Verified compilation

Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

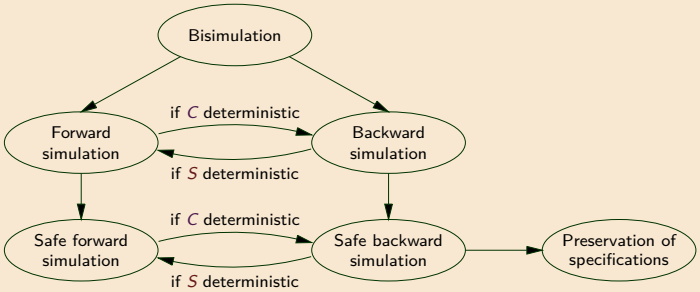
Verified compilation principles

"The striking thing about our CompCert results is that ***the middle-end bugs we found in all other compilers are absent****. [...] we have devoted about six CPU-years to the task."*

Finding and Understanding Bugs in C Compilers (2011)
X. Yang, Y. Chen, E. Eide, J. Regehr

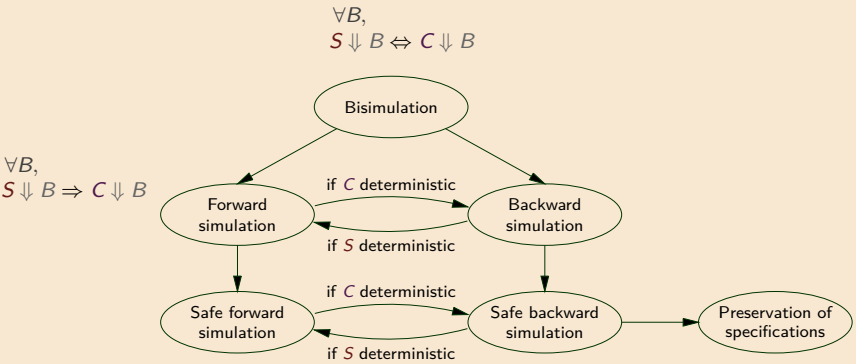
$P \Downarrow B$: program P has observable behaviour B[1]

$\forall B,$
$S \Downarrow B \Leftrightarrow C \Downarrow B$



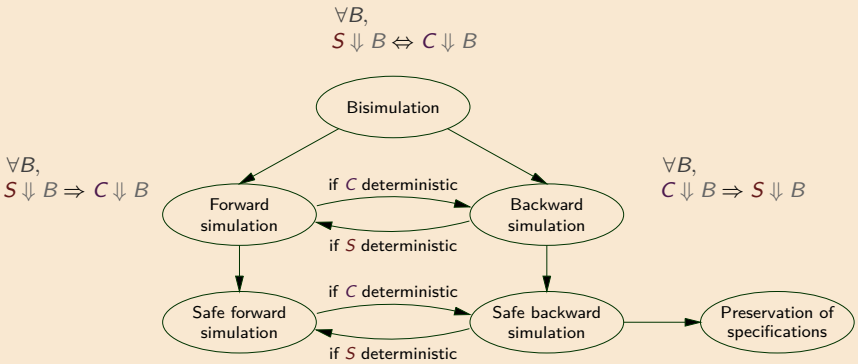
[1]Taxonomy from *A formally verified compiler back-end* (2009) by Xavier Leroy

$P \Downarrow B$: program P has observable behaviour B[1]



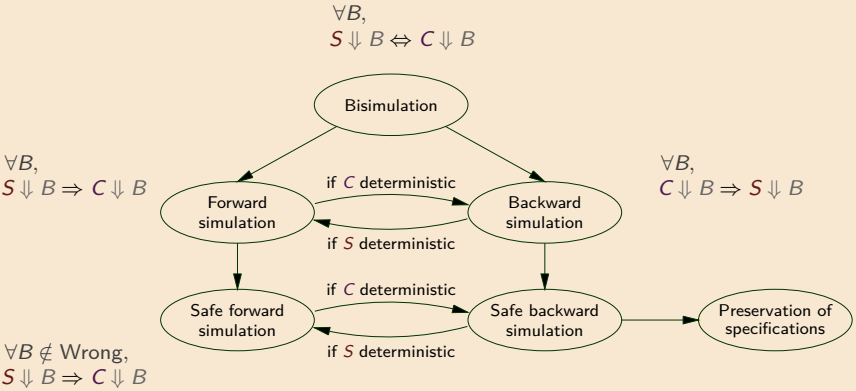
[1]Taxonomy from *A formally verified compiler back-end* (2009) by Xavier Leroy

