Defining the Undefinedness of C *

Chris Hathhorn
University of Missouri, USA
crhr38@missouri.edu

Chucky Ellison Grigore Roșu
University of Illinois at Urbana-Champaign, USA
{celliso2, grosu}@illinois.edu

Abstract

We present a "negative" semantics of the C11 language—a semantics that does not just give meaning to correct programs, but also rejects undefined programs. We investigate undefined behavior in C and discuss the techniques and special considerations needed for formally specifying it. We have used these techniques to modify and extend a semantics of C into one that captures undefined behavior. The amount of semantic infrastructure and effort required to achieve this was unexpectedly high, in the end nearly doubling the size of the original semantics. From our semantics, we have automatically extracted an undefinedness checker, which we evaluate against other popular analysis tools, using our own test suite in addition to a third-party test suite. Our checker is capable of detecting examples of all 77 categories of core language undefinedness appearing in the C11 standard, more than any other tool we considered. Based on this evaluation, we argue that our work is the most comprehensive and complete semantic treatment of undefined behavior in C, and thus of the C language itself.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords Undefined behavior, Programming language semantics, C11, K Framework

1. Introduction

A programming language specification or semantics has dual duty: to describe the behavior of correct programs and to identify incorrect programs. Many formal semantics of various parts of C (e.g., Norrish [24], Papaspyrou [25], Blazy and Leroy [1], Ellison and Roșu [6]) have tended to focus on the meaning of correct programs. But the "positive" semantics of C is only half the story. If we make the assumption that programs are well-defined, then we can ignore large swathes of the C11 standard. Most qualifiers can be ignored, for example—`const` and `restrict` are described by the standard entirely in terms of the undefinedness caused by their misuse. In well-defined programs, types are all compatible, pointer arithmetic valid, signed integers never overflow, and every function call is compatible with the type of a matching declaration in some translation unit.

* Supported in part by the U.S. Department of Education under GAANN grant number P200A100053.

PLDI'15, , June 13–17, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Our work is an extension to Ellison and Roșu [6]. In that paper, the authors focused on giving semantics to correct programs and showed how their formal definition could yield a number of tools for exploring program evaluation. But the evaluation they performed was against defined programs, and the completeness they claimed was for defined programs. In this work, in contrast, we focus on identifying undefined programs. We cover the issues we faced in moving a complete but overspecified semantics (in the sense that it gave meaning to undefined programs) to one capable of catching the core-language undefined behaviors.

To the best of our knowledge, ours is the most comprehensive semantic treatment of undefinedness in C. Undefined behavior (UB) is often considered of secondary importance in semantics, perhaps because of a misconception that capturing it might come "for free" simply by not defining certain cases. We argue that this understanding of UB as simply instances of underspecification by the standard misleads about the effort required to capture it. Undefinedness permeates every aspect of the C11 language. The standard even makes constructive use of it, describing several language features (e.g., type qualifiers), sometimes with quite complicated semantics, in terms of the behaviors they make undefined.

Our contributions include the following:

- a study of undefinedness in C, including a discussion of the various semantic techniques and considerations needed to capture all core language undefinedness,
- a first semantic treatment for some of the less-studied features of C11, such as the `restrict` qualifier and a linker phase,
- a semantics-based undefinedness checker, and
- a benchmark test suite for undefined behavior.

The tool, semantics, and test suite can be found at `https://github.com/kframework/c-semantics`.

2. Undefined Behavior

According to the C standard, undefined behavior is "behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements" [8, §3.4.3:1]. It goes on to say:

> Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). [8, §3.4.3:2]

Particular implementations of C may guarantee particular semantics for otherwise undefined behaviors, but these are then extensions of the actual C language.

The C11 standard contains a list of the 203 places it invokes undefined behavior [8, §Appendix J]. These can be divided into three broad sets:

Our classification	No.	CERT undef. behavior ids.[a]
Core language	77	#4, 8–26, 32, 33, 35–89.
compile time[b]	24	
link time[c]	8	
run time	45	
Early translation[d]	24	#2, 3, 6, 7, 27–31, 34, 90–99,
lexical	11	101, 102, 104, 107.
macros	13	
Library	101	#5, 100, 103, 105, 106,
compile time	18	108–203.
run time	83	
Other	1	#1.
Total	203	

[a] The numbers we use to identify a particular category of undefined behavior are the same as in the CERT C Secure Coding Standard [29] and correspond to the order in which clauses invoking undefinedness appear in the C11 standard (as well as the order they appear in Appendix J).

[b] Completely detectable by translation phase 7.

[c] Completely detectable by translation phase 8.

[d] Completely detectable by translation phases 1–6.

Figure 1. Breakdown of undefined behavior in C11.

- 77 involve core language features.
- 24 involve the parsing and preprocessing phases (i.e., translation phases 1–6), which are not treated by our semantics.
- 101 involve the standard library.

Plus one additional item: "A 'shall' or 'shall not' requirement that appears outside of a constraint is violated" [8, UB #1],[1] which is the justification for including many of the other items on the list. Also, each item might represent more than one kind of undefined behavior and we tend to refer to each item as a "category" because of this. For example, the item covering division by zero contains two distinct instances of undefinedness: "The value of the second operand of the `/` or `%` operator is zero" [8, UB #45].

Our classification appears in Figure 1. We categorize each behavior by the point in the translation or execution process at which it can be recognized. "Compile time" behaviors, for example, are those that can always be detected using only per-translation unit static analysis, while "link time" behaviors require whole-program static analysis to catch every instance.

Technically, even a behavior that we categorize as detectable at compile or link time might only render the individual execution that encounters it undefined, and not the whole program. But we believe that strictly-conforming[2] programs should be free of such behaviors, and that analysis tools should report them, even if they might be unreachable during execution. For example, this program has unreachable but statically-detectable undefinedness caused by attempting to use the value of a void expression [8, UB #23]:

```
int main() { if (0) (void)1 + 1; }
```

All attempts to use the value of a `void` expression can be detected statically, so we categorize this behavior as "compile time" despite the fact that only the actual execution that encounters the behavior becomes undefined.

[1] Whenever we quote the standard regarding an undefined behavior, we quote Appendix J. Technically, this appendix is non-normative and the actual wording in the normative body of the standard might differ significantly, but the appendix tends to provide a more clear and concise description.

[2] A program which does not "produce output dependent on any unspecified, undefined, or implementation-defined behavior" [8, §4:5].

In the rest of this section, we discuss the uses and implications of undefinedness in C11. Both Regehr [26] and Lattner [14] provide good introductions to this topic and their work inspired much of the discussion that follows.

Undefinedness Enables Optimizations An implementation of C does not need to handle UB by adding complex static checks that may slow down compilation or dynamic checks that might slow down execution. According to the language design principles, a C program should be "[made] fast, even if it is not guaranteed to be portable," and implementations should always "trust the programmer" [7].

