Compiler verification for fun and profit
Xavier Leroy

To cite this version:

HAL Id: hal-01076547
https://inria.hal.science/hal-01076547
Submitted on 22 Oct 2014
Compiler verification for fun and profit

Xavier Leroy
Inria Paris–Rocquencourt,
Domaine de Voluceau, BP 105, 78153 Le Chesnay, France
Email: xavier.leroy@inria.fr

ABSTRACT OF INVITED TALK

Formal verification of software or hardware systems — be it by model checking, deductive verification, abstract interpretation, type checking, or any other kind of static analysis — is generally conducted over high-level programming or description languages, quite remote from the actual machine code and circuits that execute in the system. To bridge this particular gap, we all rely on compilers and other code generators to automatically produce the executable artifact. Compilers are, however, vulnerable to miscompilation: bugs in the compiler that cause incorrect code to be generated from a correct source code, possibly invalidating the guarantees so painfully obtained by source-level formal verification. Recent experimental studies [1] show that many widely-used production-quality compilers suffer from miscompilation.

The formal verification of compilers and related code generators is a radical, mathematically-grounded answer to the miscompilation issue. By applying formal verification (typically, interactive theorem proving) to the compiler itself, it is possible to guarantee that the compiler preserves the semantics of the source programs it transforms, or at least preserves the properties of interest that were formally verified over the source programs. Proving the correctness of compilers is an old idea [2], [3] that took a long time to scale all the way to realistic compilers. In the talk, I give an overview of CompCert C [4], a moderately-optimizing compiler for almost all of the ISO C 99 language that has been formally verified using the Coq proof assistant [5].

The CompCert project is one point in a space of code generators whose verification deserves attention. For example, functional languages and object-oriented languages raise the issue of jointly verifying the compiler and the run-time system (memory management, exception handling, etc) that the generated code depends on. At the other end of the expressiveness spectrum, synchronous languages and hardware description languages also raise interesting verified generation issues, as exemplified by Pnueli’s seminal work on translation validation for Signal [6] and Braibant and Chlipala’s recent work on verified hardware synthesis [7].

Orthogonally, the integration of verification tools and compilers that are both verified against a shared formal semantics opens fascinating opportunities for “super-optimizations” that generate better code by exploiting the properties of the source code that were formally verified.

REFERENCES

ACKNOWLEDGMENTS

This work was supported by the VERASCO project (ANR-11-INSE-003) of Agence Nationale de la Recherche.