$P \Downarrow B$: program P has observable behaviour B[1]



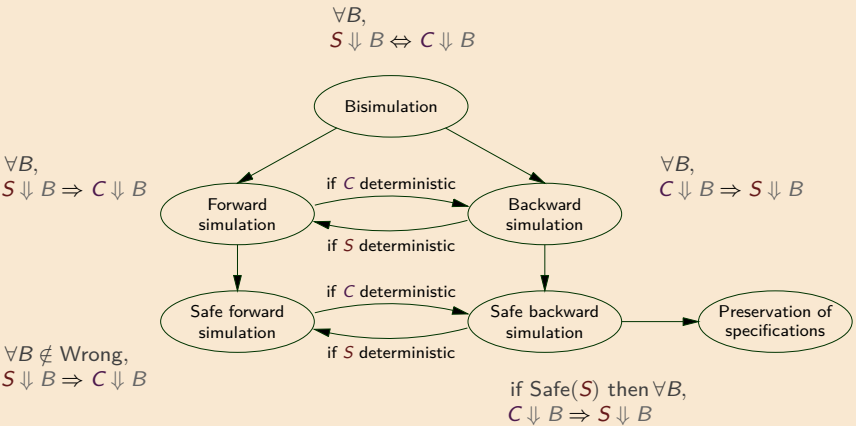
[1]Taxonomy from *A formally verified compiler back-end* (2009) by Xavier Leroy

$P \Downarrow B$: program P has observable behaviour B[1]



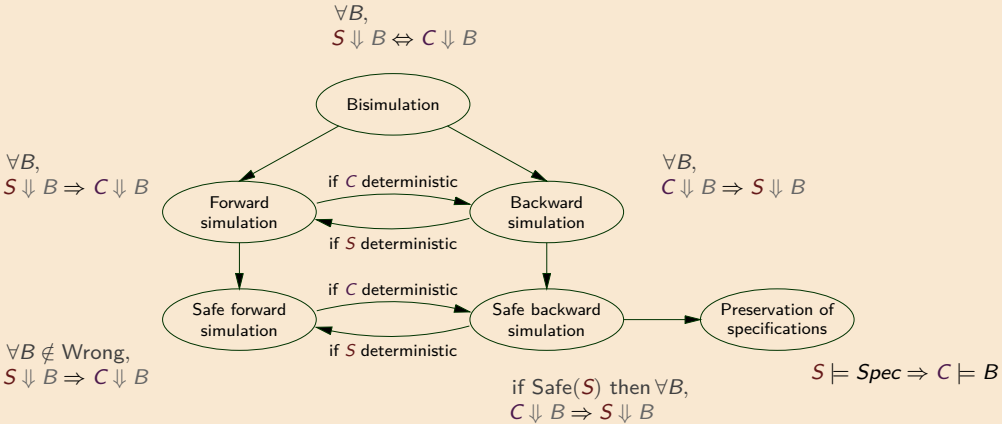
[1]Taxonomy from *A formally verified compiler back-end* (2009) by Xavier Leroy

$P \Downarrow B$: program P has observable behaviour B[1]



[1]Taxonomy from *A formally verified compiler back-end* (2009) by Xavier Leroy

$P \Downarrow B$: program P has observable behaviour B[1]



