POPL'12, ACM, pp 533-544. 2012

# An Executable Formal Semantics of C with Applications *

Chucky Ellison Grigore Roșu
University of Illinois
{celliso2, grosu}@illinois.edu

## Abstract

This paper describes an executable formal semantics of C. Being executable, the semantics has been thoroughly tested against the GCC torture test suite and successfully passes 99.2% of 776 test programs. It is the most complete and thoroughly tested formal definition of C to date. The semantics yields an interpreter, debugger, state space search tool, and model checker "for free". The semantics is shown capable of automatically finding program errors, both statically and at runtime. It is also used to enumerate nondeterministic behavior.

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

***General Terms*** Languages, Standardization, Verification.

## 1. Introduction

C provides just enough abstraction above assembly language for programmers to get their work done without having to worry about the details of the machines on which the programs run. Despite this abstraction, C is also known for the ease in which it allows programmers to write buggy programs. With no runtime checks and little static checking, in C the programmer is to be trusted entirely. Despite the abstraction, the language is still low-level enough that programmers *can* take advantage of assumptions about the underlying architecture. Trust in the programmer and the ability to write *non*-portable code are actually two of the design principles under which the C standard was written [14]. These ideas often work in concert to yield intricate, platform-dependent bugs. The potential subtlety of C bugs makes it an excellent candidate for formalization, as subtle bugs can often be caught only by more rigorous means.

In this paper, we present a formal semantics of C that can be used for finding bugs. Rather than being an "on paper" semantics, it is executable, machine readable, and has been tested against the GCC torture tests (see Section 5). The semantics describes the features of the C99 standard [13], but we often cite the text from the proposed C1X standard [15]. We use the C1X text because it will eventually supersede the C99 standard, and because it offers clearer wording and more explicit descriptions of certain kinds of behavior.

Our semantics can be considered a *freestanding* implementation of C99. The standard defines a freestanding implementation as a version of C that includes every language feature except for `_Complex` and `_Imaginary` types, and that includes only a subset of the standard library. Our semantics is the first arguably complete dynamic semantics of C (see Section 2).

Above all else, our semantics has been motivated by the desire to develop formal, yet practical tools. Our semantics was developed in such a way that the single definition could be used immediately for interpreting, debugging, or analysis (described in Section 6). At the same time, this practicality does not mean that our definition is not formal. Being written in a subset of rewriting logic (RL), it comes with a complete proof system and initial model semantics [18]. Briefly, a rewrite system is a set of rules over terms constructed from a signature. The rewrite rules match and apply everywhere, making RL a simple, uniform, and general formal computational paradigm. This is explained in greater detail in Section 3.

Our C semantics defines 150 C syntactic operators. The definitions of these operators are given by 1,163 semantic rules spread over 5,884 source lines of code (SLOC). However, it takes only 77 of those rules (536 SLOC) to cover the behavior of statements, and another 163 for expressions (748 SLOC). There are 505 rules for dealing with declarations and types, 115 rules for memory, and 189 technical rules defining helper operators. Finally, there are 114 rules for the core of our standard library. The semantics itself is described in more detail in Section 4, and is available in its entirety at `http://c-semantics.googlecode.com/`.

***Contributions*** The specific contributions of this paper include:

- a detailed comparison of other C formalizations;
- the most comprehensive formal semantics of C to date, which is executable and has been thoroughly tested;
- demonstrations as to its utility in discovering program flaws;
- constructive evidence that rewriting-based semantics scale.

***Features*** Our semantics captures every feature required by the C99 standard. We include a partial list here to give an idea of the completeness, and explain any shortcomings in Section 7. All aspects related to the below features are included and are given a direct semantics (not by a translation to other features):

- Expressions: referencing and dereferencing, casts, array indexing (`a[i]`), structure members (`->` and `.`), arithmetic, bitwise, and logical operators, sizeof, increment and decrement, assignments, sequencing (`_,_`), ternary conditional (`_?_:_`);
- Statements: `for`, `do-while`, `while`, `if`, `if/else`, `switch`, `goto`, `break`, `continue`, `return`;
- Types and Declarations: `enum`s, `struct`s, `union`s, bitfields, initializers, static storage, `typedef`s, variable length arrays;
- Values: regular scalar values (signed/unsigned arithmetic and pointer types), `struct`s, `union`s, compound literals;
- Standard Library: `malloc`/`free`, `set`/`longjmp`, basic I/O;
- Conversions: (implicit) argument and parameter promotions and arithmetic conversion, and (explicit) casts.

* Supported in part by NSA contract H98230-10-C-0294 and by (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.



*POPL'12,* January 25–27, 2012, Philadelphia, PA, USA.

## 2. Comparison with Existing Formal C Semantics

There have already been a number of formal semantics written for C. One might (rightfully) ask, "Why yet another?" We claim that the definitions so far have either made enough simplifying assumptions that for many purposes they are not C, or have lacked any way to use them other than on paper. While "paper semantics" are useful for teaching and understanding the language, we believe that without a mechanized definition, it is difficult to gain confidence in a definition's appropriateness for any other purpose. Below we highlight the most prominent definitions and explain their successes and shortcomings in comparison with our work.

***Gurevich and Huggins (1993)*** One of the earliest formal descriptions of ANSI C is given by Gurevich and Huggins [11], using abstract state machines (ASMs) (then known as evolving algebras). Their semantics describes C using four increasingly precise layers, each formal and analyzable. Their semantics covers all the high-level constructs of the language, and uses external oracles to capture the underspecification inherent in the definition of C. Their semantics was written without access to a standard, and so is based on Kernighan and Ritchie [17]. However, many behavioral details of the lowest-level features of C are now partially standardized, including details of arithmetic, type representation, and evaluation strategies. The latter has been investigated in the context of ASMs [36], but none are present in the original definition. Based on our own experience, the details involving the lowest-level features of C are incredibly complex (see Section 3.2), but we see no reason why the ASM technique could not be used to specify them.

Their semantics was never converted into an executable tool, nor has it been used in applications. However, their purpose and context was different from ours. As pointed out elsewhere [22, p. 11], their semantics was constructed without the benefit of any mechanization. According to Gurevich,[1] their purpose was to "discover the structure of C," at a time when "C was far beyond the reach of denotational semantics, algebraic specifications, etc."

***Cook, Cohen, and Redmond (1994)*** Soon after the previous definition, Cook et al. [5] describe a denotational semantics of C90 using a custom-made temporal logic for the express purpose of proving properties about C programs. Like us, they give semantics for particular implementation-defined behaviors in order to have a more concrete definition. These choices are then partitioned off so that one could, in theory, choose different implementation-defined values and behaviors.

They have given at least a basic semantics to most C constructs. We say "at least" without malicious intent—although their work was promising, they moved on to other projects before developing a testable version of their semantics and without doing any concrete evaluation.[1] Additionally, no proofs were done using this semantics.

***Cook and Subramanian (1994)*** The related work of Cook and Subramanian [4, 33] is a semantics for a restricted subset of C, based loosely on the semantics above. This semantics is embedded in the theorem prover Nqthm [2] (a precursor to ACL2). They were successful in verifying at least two functions: one that takes two pointers and swaps the values at each, and one that computes the factorial. They were also able to prove properties about the C definition itself. For example, they prove that the execution of `p = &a[n]` puts the address of the nth element of the array `a` into `p` [4, p. 122]. Their semantics is, at its roots, an interpreter—it uses a similar technique to that described by Blazy and Leroy [1] to coax an interpreter from recursive functions—but there is no description in their work of any reference programs they were capable of executing. As above, it appears the work was terminated before it was able to blossom.

[1] Personal communication, 2010.

***Norrish (1998)*** The next major semantics was provided by Norrish [22], who gives both static and dynamic formal semantics inside the HOL theorem proving system for the purpose of verifying C programs (later extended to C++ [23]). His semantics is in the Structural Operational Semantics (SOS) style, using small-step for expressions and big-step for statements. One of the focuses of his work is to present a precise description of the allowable evaluation orders of expressions. His semantics still stands as a precise representation of evaluation in C. In Section 6.3 we demonstrate how our definition captures the same behaviors.

Working inside HOL provides an elegant solution to the underspecification of the standard—Norrish can state facts given by the standard as axioms/theorems. To maintain executability, we chose instead to parameterize our definition for those implementation-defined choices. In that respect, our definitions conceptually complement each other—his is better for formal proofs about C, while ours is better for searching for behaviors in programs (see Section 6.3.1). Proofs of program correctness [31] as well as semantics-level proofs [8] have already been demonstrated in the framework used by our semantics, but we have not yet applied these techniques to C.

Norrish uses his definition to prove some properties about C itself, as well as to verify some strong properties of simple ($\leq$ 5 line) programs, but was unable to apply his work to larger programs. His semantics is not executable, so it has not been tested against actual programs. However, the proofs done within the HOL system help lend confidence to the definition.

