A Separation Logic for Non-determinism and Sequence Points in C Formalized in Coq

Robbert Krebbers

ICIS, Radboud University Nijmegen, The Netherlands

Abstract

The C11 standard of the C programming language does not specify the execution order of expressions. Besides, to make more effective optimizations possible (*e.g.* delaying of side-effects and interleaving), it gives compilers in certain cases the freedom to use even more behaviors than just those of all execution orders.

Widely used C compilers exploit this freedom given by the C standard for optimizations, so it should be taken seriously in formal verification. In [3], we have presented an operational and axiomatic semantics (based on separation logic) for non-determinism and sequence points in C. Soundness of the axiomatic semantics is proved with respect to the operational semantics. This proof has been fully formalized using the Coq proof assistant.

1 Introduction

The C programming language [2] is not only among the most popular programming languages in the world, but it is also among the most dangerous programming languages. Due to weak static typing and the absence of runtime checks, it is extremely easy for C programs to have bugs that make the program crash or behave badly in other ways: NULL-pointers can be dereferenced, arrays can be accessed outside their bounds, memory can be used after it is freed, *etc.*

Instead of forcing compilers to use a predefined execution order for expressions (*e.g.* left to right), the C standard does not specify it. This is a common cause of portability and maintenance problems, as a compiler may use an arbitrary execution order for each expression. Hence, to prove the correctness of a C program with respect to an *arbitrary* compiler, one has to verify that each possible execution order is legal and gives the correct result. To make more effective optimizations possible (*e.g.* delaying of side-effects and interleaving), the C standard requires the programmer to ensure that all execution orders satisfy certain conditions. If these conditions are not met, the program may do anything. Let us take a look at an example where one of those conditions is not met.

```
int main() {
  int x; int y = (x = 3) + (x = 4);
  printf("%d %d\n", x, y);
}
```

By considering all possible execution orders, one would naively expect this program to print `4 7` or `3 7`, depending on whether the assignment `x = 3` or `x = 4` is executed first. However, the *sequence point restriction* does not allow an object to be modified more than once (or being read after being modified) between two *sequence points* [2, 6.5p2]. A sequence point occurs for example at the end `;` of a full expression, before a function call, and after the first operand of the conditional `? :` operator [2, Annex C]. Hence, both execution orders lead to a sequence point violation, and are thus illegal. As a result, the execution of this program exhibits *undefined behavior*, meaning it may do literally anything. Indeed, when compiled with `gcc -O1` (version 4.7.2), it prints `4 8`, which does not correspond to any of the execution orders.

2 Approach

As a step towards taking non-determinism and the sequence point restriction seriously in C verification, we extend our axiomatic semantics with a Hoare judgment $\{P\}\, e\, \{Q\}$ for expressions. Since expressions not only have side-effects but primarily yield a value, the postcondition Q is a function from values to assertions. Intuitively, $\{P\}\, e\, \{Q\}$ means that if P holds beforehand, and execution of e yields a value v, then $Q\, v$ holds afterwards.

Apart from partial program correctness, the judgment $\{P\}\, e\, \{Q\}$ ensures that e exhibits no undefined behavior. To deal with the unrestrained non-determinism in C, we observe that non-determinism in expressions corresponds to a form of concurrency, which separation logic is well capable of dealing with. Inspired by the rule for the parallel composition of separation logic (see [4]), we propose the following kind of rules for each operator $\circledcirc$.

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_l * P_r\}\, e_1 \circledcirc e_2\, \{Q_1 * Q_2\}}$$

The idea is that, if the memory can be split up into two disjoint parts (using the separating conjunction $*$), in which the subexpressions e_1 respectively e_2 can be executed safely, then the full expression $e_1 \circledcirc e_2$ can be executed safely in the whole memory.

To ensure no sequence point violations occur, we use separation logic with permissions [1], extended with a special class of *locked* permissions. The singleton assertion becomes $e_1 \overset{\gamma}{\mapsto} e_2$ where γ is the permission of the object e_2 at address e_1. The inference rules are set up in such a way that reads and writes are only allowed for objects that are not locked, and moreover such that objects become locked after they have been written to. At constructs that contain a sequence point, the inference rules ensure that these locks are released.

3 Formalization in Coq

All our proofs have been fully formalized using the Coq proof assistant. We used Coq's notation mechanism combined with unicode symbols and type classes for overloading to let the Coq code correspond as well as possible to the definitions on paper. Coq's type classes are also used to provide abstract interfaces for commonly used structures like finite sets and finite partial functions, so that we were able to prove theory and implement automation in an abstract way. Because the semantics is rather big, it is quite cumbersome to prove properties about it without automation, to this end, we have automated many steps of the proofs. Our Coq code, available at `http://robbertkrebbers.nl/research/ch2o`, is about 10 000 lines of code including comments and white space. Apart from that, our library on general purpose theory (finite sets, finite functions, lists, *etc.*) is about 9 000 lines.

References

[1] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, 2005.

[2] International Organization for Standardization. *ISO/IEC 9899-2011: Programming languages – C.* ISO Working Group 14, 2012.

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

[4] Peter W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, volume 3170 of *LNCS*, pages 49–67, 2004.