draft version

Subtleties of the ISO C standard

author(s) omitted because of double blind reviewing

Abstract

In our efforts to formalize C11 (the ISO standard of the C programming language) we discovered many unexpected subtleties that make formalization of that standard difficult. Most of these difficulties are the result of the C standard giving compilers room for strong optimizations based on aliasing analysis.

We discuss some of these subtleties and indicate how they may be addressed in a formal C semantics. Furthermore, we discuss why the C standard should address the possibility of stack overflow, and we argue that evaluation of C expressions does not preserve typing in the presence of variable length array types.

Keywords C programming language, system programming languages, formal methods, programming language standardization,

1. Introduction

1.1 Problem

Current programming technology is rather fragile: programs regularly crash, hang, or even allow viruses to have free reign. An important reason is that a lot of programs are developed using low-level programming languages. One of the most extreme instances is the widespread use of the C programming language. In the TIOBE programming language popularity index [27] C is (fall 2012) in the top position.

Whereas most modern programming languages require an exception to be thrown when exceptional behavior occurs (*e.g.* when dereferencing a `NULL` pointer, when accessing an array out of its bounds, or on integer overflow), C [12, 14] does not impose such requirements. Instead, it classifies these behaviors as *undefined* and allows a program to do literally anything in such situations [12: 3.4.3]. On the one hand, this allows a compiler to omit runtime checks and to generate more efficient code, but on the other hand these undefined behaviors often lead to security vulnerabilities [6, 18, 28].

There are two main approaches for improving this situation:

- Switch to a more modern and higher level programming language. This approach reduces the number of programming errors, and if there still is an error, the chance of it being used by an exploit is much lower. One disadvantage of this approach is that there will be a thicker layer between the program and the hardware of the system. This costs performance, both in execution speed and in memory usage, but it also means a reduction in control over the behavior of the system. Especially for embedded systems and operating system kernels this is an undesired consequence.
- Stick to a low-level programming language like C, but add a formal methods layer on top of it to establish that programs do not exhibit undefined behavior. Such a layer might allow the developer to annotate their programs with invariants and to prove that these invariants indeed hold. To be practical most of these invariants should be proven automatically, and the remaining ones by interactive reasoning.

  This approach is an extension of *static analysis*. But whereas static analysis tools often yield false-positives, interactive reasoning allows the developer to prove that a false-positive is not an actual error.

  For functional correctness, this approach has also been successful. For example, there are various projects to prove the C source code of a microkernel operating system correct [4, 15].

There are many tools for the second approach, like VCC [4], Verifast [13] and Frama-C [24]. However, these tools do not use an explicit formal C semantics and only implicitly ‘know’ about the semantics of C. Therefore the connection between the correctness proof and the behavior of the program when compiled with a real world compiler is shallow. The soundness of these tools is thus questionable [9].

For this reason, we recently started a project to provide a formal semantics of the C programming language. This semantics was to be developed for interactive theorem provers, allowing one to base formal proofs on it. Although there already exist various versions of a formal semantics of significant fragments of C (see Section 1.3 for an overview), our goal was to formalize the ‘official’ semantics of C, as written down in the C11 standard. We intended not to gloss over the more difficult aspects of C and to provide a formal semantics of the whole language.

Unfortunately, our formalization efforts have turned out to be much harder than we anticipated because the C11 standard turned out to be much more difficult to formalize than expected. We were aware that C11 includes many features, so that we would need to write a large formalization to include them all. Also, since the standard is written in English, we knew we had to deal with inherent ambiguity and incompleteness. But we had not realized how difficult things were in this respect.

Already, the very basis of our formalization, the memory model, turned out to be almost impossible to bring into line with the standard text. The main problem is that C allows both *high-level* (by means of typed expressions) and *low-level* (by means of bit manipulation) access to the memory. The C99 and C11 standards have introduced various restrictions on the interaction between these two levels to allow compilers to make more effective non-aliasing hypotheses based on typing. As also observed in [11, 23] these restrictions have lead to unclarities and ambiguities in the standard text. We claim that such issues are inherent for programming languages that wish to combine high-level and low-level memory access while still allowing strong optimizations.

1.2 Approach

The aim of this paper is to discuss the situation. We describe various issues by small example programs, and discuss what the C11 standard says about them, and how a formal semantics may handle these.

As part of this work, we have developed an (implicit) prototype of a C11 semantics in the form of a large Haskell program. This program can be seen as a *very* critical C interpreter. If the standard says that a program has undefined behavior, our Haskell interpreter will terminate in a state that indicates this.

The intention of our prototype was to develop a clear semantics of the high-level part. To this end, we postponed low-level details as bytes, object representations, padding and alignment. Due to the absence of low-level details, we were able to support features that are commonly left out, or handled incorrectly, in already existing formal versions of C. In particular, we treat effective types, the common initial segment rule, indeterminate values, pointers to one past the last element, variable length arrays, and `const`-qualified objects. But even without the low-level part, we experienced many other difficulties, that are also described in this paper.

Our prototype is currently being ported to the interactive theorem prover Coq. In the final version of the paper we will give the URL of the source of the Haskell version, but because of double blind reviewing it is omitted here.

While working on a formal version of the C11 standard, we had four rules that guided our thinking:

1. If the standard is absolutely clear about something, our semantics should not deviate from that. That is, if the standard clearly states that certain programs should not exhibit undefined behavior, we are not allowed to take the easy way out and let *our* version of the semantics assign undefined behavior to it.
2. If it is *not* clear how to read the standard, our semantics should err on the side of caution. Generally, this means assigning undefined behavior as we do not want our semantics to allow one to prove that a program has a certain property, when under a different reading of the standard this property might not hold.
3. Any C idiom that is heavily used in practice should not be considered to exhibit undefined behavior, even if the standard is not completely clear about it.
4. If real-world C compilers like GCC and clang in ISO C mode exhibit behavior that is in conflict with a straightforward reading of the standard, but that can be explained by a contrived reading of the standard, our semantics should take the side of the compilers and allow their behavior.