[1]Taxonomy from *A formally verified compiler back-end* (2009) by Xavier Leroy

Verified compilation:

$$\forall S, C, \quad Comp(S) = \mathtt{OK}(C) \Longrightarrow S \approx C$$

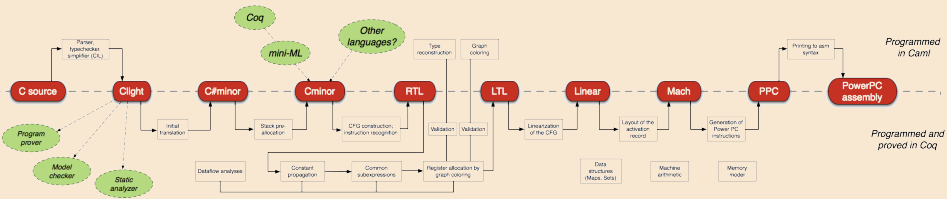
Translation validation:

If $Comp(S) = \mathtt{OK}(C)$ and $Validate(S, C) = \text{true}$
then deem C trustworthy

Q: does the difference matter?[2]

[2] More details in *A formally verified compiler back-end* (2009) by Xavier Leroy

Existing projects

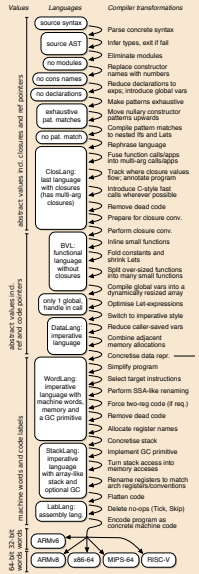


Language: (most of) standard C

Implementation: mostly Rocq/Coq

Proofs: backwards simulation

Approach: mostly verified
(some translation validation)



CAKEML

A Verified Implementation of ML

Language: (variant of) Standard ML

Implementation: HOL4

Proofs: forwards simulation

Approach: almost entirely verified

Building on CakeML

Several smaller projects reuse parts of existing verified compilers:

Candle (HOL)

Abrahamsson et al
ITP'22

PureCake (Haskell)

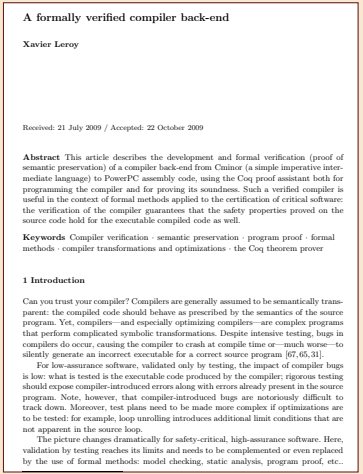
Kanabar et al
PLDI'23

Brack (Scheme)

Lasnier et al
CPP'26

Reading

Paper 1: CompCert (2009)



A formally verified compiler back-end

Xavier Leroy

Received: 21 July 2009 / Accepted: 22 October 2009

Abstract This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the certification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Keywords Compiler verification · semantic preservation · program proof · formal methods · compiler transformations and optimizations · the Coq theorem prover

1 Introduction

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compiler to crash at compile time or—much worse—to silently generate an incorrect executable for a correct source program [67,65,31].

For low-assurance software, validated only by testing, the impact of compiler bugs is low: what is tested is the executable code produced by the compiler; rigorous testing should expose compiler-introduced errors along with errors already present in the source program. Note, however, that compiler-introduced bugs are notoriously difficult to track down. Moreover, test plans need to be made more complex if optimizations are to be tested: for example, loop unrolling introduces additional limit conditions that are not apparent in the source loop.

The picture changes dramatically for safety-critical, high-assurance software. Here, validation by testing reaches its limits and needs to be complemented or even replaced by the use of formal methods: model checking, static analysis, program proof, etc..

"For the last four years, we have been working on the development of a *realistic*, *verified* compiler called Compcert.

By *verified*, we mean a compiler that is accompanied by a machine-checked proof that the generated code behaves exactly as prescribed by the semantics of the source program (semantic preservation property).

By *realistic*, we mean a compiler that could realistically be used in the context of production of critical software."