***Papaspyrou (2001)*** A denotational semantics for C99 is described by Papaspyrou [24, 25] using a monadic approach to domain construction. The definition includes static, typing, and dynamic semantics, which enables him not only to represent the behavior of executing programs, but also check for errors like redefinition of an identifier in the same scope. Papaspyrou, Norrish, and Cook et al. each give a typing semantics in addition to the dynamic semantics, while we and Blazy and Leroy (below) give only dynamic semantics.

Papaspyrou represents his semantics in Haskell, yielding a tool capable of searching for program behaviors. This was the only semantics for which we were able to obtain a working interpreter, and we were able to run it on a few examples. Having modeled expression non-determinism, and being denotational, his semantics evaluates a program into a set of possible return values. However, we found his interpreter to be of limited capability in practice. For example, using his definition, we were unable to compute the factorial of six or the fourth Fibonacci number.

***Blazy and Leroy (2009)*** A big-step operational semantics for a subset of C, called Clight, is given by Blazy and Leroy [1]. While they do not claim to have given semantics for the entirety of C, their semantics does cover most of the major features of the language and has been used in a number of proofs including the verification of the optimizing compiler CompCert.

To help validate their semantics, they have done manual reviews of the definition as well as proved properties of the semantics such as determinism of evaluation. They additionally have verified semantics-preserving transformations from their language into simpler languages, which are easier to develop confidence in. Their semantics is not directly executable, but they describe a mechanism by which they could create an equivalent recursive function that would act as an interpreter. This work has not yet been completed.

Clight does not handle non-determinism or sub-expressions with side effects. However, since publication, they have added a new front-end small-step definition called CompCert C that does handle these features, and is also being used to handle `goto`.[3]

[2] Personal communication, 2010.

[3] Personal communication, 2011.

| | Definition | | | | | | |
|---|---|---|---|---|---|---|---|
| Feature | GH | CCR | CR | No | Pa | BL | ER |
| Bitfields | ● | ◐ | ○ | ○ | ◐ | ○ | ● |
| Enums | ◐ | ● | ○ | ○ | ● | ○ | ● |
| Floats | ○ | ○ | ○ | ○ | ● | ● | ● |
| String Literal | ○ | ● | ○ | ○ | ● | ● | ● |
| Struct as Value | ○ | ○ | ○ | ● | ○ | ○ | ● |
| Arithmetic | ◐ | ● | ● | ○ | ● | ● | ● |
| Bitwise | ○ | ● | ○ | ○ | ● | ● | ● |
| Casts | ◐ | ◐ | ○ | ◐ | ◐ | ● | ● |
| Functions | ● | ● | ◐ | ● | ● | ● | ● |
| Exp. Side Effects | ● | ● | ○ | ● | ● | ○ | ● |
| Break/Continue | ◐ | ● | ◐ | ● | ● | ● | ● |
| Goto | ◐ | ○ | ○ | ○ | ● | ○ | ● |
| Switch | ◐ | ● | ○ | ○ | ● | ◐ | ● |
| Longjmp | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Malloc | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Variadic Funcs. | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Feature | GH | CCR | CR | No | Pa | BL | ER |

●: Fully Described ◐: Partially Described ○: Not Described

*GH* represents Gurevich and Huggins [11], *CCR* is Cook et al. [5], *CR* is Cook and Subramanian [4], *No* is Norrish [22], *Pa* is Papaspyrou [25], *BL* is Blazy and Leroy [1], and *ER* is our work.

**Figure 1.** Dynamic Semantics Features

We condense our study of related works in Figure 1. For interested parties, this chart may be contentious. However, we believe that it is useful, both for developers of formal semantics of C and for users of them, to give a broad (though admittedly incomplete) overview of the state of the art of the formal semantics of C. Also, it may serve as an indication of the complexity involved in the C language, although not all features are equally difficult.

We did our best to give the authors the benefit of the doubt with features they explicitly mentioned, but the other features were based on our reading of their semantics. We have also discussed our views with the authors, where possible, to try and establish a consensus. Obviously the categories are broad, but our intention is to give an overview of some of the more difficult features of C. We purposefully left off any feature that all definitions had fully defined.

Finally, there are a number of other emergent features, such as multi-dimensional arrays, that are difficult to discern correctness through simple inspection of the formal semantics (i.e., without testing or verifying it). It is also difficult to determine if feature pairs work together—for example, does a definition allow bitfields inside of unions? We decided to leave most of these features out of the chart because they are simply too hard to determine if the semantics were complete enough for them to work properly.

## 3. Background

In this section we give a little background on the C standard, including some important definitions. We additionally explain the rewriting formalism we use to give our semantics of C.

### 3.1 C Standard Information

The C standard uses the idea of undefined and partially defined behaviors in order to avoid placing difficult requirements on implementations. It categorizes the particular behaviors of any C implementation that are not fully defined into four categories: unspecified, implementation-defined, undefined, and locale-specific behavior. For the purposes of this paper, we focus on three of these [15, §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.

**undefined behavior** Behavior, upon use of a non-portable or erroneous program construct or [data, with] no requirements.

An example of unspecified behavior is the order in which the arguments to a function are evaluated. An example of implementation defined behavior is the size of an `int`. An example of undefined behavior is referring to an object outside of its lifetime.

To put these definitions in perspective, for a C program to be maximally portable, "it shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior" [15, §4.5]. This is called "strictly conforming". However, programmers use C for many inherently non-portable tasks, such as writing device drivers. The standard offers another level of conformance (called "conforming") where the program may rely on implementation-defined or even unspecified (but never undefined) behavior. Based on this, our definition is parametric in implementation-defined behaviors, and uses symbolic computation to describe unspecified behaviors. As much as possible, this behavior is kept separate from the semantics underlying the high-level (defined for all implementations) aspects of the language. More details about our parameterization are described in Section 4.5, and about our use of symbolic values in Section 6.2.2.

### 3.2 Why Details Matter

It is tempting to gloss over the details of C's arithmetic and other low-level features when giving it a formal semantics. However, C is designed to be translatable to machine languages where arithmetic is handled by any number of machine instructions. The effects of this overloading are easily felt at the size boundaries of the types. It is a common source of confusion among programmers, and so a common source of bugs. Here we give a few examples that reveal even apparently simple C programs can involve complex semantics.

For the purposes of these examples, assume that `int`s are 2 bytes (capable of representing the values −32768 to 32767) and `long int`s are 4 bytes (−2147483648 to 2147483647). Also, unless specified, in C a type is assumed to be signed.[4] In the following program, what value does `c` receive [34, Q3.14]?

```
int a = 1000, b = 1000;
long int c = a * b;
```

One is tempted to say 1000000, but that misses an important C-specific detail. The two operands of the multiplication are `int`s, so the multiplication is done at the `int` level. It therefore overflows ($1000 * 1000 = 1000000 > 32767$), which, according to the C standard, makes the expression undefined.

What if we make the types of `a` and `b` unsigned (0 to 65535)?

```
unsigned int a = 1000, b = 1000;
long int c = a * b;
```

Here, the arithmetic is again performed at the level of the operands, but overflow on *unsigned* types is completely defined in C. The result is computed by simply reducing the value modulo one more than the max value [15, §6.3.1.3:2]. 1000000 mod 65536 gives us 16960.

One last variation—`signed char`s are one byte in C (−128 to 127).[5] What does `c` receive?

```
signed char a = 100, b = 100;
int c = a * b;
```

[4] Except `char`s and bitfields, whose signedness is implementation-defined.

[5] Bytes are only required to be at least 8 bits long.

Since the chars are signed, then based on the first example above the result would seem undefined ($100 * 100 = 10000 > 127$). However, this is not the case. In C, types smaller than `int`s are promoted to `int`s before doing arithmetic. There are essentially implicit casts on the two operands: `int c = (int)a * (int)b;`. Thus, the result is actually 10000.

While the above examples might seem like a game, the conclusion we draw is that it is critical when defining the semantics of C to handle *all* of the details. The semantics at the higher level of functions and statements is actually much easier than at the level of expressions and arithmetic. These issues are subtle enough that they are very difficult to catch just by manually inspecting the code, and so need to be represented in the semantics if one wants to find bugs in real programs. Even though errors related to the above details continue to be found in real compilers [35], previous semantics for C either did not give semantics at this level of detail, or were not suitable for identifying programs that misused these features. This is one of our primary reasons for wanting an executable semantics. We give some of the rules associated to binary arithmetic in Section 4.4.4.

### 3.3 Rewriting Logic and $\mathbb{K}$

To give our semantics, we use a rewriting-based semantic framework called $\mathbb{K}$ [28], inspired by RL [18]. In particular, our semantics is written using the K-Maude tool [32], which takes $\mathbb{K}$ rewrite rules and translates them into Maude [3]. Maude is a rewriting-logic engine that provides facilities for the execution and analysis of rewriting-logic theories.