Undefinedness can sometimes enable powerful optimizations. One example is the alias analysis enabled by the `restrict` qualifier (see Section 3.2) and the strict aliasing rules (see Section 3.4), which allow compilers to infer information about how an object can be modified, potentially reducing memory accesses. Lattner [14] gives another canonical example:

```
for (int i = 0; i <= N; ++i) { ... }
```

Because of the undefined behavior caused by signed overflow [8, UB #36], a compiler can often assume that such a loop will iterate exactly `N + 1` times. If `i` were instead declared as `unsigned int`, making overflow defined, the compiler would now need to consider the possibility that the loop will never terminate (e.g., were `N` to be `UINT_MAX`).

Undefinedness Causes Lots of Problems Programmers often expect a compiler, in the presence of undefinedness, to generate an executable that behaves in some reasonable way. In fact, compilers do many unexpected things when translating programs that invoke undefinedness. For example:

```
int main() { *NULL; return 0; }
```

This will *not* cause GCC, Clang, nor ICC[3] to generate a program that raises an error when run. Instead, these compilers simply ignore the dereference.

Implementations are free to assume that undefined behavior will not occur. This assumption leads to many strange consequences. An example from Nagel [20]:

```
int x = f(); if (x + 1 < x) { ... }
```

Programmers might think to use a construct like this in order to handle possible overflow. But according to the standard, `x + 1` can never be less than `x` unless undefined behavior occurred. Therefore, a compiler is justified in removing the branch entirely, and GCC, Clang, and ICC all do this. Despite these compilers only supporting two's complement arithmetic, where `INT_MAX + 1 == INT_MIN`, they will take advantage of the undefinedness when optimizing.

Furthermore, undefinedness potentially invalidates the entire execution on which it occurs, affecting behavior that "came before" the undefinedness, due to compiler optimizations [6, §5.1.2.3]. The compiler is not required to prove an expression will not invoke undefinedness before reordering it in some way that would be semantics-preserving assuming well-definedness. For example:

```
int f(int x) {
      int r = 0;
      for (int i = 0; i < 5; i++) {
            printf("%d\n", i);
            r += 5 / x;
      }
      return r;
}
```

Despite the potential division by zero occurring after the `printf` lexically, it is not correct to assume that this function will "at least" print 0 to the screen. In practice, an optimizing compiler will notice that the expression `5 / x` is invariant in the loop and

[3] GCC v 4.8.2, Clang v 3.4, and ICC v 14.0 everywhere mentioned.

may choose to move it to before the loop. Clang and ICC both do this at optimization levels above 0. The `printf`, in this example, might never be reached if undefined behavior occurs in the statement that lexically follows it.

Undefinedness can also depend on implementation details. The standard defines two additional kinds of behavior:

implementation-defined Unspecified behavior where each implementation documents how the choice is made.

unspecified behavior Use of an unspecified value, or other behavior [with] two or more possibilities and [...] no further requirements on which is chosen in any instance. [8, §3.4]

Whether a program encounters undefinedness can depend on the choices made by an implementation regarding behaviors from these two categories. The standard, therefore, is often not enough to diagnose undefinedness—the manual for a particular implementation can also be required.

While implementation-defined behavior must be documented, unspecified behavior has no such requirement [8, §3.19.1]. An implementation is allowed to choose different semantics for different occurrences of the same unspecified behavior, and may even change between them at run time.

An example of unspecified behavior is evaluation order. Because the evaluation order of many expressions is unspecified in C, an implementation may take advantage of undefined behavior found on only some of these orderings. For example, any implementation is allowed to "miscompile" the program

```
int d = 5;
int setDenom(int x) { return d = x; }
int main() { return (10/d) + setDenom(0); }
```

because there is an evaluation strategy that would set `d` to `0` before doing the division. While GCC generates an executable containing no run time error, the CompCert compiler [15] generates code that exhibits a division by zero.

The security implications of UB are perhaps the most dire of all. The ability to do arbitrary computation often lives in the undefinedness caused by buffer overflows. The standard places no limitations on undefined behavior—all undefinedness, therefore, is a potential security hole.

Undefinedness is Hard to Detect Detecting undefined behavior is undecidable in general:

```
int main() { guard(); 5 << -1; }
```

Whether this program invokes undefined behavior[4] depends on whether `guard()` terminates. But proving `guard()` terminates, even with run time information (i.e., knowing the state of the machine when `guard()` is called), is undecidable.

Although it is impossible (in general) to prove that a program is free of undefinedness, this raises the question of whether one can *monitor* an execution for undefined behaviors. As we saw above, a smart compiler may detect undefinedness statically and generate code that does not contain the same behaviors. A monitor, therefore, might not detect all undefined behavior if the analysis were based only on the output from such a compiler, even though the original program contained it. If we instead assume we will monitor the code as run on an abstract machine, we can give more concrete answers.

First, it is both decidable and feasible to monitor an execution and detect any undefined behavior, as long as the program is deterministic. By deterministic, we mean there is only a single path of execution (or all alternatives join back to the main path after a bounded number of steps). It is feasible because one could simply check the list of undefined behaviors against all alternatives before executing any step. Because all decisions would be joinable, only a fixed amount of computation would be needed to check each step.

For non-deterministic single-threaded programs, one may need to keep arbitrary amounts of information, making monitoring for undefined behavior decidable but intractable. Consider this program:

```
int r = 0;
int flip() { /* non-determ. return 0 or 1 */ }
int main() { while(1) { r = (r << 1) + flip(); } }
```

At iteration n of the loop above, `r` can be any one of 2^n values.[5] Because undefinedness can depend on the particular value of a variable, all these possible states would need to be stored and checked at each step of computation by a monitor.

If multiple threads are introduced, then even monitoring for undefined behavior becomes undecidable. The reason is similar to the original argument—because there are no fairness restrictions on thread scheduling, at any point, the scheduler can decide to let a long-running thread continue running:

```
// Thread 1.                    // Thread 2.
while (guard()); x = 0;         5 / x;
```

In this example, if one could show that the loop must eventually terminate, then running thread 1 to completion followed by thread 2 would exhibit undefined behavior. But showing that the loop terminates is undecidable.

The tool we extract from our semantics combines static analysis with a monitor, like the one described above. By default, it explores a single evaluation strategy when encountering non-determinism due to unspecified evaluation order in expressions, but it can also search the other possible strategies for undefinedness. We discuss our approach below.

3. A Semantics for Catching Undefinedness

We developed our semantics in the rewriting-based $\mathbb{K}$ semantic framework[6] [27], inspired by rewriting logic (RL) [17]. RL organizes term rewriting *modulo equations* (namely associativity, commutativity, and identity) as a logic with a complete proof system. The central idea behind using such a formalism for the semantics of languages is that the evolution of a program can be clearly described using rewrite rules. A rewriting semantics in $\mathbb{K}$ consists of a syntax (or signature) for describing terms and a set of rewrite rules that describe steps of computation. Given some term allowed by a signature (e.g., a program together with input), deduction consists of the application of the rules to that term, yielding a transition system for any program.