Paper 2: CakeML (2014)

Principles

Systems

Reading

CakeML: A Verified Implementation of ML

Ramana Kumar [*1] Magnus O. Myreen [†1] Michael Norrish [2] Scott Owens [3]

[1] Computer Laboratory, University of Cambridge, UK
[2] Canberra Research Lab, NICTA, Australia[‡]
[3] School of Computing, University of Kent, UK

Abstract

We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop (REPL) in x86-64 machine code. Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

Our contributions are twofold. The first is simply in building a system that is end-to-end verified, demonstrating that each piece of such a verification effort can in practice be composed with the others, and ensuring that some of the pieces rely on any over-simplifying assumptions. The second is developing novel approaches to some of the more challenging aspects of the verification. In particular, our formally verified compiler can bootstrap itself: we apply the verified compiler to itself to produce a verified machine-code implementation of the compiler. Additionally, our compiler proof handles diverging input programs with a lightweight approach based on logical timeout exceptions. The entire development was carried out in the HOL4 theorem prover.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, Formal methods; F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification, Specification techniques, Invariants

Keywords Compiler verification; compiler bootstrapping; ML; machine code verification; read-eval-print loop; verified parsing; verified type checking; verified garbage collection.

[*] supported by the Gates Cambridge Trust
[†] supported by the Royal Society, UK
[‡] NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01. . . $15.00
http://dx.doi.org/10.1145/2535838.2535841

1. Introduction

The last decade has seen a strong interest in verified compilation; and there have been significant, high-profile results, many based on the CompCert compiler for C [1; 14, 16, 29]. This interest is easy to justify: in the context of program verification, an unverified compiler forms a large and complex part of the trusted computing base. However, to our knowledge, none of the existing work on verified compilers for general-purpose languages has addressed all aspects of a compiler along two dimensions: one, the compilation algorithm for converting a program from a source string to a list of numbers representing machine code, and two, the execution of that algorithm as implemented in machine code.

Our purpose in this paper is to explain how we have verified a compiler along the full scope of both of these dimensions for a practical, general-purpose programming language. Our language is called CakeML, and it is a strongly typed, impure, strict functional language based on Standard ML and OCaml. By verified, we mean that the CakeML system is ultimately x86-64 machine code alongside a mechanically checked theorem in higher-order logic saying that running that machine code causes an input program to yield output or diverge as specified by the semantics of CakeML.

We did not write the CakeML compiler and platform directly in machine code. Instead we write it in higher-order logic and synthesise CakeML from that using our previous technique [22], which puts the compiler on equal footing with other CakeML programs. We then apply the compiler to itself, *i.e.*, we *bootstrap* it. This avoids a tedious manual refinement proof relating the compilation algorithm to its implementation, as well as providing a moderately large example program. More specifically,

- we write, and can run, the compiler as a function in the logic, and we synthesise a CakeML implementation of the compiler inside the logic;
- we bootstrap the compiler to get a machine-code implementation inside the logic; and
- the compiler correctness theorem thereby applies to the machine-code implementation of the compiler.

Another consequence of bootstrapping is that we can include the compiler implementation as part of the runtime system to form an interactive read-eval-print loop (REPL). A verified REPL enables high-assurance applications that provide interactivity, an important feature for interactive theorem provers in the LCF tradition, which were the original motivation for ML.

Contributions

- Semantics that are carefully designed to be simultaneously suitable for proving meta-theoretic language properties and for supporting a verified implementation. (Section 3)
- An extension of a proof-producing synthesis pathway [22] originally from logic to ML, now to machine code (via verified compilation). (Sections 4–6, 10)

"We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop (REPL) in x86-64 machine code.

Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping."