Of course there is a tension between the second and third rule. Furthermore, the fourth rule is a special case of the second, but we included it to stress that compiler behavior can be taken as evidence of where the standard is unclear.

1.3 Related Work

This related work section consists of three parts: discussion of related work on unclarities in the C standard, discussions of related work on undefined behavior, and a brief comparison of other versions of a formal semantics of C.

An important related document is a post by Maclaren [23] on the standard committee's mailing list where he expresses his concerns about the standard's notion of an *object* and *effective type*, and discusses their relation to multiprocessing. Like our paper, he presents various issues by considering example programs. Most importantly, he describes three directions to a more consistent C standard. We will treat those in Section 3.

The standard committee's website contains a list of defect reports. These reports describe issues about the standard, and after discussion by the standard committee, they may lead to a revision or clarification of the official standard text. Defect Report #260 [11] raises similar issues as we do and will be discussed thoroughly throughout this paper.

There is little related work on undefined behavior and its relation to bugs in both programs and compilers. Wang *et al.* [28] classified various kinds of undefined behavior and studied its consequences to real-world systems. They have shown that undefined behavior is a problem in practice and that various popular open-source projects (like the Linux kernel and PostgreSQL) use compiler workarounds for it. However, they do not treat the memory model, and non-aliasing specifically, and also do not consider how to deal with undefined behavior in a formal C semantics.

Yang *et al.* [29] developed a tool to randomly generate C programs to find compiler bugs. This tools has discovered a significant number of previously unknown bugs in state of the art compilers. In order to do this effectively, they had to minimize the number of generated programs that exhibit undefined behavior. However, they do not seem to treat the kinds of undefined behavior that are considered in this paper.

Lastly, we will briefly compare the most significant already existing formal versions of a C semantics (there will be a more extensive discussion of these in Section 3). There are also many others like [5, 19, 26], but these only cover small fragments of C or are not recent enough to include the troublesome features of C99 and C11 that are the topic of this paper.

Norrish defined a semantics of a large part of C89 in the interactive theorem prover HOL [25]. His main focus was to precisely capture the non-determinism in evaluation of expressions and the standard's notion of *sequence points*. However, the problems described in our paper are due to more recent features of the standard than Norrish's work.

Blazy and Leroy [2] defined a semantics of a large part of C in the interactive theorem prover Coq to prove the correctness of the optimizing compiler CompCert. CompCert treats some of the issues we raise in this paper, but as its main application is to compile code for embedded systems, its developers are more interested in giving a semantics to various undefined behaviors (such as wild pointer casts) and to compile those in a faithful manner, than to support C's non-aliasing features to their full extent (private communication with Leroy).

A project which builds on the CompCert semantics [3], is the CerCo project [1]. The goal of that project is to prove a compiler to be complexity preserving.

Ellison and Rosu [7, 8] defined an executable semantics of the C11 standard in the $\mathbb{K}$-framework, and described various ways to deal with undefined behavior in a semantics of a C-like language. Although their semantics is very complete, has been thoroughly tested, and has some interesting applications, it seems infeasible to be used for interactive theorem provers. Besides, it is unclear whether their current memory model conforms to the C standard with respect to the issues we present.

1.4 Scope

It might seem that this paper is only of interest to those who are specifically interested in C. However, the issues that we discuss are not specific to just C. Any programming language that combines the following two design goals will have to deal with the kind of problems discussed in this article:

- It can be used for system programming, *i.e.*, it allows low-level access to the bit level representation of the data that is stored in the memory.
- It has pointers, and allows a compiler to perform aggressive optimizations based on aliasing analysis.

Furthermore, this paper is not just a 'list of bugs in the standard'. Given that the standard text does not use a mathematical formalism, bugs are almost inevitable, but generally not a big problem in practice and easily fixed. We will argue that the issues we present are more fundamental than just 'bugs' and are not easily resolved.

1.5 Contribution

The contribution of this paper is fourfold:

1. We indicate various subtleties of the C11 memory model and type system that we discovered while working on our formal semantics (Section 3, 4, 5 and 7).
2. We argue for some imperfections of C11: absence of (an abstract variant of) stack overflow (Section 6), and lack of preservation of typing (Section 7).
3. We present many small example programs that can be used as a 'benchmark' for comparing different formal versions of a C semantics.
4. We give some considerations on how to best proceed with formalizing the C standard, given that the existing standard text is imprecise and maybe even inconsistent.

2. Undefined behavior

The C standard uses the following notions of under-specification.

- **Unspecified behavior** [12: 3.4.4]. Constructs for which the standard provides two or more possibilities, *e.g.* order of evaluation. The behavior may vary for each use of the construct.
- **Implementation defined behavior** [12: 3.4.1]. Unspecified behavior, but the implementation has to document its choice, *e.g.* size of integer types, endianness.
- **Undefined behavior** [12: 3.4.3]. constructs for which the standard imposes no requirements at all, *e.g.* dereferencing a `NULL`-pointer, integer overflow.

Under-specification is used extensively to make C portable, and to allow compilers to generate fast code. An important feature of undefined behavior is that it is dynamic, rather than static (many undefined behaviors are even impossible to detect statically). When it occurs, a program is allowed to do literally anything, so as to avoid compilers having to insert (possibly expensive) dynamic checks to handle corner cases.

C's excessive use of undefined behavior (it describes more than 200 circumstances in which it occurs [12: J.2]) is quite different from more modern programming languages. There, most incorrect programs are ruled out statically by a strong type system (as in Haskell for example), or a program is required to throw an exception (as in Java for example).

Undefined behavior is important to allow strong optimizations. Based on undefined behavior, a compiler may assume that certain behaviors cannot occur, and optimize accordingly. We give examples of this in this paper.