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 with input), deduction consists of the application of the rules to that term. This yields a transition system for any program. A single path of rewrites describes the behavior of an interpreter, while searching all paths would yield all possible answers in a nondeterministic 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 2. 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., callStack), etc. As this is all best understood through an example, let us consider a typical rule for a simple imperative language (see Section 4.4.2 for the equivalent rule in C) for finding the address of a variable:

$$\frac{\langle \&X \curvearrowright \cdots \rangle_{\mathsf{k}} \ \langle \cdots X \mapsto L \cdots \rangle_{\mathsf{env}}}{L}$$

We see here two cells, k and env. 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 rule above says that if the next thing to be evaluated (which here we call a redex) is the application of the referencing operator (&) to a variable $X$, then one should match $X$ in the environment to find its location $L$ in memory. With this information, one should transform the redex into that location in memory, $L$.

This example exhibits a number of features of $\mathbb{K}$. First, rules only need to mention those cells (again, see Figure 2) relevant to the rule. The rest of the cell infrastructure 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 actually quite useful. The above rule would be written out as a traditional rewrite rule like this:

$$\langle \&X \curvearrowright \kappa \rangle_{\mathsf{k}} \ \langle \rho_1, X \mapsto L, \rho_2 \rangle_{\mathsf{env}} \Rightarrow \langle L \curvearrowright \kappa \rangle_{\mathsf{k}} \ \langle \rho_1, X \mapsto L, \rho_2 \rangle_{\mathsf{env}}$$

Items in the k cell are separated with "$\curvearrowright$", which can now be seen. The $\kappa$ and $\rho_1, \rho_2$ 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 (RHS). 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 2). These intervening cells are automatically inferred in $\mathbb{K}$, which keeps the rules more modular.

Going back to $\mathbb{K}$, we use "$\cdot$" to represent the unit element of any algebraic lists or sets (including the "$\curvearrowright$" list). We also use "—" to stand for a term that we do not care to name. Finally, in order to get the redexes to the top of the k cell (i.e., in order to identify which positions in the syntax tree can be reduced next), the grammar of C is annotated with additional "strictness" annotations. For example, for addition, we say that

$$Exp ::= Exp + Exp \ \ [\text{strict}]$$

meaning that either argument of the addition operator can be taken out for evaluation, nondeterministically. In contrast, the `if` construct looks like this:

$$Stmt ::= \texttt{if (}Exp\texttt{)} \ Stmt \ \ [\text{strict(1)}]$$

indicating that only the first argument can be taken out for evaluation. The two annotations above cause the following six rules to be automatically generated:

$$\frac{\langle E_1 + E_2 \cdots \rangle_{\mathsf{k}}}{E_1 \curvearrowright \square + E_2} \quad \frac{\langle E_1 + E_2 \cdots \rangle_{\mathsf{k}}}{E_2 \curvearrowright E_1 + \square} \quad \frac{\langle \texttt{if (}E\texttt{)}\ S \cdots \rangle_{\mathsf{k}}}{E \curvearrowright \texttt{if (}\square\texttt{)}\ S}$$

$$\frac{\langle V \curvearrowright \square + E_2 \cdots \rangle_{\mathsf{k}}}{V + E_2} \quad \frac{\langle V \curvearrowright E_1 + \square \cdots \rangle_{\mathsf{k}}}{E_1 + V} \quad \frac{\langle V \curvearrowright \texttt{if (}\square\texttt{)}\ S \cdots \rangle_{\mathsf{k}}}{\texttt{if (}V\texttt{)}\ S}$$

Here, $E_1$, $E_2$, and $E$ represent unevaluated expressions and $V$ represents an evaluated expression (i.e., a value). While these are the rules generated by K-Maude, in the theory of $\mathbb{K}$ they can apply anywhere (not just at the top of the k cell). There are additional annotations for specifying more particular evaluation strategies, and can be found in documentation on $\mathbb{K}$ [28]. We also give names to certain contexts that are evaluated differently. For example, the left-hand side (LHS) of an assignment is evaluated differently than the RHS. The use of this is described in Section 4.4.1.

## 4. The Semantics of C in $\mathbb{K}$

In this section, we describe the different components of our definition and give a number of example rules from the semantics.

### 4.1 Syntax

We use the FrontC parser, with additions made and included in CIL [19], an "off-the-shelf" C parser and transformation tool. FrontC itself parses only ANSI C (C90), but CIL extended it with syntax for C99. We use *only* the parser here, and none of the transformations of CIL; we give semantics directly to the abstract syntax tree generated by the parser. The FrontC parser (with C99 extensions) is used by a number of other tools, including CompCert [1] and Frama-C [6].

### 4.2 Configuration (Program + State)

The configuration of a running program is represented by nested multisets of labeled cells, and Figure 2 shows the most important cells used in our semantics. While this figure only shows 17 cells,

$$\left\langle \langle K\rangle_{\mathsf{k}} \left\langle \left\langle \begin{array}{l} \langle Map\rangle_{\mathsf{env}}\ \langle Map\rangle_{\mathsf{types}}\ \langle Map\rangle_{\mathsf{structs}}\ \langle List\rangle_{\mathsf{loopStack}}\ \langle Bag\rangle_{\mathsf{locsWrittenTo}}\ \langle K\rangle_{\mathsf{currFunction}} \\ \langle Map\rangle_{\mathsf{genv}}\ \langle Map\rangle_{\mathsf{gtypes}}\ \langle Map\rangle_{\mathsf{gstructs}}\ \langle Map\rangle_{\mathsf{mem}}\ \langle Map\rangle_{\mathsf{malloced}}\ \langle Map\rangle_{\mathsf{gotoMap}} \end{array} \right\rangle_{\mathsf{control}} \langle List\rangle_{\mathsf{callStack}} \right\rangle_{\mathsf{local}} \right\rangle_{\mathsf{T}}$$

**Figure 2.** Subset of the C Configuration

we use over 60 in the full semantics. The outer T cell contains the cells used during program evaluation: at the top, a k cell contains the current computation itself and a local cell holds a number of cells related to control flow, and below, there are a number of cells dealing with global information.

In the local cell, there is a callstack used for calling and returning from functions, and a control cell which gets pushed onto the call stack. Inside the control cell, there is a local variable environment (env), a local type environment (types), local aggregate definitions (structs), a loop stack, a record of the locations that have been written to since the last sequence point (Section 4.6), and the name of the current function. The cells inside the control cell were separated in this manner because these are the cells that get pushed onto the call stack when making a function call.

Outside the local cell are a number of global mappings, such as the global variable environment (genv), the global type environment (gtypes), global aggregate definitions (gstructs), the heap (mem), the dynamic allocation map (malloced), and a map from function-name/label pairs to continuations (for use by `goto` and `switch`).

### 4.3 Memory Layout

Our memory is essentially a map from locations to blocks of bytes. It is based on the memory model of both Blazy and Leroy [1] and Roşu et al. [30] in the sense that the actual locations themselves are symbolic numbers. However, it is more like the former in that the actual blocks of bytes are really maps from offsets to bytes.

Below we see a snippet of a memory cell, holding four bytes:

$$\langle \cdots 32 \mapsto \mathrm{obj}\,(4,\ (0 \mapsto 7,\ 1 \mapsto 23,\ 2 \mapsto 140,\ 3 \mapsto 4)) \cdots \rangle_{\mathsf{mem}}$$

This says that at symbolic location 32, there is an object whose size is 4 bytes; those bytes are 7, 23, 140, and 4. All objects are broken into individual bytes, including aggregate types like arrays or `struct`s, as well as base types like integers.

Our pointers are actually base/offset pairs, which we write as $\mathrm{sym}(B) + O$, 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 $B < B'$ (Section 6.2.2). It is better to think of the 32 above as representing "object 32", as opposed to "location 32".

When looked up, the bytes are interpreted depending on the type of the construct used to give the address. The simplest example possible is dereferencing a pointer $\mathrm{sym}(32)+2$ of type `unsigned char*`, which would simply yield the value 140 of type `unsigned char`. Looking up data using different pointer types requires taking into account a number of implementation-defined details such as the use of signed magnitude, one's, or two's complement representation, or the order of bytes (endianness). These choices are made parametric in the semantics, and can be configured depending on which implementation a user is interested in working with (Section 4.5).

When new objects (`int`s, arrays, `struct`s, etc.) get allocated, each is created as a new block and is mapped from a new symbolic number. The block is allowed to contain as many bytes as in the object, and accesses relative to that object must be contained in the block. We represent information smaller than the byte (i.e., bitfields) by using offsets within the bytes themselves. While it might seem that it would be more consistent to treat memory as mappings from bit locations to individual bits, bitfields themselves are not addressable in C, so we decided on this hybrid approach.

### 4.4 Semantics

We now give the flavor of our semantics by examining a few of the 1,163 rules. For the rules below, recall that in $\mathbb{K}$ what is above the line is considered the LHS of the rule, while what is below the line is considered the RHS. Parts of a rule without a line at all are considered to be on both sides of the rule.

