Experiments in validating formal semantics for C

Sandrine Blazy

Experiments in validating formal semantics for C

Sandrine Blazy

ENSIIE and INRIA Rocquencourt
Sandrine.Blazy@ensiie.fr

Abstract. This paper reports on the design of adequate on-machine formal semantics for a certified C compiler. This compiler is an optimizing compiler, that targets critical embedded software. It is written and formally verified using the Coq proof assistant. The main structure of the compiler is very strongly conditioned by the choice of the languages of the compiler, and also by the kind of semantics of these languages.

1 Introduction

C is still widely used in industry, especially for developing embedded software. The main reason is the control by the C programmer of all the resources that are required for program execution (*e.g.* memory layout, memory allocation) and that affect performance. C programs can therefore be very efficient, but the price to pay is a programming effort. For instance, using C pointer arithmetic may be required in order to compute the address of a memory cell.

However, a consequence of this freedom given to the C programmer is the presence of run-time errors such as buffer overflows and dangling pointers. Such errors may be difficult to detect in programs, because of C type casts and more generally because the C type system is unsafe.

Many static analysis tools attempt to detect such errors in order to ensure safety properties that may be ensured by more recent languages such as Java. For instance, CCured is a program transformation system that adds memory safety guarantees to C programs by verifying statically that memory errors cannot occur and by inserting run-time checks where static verification is insufficient [1]. Another example is Cyclone, a type-safe dialect of C that makes programs invulnerable to errors such as those detected by CCured [2].

More generally, formal program verification ensures that a program does what is stated in its specification, written as assertions using logical formulas. Assertions express expected properties of the program. Formal program verification generates proof obligations that are usually discharged towards external proof assistants or decision procedures (*e.g.* [3, 4]).

Recently, separation logic has been defined as an extension of Hoare logic for reasoning about programs that manipulate pointer structures [5]. Separation logic aims at proving fine-grained properties about pointers and memory footprints, such as non overlapping (*i.e.* separation) between memory regions. Contrary to Hoare logic, separation logic allows local reasoning on the memory footprint of a statement. Such a local reasoning facilitates program proofs.

Some decision procedures have been defined for separation logic, but they are dedicated to toy languages (*e.g.* [6]). Shallow embeddings of separation logic in theorem provers have also been defined (*e.g.* [7–9], where [9] handles a large subset of C).

To sum up, there are several ways to avoid errors in C programs. But, once a property has been formally verified in a C program, how can we ensure that it is also verified by the machine code generated by a C compiler? Bugs in the compiler may invalidate the verification of the source code. It is then necessary to formally verify a property expressing the equivalence between a source code and its corresponding machine code. Several kinds of equivalence may be defined; they may be more or less hard to formally verify. The equivalence may be the transcription in the target language of the property that has already been formally verified on the source program. For instance, the equivalence can express memory safety of the target program. Another way to formally verify that two programs are equivalent consists in verifying that they are computing the same observable results. A program execution is thus abstracted in the computation of input-output and final values of the program.

A way to avoid errors in C programs is to formally verify the C compiler itself. This yields a certified compiler, that is a compiler equipped with a machine-checked proof that the semantics of the source code is preserved along the compilation process [10, 11]. The comparison between this approach and other well-known approaches such as translation validation and proof-carrying code is detailed in [10].

We are currently developing a realistic, certified compiler called CompCert that targets critical embedded software [10, 12, 13]. [10] details the back-end of this compiler, [12] explains its memory model and [13] presents its first front-end. Since then, the front-end has been rebuilt in order to handle a larger subset of C and to facilitate the proof of semantic preservation. The formal semantics have also become more precise. The rest of this paper reports on the design of formal semantics for the CompCert compiler. It discusses the different kinds of semantics that have been studied for the languages of the compiler.

2 Formal semantics for certified compilation

This section describes the CompCert compiler, and details the design and the validation of its formal semantics.

2.1 The CompCert certified compiler

The source language of the CompCert compiler is a large subset of C. Only `goto` and `longjmp` statements are not handled by the CompCert compiler. The target language is the assembly language for the PowerPC architecture. The CompCert compiler is an optimizing compiler. It accomplishes a series of program transformation phases. A program transformation is either a translation to a lower level language or a program optimization. The formal verification of the compiler consists in formally verifying each of its phases. All the formal verifications are developed and machine-checked using the Coq proof assistant. The main optimizations of CompCert are constant propagation, common subexpression elimination and instruction scheduling. Thus, the CompCert compiler generates compact and reasonably efficient target code.

There are 6 intermediate languages in the CompCert compiler. Thus, we have defined a formal semantics for each of the 8 languages of the CompCert compiler. Each formal semantics relies on a memory model that is common to all the languages of the compiler. The CompCert compiler ensures memory safety, mainly concerning reads and writes in memory [12].

