A Coq-based synthesis of Scala programs which are correct-by-construction

Youssef El Bakouny
CIMTI - ESIB
Saint-Joseph University
Beirut, Lebanon
CEDRIC
CNAM
Paris, France
Youssef.Bakouny@usj.edu.lb

Tristan Crolard
CEDRIC
CNAM
Paris, France
Tristan.Crolard@cnam.fr

Dani Mezher
CIMTI - ESIB
Saint-Joseph University
Beirut, Lebanon
Dany.Mezher@usj.edu.lb

Abstract

The present paper introduces Scala-of-Coq, a new compiler that allows a Coq-based synthesis of Scala programs which are "correct-by-construction". A typical workflow features a user implementing a Coq functional program, proving this program's correctness with regards to its specification and making use of Scala-of-Coq to synthesize a Scala program that can seamlessly be integrated into an existing industrial Scala or Java application.

Keywords Formal methods, Functional programming, Coq, Scala, Compilation

ACM Reference format:
Youssef El Bakouny, Tristan Crolard, and Dani Mezher. 2017. A Coq-based synthesis of Scala programs which are correct-by-construction. In *Proceedings of FTFJP'17, Barcelona , Spain, June 18-23, 2017,* 2 pages.
DOI: 10.1145/3103111.3104041

1 Introduction

In our modern world, software bugs are becoming increasingly detrimental to the engineering industry. Since software components are interconnected, a bug in one component is likely to affect others, causing a system-wide failure. Additionally, software bugs have often created system vulnerabilities that are exploitable by malicious users [10].

These bugs are essentially due to a lack of rigor in the software engineering field where the correctness of a program is thoroughly tested but rarely proven. In fact, current industrial software development practices rely solely on the expensive and time-consuming development of tests in an attempt to hunt down bugs. For non-trivial cases, this approach can never guarantee that a software is correct with regards to its specification.

As a result, we have recently witnessed interesting initiatives aimed at developing new science, technologies and tools capable of responding to the aforementioned software engineering challenges. A remarkable example of such an initiative is a U.S. National Science Foundation (NSF) expedition in computing project called "the Science of Deep Specification (Deep Spec)" [10].

These initiatives make use of "formal methods", potentially as a complement to software testing, with the goal of providing the means of developing software that is correct-by-construction. These formal methods can be defined as a set of techniques and mathematical theories allowing both the specification of a software system in an unambiguous manner, and the rigorous verification of the conformity of this software system with regards to its specification. This verification can be done by proving that the program satisfies a set of lemmas or theorems constituting its specification.

2 Proof assistants

Since the manual checking of realistic program proofs is impractical or, to say the least, time-consuming; several proof assistants have been developed to provide machine-checked proofs. Isabelle/HOL [8] and Coq [6] are currently the world's two leading proof assistants.

Both of these proof assistants allow the interactive construction of formal proofs; Coq is based on a logical framework known as the calculus of inductive constructions [11], while Isabelle/HOL is based on high order classical logic. Coq supports code generation in Objective Caml, Haskell and Scheme [6], while Isabelle/HOL supports code generation in Objective Caml, Haskell, SML and *Scala* [3].

The code generation capabilities of these proof assistants enable the synthesis of programs which are correct-by-construction. However, they do not address the verification needs of existing software programs. This need is covered by other initiatives such as Leon [1], Why3 [2] and, also, Sireum Logika[1]. Leon and Logika allow the automatic verification of *Scala* programs. Leon can also resort to Isabelle/HOL machine-checked proofs when its automatic verification mechanism fails to give an answer. Similarly, Why3 allows the automatic verification of Objective Caml programs with the option of resorting to Coq on failure.

Coq has been successfully used to implement CompCert [5], the world's first and only formally verified C compiler. It is also widely used in several research projects including the aforementioned DeepSpec project, hence our interest in this particular proof assistant.

3 The Scala Programming Language

The Scala programming language [9] was developed by Martin Odersky and his "École Polytechnique Fédérale de Lausanne" (EPFL) team in 2003. It is currently of high interest to several industrial projects thanks to its practical fusion of functional and object-oriented programming; as well as its seamless interoperability with Java.

