CertiCoq: A verified compiler for Coq

Abhishek Anand[1] Andrew W. Appel[2] Greg Morrisett[1] Zoe Paraskevopoulou[2]
Randy Pollack[3] Olivier Savary Bélanger[2] Matthieu Sozeau[4] Matthew Weaver[2]

[1] Cornell University [2]Princeton University [3]Edinburgh University [4]Inria

CoqPL workshop, Paris, January 21, 2017

Abstract

CertiCoq is a mechanically verified, optimizing compiler for Coq that bridges the gap between certified high-level programs and their translation to machine language. We outline its design as well as the main foundational and engineering challenges involved in building and certifying a compiler for Coq in Coq.

1. Introduction

Certified programs, i.e. programs that come equipped with machine-checked proofs of specifications, are becoming more and more prevalent since the emergence of dependently typed languages that allow the users to express and prove properties of programs within the language itself. Notably, Coq has been used to certify a considerable amount of realistic software, including compilers [7], web servers [5] and databases [9]. A commonly used practice in software certification within Coq is to write programs in Gallina, the core language of Coq, mechanically verify them, and then use one of the provided extraction mechanisms in order to obtain an executable version of the certified program. However, the existing extraction pipelines and the compilers of the languages they are targeting are not certified, and thus they provide no guarantee that the low-level code will match the specification of the source program. CertiCoq aims to bridge this gap by providing a verified extraction pipeline from Coq to machine language. Currently, we are targeting CompCert C light, thus we can compose with the CompCert compiler (and its correctness proof) to get a verified compilation pipeline from Gallina to machine language. But we are also exploring an LLVM back end, for composition with a possible future verified LLVM [13]. All phases of our compiler are written in Gallina, and we are proving them correct in Coq. Most phases have already been proven correct while the rest have proofs in progress.

The area of formal compiler certification has grown extensively since the CompCert [7] project. Significant efforts have been made in building certified compilers for functional languages, including CakeML [12], which has reached a remarkable level of realism, and the compositional Pilsner compiler [11]. Although correct compilation is important on its own sake, the source languages of these compilers lack a proof theory that allows the user to certify properties about the source programs, and thus they do not provide end-to-end correctness guarantees. Towards this end, the VST project provides a program logic to reason about deep embeddings of C programs within Coq. The program logic is proved sound with respect to the C semantics and the C program is compiled with the CompCert C compiler, providing an end-to-end correctness specification. The great advantage of Gallina over C is that its proof theory (CiC) is much easier to use than C's proof theory (separation logic).

In the following sections we present the guiding principles and the design of CertiCoq, and we give an overview of the research challenges tackled by this effort.

2. Principles

Principle 1: Erase the types! Compiler correctness is a stronger property than type preservation, anyway. We compile through a series of untyped IRs, maintaining a record of useful information on the side—such as the arities of data constructors of inductive data types, from which we can derive tagged data representations in memory. With erased types, we avoid the need for a *full* Coq-in-Coq; all we need to deeply embed is evaluation semantics of the computational part.

Principle 2: Use properties of the source language to simplify reasoning about transformations! For example, we can restrict our reasoning to terminating programs since Coq is strongly normalizing. This way we avoid backward simulations (forward simulation proofs are much simpler) and avoid proving preservation of divergence.

Principle 3: Write compilers in a purely functional language with a good proof theory! This principle is already well established [7]; we choose Coq.

3. Structure of the Compiler

Our compiler is a pipeline of several phases through several intermediate languages. For each phase we prove that (weak) call by value big-step semantics are preserved.

$\boxed{L_0\text{: Gallina, repr. in Ocaml data constructors in the Coq kernel}}$

Annotate proofs, reify into Coq using *template-coq*.

$\boxed{L_1\text{: Gallina, represented in Coq inductive data constructors}}$

Proof erasure: replace proofs by noninformative nonces [8].
Erase type labels (λ, $\forall$, `match`).

$\boxed{L_2\text{: Type-erased Gallina}}$

η-expand constructors, flatten spine applications to unary.

$\boxed{L_3\text{: Simplified Gallina}}$

Close program by let-binding definitions from the environment.

$\boxed{L_4\text{: Closed Gallina}}$

Move from deBruijn bindings to named, abstract binding trees to use the SquiggleLazyEq theory of substitution [1].

$\boxed{L_{4a}\text{: Named } \lambda \text{ calculus}}$

CPS conversion: similar to [4].

$\boxed{L_5\text{: named CPS with substitution-based evaluation}}$

Make variable names globally unique, ensure that arguments to constructors are variables.

$\boxed{L_6\text{: machine-oriented CPS with environment-based evaluation}}$

Uncurrying: Uncurry function applications using η-expansions.
Shrink-reductions: function inlining, constant folding, dead variable elimination using an efficient algorithm [3] adapted and improved.

