DEFINING THE UNDEFINEDNESS OF C11: PRACTICAL SEMANTICS-BASED PROGRAM ANALYSIS

A Dissertation presented to
the Faculty of the Graduate School
at the University of Missouri

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

---

by
CHRIS HATHHORN
Dr. William L. Harrison, Dissertation Supervisor
DEC 2017

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

DEFINING THE UNDEFINEDNESS OF C11:
PRACTICAL SEMANTICS-BASED PROGRAM ANALYSIS

presented by Chris Hathhorn, a candidate for the degree of Doctor of Philosophy, and hereby certify that, in their opinion, it is worthy of acceptance.

_____

Dr. William L. Harrison

_____

Dr. Rohit Chadha

_____

Dr. James Keller

_____

Dr. Sean Goggins

_____

Dr. Grigore Roșu

Dedication

ACKNOWLEDGMENTS

Contents

List of Figures

ABSTRACT

This thesis extends the work of Ellison and Roşu [13, 12] but focuses on the "negative" semantics of the C11 language—the semantics required to not just give meaning to correct programs, but also to reject undefined programs. We investigate undefined behavior in C and discuss the techniques and special considerations needed to formally specify it. Using these techniques, we have modified and extended a semantics of C into one that captures undefined behavior. The amount of semantic infrastructure and effort required to achieve this was unexpectedly high, in the end more than tripling the size of the original semantics.

From our semantics, we automatically extract kcc, a tool for checking real-world C programs for undefined behavior and other common programmer mistakes. Previous versions of this tool were used primarily for testing the correctness of the semantics, but we have improved it into a tool for doing practical analysis of real C programs. It beats many similar tools in its ability to catch a broad range of undesirable behaviors. We demonstrate this with comparisons based on our own test suite in addition to third-party benchmarks. Our checker is capable of detecting examples of all 77 categories of core language undefinedness appearing in the C11 standard, more than any other tool we considered. Based on this evaluation, we argue that our work is the most comprehensive and complete semantic treatment of undefined behavior in C, and thus of the C language itself.

Chapter 1

Introduction

> [Undefined behavior] is absolutely essential for standardization purposes, since otherwise the language will be impossible to implement efficiently on differing hardware designs. C. A. R. Hoare [21]

Most formal semantics of various parts of C (e.g., Norrish [40], Papaspyrou [41], Blazy and Leroy [5], Ellison and Roşu [13]) have focused on the meaning of correct programs. But for the C language, the semantics of correct programs is only half the story. If we make the assumption that programs are well-defined, then we can ignore large swathes of the C11 standard. Most qualifiers can be ignored, for example—`const` and `restrict` are described by the standard entirely in terms of the undefinedness caused by their misuse. In well-defined programs, types are all compatible, pointer arithmetic valid, signed integers never overflow, and every function call is compatible with the type of a matching declaration in some translation unit.

Our work extends Ellison and Roşu [13]. Their work focused on giving semantics to correct programs and they evaluated their semantics against well-defined programs. This work, in contrast, focuses on identifying undefined programs. Ultimately, our goal is to do practical semantics-based program analysis of real C programs. This work has produced, as its primary artifact, an operational semantics of C[1] able to distinguish genuine C programs—large, real-world programs that exercise (yet adhere to) all parts of the C11 standard—from the undefined.

A note on the significance of undefined behavior: in the C language, undefined behavior accounts for many (if not most) programmer errors. The notorious reputation C programs have for poor security and exploitability is entirely attributable to undefined behavior. Buffer overruns, signed integer overflow, and using invalid pointers are common examples, but undefined behavior can be even more subtle. As compilers become more sophisticated, they are increasingly able to optimize under the assumption that programs contain no undefinedness to surprising and sometimes dangerous effect when a program actually does contain undefined behavior [30, 44]. And as we will see, undefined behavior so permeates C11 that it would be unlikely to find a C program of a non-trivial size without it.

In the next section, we define undefined behavior and where it appears in the standard, give some examples of the undefined behavior covered by our

[1]Our semantics is open-source and available online at `https://github.com/kframework/c-semantics`.

semantics, and attempt to define what the standard might actually require of implementations when it invokes undefined behavior. We also describe our formal operational semantics of the C11 standard (implemented in the $\mathbb{K}$ framework) and the tool we extract from our semantics (kcc) to do analysis of real C programs. In Section 4.1, we describe some related work and evaluate our tool against some state-of-the-art analyzers. kcc can beat such tools in terms of its broad coverage and lack of false positives [17, 11].

1.1 The semantics of undefined behavior (UB)

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" [24, §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). [24, §3.4.3:2]

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

The standard also defines two further categories of behavior: unspecified and implementation-defined behavior, which are somewhat less dire than the undefined. In the former case, implementations are given a choice among a set of possible values or behaviors (such as the order of evaluation of function arguments), with no requirement that an implementation make the same choice in every instance. Implementation-defined behavior differs from unspecified behavior only in the requirement that an implementation must document its choice. And undefined behavior can depend on these choices, so consulting an implementation's manual might be required in order to determine whether something is actually a well-defined C program. In the rest of this thesis, we will focus on undefined behavior, and not unspecified or implementation-defined behavior, which are conventionally modeled by non-determinism and parameterization, respectively.

The C11 standard contains a list of the 203 places it invokes undefined behavior [24, §J]. These can be divided into three broad sets:

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

Plus one additional item: "A 'shall' or 'shall not' requirement that appears outside of a constraint is violated" [24, UB #1], [2] which is the justification for including

[2] Whenever we quote the standard regarding an undefined behavior, we quote Annex J. However, this annex is non-normative and the actual wording in the normative body of the

many of the other items on the list. Also, each item might represent more than one kind of undefined behavior and we tend to refer to each item as a "category" because of this. For example, the item covering division by zero contains two distinct instances of undefinedness: "The value of the second operand of the / or % operator is zero" [24, UB #45].

Our classification appears in Figure 1.1. We categorize each behavior by the point in the translation or execution process at which it can be recognized. "Compile time" behaviors, for example, are those that can always be detected using only per-translation unit static analysis, while "link time" behaviors require whole-program static analysis to catch every instance.

Technically, even a behavior that we categorize as detectable at compile or link time might only render the individual execution that encounters it undefined, and not the whole program. But we believe that strictly-conforming[3] programs should be free of such behaviors, and that analysis tools should report them, even if they might be unreachable during execution. For example, this program has unreachable but statically-detectable undefinedness caused by attempting to use the value of a void expression [24, UB #23]:

```
1 int main() {
2       if (0) (void)1 + 1;
3 }
```

standard might differ significantly, but the appendix tends to provide a more clear and concise description.

[3] A program which does not "produce output dependent on any unspecified, undefined, or implementation-defined behavior" [24, §4:5].

Classification	UBs	Coverage	CERT UB ids.[a]
Core language	77	~77 (100%)	4, 8–26, 32, 33, 35–89.
compile time[b]	23	23	
link time[c]	8	8	
run time	46	46	
Early translation[d]	24	~1 (4%)	2, 3, 6, 7, 27–31, 34, 90–99, 101, 102, 104, 107.
lexical	11	1	
macros	13	0	
Standard library	101	~49 (49%)	5, 100, 103, 105, 106, 108–203.
compile time	17	6	
run time	84	43	
Other	1	0	1.
Total	203	~127 (63%)	

[a]The numbers we use to identify a particular category of undefined behavior are the same as in the CERT C Secure Coding Standard [45] and correspond to the order in which each behavior appears in Annex J of C11 (see Appendix A).

[b]Completely detectable by translation phase 7.

[c]Completely detectable by translation phase 8.

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

Figure 1.1. Breakdown of undefined behavior in C11.

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

Just giving a semantics for correct programs (as Ellison and Roşu [13] did) is almost never enough to catch undefined ones. In fact, more of our time has been spent tailoring our semantics to catch incorrect programs than was spent developing the original semantics for correct programs. Capturing undefined

	Rewrite rules	Files	SLOC
Translation-specific semantics	1200	75	5680
Execution-specific semantics	370	37	2380
Common (used in both trans. and exec.)	1700	64	5960
Library semantics	860	24	4120
Total	4130	200	18140
The kcc script, other glue (Perl, shell)		12	1430
Unit/regression test suite (in C)		2650	56330

Figure 1.2. The size of different parts of the semantics in term of physical source lines of code (SLOC), excluding blank lines and comments, and the number of rewrite rules, as of November 2017. The test suite mentioned above is the set of programs we execute with the semantics after every change to ensure its correctness.

behavior has more than tripled the size of the semantics, from 1163 rewrite rules in the original semantics [13] to over 4000 in our current version (see Figure 1.2 for details). While many undefined behaviors are fairly trivial to catch and only require a more precise semantics, others need a complicated reworking of models for core language features. When it comes to C, the semantics of correct programs is only half the battle—the easier, better-understood half, we argue.

Figure 1.1 also lists the current degree to which our semantics captures each category of behavior. This is an approximation based on testing—a behavior being "covered" means we have example programs exhibiting the behavior that our semantics rejects (as well as similar, but correct, programs that control for false positives and that our semantics accepts). Although our semantics does not treat translation phases 1–6 (parsing and preprocessing), the infrastructure we use

to execute the semantics will identify some issues with programs during these phases.

1.2 Examples of undefined behavior

As mentioned above, undefined behavior permeates all parts of the C language. Below, we present a few examples of the many varieties of undefined behavior our semantics covers.

Unsequenced side-effects. Unsequenced writes or an unsequenced write and read of the same object is undefined [24, UB #35]. For example, this program would seem to return 3, and it does when compiled with Clang or ICC:

```
1 int main() {
2       int x = 0;
3       return (x = 1) + (x = 2);
4 }
```

However, it is actually undefined because multiple writes to the same location must be sequenced [24, UB #35], but the operands in the addition have an unspecified evaluation order. Compiled with GCC, this same program returns 4.

Strict aliasing violations. Each object in memory, according to the strict aliasing rules, has an "effective type" and a compiler is allowed to assume accesses to such objects will only occur through an lvalue expression of either a compatible type (modulo qualifiers and signedness) or a character type. The effective type

of an object is its declared type or (in the case of memory allocated with `malloc`) the type of the lvalue through which the last store occurred. For example, line 4 below results in undefined behavior according to the strict aliasing rules [24, UB #37]:

```
1 int main() {
2       int x = 0;
3       long *p = (long*)&x;
4       *p;
5 }
```

These aliasing restrictions [24, §6.5] are intended to allow optimizations using type-based alias analysis and are often surprising to C programmers who envision an untyped memory where stored objects are just unstructured swathes of bytes.

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

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

```c
int main() {
      const char p = 'x';
      *(char*)&p = 'y';
}
```

The ability to manipulate the "object representation" of objects through pointers to `char` is an important feature of C, yet writing to any part of an object declared with `const` through such a pointer still invokes undefinedness. [24, UB #64]

The `restrict` qualifier. The standard gives clear license for implementations to elide it: "deleting all instances of [`restrict`] from all preprocessing translation units composing a conforming program does not change its meaning" [24, §6.7.3:8]. But, as with `const`, deleting all instances of `restrict` potentially causes a program containing undefinedness to become well-defined. In fact, the standard gives the meaning of the `restrict` qualifier entirely in terms of what becomes undefined, in a whole section section devoted to the topic [24, §6.7.3.1, UB #68, 69]. In a sense, the entire purpose of qualifiers like `const` and `restrict` is to allow programmers the ability to add extra undefinedness to their programs.

These global-scope declarations

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

assert, according to the standard, that "if an object is accessed using one of a, b, or c, and that object is modified anywhere (at all) in the program, then it is

not also accessed through the other two" [24, §6.7.3.1]. And two categories of undefined behaviors are dedicated to the `restrict` qualifier:

- An object which has been modified is accessed through a `restrict`-qualified pointer to a const-qualified type, or through a `restrict`-qualified pointer and another pointer that are not both based on the same object. [24, UB #68]
- A `restrict`-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment. [24, UB #69]

Invalid pointers. One of the most canonical examples of undefined behavior might be dereferencing an invalid pointer, but, in fact, the dereference step is often not required to invoke undefined behavior:

```
1 int main() {
2       int *p;
3       { int x = 0; p = &x; }
4       p;
5 }
```

Any use at all of an invalid pointer, such as on the fourth line above, is undefined behavior [24, UB #10].

Pointer provenance. Consider this example, inspired by Defect Report #260 [23].

