Formalizing C in Coq

Robbert Krebbers   Freek Wiedijk
ICIS, Radboud University Nijmegen, The Netherlands
mail@robbertkrebbers.nl   freek@cs.ru.nl

Abstract

The CH_2O project at the RU Nijmegen works towards developing a Coq formalization of a significant fragment of the C programming language as described by the C11 standard. In this project, we have developed a (non-deterministic small step) operational and executable semantics of a typed C core language, a type correct translation of actual C programs into this core language, and extensions of separation logic to reason about subtle features of C.

In recent work (Krebbers & Wiedijk, 2014), we have turned the executable semantics into an interpreter. This interpreter, which is written almost entirely in Coq, can be extracted to OCaml to explore all defined and undefined behaviors of C programs.

In our CoqPL talk, we will describe the internals of our interpreter, a standard library for Coq that we have developed, and Coq features that were essential to the development.

1. Introduction

The C programming language [3] gives close control over the machine, has a high runtime efficiency, and still is very portable, making it a very popular programming language. Of course, when using a low-level language like C, it becomes very easy to make mistakes with potentially disastrous consequences.

An approach to remedy this situation is to use *proofs* to establish the safety of C programs. That way one gets all performance, control and portability benefits, but without the dangers. Such approaches range from static analysis (which is by nature incomplete) to systems where a user can interactively reason about programs.

Still with systems like this, it is often unclear whether it matches the compiler that is used (or might be used in the future), because most of these systems are about a version of C that is quite specific. Moreover, the semantics of C implicit in such a system is generally not made explicit, making it hard to establish that it is error-free.

For this reason, the CH_2O project [4–9] is developing an *explicit* formal semantics that should match the official description of C, the C11 standard [3], as closely as possible. If one proves something about a program with respect to the CH_2O semantics, it will behave that way with *any* C11 compliant compiler. There are two projects that are very close:

- The CompCert project by Leroy *et al.* [11] has a formal semantics for a C-like language, called CompCert C, in Coq. It moreover has an optimizing compiler written in Coq that has been proved correct with respect to this semantics.
- Ellison and Rosu [2] have a formal semantics of C in the $\mathbb{K}$-framework. They also explicitly model the C11 standard.

In both projects, as well as in ours, there is a *formal* description of a significant part of C close to the C11 standard and an *executable* interpreter that matches the semantics precisely. However, both projects also differ from our work:

- **Proof infrastructure.** Unlike Ellison and Rosu, we have a proof infrastructure, and establish metatheoretical properties.
- **Explicit typing.** We have type judgments, and prove properties of our language like type preservation and progress.
- **Formal translation from abstract syntax.** Unlike CompCert, we process the abstract syntax of a C program to an intermediate language *inside* Coq, and prove that this translation always yields a well-typed program.
- **Core language.** Unlike Ellison and Rosu, by first going to a *core* language, the description of some semantic features (like loops) becomes more principled and simpler.
- **Closer to the C11 standard.** CompCert makes choices for implementation defined behavior (*e.g.* integer representations) and gives a semantics to various undefined behaviors (such as aliasing violations). For some programs Ellison and Rosu's semantics is less precise than ours (see [4, 5, 9, 10]), because in our memory model data objects are structured like trees.

2. Exploring the C11 semantics

To be able to test our semantics, we developed an 'interpreter' [10], which does not execute a program according to *one* interpretation of the C standard, but rather calculates *all* behaviors of the program that are allowed. If the given program has undefined behavior, our interpreter will explicitly state this undefinedness.

Our interpreter is also different from compilers or interpreters that insert tests for undefined behaviors as a protection. Those compilers or interpreters generally only follow one possible execution path. Instead, our interpreter is not primarily meant to be a debugging tool, but instead is an *exploration* tool, intended to explore the implications of the C standard [1, 14].

The CH_2O interpreter involves 4 passes to get from C source code to the behavior of the program. These are shown in Figure 1, and involve two languages:

CH_2O core C is quite abstract, and not very close to actual C.

CH_2O abstract C is very close to the abstract syntax trees of C.

Metatheoretical results, like properties of the memory model [4], soundness of separation logic [5, 6, 9], and type preservation and progress [9] have been proved in Coq with respect to CH_2O core C.