Lambda lifting: replace calls to known functions with calls to functions whose free variables have become formal parameters. This "unboxes" closure environments.
Closure conversion: replace functions with pairs of closed functions (with an extra environment parameter) and their environment.
Hoisting: unnest functions and move them to top-level.

$\boxed{L_{6c}\text{: CPS with no nested functions}}$

Generate stackless closure-passing code in the style of SML/NJ [2] and Manticore [6], represented as tail calls (and function pointers) in C.

$\boxed{L_7\text{: CompCert C light}}$

Compile with CompCert.

$\boxed{L_\infty\text{: assembly language}}$

4. Research Challenges and Future Directions

Axioms. Coq's logic is known to be consistent with axioms like UIP and functional extensionality, which are used in several Coq developments. However, axioms can wreak havoc with Coq's computation – they can cause programs to get stuck. Following Letouzey [8], early phases of our compiler eliminate proofs (including axioms) so that they cannot get into the way of computations. We plan to extend Letouzey's analysis of the method by developing mechanized proofs of stronger properties (observational equivalence instead of just the agreement of head constructors), and that these properties hold even in the presence of some blessed axioms. Furthermore, to enable optimizations such as hoisting a closed term out of a lambda, we may need to extend Letouzey's method to keep track of enclosing binders during erasure.

Effects. Even though Coq is a pure language, one can encode effects as pure monadic values denoting actions like reading and writing memory or files [9, 10]. Because this approach does not sacrifice purity, or extend Coq with any new construct, no change would be needed to the initial phases of the compiler (L0-L6). In particular, we would not need to worry about effects in optimizations such as common subexpression elimination[1]. We plan to design a coinductively defined monad in Coq to encode common effectful primitive actions in C. We will also investigate a specification of how some of those actions change the world (e.g. heap) and the soundness of the specification w.r.t. the CompCert C semantics.

Compositionality. We aim to prove the compiler correct in a compositional way. This compositionality property is three-fold: (a.) the correctness statements of each phase should compose vertically, (b.) CertiCoq's end-to-end theorem should compose with CompCert's end-to-end theorem, and (c.) each one of these correctness statements should compose horizontally in order to support separate compilation and linking. Our goal is to show semantics preservation; since we are only interested in safe programs and the semantics is deterministic, semantics preservation implies semantics refinement.

Portability. By targeting C instead of machine language, we can leverage the verified phases of CompCert while gaining portability to all of CompCert's target machines (x86-32, x86-64, ARM, Power, Risc-V). But this comes at a cost: C's calling conventions are not a perfect match for continuation-based (all tail-call) programs. There may be simple mitigations of this problem, e.g., equip CompCert with different calling conventions for C, as is already done in the GHC→LLVM compiler.

[1] Currently, because of the lack of common subexpression elimination, two definitionally equal Coq programs, when executed after extraction to OCaml, may have drastically different asymptotic complexity – see https://sympa.inria.fr/sympa/arc/coq-club/2016-01/msg00177.html

Interface. We use Ocaml data representations with C calling conventions, so our compiled Gallina programs should be easily callable from Ocaml or C.

Garbage collection. We have a high-performance generational garbage collector. It is particularly simple because Gallina is *purely* functional; unlike imperative languages such as ML (ref-update) and Haskell (thunk-update), there are never stores to already allocated data. A project is underway in Singapore to prove this collector correct using Verifiable C. Our programs are also compatible with the Ocaml collector.

5. Conclusion

This work in progress should soon yield a compiler that is **useful** (for compiling Gallina programs that you have proved correct in Coq), **efficient** (in compilation time), **optimizing** (competitive with Ocaml), **verified** (with proofs in Coq), **compatible** (with C and Ocaml), and **elegant**. Regarding the last point, we have carefully structured the intermediate languages so that each phase does a clearly specified transformation whose proof is clear and maintainable.

Acknowledgments. This research was supported in part by NSF grants CCF-1407790, CCF-1407794, and CCF-1521602.

References

[1] A. Anand. Exploiting uniformity in substitution: The Nuprl term model. International Conference on Mathematical Software, 2016. URL https://github.com/aa755/SquiggleEq.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[3] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *J. Funct. Program.*, 7(5):515–540, 1997.

[4] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 211–225. Springer, 2007.

[5] S. L. Dongseok Jang, Zachary Tatlock. Establishing browser security guarantees through formal shim verification. In *USENIX Security Symposium*, page 8, 2012.

[6] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming*, pages 37–44. ACM, 2007. ISBN 978-1-59593-690-5.

[7] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 42–54. ACM, 2006.

[8] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Univ. Paris-Sud, July 2004.

[9] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. POPL, pages 237–248. ACM, 2010.

[10] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP'08*. ACM, 2008.

[11] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *ICFP'15*, pages 166–178. ACM, 2015.

[12] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A new verified compiler backend for CakeML. In *ICFP'16*, pages 60–73. ACM, 2016.

[13] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL'12*, pages 427–440. ACM, 2012.