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

Tuan-Hung Pham
University of Minnesota
Minneapolis, MN 55455, USA
hung@cs.umn.edu

Michael W. Whalen
University of Minnesota
Minneapolis, MN 55455, USA
whalen@cs.umn.edu

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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification

Keywords

Decision procedures, satisfiability, unrolling

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 [5] 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.

ESEC/FSE '13, August 18–26, 2013, Saint Petersburg, Russia

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 some decision procedures for algebraic data types [13, 15, 16], or could be in the form of recursively defined functions, as in the DRYAD logic introduced by Madhusudan et al. [10]. Tools have been created to reason about these applications, such as the Leon verification system [16] 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[1], an open source tool to reason about algebraic data types with abstractions that is conformant with the SMT-Lib 2.0 format [2]. 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 [5]. 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. Section 5 presents related work. Finally, we conclude and outline some future work in Section 6.

2. ALGORITHM

RADA works based on our unrolling-based decision procedure to reason about recursive functions with abstractions [13]. The input of the procedure is a formula ϕ in a logic[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 $\alpha : \tau \to \mathcal{C}$ 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 RealTree that represents a binary tree of real numbers. Each node of

[1] `http://crisys.cs.umn.edu/rada/`.
[2] See [15] for a full description of the syntax and semantics of the logic.

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. In this example, $\mathcal{E}$, $\mathcal{C}$, and α are Real, Real, and SumTree, respectively.