#### 4.4.1 Lookup and Assignment

We first consider one of the most basic expressions—the identifier. According to the standard, "An identifier is a primary expression, [...] designating an object (in which case it is an lvalue) or a function (in which case it is a function designator)" [15, §6.5.1:2]. Although in informal language an "lvalue" is an expression that appears on the LHS of an assignment, this is not the case according to the C standard. An lvalue can be more accurately thought of as any expression that designates a place in memory; a footnote in the standard suggests it might better be called a "locator value" [15, §6.3.2.1:1]. We denote lvalues with brackets; an lvalue that points to location $L$ which is of type $T$ is denoted by $[L]:T$. With this in mind, here then is our lookup rule:

$$\frac{\langle\ X\ \cdots\rangle_{\mathsf{k}}\ \langle \cdots X \mapsto L \cdots\rangle_{\mathsf{env}}\ \langle \cdots X \mapsto T \cdots\rangle_{\mathsf{type}}}{[L]:T}$$

This rule is actually very similar to the example address-of rule we gave in Section 3.3. It says that when the next thing to evaluate is the program variable $X$, both its location $L$ and its type $T$ should be looked up (in the env and type cells), and the variable should be replaced by an lvalue containing those two pieces of information. We distinguish between objects and functions based on type.

In almost all contexts, this lvalue will actually get converted to the value at that location:

> Except when it is the operand of the `sizeof` operator, the unary & operator, the ++ operator, the -- operator, or the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue) [13, §6.3.2.1:2].

We call these contexts "reval", for "right" evaluation. Here is the rule for simplifying lvalues in the "right value" context:

$$\frac{\mathrm{reval}([L]:T)}{\mathrm{read}(L,T)} \qquad \text{where } \neg(\mathrm{isArrayType}(T) \vee \mathrm{isFunctionType}(T))$$

The rule for "read" then does the actual read from memory. Its evaluation involves a series of rules whose job is to determine the size of the type, pull the right bytes from memory, and to piece them together in the right order to reconstruct the value. There are over 10 highly technical rules defining "read", just for integer types alone. This process results in a normal value, instead of an lvalue, which we represent simply as $V:T$.

#### 4.4.2 Reference and Dereference

We can now take a look at the rule for the & operator:

$$\frac{\langle\ \&\,([L]:T)\ \cdots\rangle_{\mathsf{k}}}{L:\mathrm{pointerType}(T)}$$

This rule says that when the next computation to be performed is taking the address of an lvalue, it should simply be converted into a "true value" holding the same address, but whose type is a pointer type to the original type. We can expect to find an lvalue

as the argument because the "reval" context does not include the arguments of the address operator.

The rule for dereference is similarly simple:

$$\langle \frac{*\,(L : \text{pointerType}(T))}{\text{checkDerefLoc}(L) \curvearrowright [L] : T} \cdots \rangle_{\text{k}} \qquad \text{where } T \neq \texttt{void}$$

This will first make sure that the location $L$ is allowed to be dereferenced (e.g., it is valid memory), and will then evaluate to an lvalue of the same location. As with lookup, no memory is read by default. Notice that checkDerefLoc is "blocking" the top of the k cell. As long as it stays there, no rules that match other constructs on the top of k can apply. If checkDerefLoc succeeds, it will simply evaluate to the unit of the $\curvearrowright$ construct and disappear. This is called "dissolving". Our rule for checkDerefLoc is:

$$\langle \frac{\text{checkDerefLoc}(\text{sym}(B) + O)}{\cdot} \cdots \rangle_{\text{k}} \langle \cdots B \mapsto \text{obj}(\mathit{Len}, \_) \cdots \rangle_{\text{mem}} \qquad \text{where } O < \mathit{Len}$$

Here we match the constituent parts of a location, $B$ and $O$, or base and offset as explained in Section 4.3. We then match the base part of the pointer in the memory cell, giving us an object, and check that the offset is within the bounds of the object. If this is the case, we dissolve the checkDerefLoc task.

### 4.4.3 Structure Members

The standard says, "A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. The value is that of the named member, and is an lvalue if the first expression is an lvalue" [15, §6.5.2.3:3].

Here is the rule for when the first expression is an lvalue:

$$\langle \frac{([L] : \text{structType}(S)) \texttt{.} F}{[L + \mathit{Offset}] : T} \cdots \rangle_{\text{k}} \langle \cdots S \mapsto (F \mapsto (\mathit{Offset}, T) \_) \cdots \rangle_{\text{structs}}$$

This rule finds the offset *Offset* and type $T$ of the field $F$ in struct $S$ and simply adds the offset to the base address $L$ of the struct to evaluate the expression. The result is another lvalue of the type of the field. In contrast, the rule for when the first expression is not an lvalue cannot simply work with pointers:

$$\langle \frac{(V : \text{structType}(S)) \texttt{.} F}{\text{extractField}(V, SD, S, F)} \cdots \rangle_{\text{k}} \langle \cdots S \mapsto (F \mapsto SD \_) \cdots \rangle_{\text{structs}}$$

One situation in which this arises is when a function returns a struct, and the programmer uses the function call to access a particular field, as in the expression `fun().field`. The call to `fun()` will result in a struct value, represented in the rule above by $V : \text{structType}(S)$. The helper function extractField will look at the bytes of the struct (represented by $V$) and "read" a value of the appropriate type ($SD$ contains the offset and type of the field). There are many rules shared by the extractField and read helpers, since both have to piece together bytes in implementation-defined orders to make new values.

The semantics for the arrow operator (`p->f`) is identical to that of the dot operator above after dereferencing the first subexpression:

$$E \texttt{->} F \Rightarrow (\texttt{*}E)\texttt{.}F$$

There are similar rules as above for union, where all offsets of a union's fields are 0.

### 4.4.4 Multiplication (and Related Conversions)

As mentioned in Section 3.2, the rules for arithmetic in C are non-trivial. To show this in more detail, here we give many of the rules related to integer multiplication. Here is the core multiplication rule:

$$\frac{(I_1 : T) \texttt{ * } (I_2 : T)}{\text{arithInterpret}(T, I_1 *_{\mathit{Int}} I_2)} \qquad \text{where hasBeenPromoted}(T)$$

This rule matches when multiplying values with identical, promoted types (more on promotion shortly). It then uses a helper operator "arithInterpret" to convert the resulting product into a proper value:

$$\frac{\text{arithInterpret}(T, I)}{I : T} \quad \text{where } \min(T) \leq I \wedge \max(T) \geq I$$

$$\frac{\text{arithInterpret}(T, I)}{\text{arithInterpret}(T, I -_{\mathit{Int}} (\max(T) +_{\mathit{Int}} 1))} \quad \text{where} \quad \begin{array}{l}\text{isUnsignedIntType}(T) \\ \wedge\, I > \max(T)\end{array}$$

$$\frac{\text{arithInterpret}(T, I)}{\text{arithInterpret}(T, I +_{\mathit{Int}} (\max(T) +_{\mathit{Int}} 1))} \quad \text{where} \quad \begin{array}{l}\text{isUnsignedIntType}(T) \\ \wedge\, I < \min(T)\end{array}$$

The first rule creates a value as long as the product is the range of the type. The next two rules collapse out-of-range unsigned products into range [15, §6.3.1.3:2]. By not giving rules to out-of-range signed types, we catch signed overflow here.

With the above rules defined, the question becomes how to promote and convert the types of the operands so that the core multiplication rule can take effect. First, all arithmetic in C takes place at or above the size of `int`s. This means smaller types need to be coerced into `int` or `unsigned int`.

$$\langle (\_ : \frac{T}{\text{promote}(T)}) \texttt{ * } \_ \cdots \rangle_{\text{k}} \qquad \text{where } \neg\text{hasBeenPromoted}(T)$$

The above rule (and its commutative partner) cause unpromoted multiplication operands to be promoted. Of the actual promotion, the standard says, "If an `int` can represent all values of the original type [...], the value is converted to an `int`; otherwise, it is converted to an `unsigned int`" [15, §6.3.1.1:2]:

$$\frac{\text{promote}(T)}{\texttt{int}} \qquad \text{where } \min(\texttt{int}) \leq \min(T) \wedge \max(\texttt{int}) \geq \max(T)$$

$$\frac{\text{promote}(T)}{\texttt{unsigned int}} \qquad \text{where } \neg(\min(\texttt{int}) \leq \min(T) \wedge \max(\texttt{int}) \geq \max(T))$$

Finally, in order to perform the multiplication, the types of the operands have to be identical. If the types are not identical, an implicit conversion takes place to convert the different types to a common type. There are eight rules for this given in the standard. To give an idea of their flavor, we give a few of the rules for integer conversions here. First, the rule to enable conversion:

$$\langle \frac{I_1 : T}{\text{cast}(\tau, I_1 : T)} \texttt{ * } \frac{I_2 : T'}{\text{cast}(\tau, I_2 : T')} \cdots \rangle_{\text{k}} \quad \text{where} \quad \begin{array}{l} T \neq T' \\ \wedge\, \tau = \text{arithConv}(T, T')\end{array}$$

The standard says, "if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank" [15, §6.3.1.8:1]:

$$\frac{\text{arithConv}(T, T')}{\text{maxType}(T, T')} \quad \text{where hasSameSignedness(T, T')}$$

Rank is a partial ordering on integer types based on their ranges and signedness, e.g., $\text{rank}(\texttt{short int}) < \text{rank}(\texttt{int})$. Additionally, the ranks of unsigned integer types equal the ranks of the corresponding signed integer types [15, §6.3.1.1:1]. Continuing with the conversion rules, "Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type" [15, §6.3.1.8:1]:

$$\langle \frac{\text{arithConv}(T, T')}{T} \cdots \rangle_{\text{k}} \quad \text{where} \quad \begin{array}{l}\text{isUnsigned}(T) \\ \wedge\, \text{isSigned}(T') \\ \wedge\, \text{rank}(T) \geq \text{rank}(T')\end{array}$$

and similarly for the commutative case.