The formal verification of the CompCert compiler is the proof of the following semantic preservation theorem (that is written in Coq in Fig. 1): for all source code S, if the compiler transforms S into machine code C without reporting error, and if S has well defined semantics, then C has the same semantics as S, modulo observational equivalence. Thus, S and C are considered as semantically equivalent if there is no observable difference between their executions. Let us note that the successful compilation of a program does not necessarily imply that this program has well defined semantics. An erroneous program is not conform to some formal semantics or it is not transformed successfully during the compilation process, thus simplifying the definition of both the formal semantics and program transformations.

Generally speaking, during the formal verification of a C compiler, only some events are observed. Usually, these are the final results of the C programs. Other events may also be observed, according to the precision of the semantics for C. For instance, the call graph, or the trace of read and write accesses may also be observed, together with the final results. The trust in the certified compiler increases in proportion to the number and variety of observable events. However, there is a tradeoff between the trust gained by a higher amount of observable events, and the admissible optimizations that the compiler may perform.

2.2 On-machine formal semantics

The kind of formal semantics has a strong impact on the semantic preservation properties that can be verified. There are mainly two kinds of formal semantics that are adapted to formal reasoning on program equivalence. These are operational semantics.

Big-step operational semantics (a.k.a. natural semantics) relate formally a program to its final result, and lend themselves well to the proof of compiler-like program transformations. However, these semantics apply only to terminating programs, and do not allow observing intermediate states during program execution. These two features are distinct disadvantages for the intended application domain of the CompCert compiler: embedded software is typically reactive in nature, meaning that programs do not normally terminate and their interactions with the outside world is what matters, not their final result.

Small-step operational semantics based on (finite or infinite) sequences of elementary reductions of the program source allow precise observations of the program execution and also the observation of non-terminating programs. However, such reduction semantics are difficult to exploit when proving the correctness of compiler transformations such as the generation of a control-flow graph from a structured program.

As big-step semantics are simpler than small-step semantics, they are usually preferred in order to define and reason on languages such as C, with non-local constructs (*i.e.* `return`, `continue` and `break` statements) mixed with structured programming (*e.g.* loops). A recurring issue in the formal verification of a C compiler is the development of appropriate operational semantics for the source, intermediate and target languages. Designing adequate on-machine operational semantics for C is not a trivial task. Adequate on-machine semantics are such that their associated induction principles are quite easy to formulate and the corresponding induction steps are provable without too much difficulty.

The validation of the formal semantics is another recurrent issue. The best way to validate the formal semantics for a language such as C is to prove a great deal of properties about the semantics. The formal verification of a C compiler provides an indirect but original way to validate the semantics of the C language. It is relatively straightforward to formalize operational semantics, but much harder to make sure that these semantics are correct and capture the intended meaning of programs. In our experience, proving the correctness of a translation to a lower-level language has detected many small errors in the semantics of the source and target languages, and therefore has generated additional confidence in both.

An interesting result is that the main structure of the CompCert compiler is not conditioned by the program transformations, but very strongly by the choice of the languages of the compiler, and also by the kind of semantics of these languages. Thus, the intermediate languages of the CompCert compiler have been designed in order to facilitate the proofs of translation between languages. When a proof of translation from a language L_1 to a simpler, lower-level language L_2 required to specify different concepts (*e.g.* correspondences between memory states) that made the reasoning more complex, an intermediate language L_i between L_1 and L_2 has been defined. The formal verification of both translations from and to L_i happened to be much easier to achieve than the formal verification of the translation from L_1 to L_2. This is why there are 6 intermediate languages in the CompCert compiler.

Having so many intermediate languages is not common for a compiler. From the programming point of view, it seems to be easier to define as few intermediate languages as possible and to write as few program translations as possible. From the proof point of view, it is easier to define several intermediate languages and elementary translations between slightly different languages.

Turning to the kind of formal semantics we adopted in CompCert, most of the semantics were initially of the big-step operational kind. These big-step semantics capture the final result of program execution, as well as traces of calls to input and output functions. Thus, the formal verification of the CompCert compiler proves that a target program computes the same result as its corresponding source program, and also that the trace of all input-output activities of the program is preserved by all the phases of the compiler. This addition of traces leads to a significantly stronger observational equivalence between source and machine code.

Fig.1 shows this semantics preservation theorem written in Coq. As explained previously in Section 2.1, if a program called `prog` is compiled into a PowerPC program called `tprog` without reporting error (first hypothesis of the theorem), and if the execution (in C) of `prog` terminates and yields a trace and a final integer value (*i.e.* the return value of the `main` function of `prog`), then the execution of `tprog` (in PowerPC assembly) yields the same results (*i.e.* the same trace and integer value).

```
Theorem transf_c_program_correct:
   forall prog tprog trace n,
   transf_c_program prog = Some tprog ->
   Csem.exec_program progr trace (Vint n) ->
   PPCsem.exec_program tprog trace (Vint n).
```

Fig. 1. Main theorem: semantic preservation of the CompCert compiler

We also investigated other forms of on-machine semantics in order to observe non-terminating programs. We have defined several kinds of small-step semantics for some intermediate languages, and proved the equivalence between small-step and big-step semantics for terminating programs, thus giving the opportunity to validate differently the semantics. Small-step semantics have been defined for the languages of the compiler back-end and the semantic preservation proofs have been adapted rather easily.