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

Despite the standard requiring arrays to be contiguously allocated, and therefore bit patterns representing the p and q pointers should compare equal on line 4, it does not follow that the assignments on lines 5 and 6 are both defined. In fact, the assignment on line 5 is well-defined, but the assignment on line 6 invokes undefined behavior according to the following criteria:

> Addition [...] of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated. [24, UB #47]

The moral here is that the provenance of pointers can be significant when determining whether operations performed on them are defined. When checking if a dereference is within bounds, it matters whether a pointer value was based on an array or whether it came from the & operator applied to a scalar.

Checking the type or value at the memory location before a dereference is not enough.

Translation phase. C11 lists eight "translation phases" [24, §5.1.1.2]. The first six phases involve preprocessing. The seventh corresponds to the actions taken by implementations to transform the C source files of a translation unit into a compiled object file. The eighth, which corresponds to linking, governs how multiple translation units are combined.

Our semantics gives a separate treatment to the three major phases of C implementations: compilation, linking, and execution (see Figure 3.1). This brings our semantics very close to how C compilers actually operate. In particular, kcc can separately translate translation units and later link them together to form an executable. We use this technique, for example, to pre-translate the standard library in order to speed up the time it takes the tool to interpret C programs.

As an example of translation-time undefinedness, the following program relies on a common language extension for declaring a global identifier shared between multiple translation units and will usually not elicit a warning from GCC or Clang:

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

```
1 // Trans. unit 2.
2 int x = 0;
```

But it is a language extension, and as such it relies on undefined behavior. Each of those global declarations of `x` constitutes a "tentative definition," which becomes a real definition by the end of the translation unit. The program is therefore undefined because the identifier `x` has multiple external definitions [24, UB #84].

Most C implementations handle compiling and linking in separate phases, and by the time the linking phase is reached, most typing information has been lost. As a result, for example, compilers generally will not issue a warning about the undefinedness that results from the type incompatibility of the call to `f` below [24, UB #41]:

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

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

Chapter 2

$\mathbb{K}$ framework semantics

We have developed our semantics in the rewriting-based $\mathbb{K}$ executable semantic framework[1] [2], inspired by rewriting logic [34]. Rewriting logic organizes term rewriting *modulo equations* (namely associativity, commutativity, and identity) as a logic with a complete proof system. The central idea behind using such a formalism for the semantics of languages is that the evolution of a program can be clearly described using rewrite rules. A rewriting semantics in $\mathbb{K}$ consists of a syntax (or signature) for describing terms and a set of rewrite rules that describe steps of computation. Given some term allowed by a signature (e.g., a program together with input), deduction consists of the application of the rules to that term, yielding a transition system for any program.

Besides C [13, 12, 19], $\mathbb{K}$ has been been used to define many other real-world programming languages, including modern object-oriented languages

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

like Java [6], scripting languages like JavaScript [43], Python [16], and PHP [14], and recently EVM [20], the virtual machine of the Ethereum cryptocurrency blockchain.

From a $\mathbb{K}$ operational semantics, we can derive many useful tools, such as an interpreter (e.g., kcc) and a program verifier [1]. These tools can be extracted and verification performed directly from the $\mathbb{K}$ executable operational semantics, without the need for an additional axiomatic semantics (and the burden of maintaining the necessary equivalence proofs). This is an advantage because operational semantics tend to be easier to write than other semantic styles and have a clear computational interpretation, so we can gain confidence in its correctness by rigorous testing.

To get an idea of how the $\mathbb{K}$ semantics of C works, below we present the complete $\mathbb{K}$ operational semantics for a simple imperative language (sometimes called WHILE or IMP). This same language is revisited in Chapter 5, where we discuss what C-style undefined behavior might look like in it. See Roșu and Șerbănuță [3] for a more in-depth example and discussion.

2.1 Example: the semantics of IMP

The syntax for a language defined in $\mathbb{K}$ is given in an annotated BNF style. First, we present the syntax for expressions:

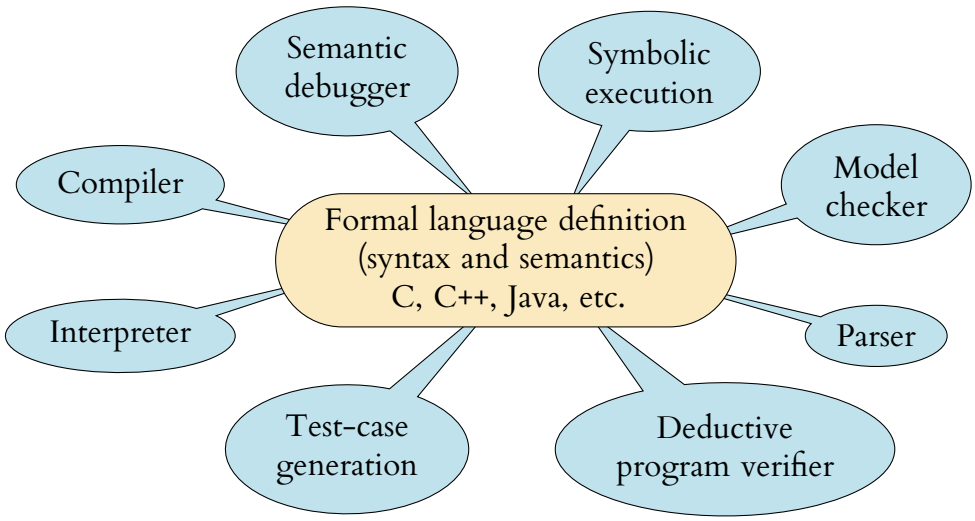


Figure 2.1. The $\mathbb{K}$ semantic framework. Semantics-based tools can be extracted and program verification performed from an executable operational semantics (without the need for an additional axiomatic semantics).

```
syntax AExp  ::= Int | Id
             | AExp "/" AExp [left, strict]
             > AExp "+" AExp [left, strict]
             | "(" AExp ")" [bracket]
syntax BExp  ::= Bool
             | AExp "<=" AExp [seqstrict]
             | "!" BExp [strict]
             > BExp "&&" BExp [left, strict(1)]
             | "(" BExp ")" [bracket]
```

The `[strict]` annotations have semantic meaning: the sub-expressions of the annotated production will be evaluated to the KResult sort before the annotated production. Next, the syntax of statements:

```
syntax Block ::= "{" "}"
             | "{" Stmt "}"
syntax Stmt  ::= Block
             | Id "=" AExp ";" [strict(2)]
             | "if" "(" BExp ")"
                  Block "else" Block [strict(1)]
             | "while" "(" BExp ")" Block
             > Stmt Stmt [left]
syntax Pgm ::= Stmt
syntax Ids ::= List{Id,","}
```

From this grammar, $\mathbb{K}$ will automatically generate a parser for IMP. To get a feel for the intended meaning of this language, here is a program in IMP for calculating the sum of numbers from 1 to 100:

```
n = 100;
sum = 0;
while (!(n <= 0)) {
  sum = sum + n;
  n = n + -1;
}
```

A $\mathbb{K}$ semantics comprises, in addition to the definition of the syntax given by the grammar above, a configuration and a set of rules. The configuration, given by a labeled, nested multi-set of cells and atomic data structures (e.g., sets, lists, maps), represents the state of the interpreter defined by the semantics. The rules, then, govern how the interpreter transitions from one state to the next by rewriting parts of the configuration.

The configuration for this language is very simple. The only pieces of state used by the semantics of IMP consist of the `k` and `sto` cells, the former being used for sequencing computation and the latter holds a map associating identifiers with their values:

$$\langle \langle \mathsf{K} \rangle_{\mathsf{k}} \langle \mathsf{Map} \rangle_{\mathsf{sto}} \rangle_{\mathsf{T}}$$

The k cell holds a list (with elements conventionally separated by the "↷" character in rewrite rules) of computations to be evaluated. Computations are sequenced by a convention of matching only the head of this list during rewriting. Sub-computations to be evaluated next are moved to the head.

Now for the rules. The semantics of arithmetic expressions in IMP are given by only two rules:

```
rule I1 / I2 => I1 /Int I2 requires I2 =/=Int 0
rule I1 + I2 => I1 +Int I2
```

Rules match a part of the configuration of the left of the "⇒" and rewrite it to what appears on the right. When the configuration cells are omitted, the rules implicitly apply only to the top of the $\langle\cdot\rangle_{\texttt{k}}$ cell. I.e., the above two rules are equivalent to these:

```
rule <k> I1 / I2 => I1 /Int I2 ... </k> requires I2 =/=Int 0
rule <k> I1 + I2 => I1 +Int I2 ... </k>
```

Rules might optionally bind parts of the configuration to a variable (I_1 and I_2 in the rules above) and have additional constraints given by the `requires` directive. The rule for division, for example, does not apply when the divisor is zero.

The semantics of boolean expressions are also straightforward:

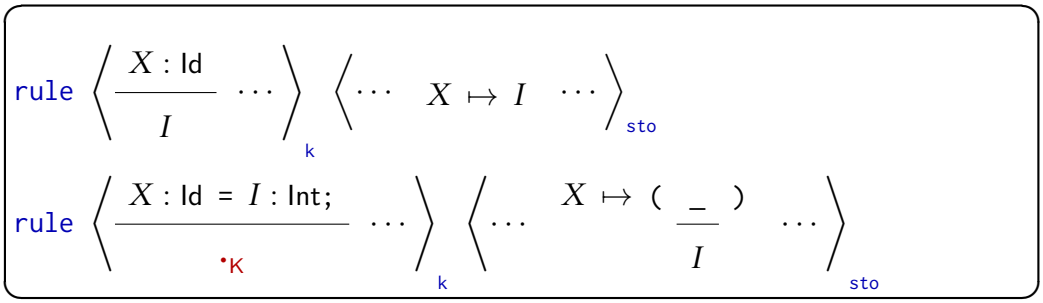
```
rule I1 <= I2 => I1 ≤Int I2

rule ! B => ¬ B

rule true && B => B

rule false && _ => false
```

And statements:

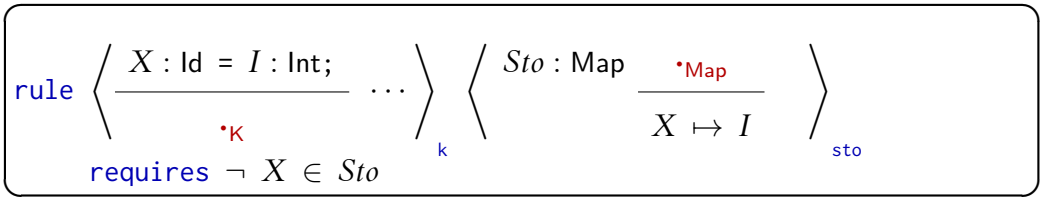
```
rule {} => •K

rule {S} => S

rule S1 : Stmt S2 : Stmt => S1 ↷ S2

rule if (true)  S else _ => S

rule if (false) _ else S => S

rule while (B) S => if (B) S while (B) S else {} [structural]
```

Notice the appearance of "$\curvearrowright$" on the right-side of the rule for sequencing the evaluation of statements. This rule sequences the evaluation of two adjacent statements by evaluating the first before the second.

Finally, we get to the semantics of variable lookup and assignment, which must access the `sto` cell to retrieve and modify the value associated with a particular variable. In these rules, we see what is implicit in the previous rules dealing with expressions and statements: all of these rules have been rewriting the head of the `k` cell. This convention facilitates sequencing computation.



These rules replace an identifier with its value in the store and, in the case of the assignment statement, update the value associated with an identifier in the store. Notice that both of these rules only apply when X is already in the store. We could add syntax to this language for declaring variables, but instead we will add another rule for assignment that allows new entries in the store to be created:



This completes the semantics of IMP. Despite the concise style of $\mathbb{K}$, the C semantics is much larger. The configuration in Figure 2.2 is only a small subset of the configuration from our C semantics, which contains around 100 such cells in the execution semantics and another 60 in the translation semantics. The $\mathbb{K}$ style of rewriting allows us to relatively easily manage this much state, however, because the rules are contextual. As another example, consider a typical rule for a simple imperative language for dereferencing a pointer:

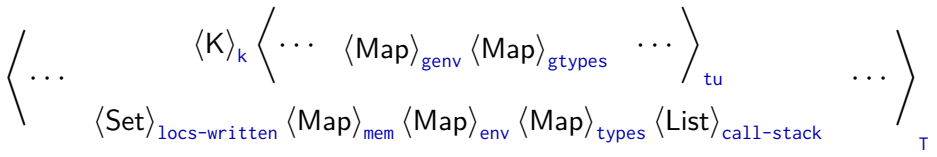
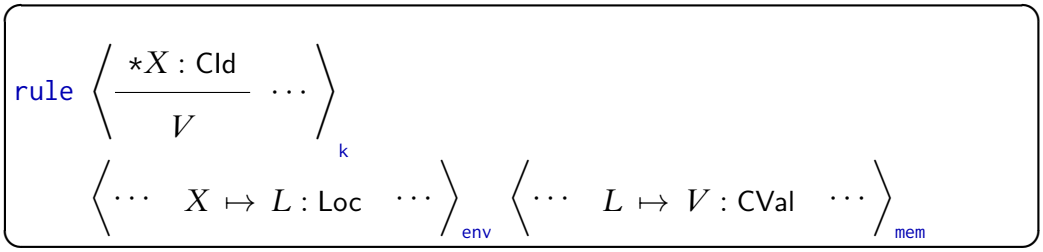


Figure 2.2. A small excerpt of the configuration used by the C semantics. The actual configuration contains over 100 cells in the execution semantics with another 60 in the translation semantics.



