Formal C semantics: CompCert and the C standard

Robbert Krebbers[1], Xavier Leroy[2], and Freek Wiedijk[1]

[1] ICIS, Radboud University Nijmegen, The Netherlands
[2] Inria Paris-Rocquencourt, France

Abstract. We discuss the difference between a formal semantics of the C standard, and a formal semantics of an implementation of C that satisfies the C standard. In this context we extend the CompCert semantics with end-of-array pointers and the possibility to byte-wise copy objects. This is a first and necessary step towards proving that the CompCert semantics refines the formal version of the C standard that is being developed in the Formalin project in Nijmegen.

1 Introduction

The C programming language [2] allows for close control over the machine, while being very portable, and allowing for high runtime efficiency. This made C among the most popular programming languages in the world.

However, C 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.* Furthermore, C programs can be developed with a too specific interpretation of the language in mind, giving portability and maintenance problems later.

An interesting possibility to remedy these issues is to *reason* about C programs. Static program analysis is a huge step in this direction, but is by nature incomplete. The use of interactive theorem provers reduces the problem of incompleteness, but if the verification conditions involved are just generated by a tool (like Jessie/Why3 [9]), the semantics that applies is implicit. Therefore, the semantics cannot be studied on its own, and is very difficult to get correct. For this reason, to obtain the best environment to reason reliably about C programs, one needs a formal semantics in an interactive theorem prover, like the Cholera [10], CompCert [6], or Formalin [3,4] semantics.

The CompCert semantics has the added benefit that it has been used in the correctness proof of the CompCert compiler. Hence, *if* one uses this compiler, one can be sure that the proved properties will hold for the generated assembly code too. However, verification against the CompCert semantics does not give reliable guarantees when the program is compiled using a different compiler.

The C standard gives compilers considerable freedom in what behaviors to give to a program [2, 3.4]. It uses the following notions of under-specification:

- *Unspecified behavior*: two or more behaviors are allowed. For example: order of evaluation in expressions. The choice may vary for each use.
- *Implementation defined behavior*: unspecified behavior, but the compiler has to document its choice. For example: size and endianness of integers.
- *Undefined behavior:* the standard imposes no requirements at all, the program is even allowed to crash. For example: dereferencing a `NULL` or dangling pointer, signed integer overflow, and a sequent point violation (modifying a memory location more than once between two sequence points).

Under-specification is used extensively to make C portable, and to allow compilers to generate fast code. For example, when dereferencing a pointer, no code has to be generated to check whether the pointer is valid or not. If the pointer is invalid (`NULL` or a dangling pointer), the compiled program may do something arbitrary instead of having to exit with a nice error message.

Like any compiler, CompCert has to make choices for implementation defined behavior (*e.g.* integer representations). Moreover, due to its intended use for embedded systems, CompCert gives a semantics to various undefined behaviors (such as aliasing violations) and compiles those in a faithful manner.

To verify properties of programs that are being compiled by CompCert, one can make explicit use of the behaviors that are defined by CompCert but not by the C standard. On the contrary, the Formalin semantics intends to be a formal version of the C standard, and therefore should capture the behavior of *any* C compiler. A blog post by Regehr [11] shows some examples of bizarre behavior by widely used compilers due to undefined behavior. Hence, Formalin has to take *all* under-specification seriously (even if that makes the semantics more complex), whereas CompCert may (and even has to) make specific choices.

For widely used compilers like GCC and Clang, Formalin is of course unable to give any formal guarantees that a correctness proof with respect to its semantics ensures correctness when compiled. After all, these compilers do not have a formal semantics. We can only argue that the Formalin semantics makes more things undefined than the C standard, and assuming these compilers "implement the C standard", correctness morally follows.

As a more formal means of validation of the Formalin semantics we intend to prove that CompCert is a refinement of it. That means, if a behavior is defined by the Formalin semantics, then the possible behaviors of CompCert match those of Formalin. As a first step into that direction, we will discuss two necessary modifications to CompCert as displayed in Figure 1. It is important to notice that this work is not about missing features in CompCert, but about *missing behaviors* of features that are in both CompCert and Formalin.

Note that even the Formalin semantics deviates from the C standard. That is because the C standard has two incompatible ways to describe data [1, Defect Report #260]. It uses a low level description of data in terms of bits and bytes called *object representations*, but also describes data *abstractly* in a way that allows various compiler optimizations. For this reason Formalin errs on the side of caution: it makes some behaviors undefined that most people consider to be defined according to the standard.

