RADA: A Tool for Reasoning about Algebraic Data Types with Abstractions

Michael Whalen and Tuan-Hung Pham

University of Minnesota

Abstract. 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. RADA operates by successively unrolling catamorphisms and uses either CVC4 and Z3 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.

1 Introduction

Reasoning about algebraic data types has been an ongoing research topic since they are ubiquitous in functional programming. Applications include reasoning about data structure algorithms and *guard* (firewall) applications that allow/disallow XML messages to pass between networks (as is performed in the Guardol [6] system). To help address the challenge, powerful SMT solvers such as CVC4 [1] and Z3 [4] have also natively supported inductive data types written in SMT format, allowing end-users to experiment with interesting problems involving recursive data structures.

To reason about inductive data types, one of the prominent approaches is to abstract these data types into values in some decidable domains. The abstraction could be in the form of a catamorphism, as in decision procedures proposed by Suter et al. [9, 10], or could be in the form of recursively defined functions, as in the Dryad logic introduced by Madhusudan et al. [8]. Tools have been created to reason about these applications, such as the Leon verification system [10] 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.

In this paper, we introduce RADA, an open source tool written in SML/NJ to reason about algebraic data types with abstractions that is conformant with the SMT-Lib 2.0 format. 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 and has successfully been integrated into the Guardol system [6]. Experiments show that our tool is reliable, fast, and works seamlessly across multiple platforms, including Windows, Unix, and Mac OS.

The rest of this paper is organized as follows. Section 2 presents the algorithm behind RADA. Next, Section 3 describes its general architecture. Section 4 shows some experimental results we did with RADA. Finally, we conclude and outline some future work in Section 5.

2 Algorithm

RADA works based on a semi-decision procedure proposed by Suter et al. [10] to reason about recursive functions with abstractions. The input of the procedure is a formula ϕ in a parametric logic[1] that consists of literals over elements of tree terms and tree abstractions generated by a catamorphism. 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 $\mathcal{C}$. For example, suppose we have a data type $\mathsf{RealTree}$ that represents a binary tree of real numbers. Each node of the tree can be either a Leaf or a $\mathsf{Node(left : RealTree, elem : Real, right : RealTree)}$. To abstract a $\mathsf{RealTree}$, we could use a function $\mathsf{SumTree} : \mathsf{RealTree} \rightarrow \mathsf{Real}$ that maps the tree into a number showing the sum of all the elements stored in the tree. In this example, $\mathcal{E}$, $\mathcal{C}$, and α are Real, Real, and $\mathsf{SumTree}$, respectively.

The decision procedure works on top of an SMT solver [2] $\mathcal{S}$ 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. Therefore, the main idea of the decision procedure is to approximate the behavior of the catamorphism by repeatedly unrolling it a certain number of times and treating the calls to the not-yet-unrolled catamorphism instances at the lowest levels as calls to uninterpreted functions. However, a uninterpreted function can return any values in its co-domain; hence, the presence of these uninterpreted functions can make the $\mathsf{sat}/\mathsf{unsat}$ result not trustworthy. To address this issue, each time the catamorphism is unrolled, a boolean control condition B is created to determine if the uninterpreted functions at the bottom level are necessary to the determination of satisfiability. That is, if B is true, the list of uninterpreted functions does not play any role in the satisfiability result.

The main steps of the procedure are shown in Algorithm 1. The input of the algorithm is a formula ϕ written in the parametric logic and a program Π, which contains ϕ and the definitions of data type τ and catamorphism α. The goal of the algorithm is to determine 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 corresponding control condition

[1] See [9] for a full description of the syntax and sematics of the parametric logic.
[2] Suter et al. [10] specifically used Z3 [4] as the underlying SMT solver.

B_i of ϕ_i, function $unrollStep(\phi_i, \Pi, B_i)$ unrolls the catamorphim 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 control condition B_{i+1} for ϕ_{i+1}. Function $decide(\varphi)$ simply calls $\mathcal{S}$ to check the satisfiability of φ and returns $SAT/UNSAT$ accordingly.

Algorithm 1: Catamorphism unrolling algorithm [10]

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

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 the control condition is 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 control condition, or both of them together. As a result, to understand the $UNSAT$ more deeply, we could try to check the satisfiability of ϕ alone, as depicted at line 7. Note that checking ϕ alone also means that the control condition is not used; consequently, the values of uninterpreted functions may contribute to the $SAT/UNSAT$ result of $decide(\phi)$. If $decide(\phi) = UNSAT$ as at line 8, we can conclude that the problem is unsatisfiable because assigning the uninterpreted functions to any values in their co-domains still cannot make the problem satisfiable as a whole. Finally, we need to consider the case $decide(\phi) = SAT$ as at line 10. Since we already know that $decide(\phi \wedge \bigwedge_{b \in B} b) = UNSAT$, the only way to make $decide(\phi)$ 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.

3 Tool Architecture

Fig. 1 shows the overall architecture of RADA, which follows closely the algorithm described in Section 2. We use CVC4 [1] and Z3 [4] as the underlying SMT solvers in RADA because of their powerful abilities to reason about recursive data types. The grammar of RADA in Fig. 2 is based on the SMT-Lib 2.0 [2] format with some new syntax for selectors, testers, data type declarations, and catamorphism declarations.