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

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

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

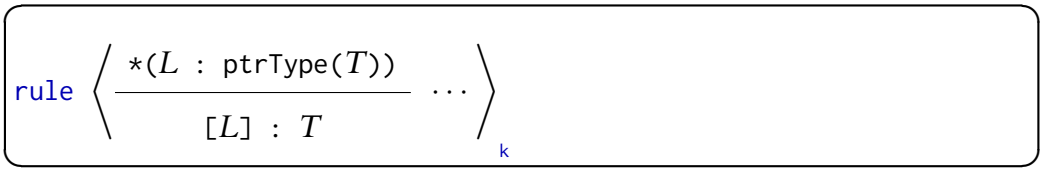
Items in the k cell are separated with "$\curvearrowright$," which can now be seen. The κ and ρ_1, ρ_2, σ_1, σ_2 take the place of the "$\cdots$" above. Nearly the entire rule is duplicated on the right-hand side. Duplication in a definition requires that changes be made in concert, in multiple places. If this duplication is not kept in sync, it leads to subtle semantic errors. In a complex language like C, the configuration structure is much more complicated, and would require actually including additional cells like genv and call-stack. These intervening cells are automatically inferred in $\mathbb{K}$, which keeps rules modular.

Chapter 3

Catching C undefined behavior

Just giving a semantics for correct programs is almost never enough to catch undefined ones. While many undefined behaviors are fairly trivial to catch and only require a more precise semantics, others need a complicated reworking of models for core language features.

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



Dereferencing a location L of type pointer-to-T yields an lvalue L of type T ($L : T$). This rule is correct according to the semantics of C—it works for

any defined program. However, it fails to detect undefined behaviors, such as dereferencing `void` or `NULL` pointers [24, UB #23, 43]. In:

```
1 int main() {
2       *NULL;
3       return 0;
4 }
```

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

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

$$\texttt{rule}\ \left\langle \frac{\texttt{*(}L\ \texttt{:}\ \texttt{ptrType(}T\texttt{))}}{\texttt{[}L\texttt{]}\ \texttt{:}\ T}\ \cdots \right\rangle$$
$$\texttt{requires}\ T \neq_{\texttt{K}} \texttt{void} \wedge L \neq^{\texttt{k}}_{\texttt{K}} \texttt{NULL}$$

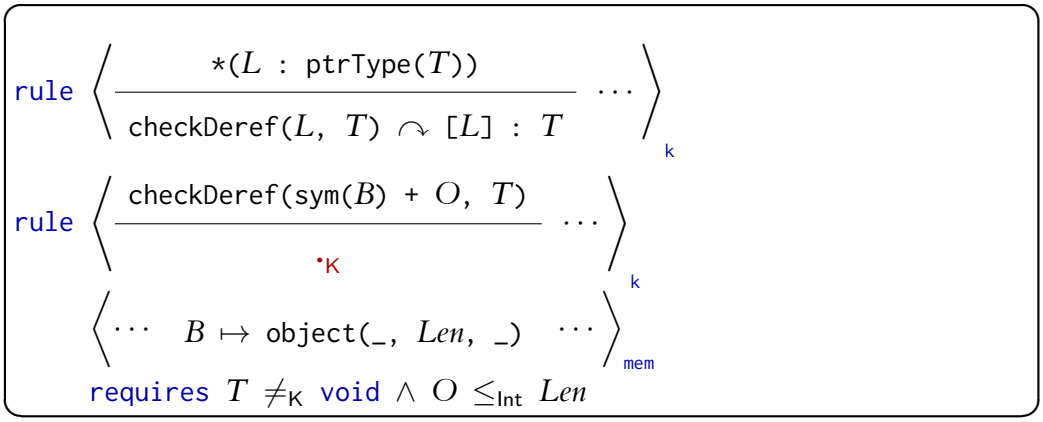
If this is the only rule in the semantics for pointer dereferencing, then the semantics will get stuck when trying to dereference `NULL` or trying to dereference a void pointer.

But we also need to eliminate the possibility of dereferencing memory that is no longer "live"—either variables that are no longer in scope, or allocated memory that has since been `free`d. Here, then, is the most verbose version of this rule that takes all this into account:

$$\texttt{rule}\ \left\langle \frac{\texttt{*(sym(}B\ \texttt{+}\ O\ \texttt{:}\ \texttt{ptrType(}T\texttt{))}}{\texttt{[sym(}B\texttt{)}\ \texttt{+}\ O\texttt{]}\ \texttt{:}\ T}\ \cdots \right\rangle$$
$$\left\langle \cdots\ B \mapsto \texttt{object(\_,}\ \mathit{Len},\ \texttt{\_)}\ \cdots \right\rangle^{\texttt{k}}_{\texttt{mem}}$$
$$\texttt{requires}\ T \neq_{\texttt{K}} \texttt{void} \wedge O \leq_{\texttt{Int}} \mathit{Len}$$

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 `sym(`B`) + `O and objects in memory are represented by a tuple containing their type, size in bytes, and the object representation as a list of bytes.

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



The actual rule for dereferencing pointers from our semantics uses a combination of these techniques, but it also must take into account misuse of the `restrict` qualifier (Section 3.3.1) and the strict aliasing rules (Section 3.2). These examples should demonstrate how the simple and straightforward rules needed for characterizing defined programs quickly become quite complicated when undefinedness must also be ruled out.

3.1 Translation, linking, and execution

C11 lists eight "translation phases" [24, §5.1.1.2]. The first six phases involve preprocessing. The seventh corresponds to the actions taken by implementations to transform the C source files of a translation unit into a compiled object file. The eighth, which corresponds to linking, governs how multiple translation units are combined:

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

Our semantics gives a separate treatment to the three major phases of C implementations: compilation, linking, and execution (see Figure 3.1). This brings our semantics very close to how C compilers actually operate. In particular, kcc can separately translate translation units and later link them together to form an executable. We use this technique, for example, to pre-translate the standard library in order to speed up the time it takes the tool to interpret C programs.

As we saw in Figure 1.1, many undefined behaviors are statically-detectable, and many of these deal with the semantics of declarations and building typing environments. It is tempting to not take these behaviors too seriously because many of these issues are readily reported on by compilers and static analyzers. But the semantics for building typing environments and dealing with the

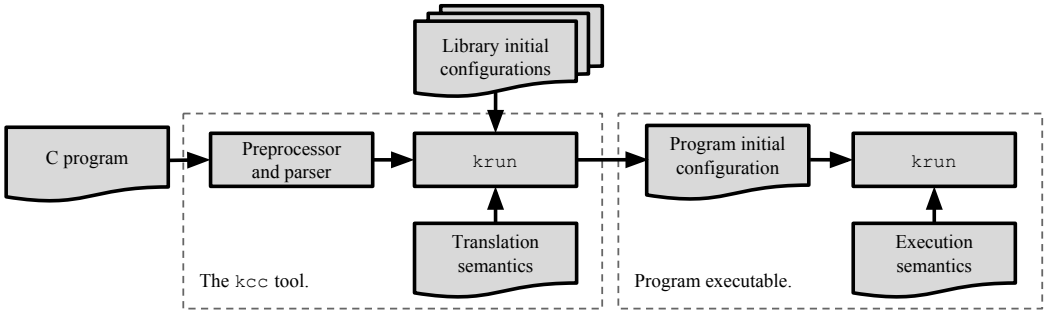


Figure 3.1. An overview of the design of the kcc tool. krun is a $\mathbb{K}$ Framework utility for extracting an interpreter from a semantics.

linkage of identifiers is notoriously tricky in C because multiple declarations, both within the same translation unit and across multiple translation units, at file scope and block scope, might refer to the same object. And even easily-detectable static undefinedness can slip through compilers and produce unexpected results. For example, consider the following program, composed of two translation units:

```c
// Trans. unit 1.
static int a[];
int main() {
        return a[0];
}
```

```c
// Trans. unit 2.
int a[] = {1, 2, 3};
```

GCC and Clang both compile this program, but they disagree about the value it returns. The GCC-compiled program returns 1, while the Clang compiled

program returns 0. Both behaviors are allowed by the standard because the `static` declaration of `a`, with an incomplete array type, invokes undefined behavior: "an object with internal linkage and an incomplete type is declared with a tentative[1] definition" [24, UB #89].

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

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

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

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

All four states are needed to prevent various kinds of malformed declarations. We must distinguish "allocated" symbols from "defined" in order to pre-

[1] "A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier `static`, constitutes a *tentative definition*." [24, §6.9.2:2]

vent multiple definitions [24, UB #84], for example, and a declaration left in the "declared" state by the end of a translation unit might be an incomplete tentative definition that must be completed and allocated.

Linking In translation phase eight, then, we combine translation units together by resolving each identifier with external linkage that appears in an expression (i.e., every identifier that is actually used) to a single definition. We also must verify that declarations for the same identifier are compatible [24, UB #15] and duplicate definitions do not exist [24, UB #84].

Most C implementations handle compiling and linking in separate phases, and by the time the linking phase is reached, most typing information has been lost. As a result, for example, compilers generally will not issue a warning about the undefinedness that results from the type incompatibility of the call to `f` below [24, UB #41]:

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

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

The CompCert compiler, due to its current reliance on an external linker, suffers from this same limitation [31]. Detecting such issues requires whole-program

static analysis and because of the subtleties of type compatibility and the rules governing the linkage of identifiers, this can be hard to get right.

Other similar cases of link time undefinedness also seem to slip by compilers without warnings. For example, the following program relies on a common language extension for declaring a global identifier shared between multiple translation units and will usually not elicit a warning from GCC or Clang:

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

```c
// Trans. unit 2.
int x;
...
```

But it is a language extension, and as such it relies on undefined behavior. Each of those global declarations of `x` constitutes a "tentative definition," which becomes a real definition by the end of the translation unit. The program is therefore undefined because the identifier `x` has multiple external definitions [24, UB #84].

3.2 Effective types

The strict aliasing rules. The notorious "strict aliasing" rules appear in §6.5 of the C11 standard:

> An object shall have its stored value accessed only by an lvalue expression that has one of the following types

1. a type compatible with the effective type of the object,
2. a qualified version of a type compatible with the effective type of the object,
3. a type that is the signed or unsigned type corresponding to the effective type of the object,
4. a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
5. an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
6. a character type. [24, §6.5]

Each object in memory, according to these rules, has an "effective type" and a compiler is allowed to assume accesses to such objects will only occur through an lvalue expression of either a compatible type (modulo qualifiers and signedness) or a character type. The effective type of an object is its declared type or (in the case of memory allocated with `malloc`) the type of the lvalue through which the last store occurred. For example, line 4 below results in undefined behavior according to the strict aliasing rules [24, UB #37], even if objects of type `int` and `long` happen to have the same size and object representation on an implementation:

```
1 int f(int * x, long * y) {
2       *x = 0;
3       *y = 1;
4       return *x;
5 }
6 int main() {
7       void * p = malloc(sizeof(int));
8       return f((int*) p, (long*) p);
9 }
```

Figure 3.2. An example of a program with undefined behavior due to violating the strict aliasing rules. It will return `0` or `1` depending on whether gcc is optimizing.

```
1 int main() {
2       int x = 0;
3       long *p = (long*)&x;
4       *p;
5 }
```

These aliasing restrictions are intended to allow optimizations using type-based alias analysis. For example, the programs in Figure 3.2 and Figure 3.3 both return either `0` or `1` when compiled with `-O0` vs. `-O2` in gcc 4.8 because it assumes, based solely on the types, that the `x` and `y` pointers cannot point to the same object.

```c
struct s { int a; int b; };
int f(struct s* x, long * y) {
        x->b = 0;
        *y = 1;
        return x->b;
}
int main() {
        void * p = malloc(sizeof(struct s));
        return f((struct s*)p, (long *) (&((struct s*)p)->b));
}
```

Figure 3.3. Another example of a program with undefined behavior due to violating the strict aliasing rules. It will return `0` or `1` depending on whether gcc is optimizing.