In $\mathbb{K}$, parts of the state are represented as labeled, nested multisets. For example, here is an excerpt from the configuration used by our semantics:

$$\left\langle \begin{array}{c} \langle K \rangle_{\mathsf{k}} \ \left\langle \ \langle Map \rangle_{\mathsf{genv}} \ \langle Map \rangle_{\mathsf{gtypes}} \ \right\rangle_{\mathsf{tu}} \ \langle Set \rangle_{\mathsf{locs\text{-}written}} \\ \langle Map \rangle_{\mathsf{mem}} \ \langle Map \rangle_{\mathsf{env}} \ \langle Map \rangle_{\mathsf{types}} \ \langle List \rangle_{\mathsf{call\text{-}stack}} \end{array} \right\rangle_{\mathsf{T}}$$

These collections contain pieces of the program state like a computation stack or continuation (e.g., k), environments (e.g., env, types), stacks (e.g., call-stack), etc. The configuration shown above is a subset of the configuration from our C semantics, which contains around 100 such cells in the execution semantics and another 60 in the translation semantics. As this is all best understood through an example, let us consider a typical rule for a simple imperative language for dereferencing a pointer (see Section 3.1 for the dereferencing rules of C):

$$\langle \underline{* \ X} \ \cdots \rangle_{\mathsf{k}} \ \langle \cdots X \mapsto L \cdots \rangle_{\mathsf{env}} \ \langle \cdots L \mapsto V \cdots \rangle_{\mathsf{mem}}$$
$$\overline{\qquad\qquad V \qquad\qquad}$$

[4] "An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression" [8, UB #51].

[5] That is, if we pretend `int` has an arbitrarily large width.

[6] `http://kframework.org`

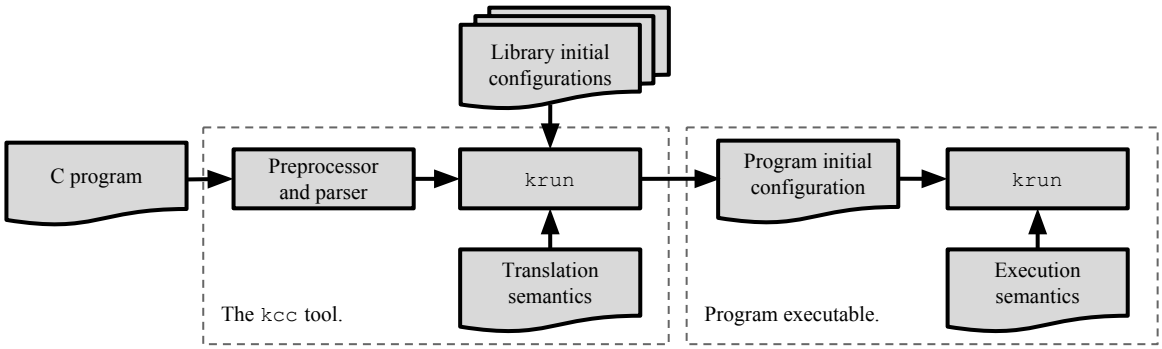

Figure 2. An overview of our tool. `krun` is a $\mathbb{K}$ Framework utility for extracting an interpreter from a semantics.

We see here three cells: k, env, and mem. The k cell represents a stack of computations waiting to be performed. The left-most (i.e., top) element of the stack is the next item to be computed. The env cell is simply a map of variables to their locations and the mem cell is a map of locations to their values. The rule above, therefore, says that if the next thing to be evaluated (which here we call a redex) is the application of the dereferencing operator (*) to a variable X, then one should match X in the environment to find its location L in memory, then match L in memory to find the associated value V. With this information, one should transform the redex into V.

This example exhibits a number of features of $\mathbb{K}$. First, rules only need to mention those cells relevant to the rule. The cell context can be inferred, making the rules robust under most extensions to the language. Second, to omit a part of a cell we write "⋯". In the above k cell, we are only interested in the current redex $*X$, but not the rest of the context. Finally, we draw a line underneath parts of the state that we wish to change—above, we only want to evaluate part of the computation, but neither the context nor the environment change.

This unconventional notation is quite useful. The above rule, written out as a traditional rewrite rule, would be:

$$\langle *\ X \curvearrowright \kappa \rangle_{\mathsf{k}}\ \langle \rho_1, X \mapsto L, \rho_2 \rangle_{\mathsf{env}}\ \langle \sigma_1, L \mapsto V, \sigma_2 \rangle_{\mathsf{mem}} \Rightarrow \langle V \curvearrowright \kappa \rangle_{\mathsf{k}}\ \langle \rho_1, X \mapsto L, \rho_2 \rangle_{\mathsf{env}}\ \langle \sigma_1, L \mapsto V, \sigma_2 \rangle_{\mathsf{mem}}$$

Items in the k cell are separated with "$\curvearrowright$," which can now be seen. The κ and $\rho_1, \rho_2, \sigma_1, \sigma_2$ take the place of the "⋯" above. Nearly the entire rule is duplicated on the right-hand side. Duplication in a definition requires that changes be made in concert, in multiple places. If this duplication is not kept in sync, it leads to subtle semantic errors. In a complex language like C, the configuration structure is much more complicated, and would require actually including additional cells like genv and call-stack. These intervening cells are automatically inferred in $\mathbb{K}$, which keeps rules modular.

Our semantics extends the work of Ellison and Roșu [6], which gave a formal semantics of C. Their semantics covered the entire freestanding C99 feature set and passed 99.2% of the GCC torture tests (a regression test suite), more than both the GCC and Clang compilers. They were able to easily test the semantics because it is executable. A wrapper script around the semantics makes the definition behave like a C compiler. We use this same technique in our work and call this tool `kcc`. See Figure 2 for an overview of the architecture.

Despite covering all of the language features, Ellison and Roșu [6] focused on what we now call the *positive semantics* of C—i.e., the semantics of correct programs—and only touched on the *negative semantics*—i.e., the rules identifying undefined programs. In fact, more of our time has been spent tailoring our semantics to catch incorrect programs than was spent developing the original semantics for correct programs. This work has nearly doubled the size of the semantics, from 1163 rewrite rules in the original, positive semantics [6], to 2155 in our current version. When it comes to C, positive semantics are only half the battle—the easier, better-understood half, we argue.

Our semantics is capable of catching 77 core language (see Figure 1) forms of undefinedness, in addition to many behaviors involving the most common library functions, such as `malloc`, `free`, and `printf`. When our tool encounters undefined behavior, it gets stuck and, for most undefined behaviors, prints out a message referencing the relevant parts of the standard. But if we do not have a message for a particular error, the reason the semantics got stuck can usually be deduced from the final configuration and a little familiarity with the semantics.

Just giving a semantics for correct programs is almost never enough to catch undefined ones. While many undefined behaviors are fairly trivial to catch and only require a more precise semantics, others need a complicated reworking of models for core language features. In the rest of this section, we cover the major issues that arise when attempting to capture the undefinedness of C with an operational semantics, as well as our solutions.

