Technical Report: Defining the Undefinedness of C

Chucky Ellison Grigore Roșu

University of Illinois

{celliso2,grosu}@illinois.edu

Abstract

This paper investigates undefined behavior in C and offers a few simple techniques for operationally specifying such behavior formally. A semantics-based undefinedness checker for C is developed using these techniques, as well as a test suite of undefined programs. The tool is evaluated against other popular analysis tools, using the new test suite in addition to a third-party test suite. The semantics-based tool performs at least as well or better than the other tools tested.

1. Introduction

A programming language specification or semantics has dual duty: to describe the behavior of correct programs and to identify incorrect programs. The process of identifying incorrect programs can also be seen as describing which programs do not belong to the language. Many languages come with static analyses (such as type systems) that statically exclude a variety of programs from the language, and there are rich formalisms for defining these restrictions. However, well-typed programs that "go bad" dynamically are less explored. Some languages choose to give these programs semantics involving exceptions, or similar constructs, while others choose to exclude these programs by fiat, stating that programs exhibiting such behaviors do not belong to the language. Regardless of how they are handled, semantic language definitions must specify these situations in some manner. This paper is about these behaviors and such specifications.

This paper is a sister paper to our previous work giving a complete, formal semantics to C [6]. In that work, we focused primarily on giving semantics to correct programs, and showed how our formal definition could yield a number of tools for exploring program evaluation. The evaluation we performed was against defined programs, and the completeness we claimed was for defined programs. In contrast, in this work we focus on identifying undefined programs. We go into detail about what this means and how to do it, and evaluate our semantics against test suites of undefined programs.

Although there have been a number of formal semantics of various subsets of C (see above paper for an in-depth comparison), they generally focus on the semantics of correct programs only. While it might seem that semantics will naturally capture undefined behavior simply by exclusion, because of the complexity of undefined behavior, it takes active work to avoid giving many undefined programs semantics. In addition, capturing the undefined behavior is at least as important as capturing the defined behavior, as it represents a source of many subtle program bugs. While a semantics of defined programs can be used to prove their behavioral correctness, any results are contingent upon programs actually being defined—it takes a semantics capturing undefined behavior to decide whether this is the case.

C, together with C++, is the king of undefined behavior—C has over 200 explicitly undefined categories of behavior, and more that are left implicitly undefined [11]. Many of these behaviors can not be detected statically, and as we show later (Section 2.6), detecting them is actually undecidable even dynamically. C is a particularly interesting case study because its undefined behaviors are truly undefined—the language has nothing to say about such programs. Moreover, the desire for fast execution combined with the acceptance of danger in the culture surrounding C means that very few implementations try to detect such errors at runtime.

Concern about undefined programs has been increasing due to the growing interest in security and safety-critical systems. However, not only are these issues broadly misunderstood by the developer community, but many tools and analyses underestimate the perniciousness of undefined behavior (see Section 2 for an introduction to its complexity), or even limit their input to only defined programs. For example, CompCert [16], a formally verified optimizing compiler for C, assumes its input programs are completely defined, and gives few guarantees if they contain undefined behavior. We provide this study of undefined behaviors in the hope of alleviating this obstacle to correct software.

To our knowledge, this is the first study of the semantics of undefined behavior (though there has been a practical study of arithmetic overflow in particular [4]). Undefinedness tends to be considered of secondary importance in semantics, particularly because of the misconception that capturing undefined behaviors comes "for free" simply by not defining certain cases. However, it can be quite difficult to cleanly separate the defined from the undefined. To see how this might be the case, consider that the negation of a context free language may not be context free, or that not all semidecidable systems are decidable. While it is true that capturing undefinedness is about not defining certain cases, this is easier said than done (see Section 4).

Our contributions include the following:

- a systematic formal analysis of undefinedness in C;
- identification and comparison of techniques that can be used to define undefinedness;
- a semantics-based tool for identifying undefined C programs;
- initial work on a comprehensive benchmark for undefined behavior in C.

The tool, the semantics, and the test suite can be found at `http://c-semantics.googlecode.com/`.

In the following sections, we formalize what undefinedness means in C and why it is important to detect (Section 2), give background information on our formalism and on our semantics (Section 3), explain techniques needed to capture undefinedness semantically (Section 4), and compare how our semantics-based tool performs against other analysis tools in identifying such behavior (Section 5).

2. Undefinedness

In this section we examine what undefinedness is and why it is useful in C. We also look into some of the complexity and strangeness of undefined behavior. We finish with a brief overview of undefinedness in other popular languages. Other good introductions to undefinedness in C (and C++) include Regehr [24] and Lattner [14]. The fact that the best existing summaries are blog posts should indicate that there is a significant lack of academic work on undefinedness.

2.1 What Undefinedness Is

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" [11, §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). [11, §3.4.3:2]

This effectively means that, according to the standard, undefined behavior is allowed to do anything at any time. This is discussed in more detail in Section 2.4. Undefined programs are *invalid* C programs, because the standard imposes no restrictions on what they can do. Of course, particular implementations of C may guarantee particular semantics for otherwise undefined behaviors, but these are then extensions of the actual C language.

2.2 Undefinedness is Useful

The C standard ultimately decides which behaviors are to be undefined and which are to be defined. One source of undefined behaviors are behaviors that are exceptional in some way, while also hard (or impossible) to detect statically.[1] If these behaviors are undefined, an implementation of C does not need to handle them by adding complex static checks that may slow down compilation, or dynamic checks that might slow down execution of the program. This makes programs run faster.

For example, dereferencing an invalid pointer may cause a trap or fault (e.g., a Segmentation Fault); more importantly, it does not do the same thing on every platform. If the language required that all platforms behave identically, for example by throwing an exception when dereferencing an invalid pointer, a C compiler would have to generate more complicated code for dereference. It would have to generate something like: for dereference of a pointer p, if p is a valid pointer, go ahead and dereference p; otherwise, throw an exception. This additional condition would mean slower code, which is something that the designers of C try to avoid: two of the design principles of C are that it should be "[made] fast, even if it is not guaranteed to be portable", and that implementations should "trust the programmer" [10].