The above equations use a number of helper operators in the side conditions—the definitions for "min" and "max" are given in Section 4.5; the other operators are defined as expected.

### 4.4.5 Malloc and Free

Here we show our semantics of `malloc` and `free`. These are functions from the standard C library that perform dynamic memory allocation and deallocation. The declarations of these functions are:

```
void *malloc(size_t size);
void free(void *ptr);
```

where `size_t` is an unsigned integer type that is implementation defined. When a programmer calls `malloc()`, an implementation

can return a new pointer pointing to a new block of memory the size specified by the programmer, or it can return `NULL` (e.g., if there is no memory available).

Here is the rule for a successful call to `malloc`:

$$\langle \frac{\texttt{malloc}(N:\texttt{size\_t})}{\text{alloc}(L,N) \curvearrowright L:\text{pointerType(void)}} \cdots \rangle_{\text{k}} \langle \cdots \frac{\cdot}{L \mapsto N} \cdots \rangle_{\text{malloced}} \quad \text{where } L \text{ is fresh}$$

If the user requests $N$ bytes, the semantics will schedule that many bytes to be allocated at a new location and record that this memory was dynamically allocated in the malloced cell. Here is the related rule for a failed called to `malloc`:

$$\langle \frac{\texttt{malloc}(\text{—})}{\text{NullPointer} : \text{pointerType(void)}} \cdots \rangle_{\text{k}}$$

This rule is usually only useful when searching the state space.

A call to `free` is meant to deallocate space allocated by `malloc`. Its rule is also straightforward:

$$\langle \underset{\cdot}{\texttt{free}(L:\text{—})} \cdots \rangle_{\text{k}} \langle \cdots \underset{\cdot}{L \mapsto N} \cdots \rangle_{\text{malloced}} \langle \cdots \underset{\cdot}{L \mapsto \text{obj}(N, \text{—})} \cdots \rangle_{\text{mem}}$$

When the user wants to free a pointer $L$, it is removed from both the malloced and mem cells. By matching these cells, the rule ensures that the pointer has not already been freed, and once applied, ensures no other rules that use that address can match into the memory.

#### 4.4.6 Setjmp and Longjmp

Finally, we show our semantics of `setjmp` and `longjmp`. These are functions from the standard C library that perform complex control flow. They are reminiscent of `call/cc`, and are often used as a kind of exception handling mechanism in C. The declarations of these functions are:

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

where `jmp_buf` is an array type "suitable for holding the information needed to restore a calling environment." A call to `setjmp` "saves its calling environment [...] for later use by the `longjmp` function." Additionally, the call to `setjmp` evaluates to zero [15, §7.13.1]. Here is our rule for `setjmp`:

$$\langle \frac{\texttt{setjmp}(L:\texttt{jmp\_buf})}{\text{write}(L,\ C\ \langle \kappa \rangle_{\text{k}}\ ) \curvearrowright 0 : \text{int}} \curvearrowright \kappa \rangle_{\text{k}} \langle C \rangle_{\text{local}}$$

Because `jmp_buf` is an array type, it will evaluate to an address $L$. In the rule above, we match the remaining computation $\kappa$ (similar to a continuation), as well as the local execution environment $C$. This includes cells like the call stack and the map from variables to locations (which we also call the environment). The rule then causes this information to be written at the location of the `jmp_buf`.

A call to `longjmp` "restores the environment saved by the most recent invocation of [`setjmp`] with the corresponding `jmp_buf` argument" [15, §7.13.2]. When the user calls `longjmp`, this address is read to find that previous context:

$$\langle \frac{\texttt{longjmp}\ (\ \overset{L:T}{\text{read}(L,T)}\ ,\text{—})}{\text{longjmp-aux}} \cdots \rangle_{\text{k}}$$

and it is then restored:

$$\langle \frac{\text{longjmp-aux}((C\ \langle \kappa \rangle_{\text{k}}\ : \text{—}),\ I:\text{int})}{(\text{if } I = 0 \text{ then } 1 \text{ else } I \text{ fi}) : \text{int}} \curvearrowright \frac{\text{—}}{\kappa} \rangle_{\text{k}} \langle \frac{\text{—}}{C} \rangle_{\text{local}}$$

This function returns the `val` that the user passes, unless this is a 0, in which case it returns 1.

It should be clear that these rules operate on the configuration itself, treating it as a first-class term of the formalism. The fact that $\mathbb{K}$ allows one to grab the continuation $\kappa$ as a term is what makes the semantics of these constructs so easy to define. This is in sharp opposition to semantic formalisms like SOS [27] where the context is a derivation tree and not directly accessible as an object inside a definition.

### 4.5 Parametric Behavior

We chose to make our definition parametric in the implementation-defined behaviors (and are not the first to do so [1, 5]). Thus, one can configure the definition based on the architecture or compiler one is interested in using, and then proceed to use the formalism to explore behaviors. This parameterization allows the definition to be "fleshed out" and made executable.

For a simple example of how the definition is parametric, our K-Maude module `C-SETTINGS` starts with:

$$\begin{aligned} \text{numBytes(signed-char)} &\Rightarrow 1 & \text{numBytes(short-int)} &\Rightarrow 2 \\ \text{numBytes(int)} &\Rightarrow 4 & \text{numBytes(long-int)} &\Rightarrow 4 \\ \text{numBytes(long-long-int)} &\Rightarrow 8 & \text{numBytes(float)} &\Rightarrow 4 \\ \text{numBytes(double)} &\Rightarrow 8 & \text{numBytes(long-double)} &\Rightarrow 16 \end{aligned}$$

These settings are then used to define a number of operators:

$$\begin{aligned} \text{numBits}(T) &\Rightarrow \text{numBytes}(T) * \text{bitsPerByte where } \neg\text{isBitfieldType}(T) \\ \text{min(int)} &\Rightarrow -_{Int}(2^{\text{numBits(int)}-_{Int}1}) \\ \text{max(int)} &\Rightarrow 2^{\text{numBits(int)}-_{Int}1} -_{Int} 1 \end{aligned}$$

Here we use a side condition to check when a type is *not* a bitfield. Finally, the above rules are used to define how an integer $I$ of type $T$ is cast to an unsigned integer type $T'$:

$$\begin{aligned} &\text{cast}(T', I:T) \Rightarrow (I \,\%_{Int}\, (\max(T') +_{Int} 1))) : T' \\ &\quad \text{where isIntegerType}(T) \wedge \text{isUnsignedIntType}(T') \wedge I > \max(T') \end{aligned}$$

Here we use helper predicates in our side conditions to make sure this rule only applies when casting from integer types to unsigned integer types. There are similar equations used to define other cases.

### 4.6 Expression Evaluation Strategy and Undefined Behavior

The C standard allows compilers freedom in optimizing code, which includes allowing them to choose their own expression evaluation order. This includes allowing them to:

- delay side effects: e.g., allowing the write to memory required by `x=5` or `x++` to be made separately from its evaluation or use;
- interleave evaluation: e.g., `A + (B * C)` can be evaluated in the order B, A, C.

At the same time, the programmer must be able to write programs whose behaviors are reproducible, and only allow non-determinism in a controlled way. Therefore, the standard makes undefined certain situations where reordering creates a "race condition". The latest treatment of this restriction is given by the C1X standard:

> 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. If there are multiple allowable orderings [...], the behavior is undefined if such an unsequenced side effect occurs in any of the orderings [15, §6.5:2].

This means that if there are two writes, or a write and a read to the same object that are unsequenced (i.e., either is allowed to happen before the other), then the expression is undefined. Examples of expressions made undefined by this clause include `(x=0)+(x=1)` and `(x=0)+x` and `x=x++` and `x=x++`, and `*p=x++`, for `int x` and `int* p=&x`. This relation is related to the concept of "sequence points", also defined by the standard. Sequence points cause the expressions they fall between to be sequenced. The most common example of a sequence point is the semicolon, i.e., the end of an expression-statement. All previous evaluations and side effects must be complete before crossing sequence points.