3.1 Expressions

To understand how a positive semantics can give meaning to undefined programs, and the general process we followed in refining and making our semantics more precise, we start with a simple example. Consider the rule for dereferencing a pointer, which defined in its most basic form is:

$$\frac{\langle *(L : \text{ptrType}(T))\ \cdots \rangle_{\mathsf{k}}}{[L] : T}$$

Dereferencing a location L of type pointer-to-T yields an lvalue L of type T ($[L] : T$). This rule is correct according to the semantics of C—it works for any defined program. However, it fails to detect undefined behaviors, such as dereferencing `void` or `NULL` pointers [8, UB #23, 43]. In:

```
int main() { *NULL; return 0; }
```

this rule would apply to `*NULL` and the result (`[NULL] : void`) would immediately be thrown away (by the semantics of "`;`"), despite the undefined dereference.

To catch these behaviors, the above rule could be rewritten:

$$\frac{\langle *(L : \text{ptrType}(T))\ \cdots \rangle_{\mathsf{k}}}{[L] : T} \quad \text{when } T \neq \texttt{void} \wedge L \neq \texttt{NULL}$$

If this is the only rule in the semantics for pointer dereferencing, then the semantics will get stuck when trying to dereference `NULL` or trying to dereference a void pointer.

But we also need to eliminate the possibility of dereferencing memory that is no longer "live"—either variables that are no longer

in scope, or allocated memory that has since been `free`d. Here, then, is the most verbose version of this rule that takes all this into account:

$$\langle \cdots B \mapsto \text{object}(\_, Len, \_) \cdots \rangle_{\text{mem}} \quad \langle \frac{*(\text{sym}(B) + O : \text{ptrType}(T))}{[\text{sym}(B) + O] : T} \cdots \rangle_{\text{k}} \quad \text{when } T \neq \texttt{void} \wedge O < Len$$

The above rule now additionally checks that the location is still alive (by matching an object in the memory), and checks that the pointer is in bounds (by comparing against the length of the memory object). Locations are represented as base/offset pairs $\text{sym}(B) + O$ and objects in memory are represented by a tuple containing their type, size in bytes, and the object representation as a list of bytes. This is explained in detail in Section 3.4.

However, all of these extra side-conditions can make rules more complicated and difficult to understand. We often embed more complicated checks into the main computation. For example, the above rule could be rewritten as two rules:

$$\langle \frac{*(L : \text{ptrType}(T))}{\text{checkDeref}(L, T) \curvearrowright [L] : T} \cdots \rangle_{\text{k}}$$

$$\langle \cdots B \mapsto \text{object}(\_, Len, \_) \cdots \rangle_{\text{mem}} \quad \langle \frac{\text{checkDeref}(\text{sym}(B) + O, T)}{\cdot} \cdots \rangle_{\text{k}} \quad \text{when } O < Len \wedge T \neq \texttt{void}$$

The actual rule for dereferencing pointers from our semantics uses a combination of these techniques, but it also must take into account misuse of the `restrict` qualifier (Section 3.2) and the strict aliasing rules (Section 3.4). These examples should demonstrate how the simple and straightforward rules needed for characterizing defined programs quickly become quite complicated when undefinedness must also be ruled out.

Unsequenced Reads and Writes Unsequenced writes or an unsequenced write and read of the same object is undefined [8, UB #35]. For example, this program would seem to return 3, and it does when compiled with Clang or ICC:

```
int main() {
     int x = 0;
     return (x = 1) + (x = 2);
}
```

However, it is actually undefined because multiple writes to the same location must be sequenced [8, UB #35], but the operands in the addition have an unspecified evaluation order. Compiled with GCC, this same program returns 4.

To catch this in our semantics, we use a technique similar to Norrish [24] and track all locations that have been modified since the last sequence point in a set called locs-written. Whenever we write to or read from a location, we first check this set to verify the location had not previously been written:

$$\langle \frac{\text{writeByte}\ (Loc, V)}{\text{writeByte'}} \cdots \rangle_{\text{k}}\ \langle S \frac{\cdot}{Loc} \rangle_{\text{locs-written}} \quad \text{when } Loc \notin S$$

$$\langle \frac{\text{readByte}\ (Loc)}{\text{readByte'}} \cdots \rangle_{\text{k}}\ \langle S \rangle_{\text{locs-written}} \quad \text{when } Loc \notin S$$

After either of the above rules have executed, the primed operations will take care of any additional checks and eventually the actual writing or reading. Finally, when we encounter a sequence point, we empty the locs-written set:

$$\langle \frac{\text{seqPoint}}{\cdot} \cdots \rangle_{\text{k}}\ \langle \frac{S}{\cdot} \rangle_{\text{locs-written}}$$

Overlapped Writes Another subtlety of assignment comes from the UB caused by assignment of an "inexactly overlapping object" [8, UB #54]. This can happen, e.g., if one variant of a union is assigned to another. If the value being assigned was read from the object being assigned to, then the overlap must be exact. To catch this, we must track the location from which values were read while evaluating the right side of the assignment.

3.2 Type Modifiers

Type modifiers in C are another good example of the extra effort needed to capture undefinedness. If we assume all programs are correct, then we can generally ignore all type modifiers. Only when modifiers are misused, such as by programs exhibiting undefinedness, do they become significant. To catch these misuses requires actually giving semantics to all the modifiers that appear in C11.

First, consider the type qualifier `const`. In C, `const` makes the qualified object unchangeable after initialization. Writes can only occur through non-`const` types [8, §6.3.2.1:1, 6.5.16:1] and attempting to write to a `const`-qualified object through a non-`const`-qualified lvalue invokes undefinedness [8, UB #64]. Therefore, in correct programs, this qualifier has no effect. But to actually catch misuse of `const` by incorrect programs requires the semantics to keep track of `const` qualifiers and check them during all modifications and conversions.

One might expect checking for `const`-correctness to be possible statically, but qualifiers can be dropped by casting pointers. For example:

```
const char p = 'x'; *(char*)&p = 'y';
```

The ability to manipulate the "object representation" of objects through pointers to `char` is an important feature of C, yet writing to any part of an object declared with `const` through such a pointer still invokes undefinedness.

Other modifiers that need similar special consideration include `restrict`, `volatile`, `_Atomic`, the alignment specifiers (`_Alignas`), and the function specifiers (`inline` and `_Noreturn`). The important point is that objects in memory generally must retain the qualifiers they were declared with in order to verify operations through pointers, which may have dropped the qualifiers. We handle this by maintaining the effective type of every object stored in memory (as described in Section 3.4). From the effective type of an object, we can calculate the effective modifiers at every offset into that object, and with this information detecting modifier-specific misuse generally becomes easy. The type qualifier `restrict`, however, requires extra effort.