To keep the language fast, the standard states that dereferencing an invalid pointer is undefined [11, §6.5.3.3:4]. This means programs are allowed to exhibit any behavior whatsoever when they dereference an invalid pointer. However, it also means that programmers now need to worry about it, if they are interested in writing portable code. The upshot of liberal use of undefined behavior is that no runtime error checking needs to be provided by the language. This leads to the fastest possible generated code, but the tradeoff is that fewer programs are portable.

2.3 Undefinedness is also a Problem

Even though undefined behavior comes with benefits, it also comes with problems. It can often confuse programmers who upon writing an undefined program, think that a compiler will generate "reasonable" behavior. In fact, compilers do many unexpected things when processing undefined programs.

For example, in the previous section we mentioned that dereferencing invalid pointers is undefined. When given this code:

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

[1] There are also undefined behaviors that are not hard to detect statically, such as "If two identifiers differ only in nonsignificant characters, the behavior is undefined" [11, §6.4.2:6], but are there for historical reasons or to make a compiler's job easier.

GCC,[2] Clang,[3] and ICC[4] will *not* generate code that segfaults, because they simply ignore the dereference of NULL. They are allowed to do this because dereferencing NULL is undefined—a compiler can do anything it wants to such an expression, including totally ignoring it.

Even worse is that compilers are at liberty to assume that undefined behavior will not occur. This assumption can lend itself to more strange consequences. One nice example is this piece of C code [21]:

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

Programmers might think to use a construct like this in order to handle a possible arithmetic overflow. However, according to the standard, `x + 1` can never be less than `x` unless undefined behavior occurred (signed overflow is undefined [11, §6.5:5]). A compiler is allowed to assume undefined behavior never occurs—even if it does occur, it does not matter what happens. Therefore, a compiler is entirely justified in removing the branch entirely. In fact, GCC 4.1.2 does this at all optimization levels and Clang and GCC 4.4.4 do this at optimization levels above 0. Even though Clang and GCC only support two's complement arithmetic, in which `INT_MAX + 1 == INT_MIN`, both compilers clearly take advantage of the undefinedness.

Here is another example where compilers take advantage of undefined behavior and produce unexpected results:

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

Because assignment is an expression in C that evaluates to "the value of the left operand after the assignment" [11, §6.5.16:3], this piece of code would seem to return 3. However, it is actually undefined because multiple writes to the same location must be sequenced (ordered) [11, §6.5:2], but addition is nondeterministic. GCC returns 4 for this program, because it transforms the code similar to the following:

```
int x = 0;
x = 1;
x = 2;
return x + x;
```

For defined programs, this transformation is completely behavior preserving. However, because it is undefined, the behavior can be, and in the case of GCC is, different than what most programmers expect.

[2] v 4.1.2 unless otherwise noted. All compilers on x86_64 with -O0 unless otherwise noted.

[3] v 3.0

[4] v 11.1

2.4 Strangeness of C Undefinedness

In one sense, all undefined behaviors are equally bad because compiler optimizers are allowed to assume undefined behavior can never occur. This assumption means that really strange things can happen when undefined behavior *does* occur. For example, undefined behavior in one part of the code might actually affect code "that ran earlier", because the compiler can reorder things. For example:

```
int main(void){
  int r = 0, d = 0;
  for (int i = 0; i < 5; i++) {
    printf("%d\n", i);
    r += 5 / d; // divides by zero
  }
  return r;
}
```

Even though the division by zero occurs after the `printf` lexically, it is not correct to assume that this program will "at least" print 0 to the screen. Again, this is because an undefined program can do *anything*. In practice, an optimizing compiler will notice that the expression `5 / d` is invariant to the loop and move it before the loop. Both GCC and ICC do this at optimization levels above 0. This means on a machine that faults when doing division by zero, nothing will be printed to the screen except for the fault. Again, this is correct behavior according to the C standard because the program triggers undefined behavior.

2.5 Implementation-Dependent Undefined Behavior

The C standard allows implementations to choose how they behave for certain kinds of behavior. So far we have discussed only undefined behavior, for which implementations may do whatever they want. However, there are other kinds of behavior, including unspecified behavior and implementation-defined behavior [11, §3.4]:

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.

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

An example of unspecified behavior is the order in which summands are evaluated in an addition. An example of implementation-defined behavior is the size of an `int`. Whether or not a program is undefined may actually depend on the choices made for an implementation regarding implementation-defined or unspecified behaviors.

2.5.1 Undefinedness Depending on Implementation-Defined Behavior

Depending on choices of implementation-defined behavior, behavior can be defined or not. For example:

```
int* p = malloc(4);
if (p) { *p = 1000; }
```

In this code, if `ints` are 4 bytes long, then the above code is free from undefined behaviors. If instead, `ints` are 8 bytes long, then the above will make an undefined memory read outside the bounds of the object pointed to by `p`. In practice, this means that programmers must be intimately familiar with the implementation-defined choices of their compiler in order to avoid potential undefinedness arising from it.

2.5.2 Undefinedness Depending on Unspecified Behavior

Like implementation-defined undefined behavior above, undefined behavior can also depend on unspecified behavior. However, while implementation-defined behavior must be documented [11, §3.19.1] so that programmers may rely on it, unspecified behavior has no such requirement. An implementation is allowed to have different unspecified behaviors in different situations, and may even change them at runtime.