In a formal semantics, unspecified behavior corresponds to non-determinism, and implementation defined behavior corresponds to parametrization by an environment describing properties of the implementation. Programs exhibiting undefined behavior are incorrect, and a formal semantics should therefore exclude those. The job of a formal semantics is thus not only to describe the meaning of correct programs, but also to exclude execution of incorrect programs.

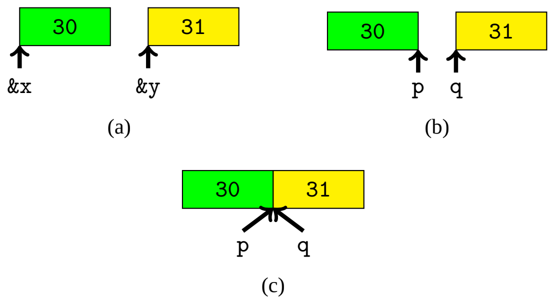


Figure 1: Adjacent blocks.

3. Pointer aliasing versus bit representations

An important feature of C is to allow both *high-level* (by means of typed expressions) and *low-level* (by means of bit manipulation) access to the memory. For low-level access, the standard requires that each value is represented as a sequence of bytes [12: 3.6, 5.2.4.2.1], called the *object representation* [12: 6.2.6.1p4, 6.5.3.4p2].

In order to allow various compiler optimizations (in particular strong non-aliasing analysis), the standard has introduced various restrictions on the interaction between these two levels of access. Let us consider the following program [11]:

```
int x = 30, y = 31;
int *p = &x + 1, *q = &y;
if (memcmp(&p, &q, sizeof(p)) == 0)
  printf("%d\n", *p);
```

Here we declare two objects `x` and `y` of type `int` and use the `&`-operator to take the address of both (Figure 1a). Increasing the pointer `&x` by one moves it `sizeof(int)` bytes ahead and yields a pointer to the right edge of the `x` block. It may seem strange that such pointers are allowed at all [12: 6.5.6p8] because they cannot be dereferenced, but their use is common programming practice when looping through arrays[1].

We store these pointers into objects `p` and `q` of type pointer to `int` (Figure 1b). The next step is to check whether these pointers `p` and `q` are equal (note: not whether the memory they point to is equal; that would be checked by `memcmp(p, q, sizeof(*p)) == 0`). We do this by using the `memcmp` function, which checks whether their object representations are equal. It is important to use bitwise comparison, instead of the ordinary `p == q`, to test whether additional information is stored. If the object representations of the two are equal, we can conclude that both pointers point to the same memory location and do not contain conflicting bounds information. From this we are allowed to conclude that `x` and `y` are allocated adjacently (Figure 1c).

Now we have ended up in a situation where the low- and high-level world are in conflict. On the one hand, `p` is a pointer to the right edge of the `x` block, and thus dereferencing it should lead to undefined behavior. On the other hand, the object representation of `p` is the same as the object representation of `q`, and so `p` and `q` should behave identically.

Although the standard itself is very unclear about these problems, in Defect Report #260 [11] the committee expressed the following judgment:

[1] Note that C allows one to use objects that do not have array type as an array of length one [12: 6.5.6p7].

> Implementations are permitted to track the origins of a bit-pattern and treat those representing an indeterminate value as distinct from those representing a determined value. They may also treat pointers based on different origins as distinct even though they are bitwise identical.

Apparently a value can contain additional information about its *origin* that is not reflected in its object representation. The reason the committee allows this, is to allow compiler optimizations that would not be correct otherwise.

To show that this is a real issue, we changed the last example slightly and compiled it with GCC:[2]

```
int x = 30, y = 31;
int *p = &x + 1, *q = &y;
if (memcmp(&p, &q, sizeof(p)) == 0) {
  *p = 10;
  printf("%d %d\n", *p, *q);
}
```

This does not give any compiler warnings and executing the program prints two distinct values '`10 31`'. Despite the fact that `p` and `q` are identical on the bit level (which follows from the fact that the `printf` is executed at all), they still behave differently on the object level, as indeed Defect Report #260 allows for.

3.1 More extreme aliasing analysis

Given that Defect Report #260 allows a compiler to take the origin of a pointer value into account, a natural question is whether that also holds for non-pointer values. Particularly, does this hold for the integer type `intptr_t`? This integer type has the property that any pointer of type `(void *)` can be converted to it, and when such an integer value is converted back to a pointer of type `(void *)`, it will compare equal to the original pointer [12: 7.20.1.4]. Consider the following program (from the GCC bug tracker[3]):

```
int x = 30, y = 31;
int *p = &x + 1, *q = &y;
intptr_t i = (intptr_t)p, j = (intptr_t)q;
printf("%ld %ld %d\n", i, j, i == j);
```

When compiled with GCC, it outputs:

```
140734814994316 140734814994316 0
```

This means that although the integer variables `i` and `j` have the same numerical value `140734814994316`, they still compare as different because the origins of their values differ.

In the reactions to the bug report from which this example was taken, this was unequivocally considered to be a bug. In other words, although some compiler writers think the license for optimizations that Defect Report #260 gives them is justified, this example was too extreme for the GCC community to be taken in that light.

Regardless of whether optimizations as in the above example are justified, it it not clear where the border of what is allowed should be. Specifically, is `p == q` allowed to evaluate to `0` in case the storage of `p` and `q` are allocated adjacently? And what about the bitwise comparison `memcmp(&p, &q, sizeof(p))`, could that also take 'origin' into account?

To use the semantics to certify an optimizing compiler, it makes sense to let these 'dangerous' pointer comparisons exhibit undefined behavior. For example, a first pass in the CompCert compiler, is to remove non-determinism as this significantly eases correctness proofs [20]. However, this means that the phase that removes non-determinism has to fix whether such a comparison is true or not. This is problematic, as future phases may change the memory layout, and therefore may change the truth of such comparisons.

