Which C semantics to embed in the front-end of a formally verified compiler?

Sandrine Blazy

ENSIIE, `Sandrine.Blazy@ensiie.fr`

We have been developing and formally verifying in Coq a moderately optimising compiler (called Compcert) for a large subset of the C language. This compiler comprises a back-end translating the Cminor intermediate language to PowerPC assembly code [3] and a front-end translating the Clight subset of C to Cminor. Clight features all the types and operators of C as well as all the structured control statements of C, but excludes unstructured control.

We have re-architected a previous front-end [1] around the use of the CIL library [2]. CIL provides an industrial-strength parser and type-checker for the C language, as well as a simplifier that eliminates or explicates many features of this language. CIL is written in Caml and is also used in other tools dedicated to the verification of C programs. As CIL performs too many simplifications, we have deactivated those that are not wanted in the context of a verified compiler.

Our formalisation of C in Coq has been extended in two ways. Firstly, the abstract syntax describes a larger subset of C. For example, recursive `struct` and `union` types have been defined using a μ operator, and a limited `switch` statement has been added in the three languages of the front-end. Secondly, the semantics of C is defined coinductively using natural semantics rules for divergence, thus modelling non-terminating programs. The proofs of semantic preservation of the front-end have also been reused and extended in order to handle our new Clight language.

The main difficulty while designing our semantics of Clight was to find the right level of abstraction between on the one hand a precise semantics enabling the proof of correctness properties of non-trivial code transformations as performed by a compiler, and on the other hand a semantics that is less strict than the C standard. For example, thanks to an abstract memory model [4], some popular violations of the C standard are specified in our semantics, but many other violations cannot be accounted for. As a result of this work, the Compcert compiler is now able to compile some realistic examples of C source code.

References

1. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM*, volume 4085 of *LNCS*, pages 460–475, 2006.
2. George C. Necula et al. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.
3. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM Press, 2006.
4. Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. To appear in JAR, 2008.