Type-preserving compilation via dependently typed syntax

Andreas Abel*

Department of Computer Science and Engineering, Gothenburg University

The *CompCert* project [Leroy, 2009] produced a verified compiler for a large fragment of the C programming language. The CompCert compiler is implemented in the type-theoretic proof assistant Coq [INRIA, 2019], and is fully verified: there is a proof that the semantics of the source program matches the semantics of the target program. However, full verification comes with a price: the majority of the formalization is concerned not with the runnable code of the compiler, but with properties of its components and proofs of these properties. If we are *not* willing to pay the price of full verification, can we nevertheless profit from the technology of type-theoretic proof assistants to make our compilers *safer* and *less likely* to contain bugs?

In this talk, I am presenting a compiler for a small fragment of the C language using *dependently-typed syntax* [Benton et al., 2012, Allais et al., 2018]. A typical compiler is proceeding in phases: parsing, type checking, code generation, and finally, object/binary file creation. Parsing and type checking make up the *front end*, which may report syntax and type errors to the user; the other phases constitute the *back end* that should only fail in exceptional cases. After type checking succeeded, we have to deal only with well-typed source programs, whose abstract syntax trees can be captured with the indexed data types of dependently-typed proof assistants and programming languages like Agda [Agda developers, 2019], Coq, Idris [Brady, 2013], Lean [de Moura et al., 2015] etc. More concisely, we shall by *dependently-typed syntax* refer to the technique of capturing well-typedness invariants of syntax trees.

Representing also typed assembly language [Morrisett et al., 1999] via dependently-typed syntax, we can write a type-preserving compiler whose type soundness is given *by construction*. In the talk, the target of compilation is a fragment of the Java Virtual Machine (JVM) enriched by some *administrative instructions* that declare the types of local variables. With JVM being a stack machine, instructions are indexed not only by the types of the local variables, but also by the types of the stack entries before and after the instruction. However for instructions that change the control flow, such as unconditional and conditional jumps, we need an additional structure to ensure type safety. Jumps are safe if the jump target has the same machine typing than the jump source. By *machine typing* we mean the pair of the types of the local variables and the types of the stack entries. Consequently, each label (i. e., jump target) needs to be assigned a machine type and can only be targeted from a program point with the same machine type. Technically, we represent labels as machine-typed de Bruijn indices, and control-flow instructions are indexed by a context of label types. We then distinguish two types of labels:

1. Join points, e. g., labels of statements following an `if-else` statement. Join points can be represented by a `let` binding in the abstract JVM syntax.

2. Looping points, e. g., labels at the beginning of a `while` statement that allow back jumps to iterate the loop. Those are represented by `fix` (recursion).

Using dependently-typed machine syntax, we ensure that *well-typed jumps do not miss*. As a result, we obtain a type-preserving compiler by construction, with a good chance of full correctness, since many compiler faults already break typing invariants. Intrinsic well-typedness also allows us to write the compiler as a total function from well-typed source to typed assembly, and totality can be automatically verified by the Agda type and termination checker.

*Supported by VR grants 621-2014-4864 and 2019-04216 and EU Cost Action CA15123.

References

Agda developers. *Agda 2.6.0 documentation*, 2019. http://agda.readthedocs.io/en/v2.6.0/.

G. Allais, R. Atkey, J. Chapman, C. McBride, and J. McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proceedings of the ACM on Programming Languages*, 2(ICFP):90:1–90:30, 2018. https://doi.org/10.1145/3236785.

N. Benton, C.-K. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012. https://doi.org/10.1007/s10817-011-9219-0.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. http://dx.doi.org/10.1017/S095679681300018X.

L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, vol. 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. https://doi.org/10.1007/978-3-319-21401-6_26.

INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.9 edition, 2019. http://coq.inria.fr/.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. http://doi.acm.org/10.1145/1538788.1538814.

J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999. https://doi.org/10.1145/319301.319345.