3.2 Three directions for improvement

Maclaren describes similar unclarities of the standard with respect to the size of array objects corresponding to pointer values [23]. He presents three directions the standard may take:

1. **The original Kernighan & Ritchie approach.** Pointer values are simply addresses, and their object representations are all that matters.
2. **Pointers carry their origin.** Each pointer value carries its *origin*: a history on how it is constructed. Certain operations (*e.g.* derefering a pointer) are disallowed if the origin conflicts.
3. **Only visible information is relevant.** This is a weaker variant of the previous one where the history is limited to a certain visibility, for example the current scope.

The first approach is taken by the semantics of Norrish [25] in HOL. Although this approach is clearly understood, and is convenient to formalize, it implies that pointers with equal object representation always have to be treated as equal. Therefore, optimizations as for example performed by GCC and allowed by Defect Report #260 are not sound with respect to Norrish's semantics.

The second approach allows the most compiler optimizations and is therefore the most appealing one. However, it is unclear how bit-level operations and library functions like `memcpy` should deal with a pointer's origin. An earlier version of the CompCert memory model by Leroy and Blazy [22], models bytes as abstract entities, instead of simple sequences of bits (even on the level of the assembly), to deal with this issue. This approach ensures that the abstract information can easily be preserved, even by byte-level operations. The drawback is of course that many bit-level operations become undefined. Hence, in a more recent version of their memory model [21], only bytes that constitute a pointer are abstract entities, whereas those that constitute integers or floating point numbers are sequences of bits.

The semantics by Ellison and Rosu [8] in the $\mathbb{K}$-framework use a similar representation. Symbolic execution is used to model bytes of pointers as abstract entities.

The third approach requires a careful definition of 'visibility'. It is unclear whether such a definition can be given that is both consistent and that allows sufficient compiler optimizations [23]. This approach therefore seems not very attractive.

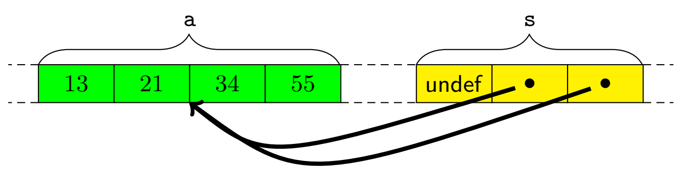
The memory model of our prototype semantics takes the second approach to its fullest extent. Our memory is a finite map from indexes to trees that expose the full structure of values, and pointers are paths through these trees. Subobjects (subarrays, fields of structures or unions) correspond to subtrees of the memory.

The origin of pointers in our semantics is much more detailed than its counterpart in CompCert [21] and in the semantics of Ellison and Rosu [8]. There, only a pointer's block and offset into that block is stored, whereas we store the entire path corresponding to the history of the construction of the pointer. In particular, since we do not flatten arrays and structures, we are able to distinguish subobjects. For example, it allows our semantics to be aware that something special is happening in the following example:

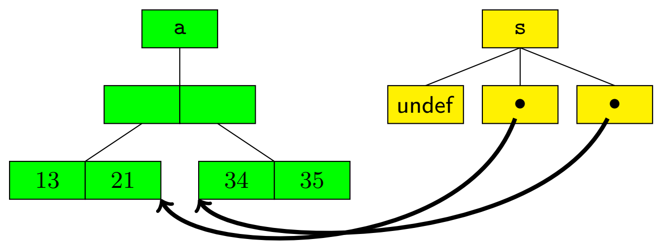
```
int a[2][2] = { {13, 21}, {34, 35} };
struct t { int *r, *p, *q; } s;
s.p = &a[0][2]; s.q = &a[1][0];
if (s.p == s.q)
  printf("%d\n", *s.p);
```

[2] Using `gcc -O2 -std=c99 -pedantic -Wall`, version 4.7.1. These compiler flags are used for all future examples.

[3] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=54945`



(a) flattened



(b) abstract

Figure 2: Example contents of the memory.

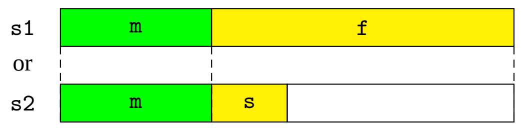


Figure 3: A union containing two structures with a common initial sequence.

Figure 2 displays the representation of the memory after executing the first three lines of this code in both a concrete memory and our abstract memory as trees.

Our semantics imposes special treatment on pointers that point to elements one past the end of an object. First of all, we do not allow these pointers to be dereferenced. Secondly, when such a pointer is used in a comparison with a pointer to another subobject, we can easily impose undefined or non-deterministic behavior. For the moment (although this probably does not follow to the C standard), we use undefined behavior to be as cautious as possible with respect to questionable situations as described in Section 3.1.

Of course, the drawback of our current memory model is that bytes are not present at all, whereas the CompCert memory model at least allows byte-level operations on objects solely consisting of integers and floating point numbers. But since the intention of our semantics was to obtain a better understanding of the high-level part of the C memory, for the moment we postponed accounting for object representations in our model.

4. The common initial sequence

C supports various data types to build more complex types: in particular, structure, union, and array types. Structures are like product types and unions are like sum types. Due to the low-level nature of C, unions are *untagged* instead of *tagged*, which means that the current variant of the union is not stored. Consider:

```
union int_or_float { int x; float y; };
```

Given an object of type `int_or_float`, it is not possible to test whether it contains the `int` or the `float` variant. This may seem unnatural, but it is in the spirit of C to let the programmer decide whether or not to store the tag.

An interesting consequence of untagged unions is the standard's *common initial sequence* rule [12: 6.5.2.3]. Consider:

```
struct t1 { int m; float f; };
struct t2 { int m; short s; };
union { struct t1 s1; struct t2 s2; } u;
```

The representation of the object `u` might look as pictured in Figure 3. Although that there may be additional space between the members (due to alignment), the standard guarantees that the integer parts always coincide. Even stronger, in this case it guarantees the following [12: 6.5.2.3p6]:

> ... it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is visible.

