Formal Proofs of Code Generation and Verification Tools

Xavier Leroy

Inria Paris-Rocquencourt, France

Abstract. Tool-assisted verification of critical software has great potential but is limited by two risks: unsoundness of the verification tools, and miscompilation when generating executable code from the sources that were verified. A radical solution to these two risks is the deductive verification of compilers and verification tools themselves. In this invited talk, I describe two ongoing projects along this line: CompCert, a verified C compiler, and Verasco, a verified static analyzer based on abstract interpretation.

Abstract of Invited Talk

Tool-assisted formal verification of software is making inroads in the critical software industry. While full correctness proofs for whole applications can rarely be achieved [6,12], tools based on static analysis and model checking can already establish important safety and security properties (memory safety, absence of arithmetic overflow, unreachability of some failure states) for large code bases [1]. Likewise, deductive program verifiers based on Hoare logic or separation logic can verify full correctness for crucial algorithms and data structures and their implementations [11]. In the context of critical software that must be qualified against demanding regulations (such as DO-178 in avionics or Common Criteria in security), such tool-assisted verifications provide independent evidence, complementing that obtained by conventional verification based on testing and reviews.

The trust we can put in the results of verification tools is limited by two risks. The first is *unsoundness* of the tool: by design or by mistake in its implementation, the tool can fail to account for all possible executions of the software under verification, reporting no alarms while an incorrect execution can occur. The second risk is *miscompilation* of the code that was formally verified. With a few exceptions [3], most verification tools operate over source code (C, Java, ...) or models (Simulink or Scade block diagrams). A bug in the compilers or code generators used to produce the executable machine code can result in an incorrect executable being produced from correct source code [13].

Both unsoundness and miscompilation risks are known in the critical software industry and accounted for in DO-178 and other regulations [7]. It is extremely difficult, however, to verify an optimizing compiler or sophisticated static analyzer using conventional testing. Formal verification of compilers, static analyzers, and related tools provides a radical, mathematically-grounded answer to

D. Giannakopoulou and G. Salaün (Eds.): SEFM 2014, LNCS 8702, pp. 1–4, 2014.
© Springer International Publishing Switzerland 2014

these risks. By applying deductive program verification to the implementations of those tools, we can prove with mathematical certainty that they are free of miscompilation and unsoundness bugs. For compilers and code generators, the high-level correctness statement is *semantic preservation*: every execution of the generated code matches one of the executions of the source code allowed by the semantics of the source language. For static analyzers and other verification tools, the high-level statement is *soundness*: every execution of the analyzed code belongs to the set of safe executions inferred and verified by the tool. Combining the two statements, we obtain that every execution of the generated code is safe.

In this talk, I give an overview of two tool verification projects I am involved in: CompCert and Verasco. CompCert [8,9] is a realistic, industrially-usable compiler for the C language (a large subset of ISO C 1999), producing assembly code for the ARM, PowerPC, and x86 architectures. It features careful code generation algorithms and a few optimizations, delivering 85% of the performance of GCC at optimization level 1. While some parts of CompCert are not verified yet (e.g. preprocessing), the 18 code generation and optimization passes come with a mechanically-checked proof of semantics preservation. Verasco [2] is an ongoing experiment to develop and prove sound a static analyzer based on abstract interpretation for the CompCert subset of C. It follows a modular architecture inspired by that of Astrée: generic abstract interpreters for the C#minor and RTL intermediate languages of CompCert, parameterized by an abstract domain of execution states, itself built as a combination of several numerical abstract domains such as integer intervals and congruences, floating-point intervals, and integer linear inequalities (convex polyhedra).

Both CompCert and Verasco share a common methodology based on interactive theorem proving in the Coq proof assistant. Both projects use Coq not just for specification and proving, but also as a programming language, to implement all the formally-verified algorithms within Coq's Gallina specification language, in pure functional style. This way, no program logic is required to reason about these implementations: they are already part of Coq's logic. Executability is not lost: Coq's extraction mechanism produces executable OCaml code from those functional specifications.

CompCert and Verasco rely crucially on precise, mechanized operational semantics of the source, intermediate, and target languages involved, from CompCert C to assembly languages. These semantics play a crucial role in the correctness statements and proofs. In a sense, the proofs of CompCert and Verasco reduce the problem of trusting these tools to that of trusting the semantics involved in their correctness statements. An executable version of the CompCert C semantics was built to enable testing of the semantics, in particular random testing using Csmith [13].

Not all parts of CompCert and Verasco need to be proved: only those parts that affect soundness, but not those part that only affect termination, precision of the analysis, or efficiency of the generated code. Leveraging this effect, complex algorithms can often be decomposed into an untrusted implementation followed by a formally-verified validator that checks the computed results for soundness and fails otherwise. For example, CompCert's register allocation pass is composed of an untrusted implementation of the Iterated Register Coalescing algorithm, followed by a validation pass, proved correct in Coq, that infers and checks equalities between program variables and registers and stack locations that were assigned to them [10]. Likewise, Verasco's relational domain for linear inequalities delegates most computations to the Verasco Polyhedral Library, which produces Farkas-style certificates that are checked by Coq-verified validators [4]. Such judicious use of verified validation a posteriori is effective to reduce overall proof effort and enable the use of sophisticated algorithms.

In conclusion, CompCert and especially Verasco are ongoing experiments where much remains to be done, such as aggressive loop optimization in CompCert and scaling to large analyzed programs for Verasco. In parallel, many other verification and code generation tools also deserve formal verification. A notable example is the verified verification condition generator of Herms et al [5]. Nonetheless, the formal verification of code generation and verification tools appears both worthwhile and feasible within the capabilities of today's interactive proof assistants.

Acknowledgments. This work is supported by Agence Nationale de la Recherche, grant ANR-11-INSE-003.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation 2003, pp. 196–207. ACM Press (2003)
2. Blazy, S., Laporte, V., Maroneze, A., Pichardie, D.: Formal verification of a C value analysis based on abstract interpretation. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 324–344. Springer, Heidelberg (2013)
3. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 469–485. Springer, Heidelberg (2001)
4. Fouilhe, A., Monniaux, D., Périn, M.: Efficient generation of correctness certificates for the abstract domain of polyhedra. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 345–365. Springer, Heidelberg (2013)
5. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 2–17. Springer, Heidelberg (2012)
6. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Comm. ACM 53(6), 107–115 (2010)
7. Kornecki, A.J., Zalewski, J.: The qualification of software development tools from the DO-178B certification perspective. CrossTalk 19(4), 19–22 (2006)
8. Leroy, X.: Formal verification of a realistic compiler. Comm. ACM 52(7), 107–115 (2009)

9. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning 43(4), 363–446 (2009)
10. Rideau, S., Leroy, X.: Validating register allocation and spilling. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 224–243. Springer, Heidelberg (2010)
11. Souyris, J., Wiels, V., Delmas, D., Delseny, H.: Formal verification of avionics software products. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 532–546. Springer, Heidelberg (2009)
12. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: Programming Language Design and Implementation 2010, pp. 99–110. ACM Press (2010)
13. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 283–294. ACM Press (2011)