The `restrict` Qualifier The standard gives clear license for a positive semantics to elide it: "deleting all instances of [`restrict`] from all preprocessing translation units composing a conforming program does not change its meaning" [8, §6.7.3:8]. But, as with `const`, deleting all instances of `restrict` potentially causes a program containing undefinedness to become well-defined. In fact, the standard gives the meaning of the `restrict` qualifier entirely in terms of what becomes undefined, in a whole section section devoted to the topic [8, §6.7.3.1, UB #68, 69]. In a sense, the entire purpose of qualifiers like `const` and `restrict` is to allow programmers the ability to add extra undefinedness to their programs.

Generally speaking, if an object is accessed through a `restrict`-qualified pointer, and it is also modified, then it should only ever be accessed through that same pointer, for the duration of the execution of the scope in which that pointer was declared. For example, these three file scope declarations

```
int * restrict a;
int * restrict b;
extern int c[];
```

assert, according to the standard, that "if an object is accessed using one of `a`, `b`, or `c`, and that object is modified anywhere in the program, then it is not also accessed through the other two" [8, §6.7.3.1].

It is not just accesses through `restrict`-qualified pointers that must be monitored, therefore, but all pointer dereferences. Furthermore, capturing this behavior requires tracking which `restrict`-qualified pointers any particular pointer or pointer expression is based on. Because the value of one `restrict`-qualified pointer variable can be assigned to another in certain cases, a particular pointer expression might be based on multiple `restrict`-qualified pointers. We handle this using the "pointer provenance" mechanism described in Section 3.5.

In addition to tracking which `restrict`-qualified pointers an expression is based on, we need two further sets of information in order to capture misuses of `restrict`:

1. a map between object addresses and the set of `restrict`-qualified pointers through which accesses to that object have occurred and
2. a map between blocks and modified object addresses.

The map in (1) associates objects stored in memory with the `restrict`-qualified pointers that have been used to access them. This is the main mechanism we use to verify future accesses also occur through expressions based on these pointers. But we also need (2) because a `restrict`-qualified pointer is associated with the block that it is declared in, which is not necessarily the same block as an access occurs in. To actually catch all misuses, therefore, requires keeping track of which locations have been modified in the scope of all blocks with associated `restrict`-qualified pointers. Whenever a write occurs, then, the location must be added to the modified set of every active block.

Now, during every dereference of a pointer expression p, that references some object x, we verify it is well-defined by forming two sets of `restrict`-qualified pointers: R, the set of active `restrict`-qualified pointers associated with x from previous accesses, and R', the set of active `restrict`-qualified pointers that p is based on. A `restrict`-qualified pointer is *active* if and only if its associated block is active and x has been modified during the execution of that block. A dereference is well-defined, then, only when $R = R'$.

3.3 The Translation Phase

C11 lists eight "translation phases" [8, §5.1.1.2]. The first six phases involve preprocessing. The seventh corresponds to the actions taken by implementations to transform the C source files of a translation unit into a compiled object file. The eighth, which corresponds to linking, governs how multiple translation units are combined:

> All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

Our semantics gives a separate treatment to the three major phases of C implementations: compilation, linking, and execution (see Figure 2). This brings our semantics very close to how C compilers actually operate. In particular, `kcc` can separately translate translation units and later link them together to form an executable. We use this technique, for example, to pre-translate the standard library in order to speed up the time it takes the tool to interpret C programs.

As we saw in Figure 1, many undefined behaviors are statically-detectable, and many of these deal with the semantics of declarations and building typing environments. It is tempting to not take these behaviors too seriously because many of these issues are readily reported on by compilers and static analyzers. But the semantics for building typing environments and dealing with the linkage of identifiers is notoriously tricky in C because multiple declarations, both within the same translation unit and across multiple translation units, at file scope and block scope, might refer to the same object. And even easily-detectable static undefinedness can slip through compilers and produce unexpected results. For example, consider the following program, composed of two translation units:

```
// Trans. unit 1.                  // Trans. unit 2.
static int a[];                    int a[] = {1, 2, 3};
int main() { return a[0]; }
```

GCC and Clang both compile this program, but they disagree about the value it returns. The GCC-compiled program returns 1, while the Clang compiled program returns 0. Both behaviors are allowed by the standard because the `static` declaration of `a`, with an incomplete array type, invokes undefined behavior: "an object with internal linkage and an incomplete type is declared with a tentative[7] definition" [8, UB #89].

Our semantics for translation phase seven primarily handles the building of typing environments for a translation unit. But we also do a simple abstract interpretation of all function bodies, during which we evaluate all constant expressions and catch cases of statically-detectable undefinedness.

Typing environments for each translation unit are built by processing the declarations. The declaration status of an identifier takes one of four states, in order of increasing definedness: declared, completed, allocated, and defined. For example, the declaration for the identifier `x` moves through all four states:

```
extern int x[];        // declared
extern int x[3];       // completed
int x[3];              // allocated
int x[3] = {1, 2, 3}; // defined
```

Each successive declaration must be compatible with previous declarations. When a declaration passes the "allocated" state, we can reserve a symbolic address for the object.

All four states are needed to prevent various kinds of malformed declarations. We must distinguish "allocated" symbols from "defined" in order to prevent multiple definitions [8, UB #84], for example, and a declaration left in the "declared" state by the end of a translation unit might be an incomplete tentative definition that must be completed and allocated.

Linking In translation phase eight, then, we combine translation units together by resolving each identifier with external linkage that appears in an expression (i.e., every identifier that is actually used) to a single definition. We also must verify that declarations for the same identifier are compatible [8, UB #15] and duplicate definitions do not exist [8, UB #84].

Most C implementations handle compiling and linking in separate phases, and by the time the linking phase has been reached, most typing information has been lost. As a result, for example, compilers generally will not issue a warning about the undefinedness that results from the type incompatibility of the call to `f` below [8, UB #41]:

```
// Trans. unit 1.                  // Trans. unit 2.
int f(int);                        int f(void) {
int main() {                           return 1;
    return f(1);                   }
}
```

The CompCert compiler, due to its current reliance on an external linker, suffers from this same limitation [16]. Detecting such issues requires whole-program static analysis and because of the subtleties of type compatibility and the rules governing the linkage of identifiers, this can be hard to get right.

Other similar cases of link time undefinedness also seem to slip by compilers without warnings. For example, the following

[7] "A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier `static`, constitutes a *tentative definition*." [8, §6.9.2:2]

program relies on a common language extension for declaring a global identifier shared between multiple translation units and will usually not elicit a warning from GCC or Clang:

```
// Trans. unit 1.              // Trans. unit 2.
int x;                         int x;
int main() { ... }             ...
```

But it is a language extension, and as such it relies on undefined behavior. Each of those global declarations of `x` constitutes a "tentative definition," which becomes a real definition by the end of the translation unit. The program is therefore undefined because the identifier `x` has multiple external definitions [8, UB #84].

3.4 Memory Model

Addresses in our memory model are symbolic base/offset pairs, which we write as $\text{sym}(B) + O$, where B corresponds to the base address of an object itself and the O represents the offset of a particular byte in the object. We wrap the base using "sym" because it is symbolic—despite representing a location, it is not appropriate to, e.g., directly compare $B < B'$ [8, §6.5.8:5]. These base addresses also encode their storage duration. This allows, for example, reading uninitialized memory with static storage duration[8] to result in zeros, while reading uninitialized memory with automatic storage duration[9] to result in an error. Our memory, then, is a map from base addresses to blocks of bytes. Each base address represents the memory of a single object.