For example, that means we are allowed to do things like:

```
int main(void) {
  u.s2.m = 20;
  printf("%d\n", u.s1.m);
}
```

So, we set the integer part via the one variant of the union, and read it out via the other. However, the following program exhibits undefined behavior as the relation to the union type is not visible in the body of the function `f`.

```
int f(struct t1 *p1, struct t2 *p2) {
  p2->m = 20;
  return p1->m;
}
int main(void) {
  printf("%d\n", f(&u.s1, &u.s2));
}
```

This restriction allows compilers to make stronger aliasing assumptions about `p1` and `p2` because their types are different. Real compilers, like GCC, happily use this, and indeed this example can be adapted for GCC such that something different from the naively expected '20' is printed.

The standard's definition of 'visible' is rather unclear, especially when a pointer to a common initial segment is passed through another function. For example, consider:

```
int *f(int *p) { return p; }
int main(void) {
  u.s2.m = 20;
  printf("%d\n", *f(&u.s1.m));
}
```

Does passing the pointer through `f` remove the visibility of the common initial segment?

The GCC manual only allows the common initial segment rule to be used when an object is accessed directly via the union type, and therefore gives no guarantees about the previous program, and even not about simpler cases like the following program:

```
int main(void) {
  u.s2.m = 20;
  int *p = &u.s1.m;
  printf("%d\n", *p);
}
```

Our semantics takes the same position as the GCC manual. We have implemented this by annotating each structure fragment in a path of a pointer with a flag whether the common initial segment rule may be used. When a pointer is stored in the memory, this flag is cleared. As a result, undefined behavior is imposed on the last two examples.

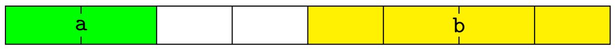


Figure 4: `struct T { short a; int b; }` on a standard 32 bits architecture.

5. Indeterminate values

Uninitialized variables and padding bytes of objects of structure type take an *indeterminate value* [12: 6.2.6.1p6]. An indeterminate value is an object that either describes an unspecified value or is a *trap representation* [12: 3.17.2]. A trap representation is an object representation that does not represent a value of the object type and reading it leads to undefined behavior [12: 6.2.6.1p5].

5.1 Uninitialized variables

Since an object of type `unsigned char` cannot have a trap value, reading it does not exhibit undefined behavior. Instead it just gives an unspecified value. This property is important to allow simple minded bit-wise copying of structures, without having to worry about padding bytes between members. For example:

```
struct T { short a; int b; } x = {10, 11}, y;
for (size_t i = 0; i < sizeof(x); i++)
  ((unsigned char*)&y)[i] =
  ((unsigned char*)&x)[i];
```

Figure 4 displays the representation of `x` on a standard 32 bits architecture with a 4 bytes alignment requirement for integers. This means that integers should be aligned at addresses that are multiples of 4, and therefore we have 2 padding bytes between the members. In case reading indeterminate values of type `unsigned char` (and in particular these padding bytes) would exhibit undefined behavior, this copy would also exhibit undefined behavior.

An interesting property of indeterminate values is that Defect Report #260 [11] allows them to change arbitrarily, so reading an indeterminate value twice might yield different results. This is useful for an optimizing compiler because it may figure out the actual lifetime of two values to be disjoint and therefore share the storage location of both. As an example (the mysterious `&x` will be explained later),

```
unsigned char x; &x;
printf("%d\n", x);
printf("%d\n", x);
```

does *not* exhibit undefined behavior (because an object of type `unsigned char` cannot contain a trap value), but Defect Report #260 allows the two printed values to be different. This is not so strange: the compiler might do liveness analysis and decide that `x` does not need to be saved on the stack when calling `printf`. And then of course `printf` might clobber the register that contains `x`.

Unfortunately, the standard is very unclear about the imposed behavior of various operations on indeterminate values, *e.g.*, what happens when they are copied or used in expressions. For example, should `y` be indeterminate after evaluating

```
unsigned char x, y; &x;
y = x/2;
```

Surely the most significant bit of `y` will be 0 after this? Or is that not something that the standard guarantees? But if `y` is *not* indeterminate after this, what about:

```
y = x/1;
```

We just changed the constant, and therefore after this statement `y` also should be determinate? But after:

```
y = x;
```

will it still not be indeterminate? This seems almost indistinguishable from the `x/1` case, but the liveness analysis argument surely now also will apply to `y`? A formal C semantics will have to take a stance on this.

Also, should the following print '0', or may it print a different value as well, because the `x` may have changed during the evaluation of the subtraction expression?

```
unsigned char x; &x;
printf("%d\n", x - x);
```

Defect Report #338 [11] remarks that on some architectures (*e.g.* IA-64) registers may hold trap values that do not exist in memory. Thus, for such architectures, programs cannot safely copy uninitialized variables of type `unsigned char` because these might reside in registers. In the C11 standard this problem has been fixed by including the following workaround [12: 6.3.2.1p2]:

> If the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

This is the reason we had to take the address of the uninitialized variables in the preceding examples. Of course even with the `&x` present, a compiler can decide to ignore it and still use a register for `x`, as this address is never used.

This workaround again shows that treating uninitialized objects as indeterminate has its difficulties. In the memory model of our semantics we keep track of uninitialized (or freed) memory by special `undef` nodes (see Figure 2). Since objects of structure or union type cannot be indeterminate, we only allow these `undef` nodes on the leaves. Again, we take the most cautious way, and let operations on these special `undef` nodes, like reading them, exhibit undefined behavior.

