Mechanized Semantics for Compiler Verification

Xavier Leroy

INRIA Paris-Rocquencourt
xavier.leroy@inria.fr

Abstract. The formal verification of compilers and related programming tools depends crucially on the availability of appropriate mechanized semantics for the source, intermediate and target languages. In this invited talk, I review various forms of operational semantics and their mechanization, based on my experience with the formal verification of the CompCert C compiler.

What does this program do, exactly? What is this program transformation or analysis supposed to do, exactly? Formal semantics is the art of providing mathematically-precise answers to these questions. It is a prerequisite to the verification of individual programs, and also to the specification (let alone verification) of programs that operate over other programs, such as static analyzers, program provers, code generators, and optimizing compilers.

Fundamental questions rarely have unique answers. Indeed, a great many different styles of semantics have been explored over the last 50 years, ranging from denotational to axiomatic to operational. In some application areas, *de facto* standards of semantics have emerged, such as labeled transition systems for concurrency, following Milner's seminal work on CCS and the π-calculus [1, 2], and small-step reduction semantics in the type systems community, following Wright and Felleisen's preservation-and-progress pattern for type soundness proofs [3].

The landscape of programming languages research evolves quickly, renewing interest in other forms of semantics. For example, mechanization—formalizing semantics "on machine" with the help of interactive theorem provers, rather than "on paper"—is becoming standard practice in our field. The POPLmark challenge [4] showed that elementary semantic tools such as capture-avoiding substitution can be difficult to mechanize. On the other hand, the power of proof assistants makes it easier to work with semantic styles that are difficult to get right on paper, such as step-indexed logical relations [5] or definitional interpreters [6].

Another evolution worth noting is to formalize "real world" languages, such as C and Javascript, and to prove semantic properties that go beyond type safety, such as semantic preservation for a code generation or optimization algorithm. The reduction-based semantics that work so well to prove type safety for small languages such as IMP, Mini-ML or Featherweight Java can "burst at the seams" when applied to big, messy languages such as C. Likewise, relating the executions of two programs, before and after a code transformation, is fundamentally more difficult than showing the preservation of a typing invariant throughout the execution of a single program.

In this talk, I survey some of these issues based on my personal experience with the formal verification of the CompCert C compiler [7]. As part of this effort, S. Blazy and I had to give mechanized semantics to 14 languages: a very large subset of ANSI C as the source language, assembly for the ARM, PowerPC and x86 machine architectures as target languages, and 10 intermediate languages that bridge the semantic gap between the source and target languages. This semantic engineering is a large part of the CompCert effort, first because these semantics appear prominently in the statement of compiler correctness that we prove, second because we had to change these semantics in essential ways throughout the development of CompCert, in order to prove stronger correctness statements and to accommodate progressively bigger subsets of ANSI C.

The first verifications were conducted against natural (big-step) semantics for the source language and most of the intermediate languages; only the target assembly language was in pure transition (small-step) style [8, 9]. Natural semantics lived up to its name, resulting in relatively straightforward specifications for our languages, and helping us discover the main insights of the semantic preservation proofs. However, we quickly hit limitations of natural semantics, such as its inability to describe nonterminating executions.

The second iteration of CompCert, therefore, uses a combination of small-step transition semantics with explicit call stack for most of the intermediate languages [10], and of *coinductive big-step semantics* for the source language and the first intermediate languages. Coinductive big-step semantics, as introduced by Grall and Leroy [11], enable divergence to be described by coinductive inference rules that follow the structure of executions, like natural semantics does for termination.

We then wanted to account for unstructured control (the `goto` statement) and nondeterministic evaluation order of C, and also to make provisions for a future extension towards shared-memory concurrency—many features where big-step semantics is not appropriate. We therefore switched to small-step, labeled transition semantics for the source and intermediate languages with structured control. We found reduction semantics in the style of MiniML or Featherweight Java inadequate for compiler proofs, but succeeded in using *continuation-based semantics* as introduced by Appel and Blazy [12]. These semantics carefully separate the current sub-command under execution from the execution context in which it appears, with the context being represented "inside-out" as a continuation term. This style of operational semantics is reminiscent not only of the CEK abstract machine [13], but also of polarization and focusing in proof theory and in λ-calculus [14, 15]

CompCert's journey through the landscape of operational semantics has been rather tortuous, but led to the discovery of original forms of operational semantics along the way. Are we at the end of the path? It depends on the language features we would like to model in the future. For instance, giving semantics to program fragments (compilation units) and reasoning about separate compilation and linking probably requires more compositional reasoning principles based on logical relations, in the style of Benton and Hur [16]. In all likelihood, the large-scale formal verification of compilers and static analyzer, as well as other emerging applications of semantics, will keep challenging the state of the art in semantics and exposing the need for new approaches and mechanizations.

References

1. Milner, R.: Communication and Concurrency. Prentice-Hall (1990)
2. Milner, R.: Communicating and Mobile Systems: the pi-Calculus. Cambridge University Press (1999)
3. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (1994) 38–94
4. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: 35th symposium Principles of Programming Languages, ACM Press (2008) 3–15
5. Appel, A.W., McAllester, D.A.: An indexed model of recursive types for foundational proof-carrying code. ACM Transactions on Programming Languages and Systems **23**(5) (2001) 657–683
6. Danielsson, N.A.: Operational semantics using the partiality monad. In: International Conference on Functional Programming 2012, ACM Press (2012) 127–138
7. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009) 107–115
8. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd symposium Principles of Programming Languages, ACM Press (2006) 42–54
9. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: FM 2006: Int. Symp. on Formal Methods. Volume 4085 of Lecture Notes in Computer Science., Springer (2006) 460–475
10. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4) (2009) 363–446
11. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation **207**(2) (2009) 284–304
12. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007. Volume 4732 of Lecture Notes in Computer Science., Springer (2007) 5–21
13. Felleisen, M., Friedman, D.P.: Control operators, the SECD machine and the λ-calculus. In: Formal Description of Programming Concepts III. North-Holland (1986) 131–141
14. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theoretical Computer Science **410**(46) (2009) 4747–4768
15. Curien, P.L., Munch-Maccagnoni, G.: The duality of computation under focus. In: Theoretical Computer Science - 6th IFIP International Conference, TCS 2010. Volume 323 of IFIP Advances in Information and Communication Technology., Springer (2010) 165–181
16. Benton, N., Hur, C.K.: Biorthogonality, step-indexing and compiler correctness. In: International Conference on Functional Programming 2009, ACM Press (2009) 97–108