This can be surprising to C programmers who envision an untyped memory where stored objects are unstructured swathes of bytes. In fact, the standard treats stored objects in a more nuanced (or contradictory) way: for the purposes of the aliasing restrictions, objects in memory have a type that can be updated with subsequent stores, but the standard also allows for these same objects to have their memory representation accessed and modified directly through a `char` pointer. This is further complicated by `union` variants [22], flexible array members, and objects in allocated-duration regions, which update their effective type on subsequent accesses. Below we outline our approach to these problems.

3.2.1 Semantics of the effective type check

Lvalue expressions have an associated last-access object type, which is a composite type formed from the current effective type and the compatible type of the lvalue expression based on it (which may only refer to a part of the object). By composite type, we mean the effective type with extra annotations for tracking locked union variants. When a write occurs through an lvalue expression, its associated last-access type becomes the effective type of the object.

Checking for effective type violations, then, occurs with every memory access through an lvalue (with type L) at a certain byte offset (Off) in an object. Below, LAT is either the last-access or effective type of the object being accessed, depending on whether the lvalue expression has caused previous accesses:

```
syntax K ::= chkEffType(Type, Int, EffectiveType) [function]
// Objects with a declared type.
rule chkEffType(L : Type, Off : Int, LAT : Type) => •K
     requires effCompat(L, getTypesAtOffset(LAT, Off))
// Allocated-duration objects.
rule chkEffType(_, _, dynamicType(_ : NoType)) => •K
rule chkEffType(L : Type, Off : Int, dynamicType(LAT : Type)) => •K
     requires effCompat(L, getTypesAtOffset(LAT, Off))
rule chkEffType(L : Type, 0, dynamicType(LAT : Type)) => •K
     requires effCompat(LAT, getTypesAtOffset(L, 0))
// Otherwise, this access violates the strict-aliasing rules.
rule chkEffType(_, _, _) => Effective type violation. [owise]
```

For objects with a declared type, the access is allowed if the type of the lvalue is compatible with one of the types at the access offset into the declared type. Consider the program in Figure 3.4. The lvalue expression on line 6 (`x[1].b.j`) results in 4 accesses that each trigger the above effective type check. In Figure 3.5 we give the values of the L, *Offset*, and LAT parameters for each of these checks (where the syntax `union u.b` indicates the type `union u` with the `b` variant active). After line 6, for example, the following accesses would be allowed:

```
*((int *) &x[0]);
*((struct s *) &x[1]);
```

But these result in undefined behavior:

```
1 struct s { int i; int j; };
2 union u { int a; struct s b; };
3 int main() {
4       union u x[2];
5       x[0].a = 10;
6       x[1].b.j = 20;
7 }
```

Figure 3.4. Example of an array of `union`s with different active variants. The object at `x` at the end of the execution of `main` has an effective type of an array of `union`s with different active variants.

LValue		*L*	*Off*	*LAT*
`x`	▷ `chkEffType (`	`union u [3]`	`, 0`	`, [union u.a, union u] )`
`x[1]`	▷ `chkEffType (`	`union u`	`, 12`	`, [union u.a, union u] )`
`x[1].b`	▷ `chkEffType (`	`struct s`	`, 16`	`, [union u.a, union u.b] )`
`x[1].b.j`	▷ `chkEffType (`	`int`	`, 20`	`, [union u.a, union u.b] )`

Figure 3.5. The effective type checks resulting from the lvalue expression on line 6 of Figure 3.4. For calculating the offsets, we assume `sizeof(int)` is 4 and `sizeof(union u)` is 12. The syntax `union u.b` indicates the type `union u` with the `b` variant active.

```
*((struct s *) &x[0]);
*((int *) &x[1]);
```

For allocated-duration objects (indicated by the `dynamicType` production in the effective type check above, in contrast to objects with a declared type), there are three cases where the access is allowed:

1. when there have been no previous accesses to the object (the effective type of the object is `NoType`),

2. when there has been previous access to the object such that the resulting effective type at the offset of this access is compatible with the type of this lvalue, and

3. there has been previous access to the object, and the type of this lvalue appears to be a more refined version of the effective type of the object (e.g., the type of this access is a `struct`, while the current effective type of the object is the type of the first member).

The extra provisions for access to allocated-duration objects allow the effective type of such objects to be continually refined on subsequent accesses, such as in the program in Figure 3.6 (though even for objects with a declared type, the effective type can be refined in order to record active `union` variants, as we saw above). Note that this interpretation of the implications of the effective type rules for allocated-duration objects disallows custom allocators for arbitrary objects to be written using `malloc` because an allocator would require the effective

type of the allocated region to, in effect, be a heterogenous array of distinct object types.

So the effective type check reduces to the definitions of `effCompat` and `getTypesAtOffset`. We present the semantics for `effCompat` below and then sketch the definition of `getTypesAtOffset` as an algorithm in pseudocode. First, a type L is compatible with a set of types iff it is either a `char` type or it is compatible with one of the types in the set:

```
syntax Bool ::= effCompat(Type, Set) [function]
rule effCompat(_ : CharType, _) ⇒ true
rule effCompat(L : Type, Eff : Type _) ⇒ true
     requires effCompat′(L, Eff)
rule effCompat(L : Type, (Eff : Type ⇒ •Set) _)
     requires ¬ effCompat′(L, Eff)
rule effCompat(_, _) ⇒ false [owise]
```

Now the definition of `effCompat′` covers the remaining items 1–5 of the strict aliasing rules (see Section 3.2[24, §6.5]):

```c
#include <stdlib.h>
struct foo { int x; char * y; };
struct bar { struct foo x; };
int main() {
  void *p = calloc(sizeof(struct bar), 1);
  int *p2 = p;            // The effective type of *p after each line:
  *p2 = 1;                //  ∅ ⇒ int
  struct foo * p3 = p;
  p3->y = "foo";          //  ⇒ struct foo { int; char *; }
  struct bar * p4 = p;
  p4->x.y = "bar";        //  ⇒ struct bar { struct foo; }
  struct bar (*p5)[1] = p;
  (*p5)[0].x.y = "baz";   //  ⇒ struct bar [1]
  free(p);
}
```

Figure 3.6. This program is allowed by our treatment of effective types. The type of the object at *p is refined by each subsequent compatible access, as described in the comments.

```
syntax Bool ::= effCompat'(Type, Type) [function]
rule effCompat'(L : Type, Eff : Type) => true
     requires stripQualifiers(L) =Type stripQualifiers(Eff)
          ∧ getQualifiers(Eff) ⊆Quals getQualifiers(L)
rule effCompat'(L : SignedIntegerType, Eff : Type) => true
     requires correspondingUnsigned(L) =Type stripQualifiers(Eff)
          ∧ getQualifiers(Eff) ⊆Quals getQualifiers(L)
rule effCompat'(L : UnsignedIntegerType, Eff : Type) => true
     requires ¬ isBoolType(L)
          ∧ correspondingSigned(L) =Type stripQualifiers(Eff)
          ∧ getQualifiers(Eff) ⊆Quals getQualifiers(L)
rule effCompat'(_, _) => false [owise]
```

The `getTypesAtOffset` production handles getting all types at a particular offset, which may changed based on which union variant is active. For example,

$$\texttt{getTypesAtOffset(int, 0)} = \{\texttt{int}\}.$$

Similarly, for the definitions of `union u` and `struct s` given above,

$$\texttt{getTypesAtOffset(union u.b, 0)} = \{\texttt{union u}, \texttt{struct s}, \texttt{int}\}.$$

We present the definition of `getTypesAtOffset` in Figure 3.7 as a pseudocode algorithm.

function `getTypesAtOffset`(τ : Type, $\delta : \mathbb{N}$) : Set

  if $\delta = 0$ **then**

    if τ is a `union` with active variant τ' **then**

      return $\{\tau\} \cup$ `getTypesAtOffset`(τ', 0)

    else if τ is an aggregate with member type τ' at offset 0 **then**

      return $\{\tau\} \cup$ `getTypesAtOffset`(τ', 0)

    else return $\{\tau\}$

  else

    if τ is a `union` with active variant τ' **then**

      return `getTypesAtOffset`(τ', δ)

    else if τ is an aggregate with member type τ' at offset δ' **then**

      return `getTypesAtOffset`(τ', $\delta - \delta'$)

    else return $\emptyset$

Figure 3.7. The definition of `getTypesAtOffset`.

When a write actually occurs, the new effective type of the object in memory becomes the last access type associated with the lvalue (via the provenance mechanism):

```
syntax K ::= write(SymLoc, CValue, Type)
rule write(Loc : SymLoc, V : CValue, T : Type)
     ⇒ updateEffectiveType(Loc)
     ↷ ...
```

Where `updateEffectiveType` will set the new effective type to the last-access type associated with the lvalue, if it exists:

```
syntax K ::= updateEffectiveType(SymLoc) [function]
rule updateEffectiveType(Loc : SymLoc)
     ⇒ setEffectiveType(base(Loc), getLastAccessType(Loc))
     requires hasLastAccessType(Loc)
rule updateEffectiveType(Loc : SymLoc) ⇒ •K [owise]
```

The actual effective type of an object is stored with the stored value of the object in the $\langle\cdot\rangle_{\texttt{mem}}$ cell:

```
syntax K ::= setEffectiveType(SymBase, EffectiveType)
rule ⟨ setEffectiveType(B : SymBase, T : EffectiveType) ... ⟩k
     ⟨ ... B ↦ object( A : EffectiveType , _, _) ... ⟩mem
                       ──────────────────
                       mergeVariants(A, T)
```

Require: τ, τ' are compatible types.

function `mergeVariants`(τ : Type, τ' : Type) : Type

if τ, τ' are `union`s with the same active variant, with types ν, ν' **then**

return `setActiveVariantType`(τ', `mergeVariants`(σ_i, σ'_i))

else if τ, τ' are aggregates with n element types σ_1, ..., σ_n, σ'_1, ..., σ'_n **then**

for $i = 1$ **to** n **do**

$\tau' \leftarrow$ `setElementType`(τ', `mergeVariants`(σ_i, σ'_i))

return τ'

else return τ'

Figure 3.8. The definition of `mergeVariants`.

The `mergeVariants` production handles the case of merging arrays of types containing `union`s with different active variants, such as in Figure 3.4. We sketch the definition of `mergeVariants` in Figure 3.8 in pseudocode.

3.2.2 Flexible array members

Since C99, the standard has allowed the last member of a `struct` to be an array with an unspecified size, which is then sized to fit the allocated memory area. For example, the last member of this struct is a flexible array member:

```
1 struct s {
2       int x;
3       int y;
4       int z[];
5 };
6 int main(void) {
7       p = malloc(sizeof(struct s) + 40);
8 }
```

3.2.3 Pointer bounds-checking refinement

As we saw in Section 3.4, two pointers that share the same value and compare as equal can still point to two different objects. But we allow conversions between such pointers in some cases. For example, a pointer to the initial field of a struct object can be converted to a pointer to the struct object by casting it to the type of the struct:

```c
#include <string.h>
struct s {
        int x;
        int y;
};
int main() {
        struct s a;
        memset(&a, 0, sizeof(struct s));              // Allowed.
        memset(&a.x, 0, sizeof(struct s));            // Error!
        memset((struct s*)&a.x, 0, sizeof(struct s)); // Allowed.
}
```

Whether such a conversion takes place depends on the current effective type of the object being referenced.

3.3 Type qualifiers

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

First, consider the type qualifier `const`. In C, `const` makes the qualified object unchangeable after initialization. Writes can only occur through non-`const`

types [24, §6.3.2.1:1, 6.5.16:1] and attempting to write to a `const`-qualified object through a non-`const`-qualified lvalue invokes undefinedness [24, UB #64]. Therefore, in correct programs, this qualifier has no effect. But to actually catch misuse of `const` by incorrect programs requires the semantics to keep track of `const` qualifiers and check them during all modifications and conversions.

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

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

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

Other modifiers that need similar special consideration include `restrict`, `volatile`, `_Atomic`, the alignment specifiers (`_Alignas`), and the function specifiers (`inline` and `_Noreturn`). The important point is that objects in memory generally must retain the qualifiers they were declared with in order to verify operations through pointers, which may have dropped the qualifiers. We handle this by maintaining the effective type of every object stored in memory (as described in Section 3.2). From the effective type of an object, we can calculate the effective modifiers at every offset into that object, and with this information detecting modifier-specific misuse generally becomes easy. The type qualifier `restrict`, however, requires extra effort.

3.3.1 Restrict qualifier

This section describes the algorithm we use to detect the misuses of the `restrict` qualifier that result in undefined behavior. Little work on attempting to catch these sorts of bugs in C programs appears to exist in related work.