5.2 Pointers to freed memory

The standard states that the value of a pointer variable becomes indeterminate when the object it points to reaches the end of its lifetime [12: 6.2.4]. In particular this means that whenever some memory is freed, all pointers to it become indeterminate. For example, assuming the `malloc`s do not fail, the following program can still exhibit undefined behavior

```
int *p = malloc(sizeof(int));
free(p);
int *q = malloc(sizeof(int));
printf("%d\n", p == q);
```

because the value of the pointer `p` has become indeterminate and now can be a trap value. Of course, we can still compare the bit patterns of both pointers, and if they are equal, try to use `p` instead.

```
int *p = malloc(sizeof(int));
free(p);
int *q = malloc(sizeof(int));
if (memcmp(&p, &q, sizeof(p)) == 0)
  *p = 10;
```

Again, Defect Report #260 [11] states that this program exhibits undefined behavior.

The fact that a pointer object becomes indeterminate after the block it points to has been freed means that if we copy pointers to various places in memory, then all copies should become indeterminate and not just the argument of `free` (which does not even need to be an lvalue). This means that a `free` operation will affect the formal memory state globally. And what about individual bytes of a pointer that have been copied? Will these also become indeterminate after freeing that pointer?

5.3 Padding bytes

The standard states that when a value is stored in a member of an object of structure or union type, padding bytes take an unspecified value [12: 6.2.6.1p6], and that when a value is stored in a member of a union type, bytes that do not correspond to that member take an unspecified value [12: 6.2.6.1p7]. Consider:[4]

```
union { int a[1]; int b[2]; } x;
x.b[0] = 10; x.b[1] = 11;
x.a[0] = 12;
printf("%d\n", x.b[1]);
```

This example can lead to undefined behavior, as assigning to `x.a` makes the bytes of `x.b` that do not belong to `x.a` unspecified, and thus `x.b[1]` possibly indeterminate. Taking Defect Report #260 into account, it is unclear whether the bytes that belong to `x.b[1]` after this may change arbitrarily.

This program also suggests that the representation of pointers in a formal semantics should contain information describing which parts of the memory should become indeterminate upon assignment to them. If instead of assigning to `x.a[0]` directly we do:

```
int *p = &x.a[0];
*p = 12;
```

`x.b[1]` will *still* become indeterminate. But the assignment to `p` might happen anywhere in the program, even in a context where the union type is not visible at all.

6. Stack overflow

A programming language implementation typically organizes its storage into two parts: the *stack* and the *heap*. On a function call, the stack is extended with a frame containing the function's arguments, local variables and return address (the address of the instruction to be executed when the function is finished). The heap contains dynamically allocated storage.

The standard abstracts from implementation details like these, and thus also allows implementations that do not organize their memory in this way. We agree that this is the right approach, but we do believe it should at the least account for (an abstract version of) the problem of *stack overflow*. Unfortunately, the notion of stack overflow is not mentioned by the standard [12] or the standard's rationale [10] at all. This is very troublesome, as for most actual implementations stack overflow is a real problem. Let us consider the following function.

```
/*@ decreases n; ensures \result == n; */
unsigned long f(unsigned long n) {
  if (n != 0) return f(n - 1) + 1;
  else return 0;
}
```

With most tools for C verification one can prove that the function `f` behaves as the identity function (for example, the Jessie plug-in for Frama-C [24] allowed us to do so). However, in a real C implementation, a call like `f(10000000)` will *not* return `10000000`, but will crash with a message like '`segmentation fault`'.

Stack overflow does not necessarily have to result in a crash with an error message, but might also overwrite non-stack parts of the memory (possibly putting the address of virus code there). Also, it can occur without function calls. For example, the program

```
int main(void) { int a[10000000]; }
```

might also crash[5].

This all means that a proof of correctness of a program with respect to the standard only guarantees correctness relative to the assumption that the stack does not overflow. As we have seen, this assumption does not hold in general. But worse, it is not even clear for which n the program

```
int main(void) { int a[n]; }
```

is guaranteed not to overflow the stack. On a microcontroller this might already happen for rather small n. All this means that a correctness proof of a program with respect to the C standard actually does not guarantee anything.

The obvious way to change the text of the C standard to address this issue would be to add to [12: 6.5.2.2] something like:

> A function call might overflow the stack, in which case the behavior is undefined. It is implementation defined under what circumstances this is guaranteed not to happen.

It is important that it is not just *unspecified* when these overflows happen, for then it still would not be possible to formally guarantee anything about a program from the text of the standard.

A consequence of this addition would be that *strictly conforming programs* no longer exist, as one of their defining properties is that they 'shall not produce output dependent on ... implementation defined behavior' [12: 4.5]. Of course, once one takes stack overflow into account no program has that property anymore. Another consequence is that then *all* C implementations become *conforming*, as that notion is defined by quantification over all strictly conforming programs [12: 4.6]. Therefore, the definition of conforming C implementations should also be adapted.

Section 5.2.4.1 of the standard states that each implementation 'shall be able to translate and execute at least one program that [satisfies a list of properties].' This seems to imply that for each implementation at least *one* program should exist that does *not* crash with stack overflow. However, this does not imply that there exists a program that *all* implementations will execute correctly (the program might be different for different implementations), and hence it still does not imply the existence of any strictly conforming program. Also, the section really seems to be about lower bounds for limits of the compilation process: limits on the execution of the program are not really taken into account.

The fact that the standard does not allow to portably test for stack overflow is to us one of the main omissions of the language. A call to `malloc` will return a `NULL` pointer when there is no storage available, which means that a program can test for that situation. But there is no counterpart to this for function calls, or for entering a block. Variable length arrays, discussed in Section 7, make this an even bigger omission.

Nearly all existing formal C semantics seem to ignore stack overflow entirely. For example CompCert [20] models an infinite memory and stack, even on the level of the assembly. Although this is undesirable, it is unclear how to deal with stack overflow

[4] Adapted from Shao Miller, Bounds Checking as Undefined Behaviour, `comp.std.c` newsgroup, July 29, 2010.