One such example is evaluation order. Because evaluation order is almost completely 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" this code:

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

because there is an evaluation strategy (e.g., right-to-left) that would set `d` to `0` before doing the division. While GCC compiles this code and generates an executable containing no runtime error, CompCert [16], a formally verified optimizing compiler for C, generates code that exhibits a division by zero. Both of these behaviors are correct because the program contains reachable undefined behavior. In practice, this means that any tool seeking to identify all undefined behaviors must search all possible evaluation strategies.

2.6 Difficulties in Detecting Undefined Behavior

In general, detecting undefined behavior is undecidable even with dynamic information. Consider the following example:

```
int main(void){
  guard();
  5 / 0;
}
```

The undefinedness of this program is based on what happens in the `guard()` function. Only if one can show that `guard()` will terminate can one conclude that this program has undefined behaviors. However, showing that `guard()` terminates, even with runtime information, is undecidable.

Although it is impossible (in general) to prove that a program is free from undefined behaviors, this raises the question of whether one can *monitor* for undefined behaviors. The question is somewhat hard to pin down—as we saw in Section 2.3, a smart compiler may detect undefined code statically and generate target code that does not contain the same behaviors. This means a monitor or even state-space search tool would not be able to detect such undefined behavior at runtime, 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 the 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 nondeterministic single-threaded[5] programs, one may need to keep arbitrary amounts of information, making the problem decidable but intractable. Consider this program:

```
int r = 0;
int flip() {
  // return 0 or 1 nondeterministically
}
int main(void){
  while(true){
    r = (r << 1) + flip();
  }
}
```

At iteration n of the loop above, `r` can be any one of 2^n values. 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. The above argument could be reformulated to encode `r` using allocated memory, avoiding the limited sizes of builtin types like `int`, but the presentation would be more complicated.

If multiple threads are introduced, then the problem 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()) {}         5 / d;
d = 0;
```

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. However, showing that the loop terminates is undecidable.

2.7 Undefinedness in Other Languages

It should be clear at this point that undefinedness is a huge part of the C language, but other languages also have undefined behavior. The documentation or specifications of

[5] Threads were added to C in C11 [11].

many popular languages identify undefined programs that are allowed to do anything (including crash). For example, LLVM includes a number of undefined behaviors, including calling a function using the wrong calling convention [15]. Scheme's specification describes undefined behavior in relation to `callcc` and `dynamic-wind` [13, p. 34]. Even Haskell, an otherwise safe and pure language, has undefined behavior in a number of unsafe libraries, such as `Unsafe.Coerce` and `System.IO.Unsafe`. There are many other examples of this in other programming languages, including Perl [5] and Ruby [27].[6]

Even languages without undefined behavior run into many of the same specification problems. Any language with constructs having exceptional behavior, such a division by zero, needs to be able to specify or define the behavior of these cases. These kinds of behavior are invariably of the form, "the —– construct is defined as —–. However, in some special case —–, it raises an exception instead." For example, the Java standard states,

> The binary `/` operator performs division, producing the quotient of its operands [...] if the value of the divisor in an integer division is 0, then an `ArithmeticException` is thrown. [8, §15.17.2]

Similarly, the SML Basis Library standard states:

> [`i div j`] returns the greatest integer less than or equal to the quotient of `i` by `j`. [...] It raises [...] Div when `j` = 0. [7]

This pattern comes up frequently enough in most languages that it is worthy of investigation. We investigate ways of formally specifying such behaviors in Section 4.

3. $\mathbb{K}$ Semantics of C

In this section we give a brief introduction to the $\mathbb{K}$ Framework, which will be useful background information for Section 4, as well give a brief summary of our previous work giving a complete formal semantics to C [6].

3.1 $\mathbb{K}$ Framework

To give our semantics, we use a rewriting-based semantic framework called $\mathbb{K}$ [25], inspired by rewriting logic (RL) [18]. RL organizes term rewriting *modulo equations* (namely associativity, commutativity, and identity) as a logic with a complete proof system and initial model semantics. The central idea behind using RL as a formalism for the semantics of languages is that the evolution of a program can be clearly described using rewrite rules. A rewriting theory consists essentially of a signature describing terms and a set of rewrite rules that describe steps of computation. Given some term allowed by signature (e.g., a program together

[6] Though the Ruby standard uses the word "unspecified", they define this to include behavior "not necessarily defined for any particular implementation" [27, §4.17].

$$\left\langle \begin{array}{l} \langle K\rangle_{\mathsf{k}}\ \langle Map\rangle_{\mathsf{genv}}\ \langle Map\rangle_{\mathsf{gtypes}}\ \langle Set\rangle_{\mathsf{locsWrittenTo}}\ \langle Set\rangle_{\mathsf{notWritable}} \\ \langle Map\rangle_{\mathsf{mem}}\ \Big\langle \Big\langle \langle Map\rangle_{\mathsf{env}}\ \langle Map\rangle_{\mathsf{types}} \Big\rangle_{\mathsf{control}} \langle List\rangle_{\mathsf{callStack}} \Big\rangle_{\mathsf{local}} \end{array} \right\rangle_{\mathsf{T}}$$

Figure 1. Subset of the C Configuration

with input), deduction consists of the application of the rules to that term. This yields a transition system for any program.

For the purposes of this paper, the $\mathbb{K}$ formalism can be regarded as a front-end to RL designed specifically for defining languages. In $\mathbb{K}$, parts of the state are represented as labeled, nested multisets, as seen in Figure 1. 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., $\mathsf{callStack}$), etc. The configuration shown in Figure 1 is a subset of the real C configuration we use, which contains over 90 such cells. As this is all best understood through an example, let us consider a typical rule for a simple imperative language (see Section 4.1.2 for the equivalent rule in C) for finding the address of a variable:

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

We see here three cells: k, env, and mem. The k cell represents a list (or 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. The mem cell is a map of locations to their values. The rule above 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 (again, see Figure 1) 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 "$\cdots$". For example, 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—in the above case, 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:

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

Items in the k cell are separated with "$\curvearrowright$", which can now be seen. The κ and ρ_1, ρ_2, σ_1, σ_2 take the place of the "$\cdots$" above. The most important thing to notice is that 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 control and local (Figure 1). These intervening cells are automatically inferred in $\mathbb{K}$, which keeps the rules more modular.

3.2 Semantics of C

In our sister paper, we gave a complete formal semantics to C. This semantics covered the entire freestanding C99 feature set and passed 99.2% of the GCC torture tests, a regression test suite used by GCC. This was higher than the GCC and Clang compilers, and only one test short of the ICC compiler. We were able to test the semantics so easily because it is executable. With a simple wrapper script around the semantics, we made the definition act like a typical C compiler, which we call `kcc`. `kcc` can run defined programs:

```
$ kcc helloworld.c
$ ./a.out
Hello world
```

and can also report on undefined programs. If we take the third example in Section 2.3 and run it in `kcc` we get:

```
ERROR! KCC encountered an error.
==============================================
Error: 00016
Description: Unsequenced side effect on scalar
object with side effect of same object.
==============================================
Function: main
Line: 3
```

We also showed how the semantics was useful for exploring program behaviors via search or model-checking tools derived automatically from the semantics. Despite the success of this venture, our previous work focused on the *positive semantics*, i.e., the semantics of correct programs, and only touched on the *negative semantics*, i.e., the rules identifying undefined programs. In fact, at least as much of our time was spent tailoring our semantics to catch undefinedness as was spent giving semantics to correct programs.

Giving rules for correct programs is often not enough to catch undefined ones. We give a number of examples of this in the next section, but here is a simple one to give the idea. In C, arrays must have length at least 1 [11, §6.7.6.2:1&5]. However, without taking this fact into consideration, it is easy to give semantics to arrays of any non-negative length, simply by allowing the size to be any natural number. If they would be used at runtime, the problem would be detected, but simply declaring them would slip through. We had precisely this problem in earlier versions of our semantics. To detect this problem, the semantics needs an additional constraint on top of allowing any natural—it must also be non-zero. Again, this additional check is not needed for correct programs—it is only needed to detect programs that are undefined.

4. Semantics-Based Undefinedness Checking

As we explained in Section 2.7, most languages have some form of undefined, or at least exceptional, behavior. When formalizing such languages, this behavior needs to be formalized as well. We were faced with this problem when developing our formal semantics for C [6]. At first we believed that detecting undefinedness using a semantics would simply be a matter of running the program using the semantics and letting it get stuck where there was no semantic rule for a behavior. We have come to realize that in fact, quite a lot of work needs to go on to enable these behaviors to be caught.

In this section we explain a number of techniques for dealing with undefinedness semantically. In doing so, we address most of the issues used as examples in Section 2.

4.1 Using Side Conditions and Checks to Limit Rules

By bolstering particular rules with side conditions, we can catch some undefined behavior. We have employed this technique in our semantics to catch much of the undefined behavior we are capable of catching.

4.1.1 Division by Zero

The simplest example of using side conditions to catch undefined behavior is in a division. In C, the following unconditional rule gives the semantics of integer division for correct programs:

$$\left\langle \frac{I \,/\, J}{I \,/_{Int}\, J} \cdots \right\rangle_{\mathsf{k}}$$

Of course, this rule is not good for programs that *do* divide by zero. In such a case, the rule might turn "$/_{Int}$" into a constructor for integers, where suddenly terms like $5 \,/_{Int}\, 0$ are introduced into the semantics. Programs like:

```
int main(void){
  5/0;
  return 0;
}
```

might actually be given complete meanings without getting stuck, because the semicolon operator throws away the value computed by its expression.

One way to solve this issue is simply by adding a side condition on the division rule requiring "$J \neq 0$". This will cause the rule to only define the defined cases, and let the semantics get stuck on the undefined case. In addition, human-readable error messages, like the one shown in Section 3.2, can be obtained by inverting the side conditions preventing undefined behavior for occurring:

$$\left\langle \frac{I \,/\, J}{\text{reportError(“Division by zero”)}} \cdots \right\rangle_{\mathsf{k}} \quad \text{when } J = 0$$

4.1.2 Dereferencing

Another example, the dereferencing rule, defined in its most basic form is:

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

This rule says that dereferencing a location L of type pointer-to-T ($L : \text{ptrType}(T)$) yields an lvalue L of type T ($[L] : T$). This rule is completely correct according to the semantics of C [11, §6.5.3.2:4] in that it works for any defined program. However, it fails to detect undefined programs including dereferencing `void` [11, §6.3.2.1:1] or null [11, §6.3.2.3:3] pointers. In a program like:

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

this rule would apply to `*NULL`, then the result (`[NULL]` : `void`) would be immediately thrown away (according to the semantics of "`;`"). The program would then return 0 and completely miss the fact that the program was undefined.

In order to catch these undefined behaviors, it could be rewritten as:

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

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.

One major downside with this technique is in making rules more complicated and more difficult to understand. For complex side conditions involving multiple parts of the state, including cells not otherwise needed by the positive rule, this is a big problem. To take pointer dereferencing again as an example, we also want 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 is the safest (and most verbose) version of the rule:

(DEREF-SAFEST)
$$\langle \frac{*(\text{sym}(B) + O : \text{ptrType}(T))}{[\text{sym}(B) + O] : T} \cdots \rangle_{\mathsf{k}} \langle \cdots B \mapsto \text{obj}(Len, \text{—}) \cdots \rangle_{\mathsf{mem}}$$
$$\text{when } T \neq \texttt{void} \wedge O < Len$$

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 $\text{sym}(B)+O$, which is explained in detail in Section 4.3.1. The rule has become much more complicated. The beauty and simplicity of the original semantic rule has been erased, simply to catch undesirable cases.

A slight variation involves embedding the safety checks in the main computation. This is useful when the safety condition is complicated or involves other parts of the state. The above rule can be rewritten as two rules like so:

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

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

The DEREF-SAFEST-EMBEDDED rule could be rewritten to use a side condition, but this would require passing the entire context (in particular, memory) to the helper-function as an argument. This works, but the rule becomes artificially complex.

4.2 Storing Additional Information

It is not enough to add new rules or side conditions to existing rules if the semantics does not keep track of all the pertinent data to be used in the specifications.

4.2.1 Unsequenced Reads and Writes

As explained in Section 2.3, unsequenced writes or an unsequenced write and read of the same object is undefined. In order to catch this in our semantics, we keep track of all the locations that have been written to since the last sequence point in a set called $\mathsf{locsWrittenTo}$ (see Figure 1). Whenever we write to or read from a location, we first check this set to make sure the location had not previously been written to:

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

$$\langle \frac{\text{readByte}}{\text{readByte'}} (Loc) \cdots \rangle_{\mathsf{k}} \langle S \rangle_{\mathsf{locsWrittenTo}} \quad \text{when } Loc \notin S$$

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 $\mathsf{locsWrittenTo}$ set:

$$\langle \frac{\text{seqPoint}}{\cdot} \cdots \rangle_{\mathsf{k}} \langle \frac{S}{\cdot} \rangle_{\mathsf{locsWrittenTo}}$$

4.2.2 Const-Correctness

Another example of needing to keep additional information is specifying `const`-correctness. In C, a type can have the type qualifier `const`, meaning it is unchangeable after initialization. Writes can only occur through non-`const` types [11, §6.3.2.1:1, §6.5.16:1]. For correct programs, this modifier can be completely ignored, since it is only there to help the programmer catch mistakes. To actually catch those mistakes requires the semantics to keep track of `const` modifiers and to check them during all modifications and conversions.

One might think that it is possible to soundly and completely check for `const`-correctness statically—after all, it is generally not allowed to drop qualifiers on pointers [11, §6.3.2.3:2], meaning one cannot simply write code like this:

```
const char p[] = "hello";
char *q = (char*)p;
```

With this in mind, one could check that no `const`s are dropped in conversion and no writes occur through `const` types. However, this is not sufficient—there are ways around the conversion, such as this:

```
const char p[] = "hello";
char *q = strchr(p, p[0]); // removes const
```

The `strchr` library function

```
char *strchr(const char *s, int c);
```

returns a pointer to the first instance of `c` in `s`. By calling it with `p` and `p[0]`, this function returns a pointer to the same string, but without the `const` modifier. This is completely defined by itself, but if a write occurs through pointer `q`, the standard says that it is undefined [11, §6.7.3:6]. We handle this in our semantics by marking memory that was defined with `const` by placing these locations into a set named notWritable, then checking this fact during subsequent writes:

$$\langle\, \underset{\text{writeByte''}}{\underline{\text{writeByte'}\ (Loc, V)}} \cdots\rangle_{\mathsf{k}}\ \ \langle S\,\rangle_{\mathsf{notWritable}} \qquad \text{when } Loc \notin S$$

4.3 Symbolic Behavior

Through the use of symbolic execution, we can further enhance the above idea by expanding the behaviors that we consider undefined, while maintaining the good behaviors. While some of the symbolic behavior was described in our sister paper [6, §6.2.2], we explain it in more depth here.

4.3.1 Memory Locations

We treat pointers not as concrete integers, but as symbolic values. By symbolic values, we mean base/offset pairs, which we write as sym(B) + O, where B corresponds to the base address of an object itself, while 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 $B < B'$ [11, §6.5.8:5]. Our memory then is a map from base addresses to blocks of bytes. Each base address represents the memory of a single object.

This is the same technique used by Blazy and Leroy [1] and by Roșu et al. [26]. It takes advantage of the fact that addresses of local variables and memory returned from allocation functions like `malloc()` are unspecified [11, §7.20.3]. However, there are a number of restrictions on many addresses, such as the elements of an array being completely contiguous and the fields in a struct being ordered (though not necessarily contiguous).

For example, take the following program:

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

If we gave objects concrete, numerical addresses, then they would always be comparable. However, this piece of code is actually undefined according to the standard [11, §6.5.8:5]. Symbolic locations that are actually base/offset pairs allow us to detect this program as problematic. We only give semantics to pointer comparisons when the two addresses share a common base. For example:

$$\langle\, \underset{1:\texttt{int}}{\underline{(\text{sym}(B) + O : T) < (\text{sym}(B) + O' : T)}}\ \cdots\rangle_{\mathsf{k}} \qquad \text{when } O < O'$$

$$\langle\, \underset{0:\texttt{int}}{\underline{(\text{sym}(B) + O : T) < (\text{sym}(B) + O' : T)}}\ \cdots\rangle_{\mathsf{k}} \qquad \text{when } O \not< O'$$

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. If we take the following code instead:

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

the addresses of `a` and `b` are guaranteed to be in order [11, §6.5.8:5], and in fact our semantics finds the comparison to be true because the pointers share a common base.

4.3.2 Storing Pointers

Another example of the use of symbolic terms in our semantics is how we store pointers in memory. Because all data must be split into bytes to be stored in memory, the same must happen with pointers stored in memory. However, because our pointers are not actual numbers, they cannot be split directly; instead, we split them symbolically. Assuming a particular pointer sym(B) + O was four bytes long, it is split into the list of bytes:

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

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. This program demonstrates its utility:

```
int main(void) {
  int x = 5, y = 6;
  int *p = &x, *q = &y;
  char *a = (char*)&p, *b = (char*)&q;
  a[0] = b[0]; a[1] = b[1]; a[2] = b[2];
  // *p is not defined yet
  a[3] = b[3]; // needs all bytes
  return *p; // returns 6
}
```

Any particular byte-splitting mechanism would mean over-specification—a user could take advantage of it to run code that is not necessarily defined.

4.3.3 Indeterminate Memory

Another example can be seen when copying a struct one byte at a time (as in a C implementation of `memcpy()`); every byte needs to be copied, even uninitialized fields (or padding), and no error should occur [11, §6.2.6.1:4]. Because of this, our semantics must give it meaning. Using concrete, perhaps arbitrary, values to represent unknowns would mean missing some incorrect programs, so we use symbolic values that allow reading and copying to take place as long as the program never uses those uninitialized values in undefined ways. We store these unknown bytes in memory as unknown(N) where N is the number of unknown bits.

In C99, unknown values are generally not allowed to be used under the possibility that they may produce a trap (an error) [9, §6.2.6.1:5]. Similarly, in our semantics, such unknown bytes may not be used by most operations. However, exceptions are made when using an unsigned-character type [9, §6.2.6.1:3–4]—this special case is represented in our semantics by an additional rule allowing such an unknown value to be read by lvalues of the allowed type.

4.4 A Semantics-Based Undefinedness Checker

By using the three techniques described above, we improved our formal semantics of C [6] into a tool capable of recognizing a wide range of undefined behaviors. While the original semantics was capable of catching a handful of undefined behaviors, in general each additional behavior we caught involved a reworking of at least one semantic rule.

Our tool is capable of detecting undefined behaviors simply by running them through the semantics. As described in Section 3.2, this is done using a wrapper, mimicking GCC, we built around the semantics. We report on the capabilities of this tool as compared to other analysis tools in Section 5.

4.5 Suggested Semantic Styles for Undefinedness

In this section, we suggest two new specification techniques for capturing undefined or exceptional behavior based on our experience in capturing undefinedness in C. These are untested (we know of no semantic framework incorporating them), but we think they would make expressing undefined behavior much more straightforward.

4.5.1 Inclusion/Exclusion Rules

One nice way to specify exceptional behavior would be to define additional “negative” semantic rules to catch the special cases. For example, in addition to the DEREF rule given earlier, add the following two rules:

(DEREF-NEG1)

$$\langle \frac{*\,(L : \text{ptrType}(\texttt{void}))}{\text{reportError(“Cannot dereference }\texttt{void}\text{ pointers”)}} \cdots\rangle_{\mathsf{k}}$$

(DEREF-NEG2)

$$\langle \frac{*\,(\texttt{NULL} : \text{ptrType}(T))}{\text{reportError(“Cannot dereference null pointers”)}} \cdots\rangle_{\mathsf{k}}$$

For this definitional strategy to make sense, later rules must be applied before earlier rules. Each additional rule acts as a refinement on the previous rule. Simply having multiple rules is much cleaner than rules with side conditions—it allows the primary, unexceptional case to be emphasized because it is presented without side conditions. However, this strategy trades off the complexity of side conditions for the complexity of rule precedence.

It is possible for rule precedence to be supported by a semantic framework as syntactic sugar, where it automatically adds side conditions necessary to prevent earlier rules from executing first. It should be clear that one could hand-write these side conditions, but the whole point of this strategy is to avoid explicit side conditions in order to make the rules simpler.

It is not enough to consider exploring the transition system for these reportError states, since this mechanism is also useful for defined but exceptional behavior. In such cases, if one were to allow the rules to apply in any order and then analyze the resulting transition system, it would be difficult to identify which paths should be removed. Consider the following three rules for division:

$$\langle \frac{I\,/\,J}{I\,/_{Real}\,J} \cdots\rangle_{\mathsf{k}} \quad\leftarrow\quad \langle \frac{I\,/\,0}{\textsc{Infinity}} \cdots\rangle_{\mathsf{k}} \quad\leftarrow\quad \langle \frac{0\,/\,0}{\textsc{NaN}} \cdots\rangle_{\mathsf{k}}$$

These rules should be tried right to left until one matches. They are similar to the rules of IEEE-754 floating point, which evaluates division by zero to INFINITY or NAN values.

4.5.2 Declarative Specification

An additional possibility is to again start with only the original positive semantic rule, but then to add declarative specifications on top of that. For example, using LTL and configuration patterns, we could specify both

$$\Box\neg\langle *\,(L : \text{ptrType}(\texttt{void})) \cdots\rangle_{\mathsf{k}}$$
$$\text{and } \Box\neg\langle *\,(\texttt{NULL} : \text{ptrType}(T)) \cdots\rangle_{\mathsf{k}}$$

The first property states that it is never the case that the next action to perform is dereferencing a `void` pointer. The second property states that it is never the case that the next action to perform is dereferencing a null pointer. Using a temporal logic to add these negative “axioms” to the semantics has the advantage of being able to capture undefined behavior that might only occur on one path in the transition system. For example, this property states that read-write data races are not allowed:

$$\Box\neg\,(\langle\text{read}(L,T)\cdots\rangle_{\mathsf{k}}\ \langle\text{write}(L',T',V)\cdots\rangle_{\mathsf{k}}\ )$$
$$\text{when overlaps}((L,T),(L',T'))$$

5. Evaluation