A hasty read of the standard may wrongly indicate that detecting this kind of undefined behavior is an easy problem that can be checked statically. In fact, it is undecidable statically; moreover, one needs to use the entire semantics in order to check it dynamically. Consider the following example:

```
int x, y, *p = &y;
int f(void){ if (guard) { p = &x; } return 0; }
int main(void){ return (x = 5) + (*p = 6) + f(); }
```

The undefinedness of this program is based on what happens in the call to `f()`. If `f` is called before the other subexpressions in `main` are evaluated, and if the `guard` expression (which could be arbitrarily complex) is true, then the remaining expression effectively becomes `(x = 5) + (x = 6)`, which is undefined. The possible complexity of the `guard` is a witness to the (static) undecidability of this problem. The evaluation of the `guard` may make arbitrary use of the entire C language, so the entire semantics is needed in order to determine whether this program is undefined.

Based on this, note that: when two expressions are unsequenced, it means that evaluation can happen in any order. Thus, it is natural to map unsequenced behavior into nondeterministic behavior. This way, we can use state space exploration as a single mechanism to find unsequenced behavior. To identify this kind of undefined behavior naively can be incredibly computationally expensive; some optimizations are necessary to make this feasible. We offer two such optimizations below.

First, with a little case analysis of the definition of the sequencing relation, it is clear that there can be no sequenced write before a read of the same object with no intervening sequence point. This means that if in searching the semantic state space, we find an execution in which the write of a scalar object happens before a write or read of the same object with no intervening sequence point, then we can conclude that this write/write or write/read pair is unsequenced. Whenever a write is made, its location is recorded in the locsWrittenTo cell, which is emptied whenever a sequence point is crossed. This cell is first checked whenever a read or write is made to ensure that there is no conflict. This strategy has the added benefit that some undefined behaviors of this kind can be detected even during interpretation (where only a single path through the state space is explored). It is similar to the strategy used by Norrish [22].

Second, it turns out that a large subset of allowed orderings do not need to be considered in order to detect undefined behavior or possible nondeterministic behaviors. Because we are looking for writes *before* other events, we can take the liberty of applying side effects immediately instead of delaying them.

What would it mean for there to exist an expression whose definedness relied on whether or not a side effect (a write) occurs later instead of earlier? There must be three parts to the expression: a subexpression $E$ generating a side effect $X$, and, for generality's sake, further subexpressions $E'$ and $E''$. The particular evaluation where we do side effects immediately would look like $E\ X\ E'\ E''$. Because this is always a possible execution, and we assume it does not show a problem, we can conclude neither $E'$ nor $E''$ reads or writes to $X$. If there is a problem only when we delay the side effect, it can be seen in a path like $E\ E'\ X\ E''$. For this to be different than applying the changes to $X$ immediately, it means there must be some use of $X$ in the evaluation of $E'$. But this contradicts the previous assumption.

This shrinks the state space dramatically, while at the same time not missing any undefined behavior. Our semantics does capture the appropriate state space, as seen in Section 6.3.1.

### 4.7 KCC

Using a simple frontend that mimics the behavior of GCC [9], C programs are parsed and translated into a Maude term, then reduced using the rules of our formal semantics. For defined programs, this process produces indistinguishable behavior from the same C program run as native code. We call this interpreter, obtained automatically from our formal semantics, KCC. As we will show in Section 6, KCC is significantly more than an interpreter—in addition to simple interpretation, it is also capable of debugging, catching undefined behaviors, state space search, and model checking. Once KCC is installed on a system, compilation of C programs generates a single executable file (an "a.out") containing the semantics of C, together with a parsed representation of the program and a call to Maude. The output is captured by a script and presented so that for working programs the output and behavior is identical to that of a real C compiler. To emphasize the seamlessness, here is a simple transcript:

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

While it may seem like a gimmick, it helped our testing and debugging tremendously. For example, we could run the definition using the same test harness GCC uses for its testing (see Section 5). It also means people with no formal background can get use out of our semantics simply by using it as they would a compiler.

## 5. Testing the Semantics

No matter what the intended use is for a formal semantics, its actual use is limited if one cannot generate confidence in its correctness. To this aim, we ensured that our formal semantics remained executable and computationally practical.

### 5.1 GCC Torture Tests

As discussed in the previous section, our semantics is encapsulated inside a drop-in replacement for GCC, which we call KCC. This enables us to test the semantics as one would test a compiler. We were then able to run our semantics against the GCC C-torture-test [10] and compare its behavior to that of GCC 4.1.2, as well as the Intel C++ Compiler (ICC) 11.1 and Clang 3.0 r132915 (C compiler for LLVM). We ran all compilers with optimizations turned off.

We use the torture test for GCC 4.4.2, specifically those tests inside the "testsuite/gcc.c-torture/execute" directory. We chose these tests because they focus particularly on portable (machine independent) executable tests. The README.gcc for the tests says, "The 'torture' tests are meant to be generic tests that can run on any target." We found that generally this is the case, although there are also tests that include GCC-specific features, which had to be excluded from our evaluation. There were originally 1093 tests, of which we excluded 267 tests because they used GCC-specific extensions or builtins, they used the `_Complex` data type or certain library functions (which are not required of a freestanding implementation of C), or they were machine dependent. This left us with 826 tests. Further manual inspection revealed an additional 50 tests that were non-conforming according to the standard (mostly signed overflow or reading from uninitialized memory), bringing us to a grand total of 776 viable tests.

In order to avoid "overfitting" our semantics to the tests, we randomly extracted about 30% of the conforming tests and developed our semantics using only this small subset (and other programs discussed in Section 5.2). After we were comfortable with the quality of our semantics when running this subset, we ran the remaining tests. Out of 541 previously untested programs, we successfully ran 514 (95%). After this initial test, we began to use all of the tests to help develop our semantics; we now pass 770 (99.2%) of the 776 compliant tests.

| | Torture Tests Run (of 776) | |
|---|---|---|
| Compiler | Count | Percent |
| GCC | 768 | 99.0 |
| ICC | 771 | 99.4 |
| Clang | 763 | 98.3 |
| KCC | 770 | 99.2 |

The 776 tests represent about 23,500 SLOC, or 30 SLOC/file.

***Correctness Analysis*** Our executable formal semantics performed nearly as well as the best compiler we tested, and better than the others. We incorporated the passing tests into our regression suite that gets run every time we commit a change. This way, upon adding features or fixing mistakes, our accuracy can only increase.

Three of the six failed tests rely on floating point accuracy problems. Two more rely on evaluating expressions inside of function declarators, as in:

```
int fun(int i, int array[i++]) { return i; }
```

which we are not handling properly. The last is a problem with the lifetime of variable length arrays.

***Coverage Analysis*** In order to have some measure of the effectiveness of our testing, we recorded the application of every semantic rule for all of the torture tests. Out of 887 core rules (non-library, non-helper operator), the GCC torture tests exercised 805 (91%).

In addition to getting a coverage measure, this process suggests an interesting application. For example, in the GCC tests looked at above, a rule that deals with casting large values to `unsigned int` was never applied. By looking at such rules, we can create new tests to trigger them. These tests would improve both confidence in the semantics as well as the test suite itself.

### 5.2 Exploratory Testing

We have also tested our semantics on programs gathered from around the web, including programs of our own design and from open source compilers. Not counting the GCC tests, we include over 17,000 SLOC in our regression tests that are run when making changes to the semantics. These tests include a number of programs from the LCC [12] and CompCert [1] compilers. We also execute the "C Reference Manual" tests (also known as cq.c),[6] which go through Kernighan and Ritchie [17] and test each feature described in about 5,000 SLOC. When these tests are added to the GCC tests described above, it brings our rule-coverage to 98% (867/887 rules).

We can successfully execute Duff's Device [7], an unstructured `switch` statement where the `case`s are inside of a loop inside of the `switch` statement itself, as well as quines (programs whose output are precisely their source code), and a number of programs from the Obfuscated C Code Contest [21]. All of these test programs, as well as our semantics, are available from our project webpage: `http://c-semantics.googlecode.com/`.

## 6. Applications—Formal Semantics is Useful!

Here we describe applications of our formal semantics, which are in addition to the interpreter already mentioned. These tools are automatically derived from the semantics—changes made to the semantics immediately affect the tools. We are permitted this luxury because we take advantage of general purpose tools available to RL theories, of which our semantics is one. Contrast this to the nearly universal strategy of writing analysis tools independently of semantics. Instead of developing a different model for each tool, a plethora of tools can be created around a single semantic definition. These tools are essentially wrappers, or views, of the semantics.

### 6.1 Debugging

By introducing a special function "`__debug`" that acts as a breakpoint, we can turn the Maude debugger into a simple debugger for C programs. This provides the ability to step through interesting parts of execution to find out what rules of semantics are invoked in giving meaning to a program.