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 logic that is not inherently supported by $\mathcal{S}$ is the application 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, an uninterpreted function can return any values in its codomain; hence, 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 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 catamorphim; 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 main steps of the procedure are shown in Algorithm 1. The input of the algorithm is a formula ϕ written in the 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 set of control conditions 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 set of control conditions 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: Unrolling-based decision procedure in [13]

```
1  (φ, B) ← unrollStep(φ, Π, ∅)
2  while true do
3      switch decide(φ ∧ ⋀_{b∈B} b) do
4          case SAT
5              return "SAT"
6          case UNSAT
7              switch decide(φ ∧ R_α) do
8                  case UNSAT
9                      return "UNSAT"
10                 case SAT
11                     (φ, B) ← unrollStep(φ, Π, 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 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 the uninterpreted functions to any values in their ranges 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.

Completeness. The decision procedure in Algorithm 1 has been proven to be sound for all catamorphisms and complete for *monotonic* catamorphisms [13]. With monotonic catamorphisms, completeness is guaranteed if there is only one type of catamorphism (i.e., α) in the input formula. We showed in [13] that for *associative-commutative* catamorphisms, a popular subclass of monotonic catamorphisms, we can handle multiple catamorphisms while preserving the completeness of the decision procedure. Also, associative-commutative catamorphisms can be automatically detectable and they allow Algorithm 1 to terminate after a considerably small number of unrollings. Furthermore, we recently identified *parameterized associative-commutative* catamorphisms [14] that have all the features of associative-commutative catamorphisms and that allow several additional parameters to be defined in a catamorphism, making our catamorphism approach more general and efficient.

3. TOOL ARCHITECTURE

Figure 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 Figure 2 is based on the SMT-Lib 2.0 [2] 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 catamorphism declarations are not understood by these solvers. **:post-cond**, which is used to declare R_α, is optional be-

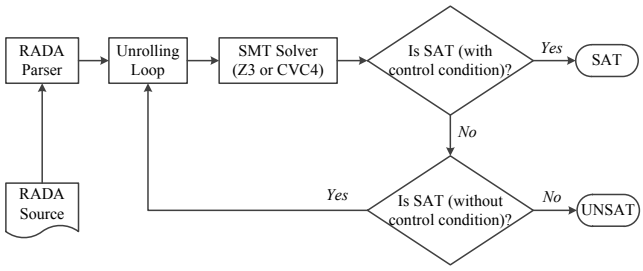

Figure 1: RADA architecture.

$\langle command\rangle_1$ ::= (**declare-datatypes** () $\langle datatype\rangle^+$)
$\langle datatype\rangle$ ::= ($\langle symbol\rangle$ $\langle datatype\_branch\rangle^+$)
$\langle datatype\_branch\rangle$ ::= ($\langle symbol\rangle$ $\langle datatype\_branch\_para\rangle^*$)
$\langle datatype\_branch\_para\rangle$ ::= ($\langle symbol\rangle$ $\langle sort\rangle$)

$\langle command\rangle_2$ ::= (**define-catamorphism** $\langle catamorphism\rangle$)
$\langle catamorphism\rangle$ ::= ($\langle symbol\rangle$ ($\langle sort\rangle$) $\langle sort\rangle$ $\langle term\rangle$ [:**post-cond** $\langle term\rangle$])

$\langle selector\_application\rangle$ ::= $\langle symbol\rangle$ $\langle symbol\rangle$
$\langle tester\_application\rangle$ ::= **is-**$\langle symbol\rangle$ $\langle symbol\rangle$

Figure 2: RADA grammar.

cause we do not need to specify R_α when α is a surjective function (e.g., function SumTree in Section 2).

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 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 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 example.

4. EXPERIMENTAL RESULTS

RADA has been successfully integrated into the Guardol system [5], replacing our implementation of the Suter-Dotta-Kuncak decision procedure [15] on top of OpenSMT [3] in Guardol. We have experimented RADA with a collection of 42 benchmark guard examples listed in Table 1. The results are very promising: all of them were automatically verified in a very short amount of time.

Table 1: Experimental results

#	Benchmark	Result	Time (s)
	Manually created benchmarks		
1	sumtree01	sat	0.098
2	sumtree02	sat	0.079
3	sumtree03	sat	0.207
4	sumtree04	unsat	0.038
5	sumtree05	sat	0.096
6	sumtree06	sat	0.097
7	sumtree07	sat	0.045
8	sumtree08	unsat	0.041
9	sumtree09	unsat	0.042
10	sumtree10	sat	0.042
11	sumtree11	sat	0.083
12	sumtree12	unsat	0.041
13	sumtree13	sat	0.036
14	sumtree14	unsat	0.114
15	mut_rec1	sat	0.077
16	mut_rec3	unsat	0.059
17	mut_rec4	unsat	0.091
18	min_max01*	unsat	0.078
19	min_max02*	unsat	0.459
20	min_max_sum01*	unsat	1.688
21	min_max_sum02*	sat	0.224
22	min_max_sum03*	sat	0.601
23	min_max_sum04*	sat	0.521
24	min_size_sum01*	unsat	1.314
25	min_size_sum02*	sat	0.262
26	negative_positive01*	unsat	0.065
27	negative_positive02*	unsat	0.421
	Manually created benchmarks containing parameterized associative-commutative catamorphisms [14]		
28	forall01	sat	0.633
29	forall02	unsat	0.455
30	exists01	sat	0.074
31	exists02	unsat	0.082
32	member01	sat	0.331
33	member02	unsat	0.458
34	ngn01	sat	0.602
35	ngn02	unsat	0.251
36	ngn_ngn01*	sat	0.801
37	ngn_ngn02*	unsat	0.315
	Complex guard benchmarks from Guardol [5]		
38	Email_Guard_Correct_All*	17 unsats	≈0.031/obligation
39	RBTree.Black_Property*	12 unsats	≈11.752/obligation
40	RBTree.Red_Property*	12 unsats	≈0.664/obligation
41	array_checksum.SumListAdd*	2 unsats	≈0.064/obligation
42	array_checksum.SumListAdd_Alt*	13 unsats	≈0.025/obligation

The set of benchmarks is divided into three parts. The first part contains simple manually created benchmarks involving normal catamorphisms. Some benchmarks in this part were used to verify interesting properties such as (1) there does not exist a tree that is both positive (i.e., all of its nodes are positive) and negative (i.e., all of its nodes are negative) and (2) the minimum value in a tree can not be bigger than the maximum value in the tree. The second group consists of ten manually created benchmarks involving parameterized associative-commutative catamorphisms [14]; some of them represent important higher-order functions such as *forall*, *exists*, and *member*. All benchmarks in the last part were automatically generated from Guardol [5] and are highly complicated; for example, the Email Guard benchmark has 8 mutually recursive data types, 6 catamorphisms, and 17 complex obligations. Benchmarks with * in

Table 1 contain multiple catamorphisms.

RADA was designed to be solver-independent, portable, and compilable on all major platforms. 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, all the benchmarks in this paper, and a screencast to illustrate the tool are available at `http://crisys.cs.umn.edu/rada/`.

5. RELATED WORK

There are some tools that support catamorphisms (as well as other functions) over algebraic data types. For example, Isabelle [11], PVS [12], and ACL2 [7] 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 [2]; 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 support for data types. One of such tools is Dafny [9], 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, which uses an unrolling-based semi-decision procedure to reason about catamorphisms [16]. 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 [4] 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 is open source while Leon is not (at the time of writing). More importantly, RADA guarantees the completeness of the results even when the input formulas have multiple catamorphisms with or without parameters for certain classes of catamorphisms [13, 14]; in this situation, it is unknown whether the decision procedure [16] used in Leon can ensure the completeness or not because the authors [16] 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.

6. 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 [5] 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.

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 [6, 8] in our tool.

7. ACKNOWLEDGEMENTS

We thank David Hardin, Konrad Slind, and Andrew Gacek for their feedback on early drafts of this paper. This work was sponsored in part by NSF grant CNS-1035715 and by a subcontract from Rockwell Collins.

8. REFERENCES

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

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

[3] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. TACAS, 2010.

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

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

[6] P. Hooimeijer and M. Veanes. An Evaluation of Automata Algorithms for String Analysis. In *VMCAI*, 2011.

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

[8] A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijer, and M. Ernst. HAMPI: A Solver For String Constraints. In *ISSTA*, 2009.

[9] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *LPAR*, 2010.

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

[11] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[12] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *CADE*, 1992.

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

[14] T.-H. Pham and M. W. Whalen. On Parameterized Abstractions in Unrolling-Based Decision Procedure for Algebraic Data Types. Technical Report 13-018, Department of Computer Science and Engineering, University of Minnesota, 2013.

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

[16] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability Modulo Recursive Programs. In *SAS*, 2011.