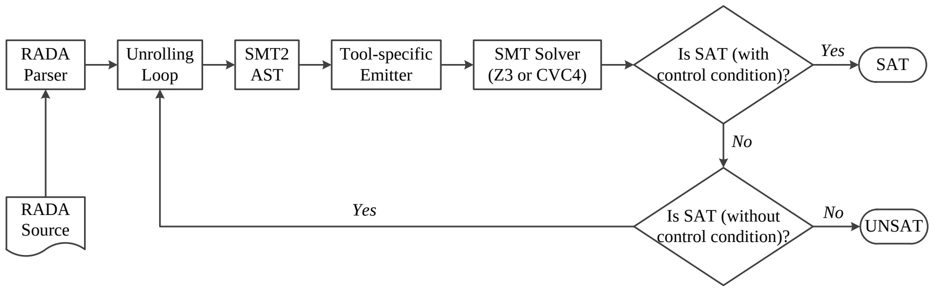

Fig. 1. RADA architecture.

⟨*command*⟩$_1$	::=	(**declare-datatypes** ⟨*datatype*⟩$^+$)
⟨*datatype*⟩	::=	(⟨*symbol*⟩ ⟨*datatype_branch*⟩$^+$)
⟨*datatype_branch*⟩	::=	(⟨*symbol*⟩ [⟨*datatype_branch_parameter*⟩$^+$])
⟨*datatype_branch_parameter*⟩	::=	(⟨*symbol*⟩ ⟨*sort*⟩)
⟨*command*⟩$_2$	::=	(**define-catamorphism** ⟨*catamorphism*⟩)
⟨*catamorphism*⟩	::=	(⟨*symbol*⟩ (⟨*sort*⟩) ⟨*sort*⟩ ⟨*term*⟩)
⟨*selector_application*⟩	::=	⟨*symbol*⟩ ⟨*symbol*⟩
⟨*tester_application*⟩	::=	*is_*⟨*symbol*⟩ ⟨*symbol*⟩]

Fig. 2. RADA grammar.

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 catamorphism declarations are not understood by these solvers. As a result, to bridge the gap between the input format of RADA and that of CVC4/Z3, each time the catamorphism is unrolled, we build an abstract syntax tree in which the catamorphism declaration is replaced by a uninterpreted function representing the behaviors of the unrolled parts of the catamorphism. Based on the abstract syntax tree, we could generate an .smt2 file that CVC4 or Z3 accepts with the help of a tool-specific emitter, which is responsible for creating a suitable .smt2 file for the solver being used.

To illustrate the grammar used in RADA, let us further examine the RealTree example briefly mentioned in Section 2. A RealTree, which could be a leaf or a root node with two subtrees and a number stored in the node, could be written in RADA syntax as follows:

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

Next, a RealTree could be abstracted into a real number representing the sum of all elements in the tree by catamorphism SumTree, which could be 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 l1 () RealTree)
(declare-fun l2 () RealTree)
(declare-fun l3 () RealTree)
(assert (= l1 (Node l2 5.0 l3)))
(assert (= (SumTree l1) 5.0))
(check-sat)
```

As expected, RADA returns sat for the above simple example.

4 Experimental Results

RADA has been successfully integrated into the Guardol system [6], replacing our implementation of the Suter-Dotta-Kuncak decision procedure [9] on top of OpenSMT [3] in Guardol. We have experimented RADA with a collection of approximately 20 benchmark guard examples. The results are very promising: all of them were automatically verified in a very short amount of time. While the majority of our benchmarks was manually designed, the most challenging one was automatically generated from Guardol. It has 8 mutually recursive data types, contains 17 complex verification conditions, and requires 65 satisfiability checking calls to SMT solvers. It took less than 3 seconds for RADA to prove the unsatisfiability of all verification conditions in this benchmark.

During our experiments on the performance of RADA, more than 100 intermediate .smt2 files involving inductive data types were also created. Not surprisingly, CVC4 and Z3 return the same sat/unsat result for each intermediate .smt2 file. Since the ability to reason about inductive data types has just recently been implemented in these SMT solvers, we believe that our collection of .smt2 files could also be served as ones of the valuable regression examples of inductive data types, especially for CVC4 and Z3[3].

RADA is written in SML/NJ, designed to be solver-independent, highly portable, and can be compiled on all major platforms. All benchmarks were run on a machine using an Intel Core I3 running at 2.13 GHz. RADA, its benchmarks, the collection of intermediate `.smt2` files, and all experimental results are available at `http://crisys.cs.umn.edu/rada`.

5 Conclusion

We have presented RADA, an open source tool to reason about inductive data types with catamorphisms. RADA was designed to be simple, efficient, portable, and easy to use. The successful uses of RADA in the Guardol project [6] make us believe that RADA not only could serve as a good research prototype tool but also holds a great promise to be used in other real world applications.

With the help of RADA, we have been able to reason about unbounded data in Guardol. However, verifying string operations in Guardol still remains a challenge and they are currently treated as uninterpreted functions in our system. Therefore, in the future, we would like to extend RADA to support a string decision procedure [7, 5, 11] in our analysis tool.

References

1. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, 2011.
2. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT*, 2010.
3. Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT Solver. In *TACAS*, 2010.
4. Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
5. Vijay Ganesh, Adam Kieżun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael Ernst. HAMPI: A Solver For String Constraints. In *ISSTA*, 2009.
6. David Hardin, Konrad Slind, Michael Whalen, and Tuan-Hung Pham. The Guardol Language and Verification System. In *TACAS*, 2012.
7. Pieter Hooimeijer and Margus Veanes. An Evaluation of Automata Algorithms for String Analysis. In *VMCAI*, 2011.
8. Parthasarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive Proofs for Inductive Tree Data-Structures. In *POPL*, 2012.
9. Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *POPL*, 2010.
10. Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability Modulo Recursive Programs. In *SAS*, 2011.
11. Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic String Verification: Combining String Analysis and Size Analysis. In *TACAS*, 2009.

[3] We have given our collection of `.smt2` files to assist the Z3 and CVC4 teams.