In the semantics, we handle this function by giving a labeled rule that causes it to evaluate to a "void" value. It is essentially equivalent to `void __debug(int i) { }`. If this function is called during execution, it starts a debugger that allows the user to inspect the current state of the program. One can step through more rules individually from there, or simply note the information and proceed. If the `__debug` call is inside a loop, the user will see a snapshot each time it reaches the expression. For example:

```
int main(void){
  for (int i = 0; i < 10; i++){ __debug(i); }
  printf("done!\n");
}
```

We can run or debug the program above as follows:

```
$ kcc debug.c
$ ./a.out # run the program normally
done!
$ DEBUG=1 ./a.out # or run it in the debugger
Debug(1)> where .
          ⟨__debug(0 : int) ···⟩k ⟨··· i ↦ L ···⟩env  ···
Debug(1)> resume .
          ⟨__debug(1 : int) ···⟩k ⟨··· i ↦ L ···⟩env  ···
```

The user can use this to see what the value of the `__debug` argument is each time through the loop, as well as the entire state of the program when the breakpoint was reached. The state presented to the user includes all of the cells of the language (Figure 2). This elided state is represented by the ellipses above. In addition to the "where" and "resume" commands, there is also a "step" command to step through the application of a single semantic rule [3, §22.1].

### 6.2 Runtime Verification

There are two main avenues through which we can catch and identify runtime problems with a program: catching undefined behavior, and symbolic execution.

#### 6.2.1 Undefined Behavior

The first mechanism is based around the idea that when something lacks semantics (i.e., when its behavior is undefined according to the standard) then the evaluation of the program will simply stop when it reaches that point in the program. We use this mechanism to catch errors like signed overflow or array out-of-bounds.

In this small program, the programmer forgot to leave space for a string terminator (`'\0'`). The call to `strcpy()` will read off the end of the array:

```
int main(void) {
  char dest[5], src[5] = "hello";
  strcpy(dest, src);
}
```

GCC will happily execute this, and depending on the state of memory, even do what one would expect. It is still undefined, and our semantics will detect trying to read past the end of the array. Because this program has no meaning, our semantics "gets stuck" when exploring its behavior. It is through this simple mechanism that we can identify undefined programs and report them to the user. By default, when a program gets stuck, we report the state of the configuration (a concrete instance of that shown in Figure 2) and what exactly the semantics was trying to do at the time of the problem. We have also begun to add explicit error messages for common problems—here is the output[7] from our tool for this code:

```
$ kcc buggy_strcpy.c ; ./a.out
ERROR encountered while executing this program.
Description: Reading outside the bounds of an object.
Function: strcpy
Line: 3
```

[6] We have been unable to determine the author or origin of this test suite. Please contact us with any information.

[7] Here and elsewhere in this section, we take the liberty to slightly simplify the output to make it fit in less vertical space.

#### 6.2.2 Symbolic Execution

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. Symbolic execution is straightforward to achieve using a rewriting-based semantics: whether a term is concrete or abstract makes no difference to the theory. Rules designed to work with concrete terms do not need to be changed in order to work with symbolic terms.

As we explained in Section 4.3, we treat pointers not as concrete integers, but as symbolic values. These values then have certain behavior defined on them, such as comparison, difference, etc. This technique is based on the idea of *strong memory safety*, which had previously been explored with a simple C-like language [30]. In this context, it takes advantage of the fact that addresses of local variables and memory returned from allocation functions like `malloc()` are unspecified [15, §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 [15, §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 relational pointer comparisons where the two addresses share a common base. Thus, evaluation gets stuck on the program above:

```
$ kcc bad_comparison.c ; ./a.out
ERROR encountered while executing this program.
Description: Cannot apply '<' to different base objects.
Function: main
Line: 3
```

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 [15, §6.5.8:5], and in fact our semantics finds the comparison to be true because the pointers share a common base.

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 [15, §6.2.6.1:5–7]. Because of this, our semantics must give it meaning. Using concrete values here 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.

### 6.3 State Space Search

We can also use our semantics to do both matching-based state search and explicit state model-checking with linear temporal logic (LTL). The basic examples below show how our semantics captures the appropriate expression evaluation semantics precisely.

#### 6.3.1 Exploring Evaluation Order

To show our semantics captures the evaluation orders of C expressions allowed by the specification, we examine some examples from related works. The results given below are not just theoretical results from our semantics, but are actual results obtained from executing the tools provided by our semantic framework.

To start with a simple example from Papaspyrou and Maćoš [26], we take a look at `x+(x=1)` in an environment where `x` is 0. This expression is undefined because the read of `x` (the lone `x`) is unsequenced with respect to the write of `x` (the assignment). Using our semantics to do a search of the behaviors of this expression finds this unsequenced read/write pair, and reports an error.

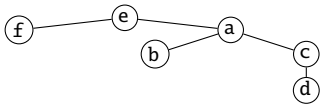
Norrish [22] offers the deceptively simple addition expression `(x=0) + (x=0)`, which in many languages would be valid. However, in C it is again a technically undefined expression due to the unsequenced assignments to `x`. Our semantics reports an error for this expression as well.

Another example in the literature is given by Papaspyrou [25], which shows how C can exhibit nondeterministic behavior while staying conforming. The driving expression is the addition of two function calls. In C, function evaluation is not allowed to interleave [15, 6.5.2.2:10], so the behavior of this program is determined solely on which call happens last:

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

If `f()` is called with the argument 2 last, then the result will be 2, and similarly for 1. Searching with our semantics gives the behaviors `{r=1}` and `{r=2}`, which are indeed the two possible results.

As a last example, we look at a more complex expression of our own devising: `f()(a(b(), c(d())))`. Except for `f()`, each function call prints out its name and returns 0. The function `f()`, however, prints out its name and then returns a function pointer to a function that prints "e". The function represented by this function pointer will be passed results of `a()`. We elide the actual function bodies, because the behavior is more easily understood by this tree:




This tree (or Hasse diagram) describes the sequencing relation for this expression. That is, it must be the case that `d` happens before `c`, that `b` and `c` happen before `a`, and that `f` and `a` happen before `e`. Running this example through our search tool gives precisely the behaviors allowed by the standard:

```
$ kcc nondet.c ; SEARCH=1 ./a.out
15 solutions found
bdcafe bdcfae bdfcae bfdcae dbcafe dbcfae dbfcae dcbafe
dcbfae dcfbae dfbcae dfcbae fbdcae fdbcae fdcbae
```

#### 6.3.2 Model Checking

In addition to the simple state search we showed above, one can also use our semantics for LTL model checking. For example, consider the following program:

```
typedef enum {green, yellow, red} state;
state lightNS = green; state lightEW = red;
int changeNS() {
  switch (lightNS) {
    case(green): lightNS = yellow; return 0;
    case(yellow): lightNS = red; return 0;
    case(red):
      if (lightEW == red) { lightNS = green; } return 0;
  }
}
...
int main(void) { while(1) { changeNS() + changeEW(); } }
```

This program is meant to represent two orthogonal traffic lights (`lightNS` and `lightEW`) at the same intersection. It provides

an implementation of an algorithm to change the state of the lights from green to yellow to red and back. We elide the nearly identical `changeEW()` function. The program takes advantage of the unspecified order of evaluation of addition in the expression `changeNS() + changeEW()` to nondeterministically choose the order in which the lights are changed.

There are a number of properties one might like to prove about this program, including safety and liveness properties. One safety property is that it should always be the case that at least one of the lights is red, or $\Box((\texttt{lightNS == red}) \vee (\texttt{lightEW == red}))$. We have added a special `#pragma`[8] allowing the programmer to write and name LTL formulae. If we call the above formula "safety", then we can invoke the model checker as follows:

```
$ kcc lights.c ; MODELCHECK=safety ./a.out
result Bool: true
```

Similarly, it is important that the lights always make progress, i.e., that it is always the case the lights will eventually become green. If we try to check $\Box\Diamond(\texttt{lightNS == green})$, we find that it does not hold of the above program:

```
$ kcc lights.c ; MODELCHECK=progress ./a.out
result ModelCheckResult: counterexample ...
```

The reason this property is not verified is that the algorithm is wrong! Because the calls to `changeNS()` and `changeEW()` can occur in any order, it is possible for either of the lights to get stuck on red. The program starts with `ns=gre, ew=red`. Consider the following execution:

```
changeNS, changeEW => ns=yel, ew=red
changeEW, changeNS => ns=red, ew=red
changeNS, changeEW => ns=gre, ew=red
```

By alternating evaluation orders, the program can change the N/S light without ever changing the E/W light. This evaluation order is highly implausable in most C compilers, but the semantics allows it. If we fix an evaluation order by changing `changeNS() + changeEW();` to `changeNS(); changeEW();`, then the property holds:

```
$ kcc lights.c ; MODELCHECK=progress ./a.out
result Bool: true
```

## 7. Limitations and Future Work

Here we delineate the limitations of our definition and explain their causes and effects.

There are two main ways in which semantics can be incomplete—under-definedness and over-definedness. Typically when one thinks of incompleteness, one thinks of failure to give meaning to correct programs. However, because we want to be able to identify incorrect or unportable programs, the semantics must be balanced appropriately between defining too much or too little. It is equally important not to give semantics to programs that should be undefined.

