# Towards the Generation of Correct Java Programs

Jolan Philippe
School of Informatics Computing and Cyber Systems
Northern Arizona University
Flagstaff, USA
jp2589@nau.edu

Frédéric Loulergue
School of Informatics Computing and Cyber Systems
Northern Arizona University, USA
Flagstaff, USA
frederic.loulergue@nau.edu

## RESEARCH POSTER EXTENDED ABSTRACT

### I. Introduction

Proof assistants such as Coq [1] can be used to develop very high assurance software such as a verified C compiler [2] and verified high performance computing programs [3]. Even when not used for a full system, using such a proof assistant to develop critical parts of a system could be interesting, for example security monitors [4] or the reconfiguration mechanism of a component based system.

Many systems are developed using Java, but it is currently not possible to obtain automatically Java code from Coq to be embedded in a larger Java development. There are some instances of Coq being used in the context of a Java system [5], [6]: either the Java code is written by hand from the Coq formalization, or there is an additional transformation step from one of the language that Coq supports, to Java bytecode. In the latter case, the style of the generated code is fixed.

Our goal is to design a plugin for Coq to be able to generate directly Java code from a Coq development in such a way that the generated code can be customized to the programming style (static, object) used in the Java development. We plan to use such code for security monitors, component based systems, high performance computing, and big data applications. We present what our system can currently generate and discuss further extensions, in particular keeping in mind that eventually we want to formally prove that the generated Java code preserves the semantics of the initial Coq code.

### II. An Overview of Coq

The Coq proof assistant can be considered as a functional programming language, but with a very rich type system that allows to express mathematical properties. By the Curry-Howard correspondence, the proofs of theorems are programs. However it is not convenient to write proof as programs and Coq offers a specific language of *tactics* to write scripts that constructs the proof term for the user.

To give a taste of how Coq code is written, Figure 1 is a small example. This example defines the type `nat` of natural numbers, defined as axiomatized by Peano. `O` is a natural number (representing the usual number 0), and if `n` is a natural number then `S` applied to `n`, written `S n`, is also a natural number. If `n` represents the usual number $n$, `S n` represents $n + 1$. `O` and `S` are called *constructors*. `add` is a recursive function defined by pattern matching on `n1`: each possible ways of constructing `n1` are considered. Finally `add_n_0` is a lemma about the operation `add`. The text between **Proof** and **Qed** is a proof script.

```
Inductive nat : Set :=
 | O : nat
 | S : nat → nat.
Fixpoint add (n1 n2 : nat) : nat :=
 match n1 with
 | O ⇒ n2
 | S n ⇒ S(add n n2)
 end.
Lemma add_n_0 (n:nat): add n O = n.
Proof. induction n; auto. simpl. now rewrite IHn. Qed.
```

Fig. 1. Short Coq Example

Clearly `nat` and `add` are definitions that have a computational content. Through its extraction mechanism [7], Coq can produce Haskell, Scheme, or OCaml code that can then be compiled and executed. During extraction, all "logical" parts of the Coq code such as `add_n_0` are removed. It is not also so simple because computational and logical parts can be intertwined. Writing types and functions that can be extracted, and proving properties about them is one possible way of using Coq [2], [3].

### III. From mini-ML to Java

The extraction of Coq [7] to other programming language requires an intermediate step where the code is transformed into Mini-ML [8], a core functional programming language. This language is syntactically more simple and it can be used as a basis for our extraction to Java code. Java is a very popular language, and there exists a lot of programming styles. Indeed, even if Java is object oriented, it is not the only way to write programs (e.g. *static* attributes). The approach used on our transformation is based on an object oriented programming style. For an inductive type $T$ defined in Mini-ML (`nat` is such a type), we define an interface Java with the same name, and for each constructor of this type, an implementing class containing a constructor and accessors.

978-1-5386-7879-4/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCS.2018.00166

```
public interface Nat {
  public Nat add(n2:Nat);
}

public class O implements Nat{
  public O (){}
  public Nat add(n2:Nat){
    return n2;
  }
}

public class S implements Nat{
  private Nat nat0;
  public S (Nat nat0){
    this.nat0 = nat0;
  }
  public getNat0 (){
    return this.nat0;
  }
  public setNat0 (Nat nat0){
    this.nat0 = nat0;
  }
  public Nat add(n2:Nat){
    return new S(nat0.add(n2));
  }
}
```