This model is the same technique used by Blazy and Leroy [1] and by Roșu et al. [28]. It takes advantage of the fact that the address of objects in memory and addresses returned by allocation functions like `malloc()` are unspecified [8, §7.20.3]. However, there are a number of restrictions on addresses, such as the elements of an array being contiguous and the fields in a struct being ordered (though not necessarily contiguous). Consider:

```
int a, b;
if (&a < &b) { ... }
```

If we gave objects concrete, numerical addresses, then they would always be comparable. However, this excerpt is actually undefined [8, UB #53]. We only give semantics to pointer comparisons when the two addresses share a common base. For example:

$$\langle \frac{(\text{sym}(B) + O : T) < (\text{sym}(B) + O' : T) \cdots}{1 : \texttt{int}} \rangle_{\text{k}} \quad \text{when } O < O'$$

Thus, evaluation gets stuck on the program above because `&a` and `&b` do not share a common base B. Of course, sometimes locations are comparable, for example:

```
struct { int a; int b; } s;
if (&s.a < &s.b) { ... }
```

The addresses of `a` and `b` are guaranteed to be in order [8, §6.5.8:5], and our semantics does allow this comparison because the pointers share a common base.

Because all data must be split into bytes to be stored in memory, the same must happen with pointers. However, because our addresses are not actual numbers, they must be split symbolically. Assuming a particular pointer $\text{sym}(B) + O$ was four bytes long, we split it into the list of bytes given by

$$\text{subObject}(\text{sym}(B) + O, 0), \text{subObject}(\text{sym}(B) + O, 1), \\ \text{subObject}(\text{sym}(B) + O, 2), \text{subObject}(\text{sym}(B) + O, 3)$$

where the first argument of subObject is the object in question and the second argument is which byte this represents. This allows the reconstruction of the original pointer, but only if given all the bytes.

[8] Objects declared at file scope or with the specifier `static`.

[9] Objects declared inside a function without `register`, `static`, or `extern`.

Indeterminate Memory An *indeterminate* value is one that may be "either an unspecified value or a trap representation," where a trap representation is "an object representation that need not represent a value of the object type" [8, §3.19]. Uninitialized block scope variables not declared `static` or `extern`, for example, take indeterminate values.

Using trap representations for indeterminate values is an implementation choice, but it is one that clearly leads to more undefinedness. In general, we use a trap representation wherever the standard allows one to be used. Then, if the value of a trap representation is ever needed when evaluating an expression, we catch the undefined behavior that results [8, UB #12]. However, an important exception must be made for the `unsigned char` type [8, §6.2.6.1:3–4]—values of this type have no trap representation. We model this in our semantics by transforming trap representations into unspecified values when read through an lvalue of `unsigned char` type. Unspecified values will also cause the semantics to get stuck in many cases, but we define more operations on them than on trap representations. This allows for the implementation of functions like `memcpy()`, for example—every byte must be copied, even indeterminate values [8, §6.2.6.1:4].

Strict Aliasing As Krebbers [12] points out, the memory model described above is not sufficient to capture the aliasing restrictions of C11 [§6.5:6–7]. These restrictions are intended to allow optimizations using type-based alias analysis. Each object in memory, according to the strict aliasing rules, has an "effective type" and a compiler is allowed to assume that accesses to such objects will only occur through an lvalue expression of either a compatible type (modulo qualifiers and signedness) or a character type. The effective type of an object is its declared type or (in the case of memory allocated with `malloc`) the type of the lvalue through which the last store occurred. For example, line 4 below results in undefined behavior according to the strict aliasing rules [8, UB #37]:

```
1  int main() {
2        int *p = malloc(sizeof(p));
3        *(long*)p = 42;
4        return *p;
5  }
```

To address this, we store the declared type of objects along with their object representations. In the case of memory allocated with `malloc`, we keep another map from symbolic addresses to the type of the last object stored at the address.

With this extra typing information, we can calculate the effective type at a certain byte offset into an object. In the case of an aggregate type, there can be ambiguity about whether an offset should have the type of the aggregate or its first element, but the strict aliasing rules allow for a pointer to an aggregate or union type to be aliased by a pointer to a type of any of its elements. We also do not need to keep track of the active variant of a union type because a pointer to a union type may be aliased to a pointer to any of the union elements, regardless of which one is "active." We catch other misuse of unions using the mechanism described in the next section.

3.5 Pointer Provenance

Consider this example, inspired by Defect Report #260 [2004]:

```
1  int main() {
2        char a[2][2];
3        char *p = &a[0][1] + 1, *q = &a[1][0];
4        if (memcmp(&p, &q, sizeof(p)) == 0) {
5             *q = 42;
6             *p = 42;
7        }
8  }
```

Even though the standard requires that arrays be contiguously allocated, and therefore the bit patterns representing the `p` and `q`

pointers should compare equal on line 4, it does not follow that the assignments on lines 5 and 6 are both defined. In fact, the assignment on line 5 is well-defined, but the assignment on line 6 invokes undefined behavior according to the following criteria:

> Addition [...] of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated. [8, UB #47]

The moral here is that the provenance[10] of pointers can be significant when determining whether operations performed on them are defined. When checking if a dereference is within bounds, it matters whether a pointer value was based on an array or whether it came from the & operator applied to a scalar. Checking the type or value at the memory location before a dereference is not enough.

Therefore, we use an approach similar to the fat pointers of Jim et al. [11] and Necula et al. [21] and make a distinction between the value that might be stored in a pointer variable and the symbolic addresses that actually pick out objects in our memory model. The former might have additional qualifiers, or tags, that carry extra information about the provenance of the address. We generalize this technique to catch many different kinds of undefinedness in our semantics. Pointer values might carry any of the following four tags, which are not part of the object representation of pointers (from the perspective of a C program), but can still follow pointer values through, e.g., function calls and memory stores. The details of when each tag will be attached to a pointer and when it might be removed depends on the tag, but often the standard is not clear about just how long and through what kinds of expressions the provenance of a pointer should remain significant.

fromUnion "When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to other members take unspecified values" [8, §6.2.6.1:7]. This tag tracks the union variant a pointer or lvalue expression is based on so we can mark the section of memory not overlapping with the active variant as unspecified. This allows some type punning[11] while still catching violations that can result from attempting to use the unspecified values in the non-overlapped part of the union.

fromArray As we pointed out above, we must track the size of an array that a pointer is based on and its current offset into the array in order to catch violations dealing with undefined pointer arithmetic and out-of-bounds pointer dereferences [8, UB #46–49].

basedOn We also must track when a pointer can be traced back to the value stored in some `restrict`-qualified pointer variable. This allows us to associate objects in memory with the `restrict`-qualified pointers used to access them and check for undefined assignments involving `restrict`-qualified pointers [8, UB #68, 69].

align We track a pointers's alignment using this same mechanism. We use this for catching the undefinedness that results from a misaligned pointer after a conversion [8, UB #25].