The standard gives clear license for implementations to elide the `restrict` qualifier: "deleting all instances of [`restrict`] from all preprocessing translation units composing a conforming program does not change its meaning" [24, §6.7.3:8]. Pascal Cuoq suggests[2], for example, that the behavior of the `restrict` keyword can (at least in part) be reproduced using something like this:

```
1 int f(int * /* restrict */ a, int * /* restrict */ b) {
2       if (a == b) *NULL;
3 }
```

Aliasing `restrict`-qualified pointers becomes undefined behavior, so we can reproduce that constraint by checking whether `a == b` and invoking undefined behavior if they are equal.

As with `const`, the standard gives the meaning of the `restrict` qualifier entirely in terms of what becomes undefined, in a whole section section devoted to the topic [24, §6.7.3.1, UB #68, 69]. In a sense, the entire purpose of qualifiers like `const` and `restrict` is to allow programmers the ability to add extra undefinedness to their programs.

Objects accessed through a `restrict`-qualified pointer, and which are modified at some point during the pointer's lifetime, form an association with pointer,

[2] `http://blog.frama-c.com/index.php?post/2012/07/25/On-the-redundancy-of-C99-s-restrict`

causing accesses via other means to be undefined behavior. For example, these global-scope declarations

```
1 int * restrict a;
2 int * restrict b;
3 extern int c[];
```

assert, according to the standard, that "if an object is accessed using one of a, b, or c, and that object is modified anywhere (at all) in the program, then it is not also accessed through the other two" [24, §6.7.3.1]. Two categories of undefined behaviors are dedicated to the `restrict` qualifier:

- An object which has been modified is accessed through a `restrict`-qualified pointer to a const-qualified type, or through a `restrict`-qualified pointer and another pointer that are not both based on the same object. [24, UB #68]
- A `restrict`-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment. [24, UB #69]

It is not just accesses through `restrict`-qualified pointers that must be monitored, therefore, but all accesses. Furthermore, capturing this behavior requires tracking which `restrict`-qualified pointers any particular lvalue expression is based on. Because the value of one `restrict`-qualified pointer variable can be assigned to another in certain cases, a particular pointer expression might be based on multiple `restrict`-qualified pointers. We handle this using the "pointer provenance" mechanism described in Section 3.4.

Consider this block, labeled here as A:

```
1 // A:
2 {
3         int x = 0;
4         int * restrict p = &x;
5 }
```

In this example, p is "based-on" the block A, and, generally speaking, restrict conflicts arise only between restrict-qualified pointers based-on a currently-active block and another pointer. An association gets formed between the restrict-qualified pointer and the object to which it points when that object becomes modified in the scope of the block on which the pointer is based. Any other access to this object while the block is active constitutes a restrict violation.

All accesses must be monitored for restrict violations, but we defer some checks to block exit. We use a stack of maps from object locations to RestrictState, which can be defined as a set representing both the associated `restrict`-qualified pointers and their degree of association:

$$\mathsf{RestrictState} \triangleq \{\bot, \mathsf{R}_S, \mathsf{U}, \mathsf{W}_S, \top\},$$
$$\text{where } S \subseteq \textit{active } \texttt{restrict}\textit{-qualified pointers}.$$

The heart of the restrict check is performed by the symmetric *join* operation, which is used to update the RestrictState associated with a location whenever an access occurs (and when the accesses occurring in the most recent block are

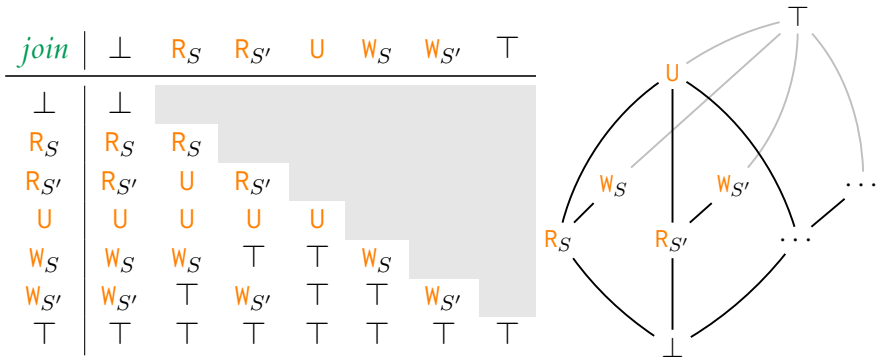
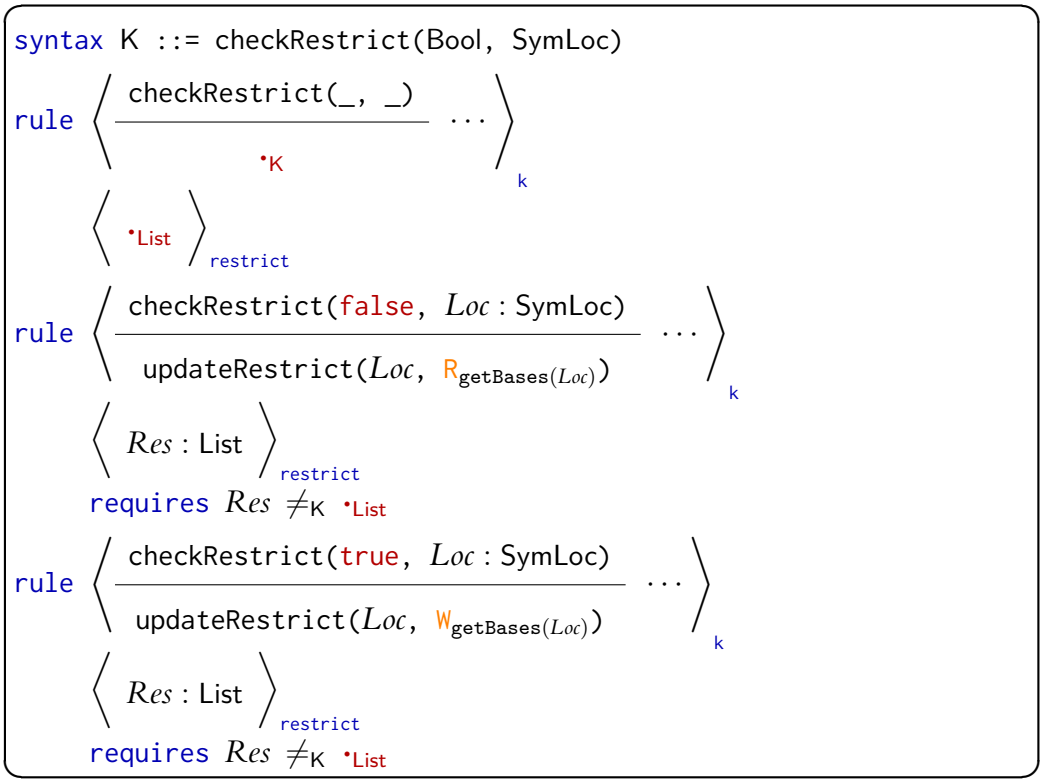


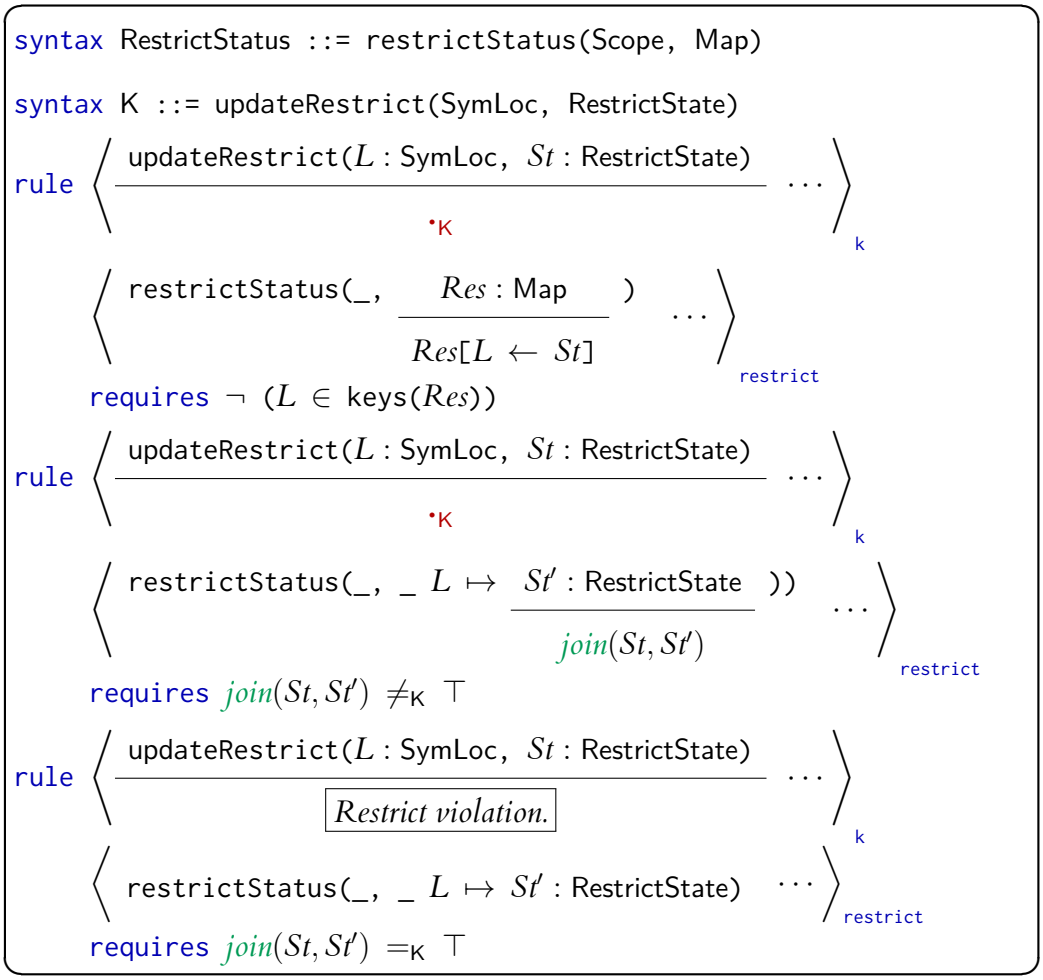
Figure 3.9. Definition of the symmetric *join* : RestrictState × RestrictState → RestrictState operation and a Hesse diagram of the RestrictState lattice, where $S, S' \subseteq$ the active `restrict`-qualified pointers such that $S \neq S'$ and $\top$ represents a `restrict` violation.

merged back with the most recent, previously encountered block). Its definition is given by the diagrams in Figure 3.9. Locations in memory start with an associated RestrictState of $\bot$, which rises toward $\top$ as accesses occur, where the $\top$ state represents a violation.

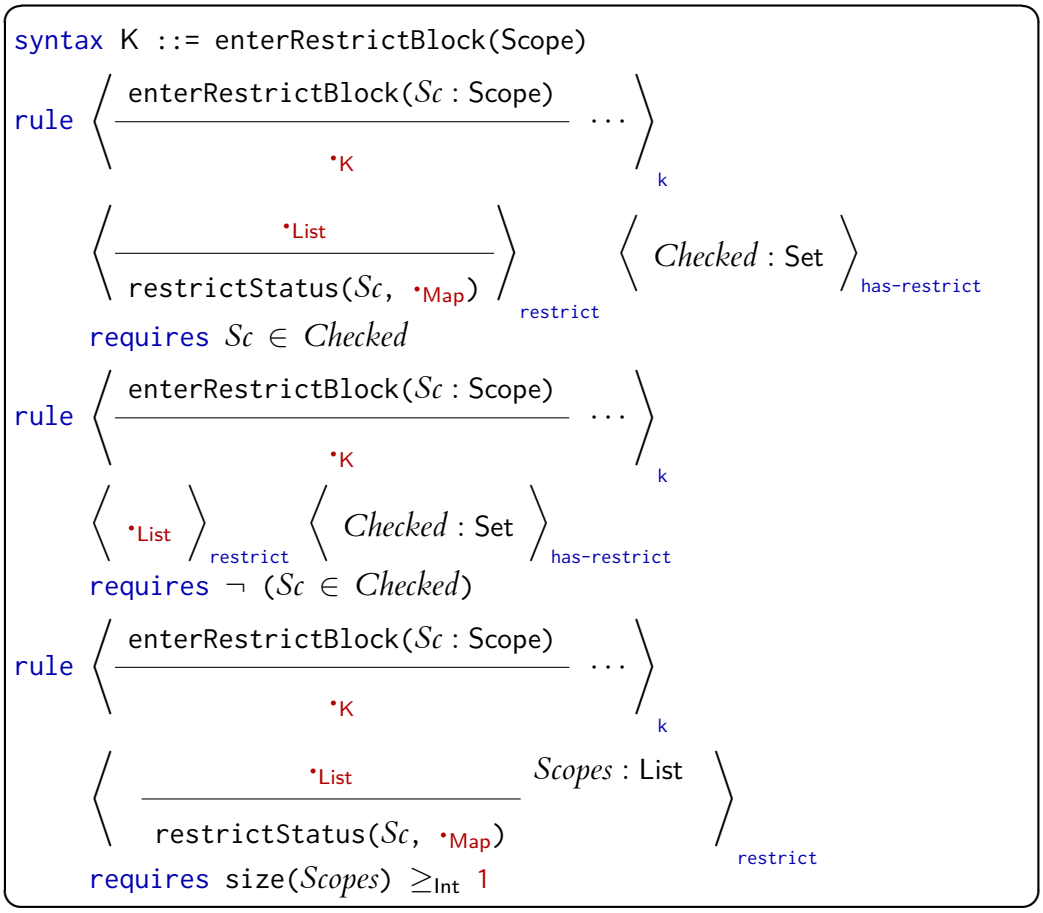
Here we present the actual lightly-cleaned semantics for the `restrict` check in terms of this *join* operation. The `checkRestrict` production is triggered for every access, where its first parameter is `true` in the case of writes, `false` for reads, and the second parameter is the location being accessed. We check if `restrict`-checking is currently enabled (i.e., the `<restrict>` cell is non-empty) before rewriting to the `updateRestrict` production:



The semantics for the rules associated with the `updateRestrict` production are also given below. We *join* the RestrictState associated with a location with the RestrictState associated with this access and raise a violation if the result is $\top$:



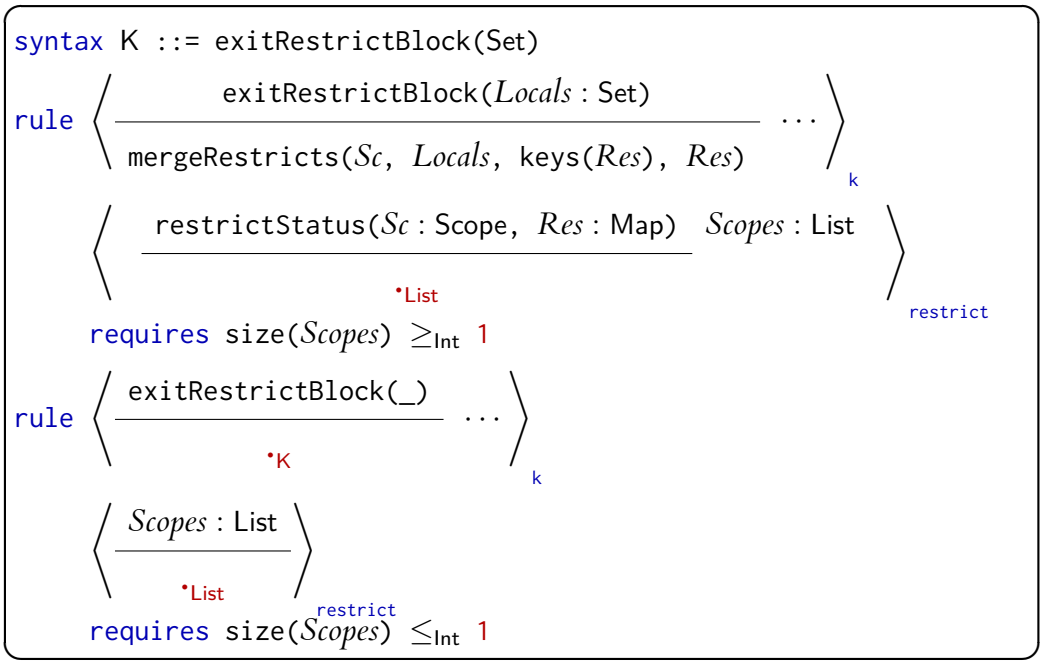
Next, we give the rules that are triggered whenever a new scope (or block) is entered or exited. Here we maintain the `<restrict>` stack and catch violations between accesses in the current scope and the most recent previously-entered scope that have been deferred:



During the translation phase of the semantics, we track which scopes actually contain `restrict`-qualified pointers and record this in the `<has-restrict>` cell. This allows us to disable checking for violations while no `restrict`-qualified pointers are (dynamically) in scope.

When entering a new scope, we push an empty map on the top of the `<restrict>` stack, so only conflicts between accesses within the new scope will be detected at the point of the access. Detection of other conflicts is deferred until block exit, when the top map is merged with the map for the previous scope

and accesses are checked for consistency. The *Locals* parameter of the `exitRestrictBlock` production is the set of local addresses going out of scope:



The merge and filter productions handle merging the RestrictStatus of the scope being exited with the next active scope. In the first phase, we drop or filter out any location or `restrict`-qualified pointer going out of scope:

```
syntax K ::= mergeRestricts(Scope, Set, List, Map)
rule mergeRestricts(Sc : Scope, Locals : Set,
          L : SymLoc Locs : List, Res : Map)
     ⇒ mergeRestrict(Sc, L, filterBases(Res[L], Sc))
       ↷ mergeRestricts(Sc, Locals, Locs, Res)
     requires ¬ (base(L) ∈ Locals)
rule mergeRestricts(Sc : Scope, Locals : Set,
          L : SymLoc Locs : List, Res : Map)
     ⇒ mergeRestricts(Sc, Locals, Locs, Res)
     requires base(L) ∈ Locals
rule mergeRestricts(_, _, •List, _) ⇒ •K
```

```
syntax K ::= mergeRestrict(Scope, SymLoc, RestrictState)
rule ⟨ mergeRestrict(_, L : SymLoc, St : RestrictState) / •K ... ⟩k
     ⟨ restrictStatus(_, Res : Map ) / Res[L ← St] ... ⟩restrict
     requires ¬ (L ∈ keys(Res))
rule ⟨ mergeRestrict(_, L : SymLoc, St : RestrictState) / •K ... ⟩k
     ⟨ restrictStatus(_, _ L ↦ St' : RestrictState ) / join(St, St') ... ⟩restrict
```

Finally, we filter out `restrict`-qualified pointers based on the block being exited:

```
syntax RestrictState ::= filterBases(RestrictState, Scope) [function]
rule filterBases(R_S : Set, Sc : Scope) => R_filterBases'(S, Sc)
rule filterBases(W_S : Set, Sc : Scope) => W_filterBases'(S, Sc)
rule filterBases(St : RestrictState, _) => St [owise]

syntax Set ::= filterBases'(Set, Scope) [function]
rule filterBases'(basedOn(_, Sc) S : Set, Sc : Scope)
     => filterBases'(S, Sc)
rule filterBases'(basedOn(B : SymBase, Sc' : Scope) S : Set, Sc : Scope)
     => basedOn(B, Sc') filterBases'(S, Sc)
     requires Sc ≠K Sc'
rule filterBases'(•Set, _) => •Set
```

3.4 Pointer provenance

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

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

Even though the standard requires that arrays be contiguously allocated, and therefore the bit patterns representing the p and q pointers should compare equal on line 4, it does not follow that the assignments on lines 5 and 6 are both defined. In fact, the assignment on line 5 is well-defined, but the assignment on line 6 invokes undefined behavior according to the following criteria:

> Addition [...] of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated. [24, UB #47]

The use of the indirection operator (&) on line 3 creates a pointer to the object identified by x, which is treated the same as a pointer to a single-element array for the purposes of pointer arithmetic. Therefore, the expression `&x + 1` is well-defined, because it results in a pointer to the element one past the end of an array, which is allowed by the standard. But to actually dereference this pointer, as is done on line 7, is undefined behavior, regardless of the fact that dereferencing q on line 6 is perfectly defined and p and q appear to be equal. This same reasoning holds even if x and y, in the above example, represented adjacent elements in a struct or even an array.

The moral here is that the provenance[3] of pointers can be significant when determining whether operations performed on them are defined. When checking if a dereference is within bounds, it matters whether a pointer value was based on an array or whether it came from the & operator applied to a scalar.

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

Checking the type or value at the memory location before a dereference is not enough.

Therefore, we use an approach similar to the fat pointers of Jim et al. [26] and Necula, McPeak, and Weimer [37] and make a distinction between the value that might be stored in a pointer variable and the symbolic addresses that actually pick out objects in our memory model. The former might have additional qualifiers, or tags, that carry extra information about the provenance of the address. We generalize this technique to catch many different kinds of undefinedness in our semantics. Pointer values might carry any of the following four tags, which are not part of the object representation of pointers (from the perspective of a C program), but can still follow pointer values through, e.g., function calls and memory stores. The details of when each tag will be attached to a pointer and when it might be removed depends on the tag, but often the standard is not clear about just how long and through what kinds of expressions the provenance of a pointer should remain significant.

`fromArray` As we pointed out above, we must track the size of an array that a pointer is based on and its current offset into the array in order to catch violations dealing with undefined pointer arithmetic and out-of-bounds pointer dereferences [24, UB #46–49].

`basedOn` We also must track when a pointer can be traced back to the value stored in some `restrict`-qualified pointer variable. This allows us to associate objects in memory with the `restrict`-qualified pointers used to access them

and check for undefined assignments involving `restrict`-qualified pointers [24, UB #68, 69].

`align` We track a pointer's alignment using this same mechanism. We use this for catching the undefinedness that results from a misaligned pointer after a conversion [24, UB #25].

`objectType` Tracks the last access type for effective type checking. When a write is made through an lvalue, its associated `objectType` becomes the new effective type of the object in memory. We also use this to track accesses to `struct`s and `union`s, which each have parts (padding in the case of a `struct` and the non-active variant in the case of a `union`) that become indeterminate during a write. Regarding `unions`: "when a value is stored in a member of an object of `union` type, the bytes of the object representation that do not correspond to other members take unspecified values" [24, §6.2.6.1:7]. From this type (which also encodes the active union variant) associated with an lvalue expression, we can mark the section of memory not overlapping with the active variant as unspecified. This allows some type punning[4] while still catching violations that can result from attempting to use the unspecified values in the non-overlapped part of the union.

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

3.5 Undefined behavior in the standard library

As Figure 1.1 illustrates, the largest source of undefined behavior in C11 is the standard library at 101 occurrences. Figure 3.10 further breaks down the sources (and our coverage) of undefined behavior in the standard library. In the following sections, we give a brief overview of our treatment of a few of the biggest sources of undefined behavior in the standard library: `stdio.h`, `stdlib.h`, `stdarg.h`, `signal.h`, and `setjmp.h`. We also give semantics to and catch undefined behavior in many other parts of the standard library (e.g., `math.h`, `string.h`, `threads.h`, `locale.h`, and `time.h`)

`stdio.h` Includes our semantics for `printf`, `scanf`, and functions for performing file input and output such as `fread` and `fwrite`. Most of our semantics for `stdio.h` consists of the semantics of the formatted input and output functions (`printf`, `scanf`, and related variations). Undefined behaviors related to these functions include invalid conversion flags and arguments not matching the type expected by a conversion specifier.

The standard also dictates the way in which input and output may be interleaved on a file stream: a call to `fflush` or a file positioning function like `fseek` must be made before receiving input on a stream that was last used for output and the reverse.

`stdlib.h` Includes the functions `malloc`, `free`, and `atexit`. The memory allocation functions (`malloc`, `aligned_alloc`, and `calloc`) are the only method for

§	Description	Header	No.	Cov.	CERT UB ids.
7.1	General		6	6	100, 103, 105, 106, 108, 109.
7.2	Diagnostics	`assert.h`	2	0	110, 111.
7.3	Complex arithmetic	`complex.h`	1	0	112.
7.4	Character handling	`ctype.h`	1	0	113.
7.5	Errors	`errno.h`	1	0	114.
7.6	Floating-point env.	`fenv.h`	4	0	115–118.
7.8	Format conv.	`inttypes.h`	1	0	119.
7.11	Localization	`locale.h`	2	2	120, 121.
7.12	Mathematics	`math.h`	2	0	122, 123.
7.13	Nonlocal jumps	`setjmp.h`	4	1	124–127.
7.14	Signal handling	`signal.h`	8	5	128–135.
7.16	Variable arguments	`stdarg.h`	8	6	136–143.
7.19	Common definitions	`stddef.h`	1	1	144.
7.20	Integer types	`stdint.h`	1	0	145.
7.21	Input/output	`stdio.h`	30	19	146–175.
7.22	General utilities	`stdlib.h`	15	8	176–190.
7.24	String handling	`string.h`	4	1	191–194.
7.25	Type-generic math	`tgmath.h`	2	0	195, 196.
7.26	Threads	`threads.h`	1	0	5.
7.27	Date and time	`time.h`	1	0	197.
7.29	Wide char. utilities	`wchar.h`	3	0	198–200.
7.30	Wide char. class.	`wctype.h`	3	0	201–203.
		Total:	101	∼49	

Figure 3.10. Breakdown of undefined behavior in the C11 standard library.

