Derivation of verification rules for C
from operational definitions

Michael Norrish
Computer Laboratory, University of Cambridge

Abstract

While a low-level, operational definition of a language's semantics is a straightforward way of specifying the behaviour of programs written in that language, it is not necessarily very suitable for formal activities such as program verification. This is clearly the case with languages such as C, where the language definition is complicated by much tedious detail. However, the work described here demonstrates that a necessarily complicated semantics for C can still be used as the basis for the generation of “axiomatic” style rules. These can then be used to support verification work in a way that is both familiar and not overly complex.

1 Introduction

The C programming language [ANS89] represents a significant challenge in applying the theory of formal semantics to a “real world” example. Many of the simplifications made in standard pedagogical expositions are overturned in spades when one seeks to define the behaviour of C programs. Four problems are particularly apparent:

- C’s expressions are not the pure idealisations of [Hoa69], where expressions of the programming language are really expressions in the containing logic. In common with other imperative languages, expressions in C can affect the state as they are evaluated. C might be considered particularly offensive in this regard as its expressions can have side-effects even in the absence of function calls.
• C’s definition allows for significant non-determinism. Not only is the order of evaluation of function arguments left unspecified, but the timing of certain forms of side-effect is also unspecified. This is not a non-determinism that a programmer can exploit, but rather one that stems from the definition’s unwillingness to constrain implementations. When formalising C’s semantics it is important to honour this non-determinism, as verification work based on such a formalisation must be able to point out places where a program relies on implementation-dependent details.

• C includes statement forms that cause premature exits from loops and function bodies (the break and return statements). These make reasoning about loops rather harder than when one can be sure that a loop’s guard will be false as it exits.

• C is subject to the problem of aliasing. Free use of pointers and the “address of” operator (&) mean that any formalisation of the language must explicitly acknowledge that variables denote locations in some form of store, not values.

Of course, these problems are typical of many third generation languages, but C retains a particular interest on pragmatic grounds. It is widely used in a great variety of fields, and despite its many flaws, seems likely to be with us for years to come.

Given the problems above, a formal statement of C’s semantics seems easiest in some form of operational framework. Such an approach has the advantage of being relatively easy to generate, and there are many examples in the field (not least the SML definition [MTH90]) from which to draw inspiration. The rest of this article briefly describes the C model I have developed (Cholera) in the HOL [Gor93] theorem prover (section 2), and in section 3 I describe how, despite its complexity, such a model can be used as the basis for the derivation of encouragingly simple inference rules. Finally, I discuss future directions of my work on Cholera.
2 The Cholera model

In essence, the Cholera\textsuperscript{1} model is a straightforward implementation of a standard natural semantics. As claimed above, the construction of the model is not particularly difficult; this section will briefly consider the ways in which the problems with C identified above can be handled by such a semantic framework.

The first problem is that of side effects in expressions. This is easily handled: the state of the model can simply be threaded through expression evaluation as well as through the execution of program statements. This makes the model’s rules slightly bulkier, slightly more awkward to deal with, but is not conceptually difficult.

The second problem is that of C’s significant non-determinisms. Where there are multiple possibilities allowed by a semantic definition, one cannot be sure how a given program phrase is to be treated by the abstract machine. This is simply handled in general: one need only provide multiple rules in the inductive relation for the same syntactic form. So, for example, there are two rules for binary operators, one where the left operand is evaluated first, and another where it is the right operand which is evaluated first. Where there are an arbitrary number of arguments to be evaluated in any order (as with the arguments passed to function calls), Cholera explicitly makes use of the notion of permutation, stating that the arguments are evaluated in the order of some permutation of the order in which they are presented.

A trickier sort of non-determinism remains however, that which specifies the action of side effects. Briefly, the C standard requires that certain expressions cause side effects on the state of the machine, but that these side effects do not necessarily take effect immediately. Instead the standard only requires that they take place before the next sequence point. A sequence point is a well-defined point within an evaluation by which stage all pending side effects will have been dealt with. Nor is there any requirement that the side effects should occur in the order in which they are generated. There is not so much a queue of pending side effects but rather a random grab-bag, from which side effects are taken at unspecified times.

This requirement of the standard does introduce rather more complexity to

\textsuperscript{1}The model is meant to presage an era of proving useful things about C in HOL, hence the name. Or one could take it to reflect the diseased nature of the language...
the model. When evaluating an expression, Cholera’s state includes a list of pending side effects. One further rule is added to the meaning relation which basically states that it is always permitted to non-deterministically select and apply a side effect from the list of those remaining after an expression has been evaluated. Sequence points are modelled by adding to the appropriate rules the requirement that the side effect list be empty when they are reached.

The third problem I identified in the introduction above was that of language primitives (such as break and return) for disrupting the flow of control. Cholera handles these in much the same way as exceptions are handled in the definition of Standard ML ([MTH90]), with a special form of “return value” from a computation. Luckily these special values do not intrude onto the semantics of expressions at all, and can be confined to the realm of the statement. There are two forms of the statement sequencing rule for example. Both “run” the first statement of the sequenced pair as normal. In one the first statement finishes normally, and the computation proceeds with the second. In the other, the first statement gives rise to an exceptional condition, and the computation of the pair stops there.

Finally, C is subject to all of the problems of aliasing which arise as a result of allowing pointers. Cholera’s state can not simply be a mapping of variable names to values, but must rather contain two mappings, one from names to locations, and another from locations to value. Cholera must also honour the C standard’s requirements on the way in which values such as arrays are laid out in memory. In a language such as SML, the definition makes it impossible to state anything about the relation of two values in memory. In C, by way of contrast, it must be the case that values with adjacent indices in an array be adjacent in memory, thereby allowing the pointer arithmetic so beloved of C programmers.