Fig. 2. Generated Java Code

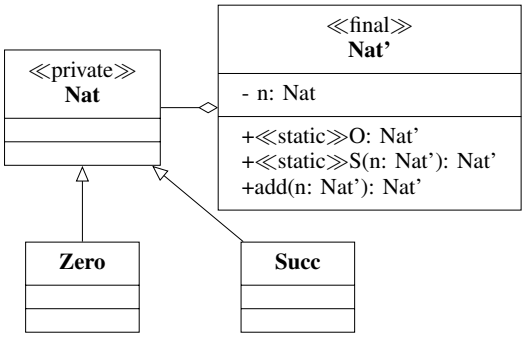


Fig. 3. Nat Class Constrained Hierarchy

In Mini-ML, as in every programming language, types are accompanied by functions to handle them. One of the strong mechanism used in functional languages is the pattern matching. A variable is matched with several patterns corresponding to the constructors of its type definition. The resulting expression attached to a pattern is derived into Java code for the corresponding class.

The Mini-ML code corresponding to the Coq code of Figure 1 is translated into the Java code given in Figure 2.

## IV. Discussion

In order to dynamically instantiate subclasses of main types (e.g $Zero$ and $Succ$ for $Nat$), a static factory method pattern can be introduced. For each instantiable object, a static function can be defined to control the creation of a class instance. The original constructors become $private$ to only allow the factory to build the instances. In addition, every constructor without fields (e.g. `O` in the `nat` context), does not need to be instantiated several times. A way to limit the number of instantiations is to use a singleton pattern in conjunction with the static factories.

As mentioned above, there exists several programming style. The proposed transformation is not the only possible one. For example, instead of using an *interface* to group the constructors into a single type, we can use an *abstract* class without any definitions.

One of the advantages of formalizing the languages with Coq is the possibility of proving the preservation of the semantic. In other words, it's possible to prove that the transformation of the Mini-ML to Java doesn't alter the semantic of the initial language. However, what if the user decide to implement the interface using her own classes? We cannot guarantee that a call to `add` has then the same semantic as in the Coq case, because of the new elements. There exists a way to block the extension of classes in Java, using the keyword *final* (e.g Figure 3). Our tool will allow the users to choose among a variety of code generation styles.

An object-oriented language seems to be an appropriate language for our extraction. Nonetheless Java has some limits including no direct support for higher-order functions. There exists alternative, such as the $Function$ interface (Java 8) to model functions with another function as parameter. Using the same interface, it is possible to create partial applications. Another lack of Java is the modular aspect. One of the strong tools available in Coq is its modular approach. Programmers can parameterize their modules and their arguments have module types (providing a set of type and value declarations and definitions). The most direct solution is the generic mechanism of Java which allows parameterizing a class by a type. In the OCamlJava project [9], Clerc proposes to constrain the generic types by the extension of a class respecting the specification of the module parameter. This is a starting point but there are still remaining challenges with this approach.

## V. Conclusion

In this paper, we presented a generator of Java programs from ML code. As numerous is the number of Java users, the language allows several approach to write semantically equivalent programs. The project merits to be deeply completed to allow a tranformation of the whole Coq language to Java, paramaterized by the user preferences.

## References


[1] The Coq Development Team, "The Coq Proof Assistant," http://coq.inria.fr.

[2] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[3] F. Loulergue, W. Bousdira, and J. Tesson, "Calculating Parallel Programs in Coq using List Homomorphisms," *Int J Parallel Prog*, vol. 45, pp. 300–319, 2017.

[4] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens, "Security monitor inlining and certification for multithreaded java," *Mathematical Structures in Computer Science*, vol. 25, no. 3, pp. 528–565, 2015.

[5] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon, "Prototyping a query compiler using coq (experience report)," *PACMPL*, vol. 1, no. ICFP, pp. 9:1–9:15, 2017.

[6] N. Gaspar, L. Henrio, and E. Madelaine, "Painless support for static and runtime verification of component-based applications," in *Fundamentals of Software Engineering - 6th International Conference, FSEN 2015 Tehran, Iran, April 22-24, 2015, Revised Selected Papers*, 2015, pp. 259–274.

[7] P. Letouzey, "Coq Extraction, an Overview," in *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, ser. LNCS, A. Beckmann, C. Dimitracopoulos, and B. Löwe, Eds., vol. 5028. Springer, 2008.

[8] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn, "A simple applicative language: Mini-ML," INRIA, Rapport Technique 529, 1986.

[9] X. Clerc, "OCaml-Java: An ML implementation for the Java ecosystem," in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 2013, pp. 45–56.