4. Evaluation

In order to evaluate our semantics and the analysis tool generated from it, we first looked for a suite of undefined programs. Although we were unable to find any test suite focusing on undefined behaviors, we did find test suites that included a few key behaviors. We consider our inability to find a definitive metric for evaluating our work to be symptomatic of the surprising lack of attention our goal of detecting strictly-conforming C programs has received. Below we mention the testing and cataloguing work we found related to undefinedness, including the Juliet Test Suite, which we use as one of our partial undefinedness benchmarks. We end this section with a description of our own undefinedness test suite.

There is an ISO technical specification for program analyzers titled "C Secure Coding Rules" [2013], suggesting programmatically enforceable rules for writing secure C code. It is similar to MISRA-C [18], whose goal was to create a "restricted subset" of C to help those using C meet safety requirements. MISRA released a "C Exemplar Suite," containing both conforming and non-conforming code for the majority of the MISRA C rules. However, these tests contain many undefined behaviors mixed into a single file, and no way to run the comparable defined code without running the undefined code. Furthermore, the MISRA tests focus on statically detectable UB. The CERT C Secure Coding Standard [29] and MITRE's "common weakness enumeration" (CWE) classification system [19] are other similar projects, identifying many causes of program error and cataloging their severity and other properties. The projects mentioned above include many undefined behaviors—for example, the undefinedness of signed overflow [8, UB #36] corresponds to CERT's INT32-C and to MITRE CWE-190.

The Juliet Test Suite NIST has released a suite of tests called the Juliet Test Suite for C/C++ [23], which is based on MITRE's CWE classification system. It contains over 45,000 tests, each triggering one of the 116 different CWEs supported by the suite. Most of the tests (~70%) are C and not C++ and they focus on statically detectable behaviors. But not all of the CWEs are actually undefined—many are simply insecure or unsafe programming practices.

Because the Juliet tests include a single undefined behavior per file and come with positive tests corresponding to the negative tests, we decided to extract an undefinedness benchmark from them. To use the Juliet tests as a test suite for undefinedness, we had to identify which tests were actually undefined. This was largely a manual process that involved understanding the meaning of each CWE. It was necessary due to the large number of defined-but-bad-practice tests that the suite contains. Interestingly, the suite contained some tests whose supposedly defined portions were actually undefined. Using our analysis tool, we were able to identify six distinct problems with these tests, which we submitted to NIST.

This extraction gave us 4113 tests, with about 96 lines per test. The tests can be divided into six classes of undefined behavior: use of an invalid pointer (buffer overflow, returning stack address, etc.), division by zero, bad argument to `free()` (stack pointer, pointer not at start of allocated space, etc.), uninitialized memory, bad function call (incorrect number or type of arguments), or integer overflow. We then ran these tests using Valgrind Memcheck [22], and the Value Analysis plugin for Frama-C [3], in addition to our tool, `kcc`. The results appear in Figure 3.

Our Undefinedness Test Suite Because we were unable to find an ideal test suite for evaluating detection of undefined behaviors, we began development of our own. As we discussed in Section 2, undefined behavior reached during an execution causes the entire execution to become undefined. This means each test in the suite must be a separate program, otherwise one undefined behavior may interact with another. In addition, each test should come with a corresponding defined test as a control, making it possible to identify false-positives in addition to false-negatives. Our suite currently includes 261 tests. These tests are much broader than the Juliet tests, covering all 77 categories of core language undefined behaviors (identified in Figure 1) as opposed to the 12 covered by the Juliet tests. We hope it will serve as a starting point for the development of a larger, more comprehensive undefinedness test suite.

We compared the same tools as before against our own test suite. This time, we also included the CompCert interpreter [2] and the

[10] This term comes from Defect Report #260 [2004].

[11] I.e., accessing a value through a union variant other than the variant of the lvalue through which the last write occured, which is allowed in some cases.

Undefined behavior	No. tests	Tools (% passed) Valgrind[a]	V. Analysis[b]	kcc
Use of invalid pointer (UB #10, 43, 46, 47)	3193	70.9	100.0	100.0
Division by zero (UB #45)	77	0.0	100.0	100.0
Bad argument to `free()` (UB #179)	334	100.0	100.0	100.0
Uninitialized memory (UB #21)	422	100.0	100.0	100.0
Bad function call (UB #38–41)	46	100.0	100.0	100.0
Integer overflow (UB #36)	41	0.0	100.0	100.0

[a] Valgrind Memcheck, v. 3.5.0, `http://valgrind.org`

[b] Frama-C Value Analysis plugin, v. Nitrogen-dev, `http://frama-c.com/value.html`

Figure 3. Comparison of analyzers against the Juliet Test Suite.

Undefined behavior	No. tests	Tools (% passed) Astrée[a]	CompCert[b]	Valgrind[c]	V. Analysis[d]	old kcc[e]	kcc
Compile time (24 UBs)	81	40.7	60.5	0.0	32.1	38.3	98.8
Link time (8 UBs)	38	47.4	84.2	0.0	42.1	23.7	100.0
Run time (45 UBs)	142	46.5	40.9	9.9	58.5	40.1	99.3
Total (77 UBs)	261	44.8	53.3	5.4	47.9	37.2	99.2

[a] The Static Analyzer Astrée, v. 14.10, `http://www.absint.com`

[b] CompCert C interpreter, v. 2.3pl2, `http://compcert.inria.fr`

[c] Valgrind Memcheck, v. 3.10.0, `http://valgrind.org`

[d] Frama-C Value Analysis, v. Neon, `http://frama-c.com/value.html`

[e] Version of the tool from Ellison and Roșu [6].

Figure 4. Comparison of analyzers against our test suite comprising 261 tests, covering 77 core language undefined behaviors. Note that some tools don't support all language features covered in the tests.

Tools	Comp. time	Link time	Run time	Total
Astrée	11	4	26	41
CompCert	15	7	23	45
Valgrind	0	0	5	5
V. Analysis	11	4	30	45
all but kcc	17	7	34	58
old kcc	10	2	23	35
kcc	24	8	45	77

Figure 5. Core language undefined behaviors each tool detects (total out of 77). We counted a tool as having coverage for a behavior if it caught the behavior in at least one test.

Astrée analyzer [4]. The former interprets programs according to the semantics of the formally-verified CompCert compiler. Like our tool, it attempts to detect undefinedness and halts when it encounters an undefined behavior. The latter, Astrée, is an abstract-interpretation based analyzer reported to detect most undefined behavior (albeit in C99, not C11). As can be seen in the tables, our tool passes most of our test cases. The two failures are due to minor bugs still outstanding at the time of this writing.

