Towards an Axiomatic Basis for C++

Gregory Malecha, Abhishek Anand, and Gordon Stewart

BedRock Systems, Inc. {gregory,abhishek,gordon}@bedrocksystems.com

Applying formal methods to mainstream languages is crucial to expanding their adoption. To this end, we describe our work on bringing verification to C++. We develop a tool, cpp2v, that converts C++ programs into abstract syntax in Coq, which we then give meaning to by an axiomatic semantics. cpp2v leverages the clang (http://clang.llvm.org) compiler infrastructure to isolate some complexities of C++ (e.g., compile-time features) without enlarging our TCB. Where it's necessary to give semantics to runtime behavior, we use reasoning principles that capture the common uses of features, choosing pragmatically to underspecify patterns (e.g., Duff's device) that are legal but uncommon. We conclude by summarizing the ways we've used the semantics so far and discuss future work.

Our tool and the accompanying Coq development is available online: https://github.com/bedrocksystems/cpp2v.

1 A Syntax

C++ has accrued many features since C. These divide cleanly into two categories: *Compile-time features* such as **template**s, **auto**, and constexpr, which are elaborated at compile time; and *runtime features* such as value categories, inheritence, and concurrency primitives, which enhance the dynamic semantics of the language.

Picking cpp2v's C++ AST to closely resemble clang's C++ IR allows us to offload many of the compile-time features of C++ to clang. For example, clang's AST already includes overload resolution information, makes constructor calls explicit, and handles the complexities of type and value-category inference.

In addition, many features that seem like runtime features are defined in the standard by their desugaring, thus they can be elaborated away by clang. For example, "foreach" loops (C++11) are defined by expanding to a **for**-loop using begin, end, and the pre-increment operator. Also, new features such as **auto** and constexpr are easily desugared into code.

2 An Axiomatic Semantics

When systems are highly nondeterministic, their operational models (e.g. small-step operational semantics) become complex as well. Rather than defining an operational model of C++, we build an axiomatic one. In our experience, the axiomatic approach tends to expose more modular structures of the underlying language than we see in purely operational models. In addition, it maps well to features of C++ such as address resolution that might be awkward to model operationally. Outside of C++, we have seen this approach employed repeatedly for example the VirtIO specification [5] describes the proper way to interact with VirtIO devices in an axiomatic style rather than attempting to describe the system operationally. As in projects like VST [1], we use separation logic, which we define in cpp2v on top of the Iris framework [2].

We build the semantics as a logic (called `mpred`) over Iris's `iProp` and enrich it with ghost state that models the state of the C++ abstract machine. To model the fact that the address of any variable can be taken, we represent program variables uniformly as resources within

```
(* program, invariant mask, thread-local information, activation record *)
Context {σ : genv} {M : coPset} {ti : thread_info} {ρ : region}.
(* expression evaluation *)
Parameters wp_lval wp_prval wp_xval : Expr → (val → FreeTemps → mpred) → mpred.
(* initializing expressions, e.g. constructors *)
Parameter wp_init : type → val → Expr → (FreeTemps → mpred) → mpred.
(* initialization lists, e.g. :field(a,b,c) *)
Parameter wpi : globname → val → Initializer → (mpred → mpred) → mpred.
(* "de-initialization" lists *)
Parameter wpd : globname → val → FieldOrBase∗obj_name → mpred → mpred.
(* statements *)
Parameter wp : Stmt → Kpreds → mpred.
```

Figure 1: The weakest (liberal) precondition definitions axiomatized in our model

the separation logic. This is a departure from VST's *Verifiable C*, which uses CompCert to distinguish address-taken local variables from temporaries.

Our semantics gives weakest (liberal) precondition assertions for the expression evaluation modes of the C++ abstract machine, for statements, and for initialization. The types of these assertions are summarized in Figure 1. Because they are parameters, we provide axioms to reason about them (not shown). For soundness, we underspecify unsupported operations, such as **virtual** function calls, which we define to have a weakest precondition of `False`.

3 Current & Future Work

The current version of `cpp2v` is a first step towards a verification toolchain for C++, but more work remains. To date, we have used `cpp2v` and the accompanying semantics to specify functional correctness of several concurrent libraries and are using it to teach systems programmers to write specifications and to reason about their programs. To that end, we are developing, and iteratively improving, automation and applying it to real-world programs.

Solidifying the Semantics. We aim to develop a semantics that is compatible with the C++ standard, but we are aware of some situations in which our current semantics deviates from the strict behavior of the standard. For example, our simplified memory model is inspired by CompCert's, which doesn't fully model the strict aliasing rules of C and C++. In practice, we avoid this incompatibility by compiling our code with `-fno-strict-aliasing` but a more complete treatment of the C++ memory model in the style of Krebbers [3] would be a more foundational way to inter-operate with modern compilers.

Language Features. Beyond solidifying the semantics, we are also working on extending them with choice features. **virtual** functions [1] is highest on the list because, while infrequently used, there are places where they significantly simplify code. Reasoning about templates pre-instantiation is a longer term goal, but the late-resolution nature of C++ templates makes it difficult to do so in a modular fashion. Concepts, a feature of C++20, *might* be a way to address this, but more research is necessary to understand that. In the meantime, we rely on reusable proof scripts to verify the most basic uses of C++ templates.

[1]Ramananandro [4] contains a Coq formalization of C++'s object system.

References

[1] A. W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[2] R. Jung et al. “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’15*. The 42nd Annual ACM SIGPLAN-SIGACT Symposium. Mumbai, India: ACM Press, 2015, pp. 637–650.

[3] R. Krebbers. “The C standard formalized in Coq”. PhD thesis. Radboud University Nijmegen, Dec. 2015.

[4] T. Ramananandro. “Mechanized Formal Semantics and Verified Compilation for C++ Objects”. An optional note. PhD thesis. Université Paris Diderot (Paris7), July 2012.

[5] *Virtual I/O Device (VIRTIO) Version 1.1*. Standard. OASIS Committee Specification, Dec. 2018.