Formally verifying a compiler: Why? How? How far?

Xavier Leroy

Formally Verifying a Compiler: Why? How? How far?

Xavier Leroy
INRIA Paris-Rocquencourt
Xavier.Leroy@inria.fr

Abstract of invited talk

Given the complexity and sophistication of code generation and optimization algorithms, and the difficulty of systematically testing a compiler, it is unsurprising that bugs occur in compilers and cause *miscompilation*: incorrect executable code is silently generated from a correct source program [5]. The formal verification of a compiler is a radical solution to the miscompilation issue. By applying formal methods (program proof) to the compiler itself, compiler verification proves, with mathematical certainty, that the generated executable code behaves exactly as prescribed by the semantics of the source program [3].

Why indulge in compiler verification? Miscompilation bugs are uncommon enough that, for ordinary, non-critical software, they are negligible compared with the bugs already present in the source program. This is no longer true, however, for critical software, as found in aircraft, medical equipment or nuclear plants, for example. There, miscompilation is a concern, which is currently addressed in unsatisfactory ways such as turning all optimizations off. The risk of miscompilation also diminishes the usefulness of formal methods (model checking, static analysis, program proof) applied at the source level: the guarantees so painfully obtained by source-level verification may not extend to the compiled code that actually runs.

How to verify a compiler? For every pass, there is a choice between 1- proving its implementation correct once and for all, and 2- leaving the implementation untrusted, but at each run feed its input and output into a *validator*: a separate algorithm that tries to establish semantic preservation between input and output code, and aborts compilation (or turns the optimization off) if it cannot. If the soundness of the validator is proved once and for all, approach 2 provides soundness guarantees as strong as approach 1, often at reduced proof costs. I will illustrate both approaches on the verification of two compilation passes: register allocation by graph coloring [2, §8], and software pipelining [4].

How far can compiler verification go? The state of the art is the CompCert compiler, which compiles almost all of the C language to PowerPC, ARM and x86 assembly and has been mechanically verified using the Coq proof assistant [1]. From a compiler standpoint, however, CompCert leaves much room for improvement: the compilation technology used is early 1980's, few optimizations are implemented, and the performance of the compiled code barely matches that of `gcc -O1`. Much future work remains, in particular on loop optimizations, which raise semantic challenges that call for principled solutions.

Speaker's bio

Xavier Leroy is a senior research scientist at INRIA near Paris, where he leads the Gallium research team. He received his Ph.D. from University of Paris 7 in 1992. His work focuses on programming languages and systems and their interface with the formal verification of software for safety and security. He is the architect and lead author of the Objective Caml functional programming language and of its implementation. He has published about 50 research papers, and received the 2007 Monpetit prize of the French Academy of sciences.

References

[1] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[2] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[3] R[obin] Milner and R[ichard] Weyhrauch. Proving compiler correctness in a mechanized logic. In Bernard Meltzer and Donald Michie, editors, *Proc. 7th Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 51–72. Edinburgh University Press, 1972.

[4] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *37th symposium Principles of Programming Languages*, pages 83–92. ACM Press, 2010.

[5] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation 2011*. ACM Press, 2011. To appear.