From these results (Figure 4), and based on the premise that our test suite includes tests for each behavior that are trivial enough that no tool looking for that behavior could miss it, we estimate the number of core language undefined behaviors each tool detects in Figure 5. These tools do not have complete coverage for many behaviors—the CompCert interpreter, for example, does not support programs comprising multiple translation units, so it necessarily will not have complete coverage for any of the behaviors we have categorized "link time."

From Figure 5 we can see that none of the other tools we tested appeared to have coverage for 19 core language undefined behaviors.[12] Many of these deal with newer features of the C language, such as the `restrict` qualifier (#68, 69), alignment specifiers (#73), variable-length arrays (#75), and the `inline` function specifier (#70). Some are tricky to monitor for, such as unsequenced side-effects (#35) and misuses of `restrict` and `const` (discussed in Section 3).

Other Related Work As we mention in the introduction, many previous semantic efforts have tended to focus on the semantics of defined programs. The recent work of Krebbers [12, 13] is a notable exception, however. His semantics, formalized in Coq, captures many of the hairier sources of undefinedness, such as expression nondeterminism and the the alias restrictions (discussed in Section 3.4).

The closest comparison for our kcc tool might be the CompCert interpreter, considered above. Note that the CompCert compiler, in contrast, being a C implementation, does not generally report on UB and is perfectly justified in taking any action upon encountering it. We also consider a few analyzers above. Some of our techniques for capturing undefinedness have precedent in the literature on such analyzers, which we try to point out in our discussion. But many of these tools, such as Valgrind, tend to have a narrow focus when it comes to detecting UB, which we think our results demonstrate.

Other promising recent work on detecting undefinedness comes from Wang et al. [30]. They charaterize undefinedness as code unstable under optimization and instead of attempting to catch the behaviors themselves, they catch code that would be affected by optimizations that aggressively take advantage of undefinedness.

5. Future Work and Conclusion

If some possible implementation is allowed by the standard to treat some expression as invoking undefined behavior, then our tool

[12] The complete list: UB #14, 22, 25, 35, 42, 60, 64–66, 68–70, 73, 75, 77, 80, 83, 86, and 89.

should treat it as undefined behavior. Our goal is to detect maximally portable, strictly-conforming programs. An example of this zeal is in our treatment of "negative zeros." The standard specifies that it is implementation-defined as to whether the two negative zero bit patterns (i.e., a sign bit of one with all ones or zeros in the value bits) might be a "trap representation or a normal value" [8, §6.2.6.2:2]. In our semantics, then, creating either of these bit patterns with a signed integer type using the bitwise operators should raise an error. In practice, however, few (if any) real implementations actually treat the negative zero bit patterns as trap representations.

Although our semantics is parameterized over, e.g., the size of various types, we need to expand this mechanism to include more implementation-defined behaviors in order to make our tool more practical for real-world applications. Often, checking for conformance with a certain implementation is more desirable than checking for strict conformance to the standard. For example, the encoding of negative integers is implementation-defined, yet two's complement is standard in all major implementations today.

Undefinedness is a feature of the C language that can facilitate aggressive optimizations. But it is also terribly subtle and the source of many bugs. In order for compilers to take full advantage of the optimization opportunities afforded by undefinedness, users must be aware of the assumptions compilers make about their code. More generally, detecting the absence of undefined behavior is an important goal on the road to fully verified software.

References

[1] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. URL http://dx.doi.org/10.1007/s10817-009-9148-3.

[2] B. Campbell. An executable semantics for CompCert C. In *Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2012. URL http://dx.doi.org/10.1007/978-3-642-35308-6_8.

[3] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *Conf. on Source Code Analysis and Manipulation (SCAM'09)*, pages 123–124. IEEE, 2009. URL http://dx.doi.org/10.1109/SCAM.2009.22.

[4] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer Berlin Heidelberg, 2005. URL http://dx.doi.org/10.1007/978-3-540-31987-0_3.

[5] C. Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012. URL http://hdl.handle.net/2142/34297.

[6] C. Ellison and G. Roșu. An executable formal semantics of C with applications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544, 2012. URL http://dx.doi.org/10.1145/2103656.2103719.

[7] ISO/IEC JTC 1, SC 22, WG 14. Rationale for international standard—programming languages—C. Technical Report 5.10, Intl. Org. for Standardization, 2003. URL http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf.

[8] ISO/IEC JTC 1, SC 22, WG 14. Defect report #260. Technical report, 2004. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

[9] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:2011: Programming languages—C. Technical report, Intl. Org. for Standardization, 2012.

[10] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC TS 17961:2013 C secure coding rules. Technical report, Intl. Org. for Standardization, 2013.

[11] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference (ATEC'02)*, pages 275–288. USENIX Association, 2002. URL http://dl.acm.org/citation.cfm?id=647057.713871.

[12] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2013. URL http://dx.doi.org/10.1007/978-3-319-03545-1_4.

[13] R. Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in C. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*, pages 101–112. ACM, 2014. URL http://dx.doi.org/10.1145/2535838.2535878.

[14] C. Lattner. What every C programmer should know about undefined behavior, 2011. URL http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[15] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. URL http://dx.doi.org/10.1145/1538788.1538814.

[16] X. Leroy. The CompCert C verified compiler: Documentation and user's manual, version 2.3. Technical report, INRIA Paris-Rocquencourt, 2014.

[17] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. URL http://dx.doi.org/10.1016/0304-3975(92)90182-F.

[18] MISRA. MISRA-C: 2004—Guidelines for the use of the C language in critical systems. Technical report, MIRA Ltd., 2004.

[19] MITRE. The common weakness enumeration (CWE) initiative, 2012. URL http://cwe.mitre.org/.

[20] T. Nagel. Troubles with GCC signed integer overflow optimization, 2010. URL http://thiemonagel.de/2010/01/signed-integer-overflow/.

[21] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 128–139. ACM, 2002. URL http://dx.doi.org/10.1145/503272.503286.

[22] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 89–100. ACM, 2007. URL http://dx.doi.org/10.1145/1250734.1250746.

[23] NIST. Juliet test suite for C/C++, version 1.0, 2010. URL http://samate.nist.gov/SRD/testsuite.php.

[24] M. Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, 1998.

[25] N. S. Papaspyrou. Denotational semantics of ANSI C. *Computer Standards and Interfaces*, 23(3):169–185, 2001.

[26] J. Regehr. A guide to undefined behavior in C and C++, 2010. URL http://blog.regehr.org/archives/213.

[27] G. Roșu and T. F. Șerbănuță. An overview of the K semantic framework. *J. Logic and Algebraic Programming*, 79(6):397–434, 2010. URL http://dx.doi.org/10.1016/j.jlap.2010.03.012.

[28] G. Roșu, W. Schulte, and T. F. Șerbănuță. Runtime verification of C memory safety. In *Runtime Verification (RV'09)*, volume 5779, pages 132–152. Springer, 2009. URL http://dx.doi.org/10.1007/978-3-642-04694-0_10.

[29] R. C. Seacord. *The CERT C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems*. 2014.

[30] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 260–275. ACM, 2013. URL http://dx.doi.org/10.1145/2517349.2522728.