Position Paper: Thoughts on Programming with Proof Assistants

Adam Chlipala
Computer Science Division
University of California, Berkeley
Berkeley, California, USA
adamc@cs.berkeley.edu

1 Introduction

Today the reigning opinion about computer proof assistants based on constructive logic (even from some of the developers of these tools!) is that, while they are very helpful for doing math, they are an absurdly heavy-weight solution to use for practical programming. Yet the Curry-Howard isomorphism foundation of proof assistants like Coq [BC04] gives them clear interpretations as programming environments.

My purpose in this position paper is to make the general claim that Coq is already quite useful today for non-trivial certified programming tasks, as well as to highlight some reasons why you might want to consider using it as a base for your next project in dependently-typed programming.

In the last year, I've tried an experiment [Chl06] of using Coq to develop dependently-typed programs of non-trivial size. My application domain has been proof-carrying code, and the idea of "certified program verifiers" in particular. Certified verifiers are essentially an optimization that replaces monolithic proofs of program correctness with re-usable program verifiers that have themselves been proved sound. Almost entirely with Coq, I've implemented a memory safety verifier for x86 machine code programs that use ML-style algebraic datatypes, with a proof of soundness defined in terms of the real x86 machine code semantics.

The remainder of this paper is a summary of what I learned from the experience, starting with a discussion of a short example of the style of code that I found to be effective and concluding with summaries of traditional programming language features that I ended up not missing and Coq features that turned out to be very helpful.

2 Programming with Optional Proofs

Dependent types can be used with inductive type families in very intricate ways. A ubiquitous example is the use of type families for terms of programming languages, where the indices of the type family for a particular meta-language term determine its object-language type. In my work, I've made do with a much more modest subset of the power available in Coq's type system.

I've stuck to dependent types with the flavor of refinement types, where additional logical predicates over values can be attached to standard types. This sort of type naturally describes the result of, for instance, a complete type inference procedure; for any input term, it returns a type known to describe that term. Of course, in most interesting program verification, the properties that we want to check are undecidable, so the type of a procedure like I just described must allow it a way to fail.

The following example illustrates how such types can be used. It uses an even simpler type family, that of optional proofs of a proposition. Let's say that we are writing a verifier that at some point must make sure that a natural number is even before proceeding. Here's Coq code for a function `isEven` to do this:

```
Definition isEven : forall (n : nat), [[even n]].
  refine (fix isEven (n : nat) : [[even n]] :=
    match n return [[even n]] with
      | 0 => Yes
      | 1 => No
      | S (S n) => pf <- isEven n;
                   Yes
    end); auto.
Defined.
```

The use of this particular form of the `Definition` command indicates that we are constructing this program partially through *interactive proof search.* After the first line, the type of the desired function is asserted as a proof goal, and the body of the definition serves as a proof script to show how to "prove" it, Curry-Howard style. Why not just write the code directly? For this toy example, that works out fine, but, for larger examples, the amount of explicit programming with proofs that this entails is intractable. We would like to take advantage of Coq's features as a proof construction and automation tool as far as possible.

Correspondingly, we begin the body of the definition using the `refine` tactic, which says "I have in mind the structure of the proof, but it has some holes left to fill in." In this case, the structure that we know is the *computational* part of the function, or precisely the part we would write in a setting without formal proofs. The form of the code that we provide gives rise to some remaining proof obligations, and these are queued as subgoals to be handled interactively in usual Coq style.

Now we can turn to the particulars of the definition. The notation `[[`P`]]` denotes the type of optional proofs of proposition P. The `refine` body makes a primitive recursive definition with a `fix` expression, pattern matching on the natural number argument `n`. The notations `Yes` and `No` have the obvious meanings, with only a `Yes` answer registering a new proof obligation. The third, recursive case of the function definition checks the evenness of another natural number before returning its answer. Optional proofs are treated as a *failure monad* in standard Haskell style, enabling the concise notation that I've used. If the recursive call fails, then the current call fails with `No`; while, if the recursive call succeeds, the current call succeeds with `Yes`, and the proof `pf` that is returned can be used in discharging the proof obligation.

The `auto` tactic is chained onto the `refine` tactic with a semicolon, directing Coq to attempt to solve automatically every proof obligation added for the function body. For this simple code, all of the obligations are solvable in this way. In general, the full power of Coq for both user-coded automation and for elaborately scripted manual proof strategies can be used.