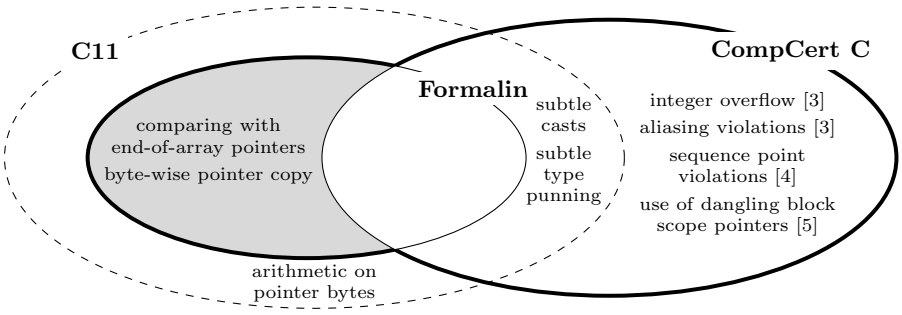

Fig. 1. We extend CompCert C with the behaviors in the shaded area. Each set in this diagram contains the programs that according to the semantics do not have undefined behavior. Since C11 is subject to interpretation, we draw it with a dashed line.

For example, in both Formalin and CompCert, adding 0 to a byte from a pointer object representation is undefined behavior. Both semantics do not just have numeric bytes, but also use symbolic bytes for pointers and uninitialized memory (see the definition `memval` of CompCert in Section 3).

Example. Using CompCert's reference interpreter, we checked that our extensions of CompCert give the correct semantics to:

```
void my_memcpy(void *dest, void *src, int n) {
   unsigned char *p = dest, *q = src, *end = p + n;
   while (p < end) // end may be end-of-array
     *p++ = *q++;
}
int main() {
  struct S { short x; short *r; } s = { 10, &s.x }, s2;
  my_memcpy(&s2, &s, sizeof(struct S));
  return *(s2.r);
}
```

In CompCert 1.12, this program has undefined behavior, for two reasons: the comparison `p < end` that involves an end-of-array pointer, and the byte-wise reads of the pointer `s.r`. Sections 2 and 3 discuss these issues and their resolution.

Sources. Our extension for end-of-array pointers is included in CompCert since version 1.13. The sources for the other extension and the Formalin semantics can be found at `http://github.com/robbertkrebbers`.

2 Pointers in CompCert

CompCert defines its memory as a finite map of blocks, each block consisting of an array of symbolic bytes (and corresponding permissions) [7]. Pointers are pairs (b, i) where b identifies the block, and i the offset into that block.

The C standard's way of dealing with pointer equality is subtle. Consider the following excerpt [2, 6.5.9p5]:

> Two pointers compare equal if and only if [...] or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.

End-of-array pointers are somewhat special, as they cannot be dereferenced, but their use is common programming practice when looping through arrays.

```
void inc_array(int *p, int n) {
  int *end = p + n;
  while (p < end) (*p++)++;
}
```

Unfortunately, end-of-array pointers can also be used in a way such that the behavior is not stable under compilation.

```
int x, y;
if (&x + 1 == &y) printf("x and y are allocated adjacently\n");
```

Here, the `printf` is executed only if `x` and `y` are allocated adjacently, which may happen as many compilers allocate `x` and `y` consecutively on the stack.

In the CompCert semantics, `x` and `y` have disjoint block identifiers, and the representations of `&x + 1` and `&y` are thus unequal. Compilation does not preserve this inequality as the blocks of `x` and `y` are merged during stack allocation. To ensure preservation of comparisons, the semantics of earlier CompCert versions (1.12 and before) required pointers used in comparisons to be *valid*. A pointer is valid if its offset is strictly within the block bounds. We weakened this restriction on pointer comparisons slightly:

- Comparison of pointers in the same block is defined only if both are *weakly valid*. A pointer is weakly valid if it is valid or end-of-array.
- Comparison of pointers with different block identifiers is defined for valid pointers only.

Our weakened restriction allows common programming practice of using end-of-array pointers when looping through arrays possible, but uses as in the second example above remain undefined. We believe that the above restriction on pointer comparisons is more sensible than the naive reading of the C standard because it is stable under compilation[3].

To adapt the compiler correctness proofs we had to show that all compilation passes preserve weak pointer validity and preserve the new definition of pointer comparisons. Furthermore, we had to modify the definition of memory injections [8] to ensure that also the offsets of weakly valid pointers remain representable by machine integers after each program transformation.

[3] Notice that the C standard already makes a distinction between pointers in the same block and pointers in different blocks, for pointer inequalities `<` and `<=` [2, 6.5.9p6].

