verifying Properties of Well-Founded Linked Lists. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 115–126, New York, NY, USA, 2006. ACM.
- [35] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference*

- on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.
- [36] Jean Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 173–177, Berlin, Heidelberg, 2007. Springer-Verlag.
- [37] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993.
- [38] K. Slind and M. Norrish. A Brief Overview of HOL4. In *Proceedings of TPHOLs*, volume 5170 of *LNCS*, pages 28–32, 2008.
- [39] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.
- [40] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.
- [41] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, 2011.
- [42] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT Solver. In *Proceedings of the 16th international conference on Tools*

and Algorithms for the Construction and Analysis of Systems, TACAS'10, pages 150–153, 2010.

- [43] B. Dutertre and L. de Moura. The Yices SMT Solver. Tool paper at <http://yices.csl.sri.com/tool-paper.pdf>, August 2006.
- [44] Clark Barrett and Cesare Tinelli. CVC3. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.
- [45] Thomas Bouton, Diego Caminha B. De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, pages 151–156, Berlin, Heidelberg, 2009. Springer-Verlag.
- [46] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT Solver. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag.
- [47] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.
- [48] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,* TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

- [49] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification, 16th International Conference, CAV 2004*, pages 175–188, 2004.
- [50] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT*, 2010.
- [51] Clark Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In *Proceedings of the 17th international conference on Computer Aided Verification, CAV'05*, pages 20–23, Berlin, Heidelberg, 2005. Springer-Verlag.
- [52] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- [53] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types. *Electronic Notes in Theoretical Computer Science*, 174(8):23–37, 2007.
- [54] Derek C. Oppen. Reasoning About Recursively Defined Data Structures. *J. ACM*, 27(3):403–411, July 1980.
- [55] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability Modulo Recursive Programs. In *Proceedings of the 18th international conference on Static analysis, SAS'11*, pages 298–315, 2011.
- [56] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [57] Sudipta Kundu, Sorin Lerner, and Rajesh K. Gupta. *High-Level Verification: Methods and Tools for Verification of System-Level Designs*. Springer, 2011.

- [58] M. Myreen. *Formal Verification of Machine-Code Programs*. PhD thesis, University of Cambridge, 2009.
- [59] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
- [60] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction, CADE-11*, pages 748–752, London, UK, UK, 1992. Springer-Verlag.
- [61] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Proceedings of the 4th Workshop on Scala, SCALA '13*, pages 1:1–1:10, 2013.
- [62] G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.
- [63] Karen Zee, Viktor Kuncak, and Martin Rinard. Full Functional Verification of Linked Data Structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, PLDI '08*, pages 349–361, 2008.
- [64] Karen Zee, Viktor Kuncak, and Martin C. Rinard. An Integrated Proof Language for Imperative Programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI '09*, pages 338–351, 2009.

- [65] Viorica Sofronie-Stokkermans. Locality Results for Certain Extensions of Theories with Bridging Functions. In *Proceedings of the 22nd International Conference on Automated Deduction, CADE-22*, pages 67–83, 2009.
- [66] Swen Jacobs and Viktor Kuncak. Towards Complete Reasoning about Axiomatic Specifications. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation, VMCAI'11*, pages 278–293, 2011.
- [67] Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive Proofs for Inductive Tree Data-Structures. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '12*, pages 123–136, 2012.
- [68] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable Logics Combining Heap Structures and Data. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11*, pages 611–622, 2011.
- [69] Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. Towards a Scalable Software Model Checker for Higher-Order Programs. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation, PEPM '13*, pages 53–62, 2013.
- [70] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate Abstraction and CEGAR for Higher-Order Model Checking. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11*, pages 222–233, 2011.
- [71] Susanna S. Epp. *Discrete Mathematics with Applications*. Brooks/Cole Publishing Co., 4th edition, 2010.

- [72] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 7th edition, 2012.
- [73] Richard P. Stanley. *Enumerative Combinatorics, Volume 2*. Cambridge University Press, 2001.
- [74] Thomas Koshy. *Catalan Numbers with Applications*. Oxford University Press, 2009.
- [75] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.
- [76] Tuan-Hung Pham and Michael W. Whalen. RADA: A Tool for Reasoning about Algebraic Data Types with Abstractions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 611–614, 2013.
- [77] P. Hooimeijer and M. Veanes. An Evaluation of Automata Algorithms for String Analysis. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI'11*. Springer, January 2011.
- [78] A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijer, and M. Ernst. HAMPI: A Solver For String Constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09*, 2009.
- [79] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated Verification of Shape, Size and Bag Properties via User-Defined Predicates in Separation Logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

Appendix A

Glossary and Acronyms

Care has been taken in this thesis to minimize the use of jargon and acronyms, but this cannot always be achieved. This appendix defines jargon terms in a glossary, and contains a table of acronyms and their meaning.

A.1 Glossary

- **Satisfiability Modulo Theories (SMT)** – A decision problem involving a combination of theories expressed in first-order logic.

A.2 Acronyms

Table A.1: Acronyms

Acronym	Meaning
SMT	Satisfiability Modulo Theories
Continued on next page	

Table A.1 – continued from previous page

Acronym	Meaning
AC	Associative-Commutative
PAC	Parameterized Associative-Commutative
SAT	Satisfiable
UNSAT	Unsatisfiable