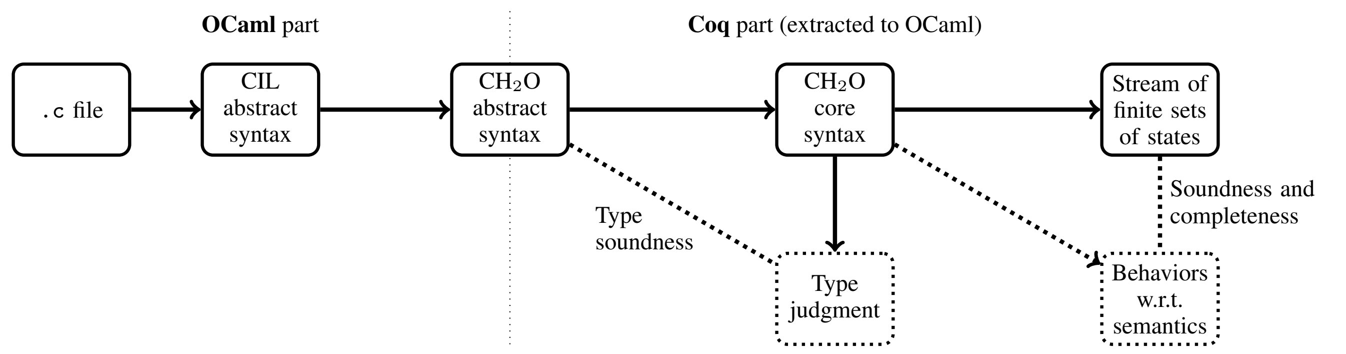
OCaml part
Coq part (extracted to OCaml)
.c file
CIL abstract syntax
CH₂O abstract syntax
CH₂O core syntax
Stream of finite sets of states
Type soundness
Type judgment
Soundness and completeness
Behaviors w.r.t. semantics

Figure 1. Overview of the architecture of the interpreter.

3. Coq formalization

All proofs in the CH_2O project have been fully formalized using Coq and we have used extraction to OCaml to obtain an interpreter to explore the behavior of C programs. This large Coq development of ∼40.000 lines uses many Coq features.

Our Coq development contains a standard library of ∼12.000 lines with results on lists, monads, maps, sets, *etc.* We have used type classes to overload notations, and to provide abstract interfaces for commonly used structures. This allowed us to prove theory and implement automation in a generic way. Our approach is inspired by the *unbundled* approach of Spitters and van der Weegen [13]. However, whereas their work heavily relies on *setoids* (types equipped with an equivalence relation), we tried to avoid setoids, and have used Leibniz equality wherever possible.

The entire development makes heavy use of finite sets and maps. Since we had to execute parts of the development, we have implemented these using radix-2 search trees to obtain logarithmic time operations. These trees are in canonical form to ensure extensional Leibniz equality of finite sets and maps.

We have also used radix-2 search trees to implement hash maps. Hash maps are used to filter out duplicates due to converging non-determinism in the reachable state set of the interpreter. Our hash maps are parametrized by a hash function that is not required to satisfy any properties. To this end, we can use the efficient OCaml standard library function `Hashtbl.hash` when extracting, and a less efficient variant in Coq itself.

The development involves a significant amount of monadic programming using the option, set, and exception monads. For example, the set monad is used to compute all behaviors of a program, and the exception monad is used to propagate error messages in the translation from abstract C. Coq's type class mechanism is used to overload monadic notations such as the *do* notation.

We have used type classes to parametrize the whole development by an abstract interface that describes implementation defined properties (such as the sizes and endianness of integers). Our interpreter can thus be used to compute the behaviors of programs on multiple architectures.

In our development, we have heavily combined interactive proofs with automated proofs. Moreover, we are using tricks like small inversions [12] to handle inversions of large inductively defined relations.

The Coq development is entirely constructive and axiom free. All details about the CH_2O Coq development can be found online at `http://robbertkrebbers.nl/research/ch2o/`.

References

[1] W. Dietz, P. Li, J. Regehr, and V. S. Adve. Understanding integer overflow in C/C++. In *ICSE*, pages 760–770, 2012.

[2] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[3] ISO. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.

[4] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*, 2013.

[5] R. Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *POPL*, pages 101–112, 2014.

[6] R. Krebbers. Separation algebras for C verification in Coq. In *VSTTE*, volume 8471 of *LNCS*, 2014.

[7] R. Krebbers, X. Leroy, and F. Wiedijk. Formal C semantics: CompCert and the C standard. In *ITP*, volume 8558 of *LNAI*, pages 543–548, 2014.

[8] R. Krebbers and F. Wiedijk. A Formalization of the C99 Standard in HOL, Isabelle and Coq. In *CICM*, volume 6824 of *LNAI*, pages 297–299, 2011.

[9] R. Krebbers and F. Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS*, volume 7794 of *LNCS*, pages 257–272, 2013.

[10] R. Krebbers and F. Wiedijk. A typed C11 semantics for interactive theorem proving, 2014. Submitted to CPP, draft available at `http://robbertkrebbers.nl/research/articles/interpreter.pdf`.

[11] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[12] J. Monin and X. Shi. Handcrafted Inversions Made Operational on Operational Semantics. In *ITP*, volume 7998 of *LNCS*, pages 338–353, 2013.

[13] B. Spitters and E. van der Weegen. Type Classes for Mathematics in Type Theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

[14] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.