3 Bytes in CompCert

CompCert represents integer and floating point values by sequences of numeric bytes, but pointer values and uninitialized memory by symbolic bytes.

```
Inductive memval: Type :=
  | Undef: memval
  | Byte: byte -> memval
  | Pointer: block -> int -> nat -> memval
  | PointerPad: memval.
```

When storing a pointer (b, i), the sequence `Pointer` b i 0, ..., `Pointer` b i 3 is stored, and on allocation of new memory a sequence of `Undef` bytes is stored (the constructor `PointerPad` is part of our extension, and is discussed later).

In the version of CompCert that we have extended, it was only possible to read a sequence of `Pointer` bytes as a pointer value. To make byte-wise reading and writing of pointers possible, we extend values with a constructor `Vptrfrag`.

```
Inductive val: Type :=
  | Vundef: val
  | Vint: int -> val
  | Vlong: int64 -> val
  | Vfloat: float -> val
  | Vptr: block -> int -> val
  | Vptrfrag: block -> int -> nat -> val.
```

Extending the functions that encode and decode values as `memval` sequences turned out more subtle than expected. The CompCert compiler back-end must sometimes generate code that stores and later restores the value of an integer register in a stack location. To preserve this value, these memory stores and loads are performed at the widest integer register type, `int`. For pointer fragments, the top 3 bytes of the in-memory representation are statically unknown, since they can result from the sign-extension of the low byte. Therefore, we abstract these top 3 bytes as the new `memval` constructor `PointerPad`.

Arithmetical operations are given undefined behavior on pointer fragments. Reading a pointer byte from memory, adding 0 to it, and writing it back remains undefined behavior. It would be tempting give an ad-hoc semantics to such corner cases, but that will result in a loss of algebraic properties like associativity.

Assignments involve implicit casts, hence `char` to `char` casts need to have defined behavior on pointer fragments to make storing these fragments possible. Since the CompCert compiler needs the guarantee that the result of a cast is well-typed (while the CompCert semantics is untyped), neutral casts perform a zero- or sign-extension instead of being the identity. However, since the top 3 bytes of the in-memory representation of pointer fragments are statically unknown, we changed the semantics of a cast from `char` to `char` to check whether the operand is well-typed (which vacuously holds for well-typed programs). If not, the behavior of the cast is undefined. This has the desired result that `char` to `char` casts can be removed in a later compilation phase.

CompCert 2.2 features a new static analysis that approximates the shapes of values, including points-to information for pointer values. Our `char` values hold more values, and thus this analysis needed some changes. For example, before our extension, the only pointer values that can be read from a given memory location are those that were stored earlier at this exact location using a pointer-wise store. With our extensions, the pointer values thus read can also come from byte-wise pointer fragments that were stored at overlapping locations.

4 Conclusion and future work

The two extensions of CompCert described in this paper succeed in giving a semantics to behaviors that were previously undefined. These extensions are a necessary step for cross validation of the CompCert and Formalin semantics. Our treatment of byte-wise copying of objects containing pointers turned out to be more involved than suggested in [7], owing to the nontrivial semantics of casts and changes to the static value analysis. If future versions of CompCert get a type system, our workaround for casts can be removed.

Another behavior that needs attention in future work is CompCert's call-by-reference passing of struct and union values, as discussed in [4]. In this case, as well as in the byte-wise copying case, the approach followed by Norrish [10] (namely, representing values as sequences of bytes identical to their in-memory representations) may provide an alternative solution, but could cause other difficulties with value analysis and compiler correctness proofs.

Acknowledgments. We thank the reviewers for their helpful comments. This work was partially supported by NWO and by ANR (grant ANR-11-INSE-003).

References

1. International Organization for Standardization: WG14 Defect Report Summary (2008), `http://www.open-std.org/jtc1/sc22/wg14/www/docs/`
2. International Organization for Standardization: ISO/IEC 9899-2011: Programming languages – C. ISO Working Group 14 (2012)
3. Krebbers, R.: Aliasing Restrictions of C11 Formalized in Coq. In: CPP. LNCS, vol. 8307, pp. 50–65 (2013)
4. Krebbers, R.: An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In: POPL. pp. 101–112 (2014)
5. Krebbers, R., Wiedijk, F.: Separation Logic for Non-local Control Flow and Block Scope Variables. In: FoSSaCS. LNCS, vol. 7794, pp. 257–272 (2013)
6. Leroy, X.: Formal verification of a realistic compiler. CACM 52(7), 107–115 (2009)
7. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research report RR-7987, INRIA (2012)
8. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. JAR 41(1), 1–31 (2008)
9. Moy, Y., Marché, C.: The Jessie plugin for Deduction Verification in Frama-C, Tutorial and Reference Manual (2011)
10. Norrish, M.: C formalised in HOL. Ph.D. thesis, University of Cambridge (1998)
11. Regehr, J.: Blog post at `http://blog.regehr.org/archives/759` (2012)