[1] http://logika.sireum.org

FTFJP'17, Barcelona , Spain

978-1-4503-5098-3/17/06…$15.00
DOI: 10.1145/3103111.3104041

Scala is the implementation language of many important frameworks, including Apache Spark, Kafka, and Akka. It also provides the core infrastructure for sites such as Twitter and Coursera.

Given the importance of Scala in the industrial world, the Coq proof assistant would highly benefit of a Scala code generation feature. For this purpose, the "Scala-of-Coq" compiler is currently being implemented.

4 Compiling Coq to Scala

The main objective of the Scala-of-Coq compiler is to allow the synthesis of human readable Scala programs that are correct-by-construction. The synthesized Scala programs are purely functional and conform to the PureScala synthax, as defined in [1].

The compiler itself is also written in Scala and its underlying Coq parser is largely based on the use of parser combinators [4], easing the construction of complex parsers out of primitive ones. The Scala library that supports these parser combinators [7] facilitated the implementation of the Coq parser without the need to resort to parser generators such as Lex/Yacc or ANTLR. This is allowing us to create and manipulate Coq's Abstract Syntax Tree (AST) directly in Scala while taking full advantage of the language's fusion of object-oriented and functional paradigms.

Using Scala-of-Coq, the Coq program in listing 1, which computes the number of nodes in a binary tree, was successfully converted to the Scala program in listing 2.

Listing 1. Source Coq Program

```
Inductive Tree : Type :=
  Leaf : Tree
| Node(l r : Tree): Tree.

Fixpoint size (t: Tree) : nat :=
match t with
  Leaf ⇒ 1
| Node l r ⇒ 1 + (size l) + (size r)
end.
```

Listing 2. Generated Scala Program

```
object CertifiedTree {
  sealed abstract class Tree
  case object Leaf extends Tree
  case class Node(l: Tree, r: Tree) extends Tree
  def size(t: Tree): Nat =
    t match {
      case Leaf => 1
      case Node(l, r) => 1 + (size(l) + size(r))
    }
}
```

It is important to note that the source Coq program can be formally verified by proving the lemmas that define its specification. For example, the lemmas in listing 3 were successfully proven using the definitions provided in listing 1. And since these Coq definitions use "nat" from the "Coq.Init.Nat" library, the generated Scala definitions also use "Nat"; an equivalent Scala implementation.

Once the source Coq program is formally proven with regards to its specification (portrayed as Lemmas), the generated Scala program can also be considered formally verified with regards to the same specification; assuming the correctness of the Scala-of-Coq compiler, the Coq proof assistant, the Scala compiler and standard library along with the Java Virtual Machine (JVM) as well as the underlying operating system and hardware infrastructure.

```
Lemma size_left: ∀ l r : Tree, size (Node l r) > size l.
Lemma size_right: ∀ l r : Tree, size (Node l r) > size r.
```

Listing 3. Coq Lemmas

Acknowledgments

The authors would like to thank the National Council for Scientific Research in Lebanon (CNRS-L)[2] for their funding, as well as Murex S.A.S[3] for providing financial support.

5 Conclusion

In conclusion, the Scala-of-Coq compiler enables users of the Coq proof assistant to generate Scala programs that are formally verified. An example of a practical use case of Scala-of-Coq depicts a Coq user that generates a verified Scala component and seamlessly integrates it into an existing industrial Scala or Java application.

Future developments will focus on the verification of existing Scala programs using Coq. This verification should also include the support of imperative side effects and a restricted subset of parallel programs.

Finally, a framework for the support of additional languages, such as Swift, should be implemented.

References

[1] 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, New York, NY, USA, 2013. ACM.

[2] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[3] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, pages 103–117, Berlin, Heidelberg, 2010. Springer.

[4] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *J. Funct. Program.*, 8(4):437–444, 1998.

[5] Xavier Leroy. Mechanized semantics for compiler verification. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 4–6. Springer, 2012.

[6] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[7] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical report, K.U.Leuven, 2008.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[9] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014.

[10] Benjamin C. Pierce. The science of deep specification (keynote). In Eelco Visser, editor, *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, page 1. ACM, 2016.

[11] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations.* Electronic textbook, 2016.

[2] http://www.cnrs.edu.lb/

[3] https://www.murex.com/