[5] In order to make this really happen with an optimizing compiler, one might need to use the array `a` to prevent it from being optimized away. For example when compiled with GCC or clang (with `-O2`) the following crashes when running it with the standard stack size:

```
int main(void) {
  int a[10000000];
  for (int i = 0; i < 10000000; i++) a[i] = i;
  return a[10];
}
```

in a formal semantics such that it still can be used for compiler correctness proofs. The following approaches are clearly not fully satisfactory:

- Change the compiler correctness proof to guarantee that a program is either compiled to a program that exhibits the correct behavior, or to a program that runs out of memory/stack. In this case compilation to a program that always runs out of memory would be correct.
- Change the compiler correctness proof to guarantee that a program is compiled correctly only if the source program have some upper bound on stack and memory usage. This approach is non-trivial because a compiler does not necessarily preserve the stack memory consumption of the program it compiles. For example, inlining of functions and spilling of variables may increase the stack usage arbitrarily [20].

Verification with stack overflow taken into account is an instance of verification of the resource consumption of a program. The CerCo project [1] aims at implementing a formally verified complexity preserving C compiler, and therefore falls in this class as well.

7. Failure of the subject reduction property

A desired property for a typed programming language is *subject reduction*, which means that evaluation preserves typing. As proven by Norrish, this property holds for (his small-step expression semantics of) C89 [25]. We will argue that due to the introduction of variable length arrays in C99, this property no longer holds. Before pursuing the problem of subject reduction, we briefly introduce variable length arrays and some other problems of these.

Before C99, arrays were required to have a size that could be determined at compile-time. To work arround this restriction, programmers had to use dynamically allocated memory (through `malloc` and `free`) for arrays of dynamic size. To loosen this restriction, C99 introduced support for *variable length arrays* (VLAs) to the language.

The first shortcoming of VLAs is related to the fact that there is no portable way to test if there is sufficient storage available (on the stack) when entering a block (see Section 6). Since most implementations use the stack to store VLAs, not being able to perform such a test, makes VLAs dangerous in practice. Consider the following program.

```
int main(void) {
  int n;
  scanf("%d", &n);
  int a[n];
}
```

Since there is no way to test if there is enough space on the stack, it is impossible to portably ensure that the program does not crash.

Another problem of VLAs is that C allows casts to variable length types. Since size expressions in such casts may impose side-effects, this makes the situation rather complex. For example, consider the following program:[6]

```
1 ? (int(*)[f(5)])0 : (int(*)[f(3)])0
```

where `f` is a function returning a positive `int`. We were unable to find anything in the standard on which of the function calls `f(5)` and `f(3)` may (or should) be evaluated. It is reasonable to allow implementations to evaluate neither of them, as programs can generally be executed correctly without needing to know the size of array bounds in pointer casts. But what about:

[6] The type `int(*)[f(5)]` should be read as ‘pointer to integer array of length `f(5)`’.

```
printf("%d\n",
  sizeof(*(1 ? 0 : (int(*)[f(3)])0)));
```

Here an implementation clearly needs to evaluate the function call, to obtain the value of the `sizeof`, even though it is in the branch of the conditional that is not taken. The standard includes the following related clause [12: 6.7.6.2p5]:

> Where a size expression is part of the operand of a `sizeof` operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated.

Consider the fact that the size expression in the ‘not taken’ branch of this example *has* to be evaluated, one may wonder if the expression in the ‘not taken’ branch of our earlier example is also allowed to be evaluated.

It also is unclear how these function calls in casts are evaluated with respect to sequence points. As they are evaluated in branches that are not taken, it seems they are exempt from the normal execution flow of an expression. But if they already can be executed before the sequence point that starts the subexpression that contains them, is there a reason they cannot be executed before the evaluation of the statement that contains them starts?

The standard’s (implicit) distinction between *static* and *run-time* typing is the reason that subject reduction breaks. This means that an expression can get a more restricted type during its evaluation. For example, statically the expression `(int(*)[f(5)])0` has type ‘pointer to integer array of variable length’, whereas at run-time it will become of the more restricted type ‘pointer to integer array of length n’ where n is the result of evaluating `f(5)`.

The previous example already indicates that one has to sacrifice either subject reduction or uniqueness of types (that is, each expression has a unique type). However, we will argue that the situation is worse, and that for a reduction semantics whose rules are applied locally, subject reduction will fail even if expressions are allowed to have multiple types. Consider:

```
1 ? (int(*)[f(5)])0 : (int(*)[3])0
```

In this example the two subexpressions `(int(*)[f(5)])0` and `(int(*)[3])0` have (static) types `int(*)[*]` and `int(*)[3]`, respectively, where T`[*]` denotes the variable length array type over T. By typing of the conditional and composite types [12: 6.5.15p6, 6.2.7] the full expression has type `int(*)[3]`.

If the function call `f(5)` gets evaluated, and returns a value different from 3, typing breaks, *i.e.*, a well typed expression is evaluated to a non-well typed expression. Luckily, the standard imposes undefined behavior on this example by means of the following clause [12: 6.7.6.2p5]:

> If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values.

Currently, our C semantics deals with this problem by allowing evaluation of size expressions at any place in a bigger expression. Only after the conditional gets reduced, we check if both arguments have compatible types, and if not, assign undefined behavior. This approach has two obvious drawbacks. First of all, it breaks subject reduction, and secondly, undefined behavior gets caught at a late moment, or might not get caught at all. For example, in

```
g() ? (int(*)[f(5)])0 : (int(*)[3])0
```

it may happen that `f(5)` is evaluated first, and returns a value unequal to 3, in which case typing has already failed, but our semantics has not yet established this. After that, it may happen that the call to `g()` does not return (for example because it invokes an `exit` or loops indefinitely), resulting in the undefined behavior not getting described by our semantics.

But does this mean that in a formal C semantics each reduction step has to establish that typing does not break in order to catch undefined behavior? This would fix some version of subject reduction, but destroys the locality of a small-step reduction semantics.