Paper 3: CertiCoq (2019)

Principles

Systems

Reading

Certified code generation from CPS to C

Olivier Savary Bélanger
Princeton University
olivier@galois.com

Matthew Z. Weaver
Princeton University
mzw@cs.princeton.edu

Andrew W. Appel
Princeton University
appel@princeton.edu

Abstract

CertiCoq is a verified-in-Coq extracter/compiler from Coq's Gallina language through CompCert C to assembly language, written as a functional program in Coq. Here we describe the implementation and Coq verification of its code generator, which translates from a continuation-passing style (CPS) intermediate language into CompCert Clight. We show how invariants over our CPS IR facilitate the generation of well behaved, efficient C code. A key point is our proved-correct interface to an external proved-correct (by other authors) generational garbage collector written in C. The semantics of C can be quite intricate, as can the design of a compiler-to-*g.c.* interface for finding roots—but the design of our CPS intermediate language facilitates a (relatively) simple implementation and correctness proof. Our measurements show that both the code generator and the generated code have good performance. Via CompCert, we have proved-correct back ends for several instruction-set architectures: x86-32, x86-64, ARM-32, ARM-64, RISC-V, and Power-PC.

CCS Concepts • **Software and its engineering** → **Formal software verification**; *Compilers*; *Correctness*; *Functional languages*; • **Theory of computation** → *Program verification*.

ACM Reference Format:
Olivier Savary Bélanger, Matthew Z. Weaver, and Andrew W. Appel. 2019. Certified code generation from CPS to C. In *Proceedings of (October 2019).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

If you prove your functional program in correct in Coq, then why entrust it to an unverified extraction/compilation pipeline? Neither Coq's extraction-to-Ocaml, nor the Ocaml compiler, nor Ocaml's runtime system is proved correct.

CertiCoq [1] is a verified-in-Coq extracter/compiler from Coq's *Gallina* functional language to assembly language, via CompCert C. The source program is extracted from the Coq kernel (its AST is reified from Ocaml datatype constructors to Coq datatype constructors) by MetaCoq [2]; this is the only phase that cannot be proved correct but it does no more than transliteration. After that (and with proofs!), we erase proofs, types, and related computationally irrelevant content [3]; constructors are eta-expanded so each constructor application is fully applied to all its arguments; the program is combined with its environment by let-binding all imported definitions, resulting in an untyped program in a simple de Bruijn functional language with inductive constructors. Then we convert to continuation-passing style (CPS) using a named representation, and we apply optimizations such as uncurrying [4], shrink-reduction [5], and lambda lifting; we closure-convert [6] into CPS terms in which all functions are closed (except for references to other closed functions in the global scope).

As we were not interested in verifying register allocation or supporting multiple back-ends for many target architectures, we translate our CPS into CompCert C light, and use CompCert as our verified register allocator and back-end code generator.

CertiCoq has a high-performance generational garbage collector, written in C and proved correct in Coq [7]. When one connects any compiler to a garbage collector, one must make an interface by which the compiler calls the collector, indicating where to find all the roots of the data graph—that is, the live local variables; and (when copying collection is used) one must be prepared for all variables of the program to be modified to point to their new locations.

Contributions. In this paper we describe the proof of correctness of our CPS-to-C translation phase: how we relate the operational semantics of CPS to the operational semantics of CompCert C, and how we reason about the graph transformation inherent in the call to the collector. These proofs are connected to proofs of the front-end phases via the CPS syntax and semantics, and to the CompCert back-end via the CompCert Clight syntax and semantics.

The artifact accompanying this paper has the code generator and its correctness proofs (not the rest of CertiCoq) plus the imported components of CompCert (that is, files leading up to the AST and operational semantics of Comp-

"CertiCoq is a verified-in-Coq extracter/compiler from Coq's Gallina language through CompCert C to assembly language, written as a functional program in Coq.

As we were not interested in verifying register allocation or supporting multiple back-ends for many target architectures, we translate our CPS into CompCert C light, and use CompCert as our verified register allocator and back-end code generator."

Principles

The relationship between backwards and forward simulation

The relationship between compiler verification and translation validation

Practicalities

How should we choose a proof assistant?

Are some languages more suited to verified compilation than others?

Economics

When is the cost worth it?

Will verified compilation eventually be the norm?