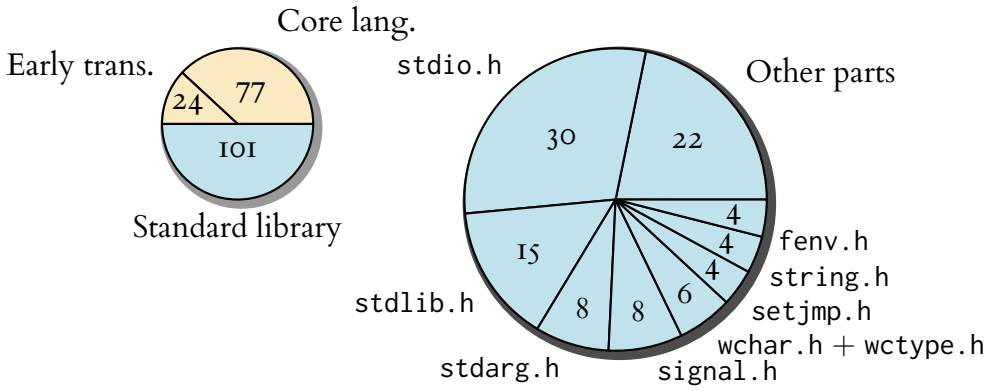


Figure 3.11. Share of undefined behaviors (out of 202) in the standard library and in each part of the standard library based on the list in Annex J [24]. Nearly 80% of the UBs from the standard library involve the functions from 9 of the library headers.

creating memory objects with an allocated duration, which exist for the entire execution of the program. Allocated duration objects are also unique in that they have no declared type, and therefore initially have no effective type. The effective type becomes the type of the lvalue through which the first write to the allocated duration region occurs. See Section 3.2 for a discussion of effective types in allocated duration regions.

`stdarg.h` Includes functions to support writing functions that accept a variable number of arguments. For example, `va_start` for initializing a `va_list` object and `va_arg` for moving to the next argument.

`signal.h` Includes functions for registering and invoking signal handlers. This behavior necessarily requires interaction with the environment a program executes in, which we do not directly model in our semantics.

`setjmp.h` The `setjmp` and `longjmp` allow a program to save and restore the current control flow state (similar to `call/cc` in Scheme). This functionality is subject to many undefined-behavior-inducing caveats, however. The function context at the time of the `setjmp` call must still be active when being restored by `longjmp`, for example. Also undefined is using `longjmp` to exit a scope where a variable-length array was declared.

Chapter 4

Interpreting programs: the kcc tool

We extract a tool from our semantics, which we call kcc. Unlike modern optimizing compilers, which have a goal to produce binaries that are as small and as fast as possible at the expense of compiling programs that may be semantically incorrect, kcc instead aims at mathematically rigorous dynamic checking of programs for strict conformance with the C11 standard. A strictly-conforming program is one that does not rely on implementation-specific behaviors and is free of all undefined behavior (see Section 5.2).

Users interface with our tool through the kcc executable, which behaves as a drop-in replacement for compilers like gcc and clang. Often, using kcc to analyze a project already using gcc (and GNU Make) is as easy as executing

```
$ CC=kcc make
```

on the command line and then running the unit tests. See Figure 4.0 for the output of kcc --help to better understand the tool's interface.

```
Usage: kcc [options]  <files>...
       kcc -help
       kcc -version
Options:
  -c                    Compile and assemble, but do not link.
  -shared               Compile and assemble into a single object file.
  -d                    Print debugging information.
  -D <name>[=[<definition>]]    Predefine <name> as a macro, with definition
                                <definition>.
  -U <name>             Undefine <name> as a macro.
  -P                    Inhibit preprocessor line numbers.
  -E                    Preprocess only.
  -I <dir>              Look for headers in <dir>.
  -iquote <dir>         Look for headers with quotation marks in <dir>.
  -isystem <dir>        Look for system headers in <dir>.
  -include <file>       Add header to file during preprocessing.
  -L <dir>              Look for shared libraries in <dir>.
  -nodefaultlibs        Do not link against the standard library.
  -std=<std>            Standard to use when building internally with
                        GCC for inline assembly. Not used by kcc directly.
  -o <file>             Place the output into <file>.
  -l <lib>              Link semantics against library in search path.
  -Wl,<args>            Add flags to linker arguments.
  -M,-MM,-MD,-MMD,-MP   Dependency generation. See documentation for GCC
                        preprocessor.
  -MF <file>            Dependency generation. See documentation for GCC
                        preprocessor.
  -MT <target>          Dependency generation. See documentation for GCC
                        preprocessor.
  -d<chars>             Debugging info from preprocessor. See documentation
                        for GCC.
  -fmessage-length=0    Write all error messages on a single line.
  -fissue-report=<file> Write issues to the specified file.
                  Format (CSV/JSON) is inferred from the specified file extension.
  -fworkspace=<dir>     Use <dir> for the workspace of kcc when reporting
                        errors.
  -Wlint                Generate lint errors for potentially undesirable
                        behaviors.*
  -flint                Generate lint errors for potentially undesirable
                        behaviors.*
  -Wno-undefined        Do not output undefined behaviors.*
  -Wno-unspecified      Do not output unspecified behaviors.*
  -Wno-implementation-defined   Do not output implementation-defined behaviors.*
  -Wno-<errcode>        Ignore specified error code.*
```

```
 -Wsystem-headers      Do not ignore errors in system headers.*
 -Wfatal-errors        Do not recover from errors reported by tool.*
 -W<errcode>           Do not ignore specified error code.*
 -Wno-ifdef=<macro>    Disable errors on lines emitted when <macro> is
                       defined.*
 -Wno-ifndef=<macro>   Disable errors on lines emitted when <macro>
is not
                       defined.*
 -Wno-file=<glob>      Disable errors in files matching <glob>.*
 -no-pedantic          Do not trigger preprocessor warnings for non-standard
                       compliant language features.
 -w                    Ignore all preprocessor warnings.
 -fheap-size=<size>    Used with -flint to detect dynamic memory overflow.*
 -fstack-size=<size>   Used with -flint to detect stack overflow.*
 -frecover-all-errors  Recover from fatal errors that would normally cause an
                       application to crash.
                       WARNING: This can change the semantics of tools like
                       autoconf which analyze the exit code of the compiler to
                       trigger unexpected or undesirable results.
 -fno-native-compilation       Disables compilation of code with native
                               compiler in order to improve error reporting in
                               programs which fail to compile under the native
                               compiler.
                               WARNING: Due to technical limitations this also
                               disables support for inline assembly.
 -x <language>         Set language for input files.
 -pthread              Enables pthread library (experimental)
 -profile <name>       Set KCC profile.
 --no-license-message  Do not print any licensing information. Use this
                       option if extra output interferes with a build system.
                       Setting the environment error KCC_NO_LICENCE_MESSAGE
                       has the same effect.
 * Indicates flags that require RV-Match from
   http://www.runtimeverification.com/match to run.
 For a complete list of other flags ignored by kcc, refer to
 ~/rv-match/c-semantics/dist/ignored-flags,
 which contains one regular expression per line.
 The following lines of output are added to this message for compatibility
 with GNU ld and libtool:
 : supported targets: elf
```

Figure 4.0. The kcc tool command-line interface (printed by kcc --help).

Consider a file `undef.c` with contents:

```
1 int main(void) {
2       int a;
3       &a + 2;
4 }
```

We compile the program with `kcc` just as we would with `gcc` or `clang`. This produces an executable named `a.out` by default, which should behave just as an executable produced by another compiler—for strictly-conforming, valid programs. For undefined or invalid programs, however, `kcc` reports errors and exits if it cannot recover. In addition to printing location information and a stack trace, `kcc` will also cite relevant sections of the standard:

```
$ kcc undef.c
$ ./a.out
 A pointer (or array subscript) outside the bounds of an object.
   at main(undef.c:3:2)
  Undefined behavior (UB-CEA1).
   see C11 section 6.5.6:8 http://rvdoc.org/C11/6.5.6
   see C11 section J.2:1 item 46 http://rvdoc.org/C11/J.2
   see CERT-C section ARR30-C http://rvdoc.org/CERT-C/ARR30-C
   see CERT-C section ARR37-C http://rvdoc.org/CERT-C/ARR37-C
   see CERT-C section STR31-C http://rvdoc.org/CERT-C/STR31-C
   see MISRA-C section 8.18:1 http://rvdoc.org/MISRA-C/8.18
   see MISRA-C section 8.1:3 http://rvdoc.org/MISRA-C/8.1
```

Unlike similar tools, we do not instrument an executable produced by a separate compiler. Instead, kcc directly interprets programs according to our formal operational semantics. See Figure 3.1 for an overview of the architecture. The semantics gives a separate treatment to the three main phases of a C implementation: compilation, linking, and execution. The first two phases together form the translation semantics, which we extract into an OCaml program to be executed by the kcc tool. The kcc tool, then, translates C programs according to the semantics, producing an abstract syntax tree (AST) as the result of the compilation and linking phases. This AST then becomes the input to another OCaml program extracted from the execution semantics.

The tool on which we have based our work was originally born as a method for testing the correctness of the operational semantics from which it was extracted [13], but the performance and scalability limitations of this original version did not make it a practical option for analysis of real programs. To this end, we have improved the tool on several fronts, most notably by implementing an OCaml-based execution engine and the ability to fall back on natively-compiled versions of functions for which we either do not have access to the source code (e.g., pre-compiled libraries) or do not support a language feature or extension used by the function (e.g., inline assembly). These improvements have allowed kcc to build and analyze programs in excess of 300k lines of code, including the BIND DNS server.

4.1 Related work and evaluation

As we mention in the introduction, many previous semantic efforts have tended to focus on the semantics of defined programs. The recent work of Krebbers [28, 27, 29] is a notable exception, however. His semantics, formalized in Coq, captures many of the trickiest sources of undefinedness, such as expression nondeterminism and the alias restrictions [24, §6.5]. He also extracts an interpreter from his semantics, but his coverage of UB is relatively narrow compared to our semantics.

Another promising recent work on detecting undefinedness comes from Wang et al. [47]. They characterize undefinedness as code unstable under optimization and instead of attempting to catch the behaviors themselves, they catch code that would be affected by optimizations that aggressively take advantage of undefinedness.

The closest comparison for our kcc tool in terms of coverage and approach might be the CompCert interpreter, which interprets programs according to the semantics of the CompCert compiler and reports on any UB it encounters. Of course, many other tools exist for analyzing C programs. Some of our techniques for capturing undefinedness have precedent in the literature on such analyzers. But many of these tools, such as Valgrind, tend to have a narrow focus when it comes to detecting UB, which we think our results demonstrate. Below, we compare kcc with some popular C analyzers on a benchmark from Toyota ITC. The other tools we consider:

- *GrammaTech CodeSonar* is a static analysis tool for identifying "bugs that can result in system crashes, unexpected behavior, and security breaches" [15].
- *MathWorks Polyspace Bug Finder* is a static analyzer for identifying "run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software" [32].
- *MathWorks Polyspace Code Prover* is a tool based on abstract interpretation that "proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code" [33].
- *Clang UBSan, TSan, MSan, and ASan (version 3.7.1)* are all `clang` modules for instrumenting compiled binaries with various mechanisms for detecting undefined behavior, data races, uninitialized reads, and various memory issues, respectively [9].
- *Valgrind Memcheck and Helgrind (version 3.10.1, GCC version 4.8.4)* are tools for instrumenting binaries for the detection of several memory and thread-related issues (illegal reads/writes, use of uninitialized or unaddressable values, deadlocks, data races, etc.) [38].
- *The CompCert C interpreter (version 2.6)* uses an approach similar to our own. It executes programs according to the semantics used by the CompCert compiler [7] and reports undefined behavior.

- *Frama-C Value Analysis (version sodium-20150201)*, like Code Prover, is a tool based on static analysis and abstract interpretation for catching several forms of undefinedness [8].

The Toyota ITC benchmark

The publicly-available[1] Toyota ITC benchmark [46] consists of 1,276 tests, half with planted defects meant to evaluate the defect rate capability of analysis tools and the other half without defects meant to evaluate the false positive rate. The tests are grouped in nine categories: static memory, dynamic memory, stack-related, numerical, resource management, pointer-related, concurrency, inappropriate code, and miscellaneous.