8. Conclusion

8.1 Discussion

We will finish our discussion of the subtleties of the C standard by asking the following questions:

1. Is the interpretation of the C standard that we presented in this paper a reasonable reading of the standard?
2. Is the C standard itself (under this interpretation) reasonable?

First of all, we claim that the standard is not fully clear. The standard committee's response in Defect Report #260 [11] is not obvious from the text of the standard. Also, Maclaren [23] presents various issues about which the standard does not have an unambiguous reading, even when taking Defect Report #260 into account.

However, the relation between Defect Report #260 and the official standard text is not even completely clear. The report *does* include a response by the standard committee and actual compilers are making use of it, and as such it should not be taken lightly. It was a clarification of the C99 version of the standard, and hence it seems obvious that there was some defect in that text. However, the parts of the text of the standard in the C11 version relevant for the issues discussed in the report have not changed from their counterpart in the C99 standard at all.

The standard makes a very clear distinction between 'normative' and 'informative' text in the standard, and the explanations of Defect Report #260 certainly are not part of the 'normative' text of the C11 standard. Therefore, it seems an option to decide to ignore the committee's response in this report, especially the notion of the *origin* of an object, which does not occur in the standard at all. In that case, one could read the standard in a 'Kernighan & Ritchie' manner. But of course, in that case various optimizations performed by actual compilers *will* be incorrect, and one will get standard compliant behavior only when compiling using flags like `-fno-strict-aliasing`. Many real world programs, like for example the Linux kernel, are compiled with such flags anyway [28], but on the other hand this attitude would mean that for compilers to be standard compliant, they would have to compile to executables with less than optimal performance, because they would need to generate more code to test for changes due to aliasing.

The second question, whether the standard can and should be improved is even more interesting. It seems very difficult (or maybe even impossible) to give a mathematically precise version of the memory model from the standard, which is a sign that one should aim for a clearer explanation of the issues involved. However, it is already not clear at all what a mathematically precise clarification of the standard should look like.

For example, in the memory model of CompCert, bytes (objects of type `unsigned char`) that constitute a pointer are *not* simple sequences of bits, but instead are abstract entities [21]. This means that treating them like numbers – calculating with them, printing them – will exhibit undefined behavior. This seems not in the spirit of the language. On the other hand, for actual practical programs, this restriction on the programs that are considered meaningful probably is not important at all.

8.2 Future work

We consider two directions for future work. On the one hand we are porting the prototype semantics that we have developed in Haskell to the theorem prover Coq. For the moment, this development mainly ignores the issues described in this paper and aims at an elegant semantics for non-local control flow, block scoped variables, expressions with side effects, and sequence points. It includes both an operational semantics and an axiomatic semantics, and proves that the axiomatic semantics is correct with respect to the operational semantics.

On the other hand we are experimenting with a formal memory model in which the issues from this paper are taken into account. We are investigating two approaches. The first is to extend the memory representation that we described in this paper (where objects are represented as trees). Instead of having values at the leaves, we use sequences of bytes at the leaves. These bytes are similar as those in CompCert [21]: they are either a concrete sequence of bits, or an abstract entity constituting a fragment of a pointer. This approach combines the best of two worlds: it still allows many byte-wise operations while incorporating the aliasing restrictions. Also, in this approach, the memory remains executable.

However, this approach still does not allow pointer bytes to be used as integers. Therefore, we are also experimenting with a memory whose state consists of a pair of an *abstract* version of the memory (based on our tree representation), and a *concrete* version of the memory (which consists of sequences of bytes encoding the data). The puzzle is how to deal with information flow between these two components. We have not been able to resolve these issues in a satisfactory way yet.

If we succeed in creating such a 'Defect Report #260-compliant' formal memory model, we will incorporate it into the operational semantics that we already are working on. We then also will be able to take a guaranteed consistent position on the issues discussed in this paper.

Acknowledgments

Acknowledgments omitted because of double blind reviewing.

References

[1] Andrea Asperi et al. Certified Complexity (CerCo). `http://cerco.cs.unibo.nl`, 2012.

[2] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[3] Brian Campbell and Randy Pollack. Executable Formal Semantics of C. Technical Report D3.1, Project FP7-ICT-2009-C-243881 CerCo, 2010.

[4] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42, 2009.

[5] Jeffrey Cook and Sakthi Subramanian. A Formal Semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, 1994.

[6] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DISCEX*, 2000.

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

[8] Chucky Ellison and Grigore Rosu. Defining the Undefinedness of C. Technical report, University of Illinois, 2012.

[9] Chucky Ellison and Grigore Rosu. Slides of [7], 2012. `http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl-slides.pdf`.

[10] International Organization for Standardization. Rationale for International Standard – Programming Languages – C, 2003. Revision 5.10.

[11] International Organization for Standardization. WG14 Defect Report Summary for ISO/IEC 9899:1999, 2008. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/`.

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

[13] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2008.

[14] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[15] Gerwin Klein et al. seL4: formal verification of an OS kernel. In *SOSP*, pages 207–220, 2009.

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

[17] Robbert Krebbers and Freek Wiedijk. Separation Logic for Non-local Control Flow, 2012. Submitted.

[18] Patrice Lacroix and Jules Desharnais. Buffer Overflow Vulnerabilities in C and C++. Rapport de Recherche DIUL-RR-0803, Université Laval, Québec, Canada, 2008.

[19] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, 2008.

[20] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[21] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, 2012.

[22] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[23] Nick Maclaren. What is an Object in C Terms?, 2001. Mailing list message, `http://www.open-std.org/jtc1/sc22/wg14/9350`.

[24] Yannick Moy and Claude Marché. *Jessie Plugin Tutorial*, Beryllium *version*. INRIA, 2009.

[25] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[26] Nikolaos Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.

[27] TIOBE Software. Programming Community Index. `http://www.tiobe.com/content/paperinfo/tpci/`, November 2012.

[28] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined Behavior: What Happened to My Code? In *APSys*, 2012.

[29] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.