With the possible exception of the treatment of side effects, the preceeding discussion should have made it clear that the development of the Cholera model for C was not particularly conceptually difficult. Of course, there were many fiddly details to get right, but the general strategy followed in the model’s construction was always reasonably clear. The real issue is whether or not such a model (and it is painfully complicated; those “fiddly details” do mount up), can ever be useful.
3 Derivation of rules on top of the model

The aim of my work with Cholera is to produce a tool that allows verification of programs written in C. If this is to be a realistic goal, the prospective user of such a tool must be able to interact with the semantics at a higher level than that of the operational definition. This is evident as soon as one begins to do even relatively simple proofs with the basic definitions. While the model is not conceptually difficult, the complexities of its internal workings can be quite overwhelming.

Inspired by [Gor89], the strategy I have chosen to reduce this burden is to provide a suite of theorems about the model which mimic the rules presented in [Hoa69]. Of course, these rules are not axioms, but are rather consequences of the definition of the model. This is advantageous for two reasons. Firstly, we can be as confident in the “axiomatic” rules as we are in the underlying model. In other words, the rules are guaranteed to be sound with respect to that model. Secondly, we are not confined to the “axioms” alone. Although one would like to be able to present as many rules as possible, and for them to be as useful as possible, this approach always allows recourse to the underlying model. We can always mix “axiomatic” and direct operational styles of reasoning.

Of course, one would not expect to be able to prove exactly the same results about C as those one typically uses with a “toy” or “for pedagogical purposes only” language. Nevertheless, I believe one of the important contributions of this work is the demonstration that one can prove rules about C which are essentially the same as the Floyd-Hoare ones. These rules differ only in that they are accompanied by various side-conditions. For example, the sequencing rule,

\[
\begin{align*}
\{P\} & \quad S_1 \quad \{P'\} \\
\{P'\} & \quad S_2 \quad \{Q\} \\
\{P\} & \quad S_1; S_2 \quad \{Q\}
\end{align*}
\]

gains a side-condition to the effect that the first statement \(S_1\) doesn’t include any return statements. (If it did, there would always be the possibility that the flow of control would never get to \(S_2\).) Clearly, this particular side-condition is too strong (an unreachable return statement leads it astray), but it illustrates an important strategic point: not only do I want Cholera...
to provide theorems that are as familiar as possible, but as far as possible, I want the various side conditions to be syntactic in nature.

The “correct” sequencing rule would include a side-condition to the effect that a return statement is not encountered in the course of executing $S_1$, but this sort of goal is not particularly amenable to automatic proof. On the other hand, the return-free side condition is one that HOL can easily discharge.

Another example of syntactic side-conditions is demonstrated in the treatment of side effects and function calls. It is a simple matter to define a predicate over expressions that is true for just those expressions that are “obviously” side effect free (variable references, arithmetic operators where the operands are also safe in the same sense, numeric constants &c). It is then possible to show that a function whose arguments are all side effect free can have its arguments evaluated in any order and still yield the same result. The resulting rule simplifies verification of programs because one need not consider all $n!$ ways of evaluating $n$ arguments in a function call. Instead just one is sufficient.

This use of syntactic conditions in my derived rules is not only convenient when one comes to prove results about programs, but the conditions also serve to characterise a class of “well-behaved” C programs. These are the programs about which it will be easy to prove verification results. Programs which fall out of the class will prove more difficult to deal with.

Another powerful use of syntax is possible in the analysis of loops. The usual rule that one wants to be able to use states that the negation of the loop’s guard is true when a loop terminates. In C this is not the case because the loop may have terminated because of a return or break statement. This means that a loop’s post-condition must instead be a disjunction: it terminated normally and the negation of the guard is true, or the loop terminated abnormally and some other post-condition holds.

I have written a function within the HOL logic that analyses loop bodies and automatically generates a post-condition corresponding to all of the abnormal ways in which a loop might exit. I have subsequently proven that the condition generated by this function does indeed hold when a loop

```c
while (P) {
    if (ex_cond)
        break;
    ...
}
```

Figure 1: Loop with abnormal exit

Another powerful use of syntax is possible in the analysis of loops. The usual rule that one wants to be able to use states that the negation of the loop’s guard is true when a loop terminates. In C this is not the case because the loop may have terminated because of a return or break statement. This means that a loop’s post-condition must instead be a disjunction: it terminated normally and the negation of the guard is true, or the loop terminated abnormally and some other post-condition holds.
terminates abnormally. The conditions generated are strong enough to be able to state that in the example of figure 1, either $P$ and $\text{ex}_\text{cond}$ will be true, or $P$ will be false.\(^2\)

## 4 Further work

Clearly one of the highest priorities with this work is to demonstrate that **Cholera** and its derived rules are a realistic basis for reasoning about C programs. To this end, I am attempting to verify a BDD algorithm written in C. The program in question performs no I/O *per se*, thereby at least allowing me to avoid having to characterise system and library calls (quite a task in itself, but orthogonal to my present concerns).

It seems likely that the process of this verification will also involve the development of further derived rules. For example I believe that despite all of C's nondeterminism, evaluation of expressions that do not include any sequence points is deterministic.\(^3\) Proof of this fact would greatly simplify much of **Cholera**'s symbolic evaluation because it would not need to consider multiple evaluation orders in many situations, but could instead proceed with whatever evaluation order was most convenient.

No doubt the trials of verification will prompt the proof of further useful rules.

## References


[Gor89] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmaniam, editors,


Much of my work is available in varying degrees of completed-ness on the World Wide Web at http://www.cl.cam.ac.uk/users/mn200/PhD/.