Another direction that we are currently investigating is coinductive big-step semantics, where evaluation rules still relate programs and program fragments to their final outcome, but some of the rules are interpreted coinductively (infinite derivation trees) instead of inductively (finite derivation trees) as usual. The coinductive interpretation enables these semantics to describe the evaluation of non-terminating programs. In coinductive semantics, inference rules are similar to those of natural semantics, provided that these rules are interpreted coinductively instead of the usual inductive interpretation, or in other terms provided that infinite evaluation derivations are considered in addition to the usual finite evaluation derivations. Such coinductive big-step semantics have been defined for smaller languages than those of the CompCert compiler [14]. These results are encouraging and in the near future, we intend to define coinductive semantics for the languages of the compiler front-end.

Lastly, we also defined a deep embedding of a separation logic for an intermediate language of the CompCert that is close to C [15]. The separation logic consists of an assertion language (with operators that are specific to separation logic) and an axiomatic semantics. We have proved the soundness of this axiomatic semantics with respect to the natural semantics used in the verification of the Compcert compiler, thus giving another opportunity to validate differently the semantics. This experiment is a first bridge between, on the one hand, program proof in the style of Hoare, and on the other hand the Compcert compiler verification effort.

2.3 Quantitative results

Most of the CompCert compiler is written directly in the Coq specification language, as pure functions. Using the extraction facility of Coq (which translates Coq specifications to Caml code), as well as the CIL library that provides an industrial-strength parser and type-checker for the C language [16], and adding a PowerPC printer written directly in Caml, we obtain Caml source code for the whole compiler, making it directly executable. To our knowledge, this is the biggest Caml code that has been automatically extracted from a Coq development. Figure 2 details the architecture if the CompCert compiler. The box in the middle of the figure represents the Coq development and shows the different translations between the languages of the compiler.

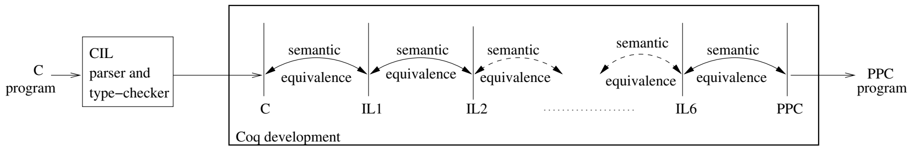

Fig. 2. Architecture of the CompCert certified compiler

Benchmarking on a set of realistic examples of C programs of a few thousand lines shows that the performance of the generated PowerPC code is entirely acceptable: performance is much better than that of the `gcc` compiler at optimization level 0, and only slightly inferior to that of `gcc` at optimization level 1.

The certified compiler represents about 45 000 lines of Coq. The formal semantics represent 8% of this code. The transformations performed by the compiler represent 13% of the code. The rest of the code correspond to the correctness proofs (22% for the lemmas and 50% for the proofs in Coq of these lemmas) and libraries (7% of the code).

3 Conclusion

In conclusion, several solutions exist for producing trusted C software. Formal program verification ensures properties written as assertions in programs. Formal compiler verification (*i.e.* certified compilation) ensures that target code behaves as prescribed by the semantics of the corresponding source code. The design of adequate on-machine semantics is crucial for program verification and certified compilation.

This paper has reported on the design of formal semantics for a certified compiler, the CompCert compiler. These formal semantics have been validated by two kinds of machine-checked proofs: correctness of translations, and semantic equivalence between different kinds of semantics. Designing a certified C optimizing compiler gives a good opportunity to validate its formal semantics for C. The future of C program verification is to connect machine-verified source programs to machine-verified compilers, and run the object code on machine-verified hardware.

References

1. George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.
2. Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In Carla Schlatter Ellis, editor, *USENIX Annual Technical Conference, General Track*, pages 275–288. USENIX, 2002.
3. Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.
4. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 398–414. Springer Verlag, 2005.
5. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.
6. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer and, editor, *Formal Methods for Components and Objects, 4th International Formal Methods for Components and Objects, 4th International Formal Methods for Components and Objects, 4th International*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
7. Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004.

8. Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and He Jifeng, editors, *8th International Conference on Formal Engineering Methods (ICFEM 2006), Macao SAR, China, October 29–November 3, 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer-Verlag, Oct. 2006.
9. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.
10. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 42–54. ACM, 2006.
11. Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *IEEE Conference on Software Engineering and Formal Methods (SEFM'05)*, 2005.
12. Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering, 7th International*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, November 2005.
13. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.
14. Xavier Leroy. Coinductive big-step operational semantics. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2006.
15. Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, Proceedings*, Lecture Notes in Computer Science, September 2007. Accepted for publication.
16. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, 2002.