Developing Certified Program Verifiers with a Proof Assistant

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

Abstract

I describe ongoing work on a new approach to foundational proof-carrying code. The key new idea is to use *certified program verifiers* to embody customized program verification strategies, specialized to particular safety policies, enforcement mechanisms, and source-level compilers. A certified verifier is an executable program that has a full correctness proof.

The particular strategy that I've been following involves using the Coq computer proof assistant as an environment for dependently typed programming, where types ensure total correctness. Elements of the development are interesting for the general insight they provide into programming with specifications.

1 Introduction

This poster describes my work on implementing program verification tools that have formal soundness proofs, using the Coq proof assistant [BC04].

The idea of certified program verifiers has important practical ramifications for foundational proof-carrying code (FPCC) [App01]. Like traditional proof-carrying code (PCC), FPCC is primarily a technique for allowing software consumers to obtain strong formal guarantees about programs before running them. It differs in that its trusted base is independent of particular source languages and compilation strategies. In this way, it gains both generality and higher assurance of soundness over the first PCC systems.

The germ of the project I'll describe comes from past work on improving the runtime efficiency of FPCC program checking. Perhaps the largest obstacle to practical use of FPCC stems from the delicate trade-offs between generality on one hand and space and time efficiency of proofs and proof checkers on the other. Program verifiers like the Java bytecode verifier have managed to creep into wide use almost unnoticed by laypeople, but naive FPCC proofs are much larger than the metadata included with Java class files and take much longer to check. It's unlikely that this increased burden would be acceptable to the average computer user.

Fundamentally, custom program verifiers with specialized algorithms and data structures have a leg up on very general proof-based verifiers. In our initial work on certified program verifiers [CCN06], we proposed getting the best of both worlds by moving up a level of abstraction: allow developers to ship their software with specialized *proof-carrying verifiers*. These verifiers have the semantic functionality of traditional program verifiers and model-checkers, but they also come with machine-checkable proofs of soundness. Each such proof can be checked *once* when a certified verifier is installed. After the proof checks out, the verifier can be applied to any number of similar programs. These later verifications require no runtime generation or checking of uniform proof objects, which we found to be the major bottleneck in previous experience with FPCC. Our paper presents performance results showing an order of magnitude improvement over all published verification time figures for FPCC systems for Typed Assembly Language [MWCG99] programs, by using a certified verifier. The verifier had a complete soundness proof, so no formal guarantees were sacrificed to win this performance.

2 Implementing Certified Verifiers with Coq

The main problem that we encountered was in the engineering issues of proof construction. We used a more or less traditional approach to program verification in proving the soundness of our verifiers, writing them in a standard programming language and extracting verification conditions that imply their soundness. Keeping the proof developments in sync with changes to verifier source code was quite a hassle. We also found that the structure of the verifier program and its proof were often very closely related, leading to what felt like duplicate work.

I decided to try investigating what could be gained by writing verifiers from the start in a language expressive enough to encode verifier soundness in its type system. My upcoming paper [Chl06] describes a case study using Coq to develop a complete memory safety verifier for x86 machine code programs that use ML-style algebraic datatypes.

My approach combines explicit dependently-typed programming with interactive and automated theorem proving inside of Coq. By providing decision procedures and other verifier ingredients with types that only admit sound implementations, it's possible to avoid implicit dependencies between programs and proofs about them. This style of programming is not new, but my results show that it can be seen through to completion in a reasonable amount of time.

My implementation uses a number of Coq type families in the style of refinement types [FP91], including some standard ones and some new ones that I define. Refinement types refine type systems by providing types that describe, for instance, "every value of type τ that satisfies logical predicate P." Standard refinement types stick to predicates that facilitate effective type inference, while Coq provides the mechanisms to use arbitrary logical predicates. Proper use of values of refinement types is validated through explicit construction of proof terms in programs.

I use types like $\{\!\{x : \tau \mid P(x)\}\!\}$, which (for an arbitrary logical predicate P) is an `option`-like type whose values are either special failure values or packages containing a value x of type τ and a proof that $P(x)$ holds. These type families can be treated as failure monads, leading to a style familiar from Haskell programming. Computations can proceed with sequences of calls to potentially-failing subroutines, where each successful call binds a return value and a proof of its correctness for use in later calls. Monadic syntax makes the code as clear visually as solutions that use, e.g., exceptions.

Coq's features for semi-automated proof construction and program extraction work very well with this style. It's possible to write programs with "proof holes" that are queued as goals for the user to prove interactively, which is a whole lot nicer than writing out proof terms manually. Program extraction also provides for a compilation from a finished Coq development into OCaml code, which can then be compiled to speedy native code. Extraction erases all proof-related parts of the program, and the Coq type system guarantees that this erasure is semantics-preserving for well-typed terms. Thus, total correctness theorems about an original Coq verifier are guaranteed to hold about the compiled version that we actually run, modulo bugs in Coq and OCaml.

The other essential technique is the use of Coq's module system to build verifiers from reusable components. Functors serve as a tool for translating a verifier at one level of abstraction into a verifier for a lower level. In this way, my final implementation is based on a stack of 8 abstraction levels, from x86 machine code to a high-level, declarative type system description. Like at the level of individual functions, dependent types make it relatively painless to glue components together, since there's no need to stare at the final product and devise a customized, global proof strategy. The module system ensures that the proof parts of the different components work smoothly together.

3 Conclusion

In the history of both traditional and foundational PCC, research has started out focusing on expressivity and the basic concepts. From there, the next steps involve techniques to improve algorithmic efficiency. Especially for FPCC, there is a need for a third stage focusing on practical issues in the development of new FPCC pieces. We shouldn't rest satisfied as long as the difficulty of implementing support for a new compiler within an FPCC framework is measured in PhD theses. I hope that the kinds of software engineering techniques that I've mentioned here can provide some insight into the challenges of this third stage.

References

[App01] Andrew W. Appel. Foundational proof-carrying code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247, 2001.

[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.

[CCN06] Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *VMCAI '06: Proceedings of the Seventh International Conference on Verification, Model Checking and Abstract Interpretation*, January 2006.

[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.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.