In the first case, we are not missing any features—we have given semantics to every feature required of a freestanding implementation of C. With this said, our semantics is not perfect. For example, we still are not passing 100% of our test cases (see Section 5). Also, our semantics of floating point numbers is particularly weak. During execution or analysis, we simply rely on an IEEE-754 implementation of floating point arithmetic provided to us by our definitional framework ($\mathbb{K}$). This is fine for interpretation and explicit state model checking, but not for deductive reasoning.

In the second case, although our semantics can catch many bad behaviors other tools cannot (e.g., we have not found any other tool that catches the undefined programs in Sections 6.2.2 or 6.3.1), there is still room for improvement. For one, our semantics aligns all types to one-byte boundaries. This means we cannot catch undefined behavior related to alignment restrictions. Note that others have worked on formalizing alignment requirements [20], but it has never been incorporated into a full semantics for C. We also do not handle type qualifiers (like `const` or `volatile`); we simply ignore them. This is safe to do when interpreting correct programs, but it means we are not detecting problems related to those features in incorrect programs. It also means that we are missing possible behaviors when searching programs that use `volatile`.

We have not yet used our C definition for doing language or program level proofs, even though the $\mathbb{K}$ Framework supports both program level [31] and semantics level proofs [8]. To do so, we need to extend our semantics with support for formal annotations (e.g., assume, assert, invariant) and connect it to a theorem prover. This is already being done for a subset of the C language [29], and we intend to apply those techniques to actual C in the future.

We still do not cover all of the standard library headers. So far, we have added library functions by need in order to run example programs, which is why we have semantics for library functions like `malloc()`, `longjmp()`, parts of `printf()`, variadic functions, and over 30 others. We intend on covering more libraries in the future, but for now, one could supplement what we provide by using implementations of libraries written in C.

In our current semantics, only some of the implementation-defined behaviors are available—the most common ones. By making the semantics parametric, we hope others can add or change implementation-defined rules to suit their needs.

Finally, we should mention the speed of our system. While it is not nearly as fast as C compiled natively, it is usable. Of the GCC torture test programs described listed in Section 5, our semantics ran over 93% of these programs in under 10 seconds (each). An additional 4% completed in 2 minutes, 2% in 5 hours, and 1% further in under 3 days. In comparison, it takes GCC about 0.05 s for each test. The reader should keep in mind that this is an interpreter obtained for free from a formal semantics. In addition, the search and model checking tools suffer the same state explosion problems inherent in all explicit-state model checking.

## 8. Conclusion

It is a shame that, despite the best efforts of over 40 years of research in formal programming languages, most language designers still consider the difficulties of defining formal semantics to outweigh the benefits. Formal semantics and practicality are not typically considered together. When C was being standardized, the standards committee explored using formal semantics, but in the end decided to use simple prose because, "Anything more ambitious was considered to be likely to delay the Standard, and to make it less accessible to its audience" [14, §6]. This is a common sentiment in the programming language community. Indeed, startlingly few "real" languages have ever been completely formalized, and even fewer were designed with formal specification in mind.

Based on our experience with our semantics, the development of a formal semantics for C could have taken place alongside the development of the standard. Within roughly 6 person-months, we had a working version of our semantics that covered more of the standard than any previous semantics. The version presented in this paper is the result of 18 person-months of work. To put this in perspective, one member of the standards committee estimated that it took roughly 62 person-*years* to produce the C99 standard [16]. We are *not* claiming that we have done the same job in a fraction of the time; obviously writing a semantics based on the standard is quite different than writing the standard itself. We are simply saying that the effort it takes to develop a rewriting-based semantics is quite small compared to the effort it took to develop the standard.

[8] A conforming way to add implementation-defined behavior to C.

The reluctance of the language community towards formal methods has not been without reason—it is not always clear that having a formal semantics earns the designer anything tangible for her effort. Commonly mentioned benefits like improving the understanding of the language or providing a model in which sound arguments about the language can be made are relatively intangible; to be accepted by the general language community, semantics needs to be shown to have concrete value beyond that of prose.

The time has come to start building analysis tools directly on formal models. Instead of building analysis tools for different languages and different versions of each language, the analysis infrastructure surrounding the semantics could be maintained independently so that one could derive tools for multiple languages simply by swapping out the semantic rules. We offer our work as one small step in this direction; we are not alone, and there are other tools including pluggable analysis architectures like Frama-C [6] and formal tools like CompCert [1] that share part of this vision.

Our semantics and its automatically generated tools have already found one serious application. Csmith [35] is a C program test generator that generates random conforming programs from a large, expressive subset of the C language. These tests are then used to perform differential testing among C compilers to find compilation bugs. To date, the Csmith team has found more than 325 bugs in common compilers like GCC and Clang. The programs Csmith generates are almost always too large (many between 1,000 and 10,000 SLOC) to submit as bug reports and need to be reduced. The reduction process is semi-automatic, but is riddled with the possibility of introducing undefined behavior. Until now, these tests would have to be carefully examined by hand for undefined behavior, because any such behavior would render the tests invalid. Our semantic tools are being used by the Csmith team to detect this undefined behavior and have allowed them to more completely automate the process and reduce the tests more aggressively.

## 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.

[2] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, second edition, 1998.

[3] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[4] J. V. Cook and S. Subramanian. A formal semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, November 1994.

[5] J. V. Cook, E. L. Cohen, and T. S. Redmond. A formal denotational semantics for C. Technical Report 409D, Trusted Information Systems, September 1994.

[6] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. *SIGPLAN Not.*, 44: 281–286, August 2009.

[7] T. Duff. On Duff's device, 1988. URL `http://www.lysator.liu.se/c/duffs-device.html`. Msg. to the comp.lang.c Usenet group.

[8] C. Ellison, T. F. Șerbănuță, and G. Roșu. A rewriting logic approach to type inference. In *19th Intl. Wkshp. on Algebraic Development Techniques (WADT'08)*, volume 5486 of *LNCS*, pages 135–151, 2009.

[9] FSF. GNU compiler collection, 2010. URL `http://gcc.gnu.org`.

[10] FSF. C language testsuites: "C-torture" version 4.4.2, 2010. URL `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`.

[11] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In *Computer Science Logic*, volume 702 of *LNCS*, pages 274–308, 1993.

[12] D. R. Hanson and C. W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[13] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:1999: Programming languages—C. Technical Report n1256, Intl. Organization for Standardization, December 1999.

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

[15] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:201x: Programming languages—C. Technical Report n1570, Intl. Organization for Standardization, August 2011.

[16] D. M. Jones. *The New C Standard: An Economic and Cultural Commentary*. Self-published, December 2008. URL `http://www.knosof.co.uk/cbook/cbook.html`.

[17] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

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

[19] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Construction*, pages 213–228, 2002.

[20] M. Nita, D. Grossman, and C. Chambers. A theory of platform-dependent low-level software. In *35th ACM Symposium on Principles of Programming Languages (POPL'08)*, 2008.

[21] L. C. Noll, S. Cooper, P. Seebach, and L. A. Broukhis. The international obfuscated C code contest, 2010. URL `http://www.ioccc.org/`.

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

[23] M. Norrish. A formal semantics for C++. Technical report, NICTA, 2008. URL `http://nicta.com.au/people/norrishm/attachments/bibliographies_and_papers/C-TR.pdf`.

[24] N. S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, Natl. Technical University of Athens, 1998.

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

[26] N. S. Papaspyrou and D. Maćoš. A study of evaluation order semantics in expressions with side effects. *J. Functional Programming*, 10(3): 227–244, 2000.

[27] G. D. Plotkin. The origins of structural operational semantics. *J. Logic and Algebraic Programming*, 60:60–61, 2004.

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

[29] G. Roșu and A. Ștefănescu. Matching logic: A new program verification approach (NIER track). In *30th Intl. Conf. on Software Engineering (ICSE'11)*, pages 868–871, 2011.

[30] 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.

[31] G. Roșu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *13th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'10)*, volume 6486 of *LNCS*, pages 142–162, 2010.

[32] T. F. Șerbănuță and G. Roșu. K-Maude: A rewriting based tool for semantics of programming languages. In *8th Intl. Wkshp. on Rewriting Logic and its Applications (WRLA'09)*, volume 6381 of *LNCS*, pages 104–122, 2010.

[33] S. Subramanian and J. V. Cook. Mechanical verification of C programs. In *ACM SIGSOFT Wkshp. on Formal Methods in Software Practice*, January 1996.

[34] S. Summit. C programming FAQs: Frequently asked questions, 2005. URL `http://www.c-faq.com/`.

[35] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *32nd Conf. on Programming Language Design and Implementation (PLDI'11)*, pages 283–294, 2011.

[36] W. Zimmermann and A. Dold. A framework for modeling the semantics of expression evaluation with abstract state machines. In *Abstract State Machines*, volume 2589 of *LNCS*, pages 391–406, 2003.