The example I've showed uses real Coq syntax, including some user-defined extensions. These extensions are the `[[`P`]]`, `Yes`, `No`, and monadic arrow notations. Expanding these notations and taking into account the results of automated proof search, we get internally an "explicit" definition like the following. One change is that `Yes` has become the constructor `PSome` applied to an explicit proof term. The `auto` tactic probably generates proof terms that aren't as nice to read as those I've used here.

```
Fixpoint isEven (n : nat) : Prop_option (even n) :=
  match n return Prop_option (even n) with
    | O => PSome even_O
    | S O => PNone
    | S (S n) => match isEven n with
                   | PNone => PNone
                   | PSome pf => PSome (even_SS pf)
                 end
  end.
```

3 Pros and Cons of Programming with Coq

3.1 Missing Programming Language Features

Coq is lacking a number of standard programming language features, and several recent language projects [CX05, WSW05] focus on bringing these features to dependently-typed programming.

Two big ones are imperativity and exceptions. I can only say anecdotally that I haven't missed either of these in the work I've described, and I'll point unconvinced readers to their local Haskell enthusiasts for further arguments. I found the failure monad style that I just sketched to be very effective in taking over for one common use of exceptions.

Then there is general recursion or the ability to write non-terminating programs in general. Coq has no separation of logic and programming language, so termination of all terms is required for soundness. I can again mention anecdotally that this only showed up once as a small inconvenience in my work, and Coq has good support for enabling a wide variety of termination arguments.

In summary, there may be areas like "systems" programming where Coq's pure, total programming model is a bad fit, but I believe that it works smoothly in a wide variety of application areas where you might want to bring dependent types to bear. The extent of Coq's programming support would probably surprise most people who haven't use it, as Coq includes a module system, a compilation toolchain that leads to fast native code, and easy integration with normal OCaml code.

3.2 How Coq Supports Effective Programming with Dependent Types

Coq has a number of features designed primarily for "proving" rather than "programming" that nonetheless turn out to be quite useful in dependently typed programming.

- The core of Coq, the Calculus of Inductive Constructions, is very small. This is desirable both because it provides a mathematically elegant and very general solution, and because it brings the trustworthiness benefits of a small trusted code base for a checker for Coq developments. The second point is important in the context of proof-carrying code.

- Languages that separate programming constructs from proof constructs lose the advantages of an idiom called "proof by reflection" [Bou97]. The basic idea of proof by reflection is that checking a proof involves running a program. For instance, in Coq one legal kind of proof of a program's correctness essentially says "Run this certified program verifier and make sure it accepts the program." The proof checker runs the verifier using the same syntactic mechanisms it uses to check proofs in general. It's possible to regain some of these advantages in a language with separate "programming" and "proving" levels by introducing a separate, more pure programming language for use in proofs, but it's nice to avoid this complication.

- Coq's tactic facility makes it easy to script custom decision procedures and use them to construct proofs arising as obligations in dependently-typed programming.

- Coq has a nice ML-style module system for structuring proof developments, programs, and combinations of the two.

- Coq has been around for a while, so there are a lot of libraries, pre-written proof-generating decision procedures, etc., available for it.

4 Conclusion

In the not-so-distant past, Coq was clunky to use and infeasible for real programming. Today, it is mature and reasonable to use for carrying out non-trivial certified programming projects. A number of key features designed originally for formalizing math turn out to play roles in enabling effective dependently-typed programming.

Languages like Epigram [MM04] have very similar foundations to Coq but focus more on programming than proving. The question of which to use seems to hinge on whether "programming" or "proving" aspects dominate the complexity of a program. Many applications in, for instance, high-level programming language semantics involve proofs that closely follow syntax, so that Epigram's features for dependent pattern matching make it a good choice. On the other hand, I think that for cases like reasoning about compilation from high-level languages to machine code, some very large, non-syntax-directed proofs will inevitably be involved, so that the biggest productivity gains are to be had by taking advantage of some serious proof organization and automation machinery. It also seems a promising direction to investigate the best ways of importing some of Coq's more recent proof-oriented features into more traditional programming settings.

References

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, pages 515–529, 1997.

[Chl06] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. In *ICFP '06: Conference record of the 11th ACM SIGPLAN International Conference on Functional Programming*, September 2006. To appear.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 66–77, 2005.

[MM04] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[WSW05] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2005.