In this section we evaluate the semantics-based approach against special-purpose analysis tools. To do so, we explain our testing methodology, which includes a third-party suite of undefined tests as well as a suite of tests we developed.

5.1 Third Party Evaluation

In order to evaluate our analysis tool, 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. Below we briefly mention work we encountered that may evolve into or develop a complete suite in the future, as well as one suite that we use as a partial undefinedness benchmark.

5.1.1 Related Test Suites

There is a proposed ISO technical specification for program analyzers for C [12], suggesting programmatically enforceable rules for writing secure C code. Many of these rules involve avoiding undefined behavior; however, the specification only focuses on statically enforceable rules. The above classification is similar to MISRA-C [19], 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 code 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, most of the MISRA tests test only statically detectable undefined behavior. The CERT C Secure Coding Standard [28] and MITRE's "common weakness enumeration" (CWE) classification system [20] are other similar projects, identifying many causes of program error and cataloguing their severity and other properties. The projects mentioned above include many undefined behaviors—for example, the undefinedness of signed overflow [11, §6.5:5] corresponds to CERT's INT32-C and to MITRE CWE-190.

5.1.2 Juliet Test Suite

NIST has released a suite of tests for security 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 of which triggers one of the 116 different CWEs supported by the suite. Most of the tests (~70%) are C and not C++. However, again the Juliet tests focus on statically detectable violations, and 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 and have submitted the list to NIST. We have not heard back from them yet. We also wrote a small script automating the process of extracting and in some cases fixing the tests, available at `http://code.google.com/p/c-semantics/source/browse/trunk/tests/juliet/clean.sh`.

This extraction gave us 4113 tests, with about 96 source lines of code (SLOC) per test (179 SLOC with the helper-library linked in). 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 a number of analysis tools, including our own semantics-based tool `kcc`. These tools include Valgrind [22],[7] CheckPointer [17],[8] and the Value Analysis plugin for Frama-C [2].[9] Although the Juliet tests are designed to exercise static analysis tools, all of the tools we tested can be considered dynamic analysis tools.[10] The results of this benchmark can be seen in Figure 2. Valgrind, and Value Analysis each took, on average, 0.5 s to run the tests, `kcc` took 23 s, and CheckPointer took 80 s. CheckPointer has a large, fixed startup time as it is mainly used to check large software projects, not 100 line programs.

Based on initial results of these tests, we improved our tool to catch precisely those behaviors we were missing. We also contacted the authors of the Value Analysis plugin with their initial results, and they were able to patch their tool within a few days to do the same thing. Because not all tools had this opportunity, the test results should not be taken as any kind of authoritative ranking, but instead suggest some ideas. First, no tool was able to catch behaviors accurately unless they specifically focused on those behaviors. This reaffirms the idea that undefinedness checking does not simply come for free (e.g., by simply leaving out cases), but needs to be studied and understood specifically. For example, Valgrind does not try to detect division by zero or integer overflow, and CheckPointer was not designed to detect division by zero, uninitialized memory, or integer overflow. This shows up very clearly in the test results. Second, tools were able to improve performance simply by looking at concrete failing tests and adapting their techniques. As an example, on its initial run on the Juliet tests, `kcc` only caught about 93%.

[7] v. 3.5.0, `http://valgrind.org/`

[8] v. 1.1.5, `http://www.semdesigns.com/Products/MemorySafety/`

[9] v. Nitrogen-dev, `http://frama-c.com/value.html`

[10] Frama-C's value analysis can be used in "C interpreter" mode [3, §2.1].

		Tools (% passed)			
Undefined Behavior	No. Tests	Valgrind	CheckPointer	V. Analysis	`kcc`
Use of invalid pointer	3193	70.9	89.1	100.0	100.0
Division by zero	77	0.0	0.0	100.0	100.0
Bad argument to `free()`	334	100.0	99.7	100.0	100.0
Uninitialized memory	422	100.0	29.3	100.0	100.0
Bad function call	46	100.0	100.0	100.0	100.0
Integer overflow	41	0.0	0.0	100.0	100.0

Figure 2. Comparison of analysis tools on Juliet Test Suite

These ideas mean it is critical that undefinedness benchmarks continue to be developed and used to refine analysis tools. Both we and the Value Analysis team found the Juliet tests useful in improving our tools; in many cases, they gave concrete examples of missing cases that were otherwise hard to identify. The identification, together with the techniques described in Section 4, enabled us to adapt our tools to catch every behavior in the suite.

5.2 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. This involved first trying to understand the behaviors, and then constructing test cases corresponding to each behavior.

5.2.1 Our Classifications

To help develop our test suite, we first tried to understand the undefined behaviors listed in the standard. Part of this involves classifying the behaviors into categories depending on difficulty. For example, the standard says: "The (nonexistent) value of a `void` expression (an expression that has type `void`) shall not be used in any way, and implicit or explicit conversions (except to `void`) shall not be applied to such an expression" [11, §6.3.2.2:1]. Depending on how one interprets the word "use", this could be a static or dynamic restriction. If static, the code:

```
if (0) { (int)(void)5; }
```

is undefined according to §6.3.2.2:1; if dynamic, it is defined since the problematic code can never be reached. The intention behind the standard[11] appears to be that, in general, situations are made statically undefined if it is not easy to generate code for them. Only when code can be generated, then the situation can be undefined dynamically. In the above example, it is hard to imagine code being generated for `(int)(void)5`, so we can conclude this is meant to be statically undefined. When there was any confusion as to the static/dynamic nature of any of the behaviors, we use the above assumption.

[11] Private correspondence with committee member.

We found that the majority of the categories of undefined behavior in C are dynamic in nature. Out of 221 undefined behaviors, 92 are statically detectable and 129 are only dynamically detectable. Because the argument for the undecidability of detecting undefinedness (Section 2.6) does not depend on the particular dynamic behavior, detecting any dynamic behavior is equally hard. This does not apply to the static behaviors, as they are undefined for static reasons and are not subject to particular control flows.

5.2.2 Our Test Suite

An ideal test suite for undefined behaviors involves individual tests for each of the 221 undefined behaviors. Some behaviors require multiple tests, e.g., "If the specification of a function type includes any type qualifiers, the behavior is undefined." [11, §6.7.3:9] requires at least one test for each qualifier. Ideally the tests would also include control-flow, data-flow, and execution-flow variations in order to make static analysis more difficult.

As we discussed in Section 2.4, dynamic undefined behavior on a reachable path (or any statically undefined behavior) causes the entire program to become undefined. This means that each test in the test suite needs to be a separate program, otherwise one undefined behavior may interact with another undefined behavior. In addition, each test should come with a corresponding defined test. This "control" test makes it possible to identify false-positives in addition to false-negatives. Without such tests, a tool could simply say all programs were undefined and receive full marks.

Our suite currently includes 178 tests covering 70 of the undefined behaviors. We hope it will serve as a starting point for the development of a larger, more comprehensive test. Our suite focuses almost entirely on the non-library behaviors, and specifically on the dynamic behaviors therein. It includes at least one test for each of the 42 dynamically undefined behaviors relating to the non-library part of the language that are not also implementation-specific. We have made our test suite and categorization available for download at `http://code.google.com/p/c-semantics/downloads/`.

These tests are much broader than the Juliet tests, covering 70 undefined behaviors as opposed to the 6 covered by the Juliet tests. However, each behavior is tested shallowly, with

Tools	Static (% Passed)	Dynamic (% Passed)
Valgrind	0.0	2.3
V. Analysis	1.6	45.3
CheckPtr.	2.4	13.1
kcc	44.8	64.0

Figure 3. Comparison of analysis tools against our tests. These averages are across undefined behaviors, and no behavior is weighted more than another.

only 2 tests per behavior on average. Some of the dynamic behaviors it tests that the Juliet suite does not include:

- If the program attempts to modify [a character string literal], the behavior is undefined. [11, §6.4.5:7]
- An object shall have its stored value accessed only by an lvalue expression that has [an allowed type]. [11, §6.5:7]
- When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object. [11, §6.5.6:9]
- If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. [11, §6.5:2]

There are many other such behaviors tested, and all are equally bad from the C standard's perspective. They can all cause a compiler to generate unexpected code or cause a running program to behave in an unexpected way.

We compared the same tools as before using our own custom made tests. The results can be seen in Figure 3. It is clear that the tools focusing on a few common undefined behaviors (Valgrind and CheckPointer) only detect a small percentage of behaviors. Both Value Analysis and kcc, which were designed to catch a large number of undefined behaviors, were able to catch a much larger number of dynamic behaviors, and in the case of kcc, many of the static behaviors as well.

6. Conclusion

In this paper we investigated undefined behaviors in C and how one can capture these behaviors semantically. We discussed three techniques for formally describing undefined behaviors. We also used these techniques in a semantics-based analysis tool, which we tested against other popular analysis tools. We compared the tools on a test suite of our own devising, which we are making publicly available, as well as on another publicly available test suite.

We hope that this work will bring more attention to the problem of undefined behavior in program verification. Undefined programs may behave in *any* way and undefinedness is (in general) undecidable to detect; this means that undefined programs are a serious problem that needs to be addressed by analysis tools. Whether this is through semantic means or some other mechanism, tools to verify the absence of undefined behavior are needed on the road to fully verified software.

References

[1] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning*, 43(3): 263–288, 2009. 8

[2] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *9th Intl. Working Conf. on Source Code Analysis and Manipulation*, SCAM'09, pages 123–124. IEEE, 2009. 10

[3] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, April 2012. 10

[4] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *ICSE'12*, 2012. To appear. 1

[5] M. Dominus. Undefined behavior in Perl and other languages, October 2007. URL http://blog.plover.com/prog/perl/undefined.html. 5

[6] C. Ellison and G. Roșu. An executable formal semantics of C with applications. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012. 1, 5, 6, 8, 9

[7] E. R. Gansner and J. H. Reppy. The Standard ML basis library, 2004. URL http://www.standardml.org/Basis/. 5

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005. 5

[9] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:1999: Programming languages—C. Committee Draft N1256, Intl. Org. for Standardization, December 1999. 9

[10] ISO/IEC JTC 1, SC 22, WG 14. Rationale for international standard—programming languages—C. Technical Report 5.10, Intl. Org. for Standardization, April 2003. 2

[11] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:2011: Programming languages—C. Committee Draft N1570, Intl. Org. for Standardization, August 2011. 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12

[12] ISO/IEC JTC 1, SC 22, WG 14. C secure coding rules. Committee Draft N1579, Intl. Org. for Standardization, September 2011. 10

[13] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33: 26–76, 1998. 5

[14] C. Lattner. What every C programmer should know about undefined behavior, May 2011. URL http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html. 2

[15] C. Lattner. LLVM assembly language reference manual, February 2012. URL http://llvm.org/docs/LangRef.html. 5

[16] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. 1, 4

[17] M. Mehlich. CheckPointer—A C memory access validator. In *11th Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM'11)*, pages 165–172. IEEE, 2011. 10

[18] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. 5

[19] MISRA Consortium. MISRA-C: 2004—Guidelines for the use of the C language in critical systems. Technical report, MIRA Ltd., October 2004. 10

[20] MITRE Corporation. The common weakness enumeration (CWE) initiative, 2012. URL `http://cwe.mitre.org/`. 10

[21] T. Nagel. Troubles with GCC signed integer overflow optimization, January 2010. URL `http://thiemonagel.de/2010/01/signed-integer-overflow/`. 3

[22] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. 10

[23] NIST. Juliet test suite for C/C++, December 2010. URL `http://samate.nist.gov/SRD/testsuite.php`. 10

[24] J. Regehr. A guide to undefined behavior in C and C++, July 2010. URL `http://blog.regehr.org/archives/213`. 2

[25] G. Roșu and T. F. Șerbănuță. An overview of the K semantic framework. *J. Logic and Algebraic Programming*, 79(6):397–434, 2010. 5

[26] G. Roșu, W. Schulte, and T. F. Șerbănuță. Runtime verification of C memory safety. In *Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 132–152, 2009. 8

[27] Ruby Standardization WG. Programming languages—Ruby. Draft, Information-technology Promotion Agency, August 2010. 5

[28] R. C. Seacord and J. A. Rafail. The CERT C secure coding standard, 2008. 10