We evaluated kcc along with the tools mentioned above on this benchmark. Our results appear in Figure 4.1 and the tools we used for our evaluation are available online.[2] Following the method of Shiraishi, Mohan, and Marimuthu [46], we report the value of three metrics: DR is the detection rate, the percentage of tests containing errors where the error was detected; $\underline{\mathsf{FPR}} = 100 - \mathsf{FPR}$, where FPR is the false positive rate; and PM is a productivity metric, where $\mathsf{PM} = \sqrt{\mathsf{DR} \times \underline{\mathsf{FPR}}}$, the geometric mean of DR and $\underline{\mathsf{FPR}}$.

Interestingly, the use of kcc on the Toyota ITC benchmark detected a number of flaws in the benchmark itself, both in the form of undefined behavior

[1] https://github.com/Toyota-ITC-SSD/Software-Analysis-Benchmark

[2] https://github.com/runtimeverification/evaluation/tree/master/toyota-itc-benchmark

that was not intended, and in the form of tests that were intended to contain a defect but were actually correct. Our fixes for these issues were accepted by the Toyota ITC authors and we used the fixed version of the benchmark in our experiments. Unfortunately, we do not have access to the MathWorks and GrammaTech static analysis tools, so in Figure 4.1 we have reproduced the results reported in Shiraishi, Mohan, and Marimuthu [46]. Thus, it is possible that the metrics scored for the other tools may be off by some amount.

4.2 Evaluation

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

There is an ISO technical specification for program analyzers titled "C Secure Coding Rules" [2013], suggesting programmatically enforceable rules for writing secure C code. It is similar to MISRA-C [35], whose goal was to cre-

Tool		Static memory	Dynamic memory	Stack-related	Numerical	Resource management	Pointer-related	Concurrency	Inappropriate code	Misc.	Avg. (unweighted)	Avg. (weighted)
kcc	DR	100	94	100	96	93	98	67	*0*	63	79	82
	FPR	100	100	100	100	100	100	100	–	100	100	100
	PM	100	97	100	98	96	99	82	*0*	79	89	91
GrammaTech CodeSonar	DR	100	89	*0*	48	61	52	70	46	69	59	68
	FPR	100	100	–	100	100	96	77	99	100	97	98
	PM	100	94	*0*	69	78	71	73	67	83	76	82
MathWorks Bug Finder	DR	97	90	15	41	55	69	*0*	28	69	52	62
	FPR	100	100	85	100	100	100	–	94	100	98	99
	PM	98	95	36	64	74	83	*0*	51	83	71	78
MathWorks Code Prover	DR	97	92	60	55	20	69	*0*	1	83	53	53
	FPR	100	95	70	99	90	93	–	97	100	94	95
	PM	98	93	65	74	42	80	*0*	10	91	71	71
UBSan + TSan + MSan + ASan (clang)	DR	79	16	95	59	47	58	67	*0*	37	51	47
	FPR	100	95	75	100	96	97	72	–	100	93	95
	PM	89	39	84	77	67	75	70	*0*	61	69	67
Valgrind + Helgrind (gcc)	DR	9	80	70	22	57	60	72	2	29	44	42
	FPR	100	95	80	100	100	100	79	100	100	95	97
	PM	30	87	75	47	76	77	76	13	53	65	65
CompCert interpreter	DR	97	29	35	48	32	87	58	17	63	52	51
	FPR	82	80	70	79	83	73	42	83	71	74	76
	PM	89	48	49	62	52	80	49	38	67	62	63
Frama-C Value Analysis	DR	82	79	45	79	63	81	7	33	83	61	66
	FPR	96	27	65	47	46	40	100	63	49	59	55
	PM	89	46	54	61	54	57	26	45	63	60	60

Figure 4.1. Comparison of tools on the 1,276 tests of the ITC benchmark. The numbers for the GrammaTech and MathWorks tools come from Shiraishi, Mohan, and Marimuthu [46].

- Blue indicates the best score in a category for a particular metric; orange emphasizes the weighted average of the productivity metric for each tool.
- DR, FPR, and PM are, respectively, the detection rate, 100 − FPR (the complement of the false positive rate), and the productivity metric.
- The final average is weighted by the number of tests in each category.
- Italics and a dash indicate categories for which a tool has no support.

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

[a]Valgrind Memcheck, v. 3.5.0, `http://valgrind.org`
[b]Frama-C Value Analysis plugin, v. Nitrogen-dev, `http://frama-c.com/value.html`

Figure 4.2. Comparison of analyzers against the Juliet Test Suite.

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

[a]The Static Analyzer Astrée, v. 14.10, `http://www.absint.com`
[b]CompCert C interpreter, v. 2.3pl2, `http://compcert.inria.fr`
[c]Valgrind Memcheck, v. 3.10.0, `http://valgrind.org`
[d]Frama-C Value Analysis, v. Neon, `http://frama-c.com/value.html`
[e]Version of the tool from Ellison and Roșu [13].

Figure 4.3. Comparison of analyzers against our test suite comprising 261 tests, covering 77 core language undefined behaviors. Note that some tools don't support all language features covered in the tests.

ate a "restricted subset" of C to help those using C meet safety requirements. MISRA released a "C Exemplar Suite," containing both conforming and non-conforming code for the majority of the MISRA C rules. However, these tests contain many undefined behaviors mixed into a single file, and no way to run the comparable defined code without running the undefined code. Furthermore, the MISRA tests focus on statically detectable UB. The CERT C Secure Coding Standard [45] and MITRE's "common weakness enumeration" (CWE) classification system [36] are other similar projects, identifying many causes of program error and cataloging their severity and other properties. The projects mentioned above include many undefined behaviors—for example, the undefinedness of signed overflow [24, UB #36] corresponds to CERT's INT32-C and to MITRE CWE-190.

The Juliet Test Suite NIST has released a suite of tests called the Juliet Test Suite for C/C++ [39], which is based on MITRE's CWE classification system. It contains over 45,000 tests, each triggering one of the 116 different CWEs supported by the suite. Most of the tests (∼70%) are C and not C++ and they focus on statically detectable behaviors. But not all of the CWEs are actually undefined—many are simply insecure or unsafe programming practices.

Because the Juliet tests include a single undefined behavior per file and come with positive tests corresponding to the negative tests, we decided to extract an undefinedness benchmark from them. To use the Juliet tests as a test suite for undefinedness, we had to identify which tests were actually undefined. This was

largely a manual process that involved understanding the meaning of each CWE. It was necessary due to the large number of defined-but-bad-practice tests that the suite contains. Interestingly, the suite contained some tests whose supposedly defined portions were actually undefined. Using our analysis tool, we were able to identify six distinct problems with these tests, which we submitted to NIST.

This extraction gave us 4113 tests, with about 96 lines per test. The tests can be divided into six classes of undefined behavior: use of an invalid pointer (buffer overflow, returning stack address, etc.), division by zero, bad argument to `free()` (stack pointer, pointer not at start of allocated space, etc.), uninitialized memory, bad function call (incorrect number or type of arguments), or integer overflow. We then ran these tests using Valgrind Memcheck [38], and the Value Analysis plugin for Frama-C [8], in addition to our tool, `kcc`. The results appear in Figure 4.2.

Our Undefinedness Test Suite Because we were unable to find an ideal test suite for evaluating detection of undefined behaviors, we began development of our own. As we discussed in Section 5, undefined behavior reached during an execution causes the entire execution to become undefined. This means each test in the suite must be a separate program, otherwise one undefined behavior may interact with another. In addition, each test should come with a corresponding defined test as a control, making it possible to identify false-positives in addition to false-negatives. The suite we have used here for this evaluation, a subset of our unit and regression tests, includes 261 tests. These tests are much broader

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

Figure 4.4. Core language undefined behaviors each tool detects (total out of 77). We counted a tool as having coverage for a behavior if it caught the behavior in at least one test.

than the Juliet tests, covering all 77 categories of core language undefined behaviors (identified in Figure 1.1) as opposed to the 12 covered by the Juliet tests. We hope it will serve as a starting point for the development of a larger, more comprehensive undefinedness test suite.

We compared the same tools as before against our own test suite. This time, we also included the CompCert interpreter [7] and the Astrée analyzer [10]. The former interprets programs according to the semantics of the formally-verified CompCert compiler. Like our tool, it attempts to detect undefinedness and halts when it encounters an undefined behavior. The latter, Astrée, is an abstract-interpretation based analyzer reported to detect most undefined behavior (albeit in C99, not C11). As can be seen in the tables, our tool passes most of our test cases. The two failures are due to minor bugs still outstanding at the time of this writing.

From these results (Figure 4.3), and based on the premise that our test suite includes tests for each behavior that are trivial enough that no tool looking for that behavior could miss it, we estimate the number of core language undefined behaviors each tool detects in Figure 4.4. These tools do not have complete coverage for many behaviors—the CompCert interpreter, for example, does not support programs comprising multiple translation units, so it necessarily will not have complete coverage for any of the behaviors we have categorized "link time."

From Figure 4.4 we can see that none of the other tools we tested appeared to have coverage for 19 core language undefined behaviors.[3] Many of these deal with newer features of the C language, such as the `restrict` qualifier (#68, 69), alignment specifiers (#73), variable-length arrays (#75), and the `inline` function specifier (#70). Some are tricky to monitor for, such as unsequenced side-effects (#35) and misuses of `restrict` and `const` (discussed in Section 3).

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

Chapter 5

Trace-based characterization of undefined behavior

5.1 Introduction

Undefinedness exists in the relationship between the specification and an implementation, so it is important to make a distinction between the semantics of the language specification (e.g., the C11 standard) and the semantics of a particular implementation when trying to understand it. In a sense, the semantics of an implementation should be a refinement of the semantics of the language specification: wherever the specification gives a certain meaning to a program, the implementation must also, but where the specification says nothing, the implementation is left unconstrained. However, the idea that undefined behavior might be allowed to propagate backwards through a trace complicates this

understanding. Retroactive or backward-propagation of undefined behavior is conventionally allowed to enable optimizations: a C implementation does not need to prove the absence of undefined behavior before reordering statements in a program. In other words, undefined behavior itself is not treated as an observable side-effect that must be preserved by implementations. But to what extent does this hold? Are implementations not required to preserve any part of a trace that eventually ends in undefined behavior?

The conventional understanding seems to be that it is "executions" or "traces" that become undefined, and not typically whole programs. As an example:

```
1 int main(int argc, char **argv) {
2       if (argc > 1) {
3             printf("a");
4             *NULL;
5       } else {
6             printf("b");
7       }
8 }
```

This program would seem to be perfectly defined (and the standard requires that it be) as long as `argc` $\leq$ 1, but the compiler should be totally justified in eliding the true branch altogether and producing code that outputs `b` unconditionally.

Below, we sketch our understanding of what exactly it is that the C11 standard requires of implementations in terms of a relation between the operational semantics of the standard and the operational semantics of an implementation.

First, we describe this idea from a high level by considering the relationship between an operational semantics of each in terms of labeled transition systems and traces in the style of Bakker and Vink [4]. Next, we introduce ELSE, a language due to Papaspyrou and Maćoš [42] of C-like expressions, and extend it with a treatment of undefined behavior in a monadic denotational style semantics implemented in Haskell. Finally, we give an operational and denotational semantics to a IMP-like language extended with undefined behavior and apply our approach to describe the kind of loop optimizations that are allowed under different interpretations of the implications of undefined behavior.

Ultimately, our goal is to gain a more precise understanding of what the C11 standard actually requires of *strictly conforming programs* and *conforming implementations* where undefined behavior is concerned. The former is defined by C11 in chapter 4:

> A *strictly conforming program* shall use only those features of the language and library specified in this International Standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit. [24, §4:5]

And a *conforming implementation* is one which accepts all strictly conforming programs (but might also accept other programs) [24, §4:6] and produces observable behavior according to the rules of the abstract machine defined by the standard [24, §5.1.2.3:6].

5.2 Undefined behavior in C11

First, we give an account of how an operational semantics of the C11 standard can be understood (in contrast to the operational semantics of a particular implementation of the standard). The C11 standard actually describes the semantics of a finite set of closely-related C languages. We call these "abstract" semantics. Each abstract semantics represents the abstract machine described by the standard parameterized by implementation-specific details. This set is finite because instances of "implementation-defined" behavior should always give a finite number of choices to implementations (e.g. the size of `int`). The abstract semantics, then, can be understood as defining a transition relation in a labeled transition system extended with an extra `undef` state:

Definition 1 (Abstract semantics). *An abstract semantics $\mathcal{A}$ is a finitely-branching, non-deterministic transition system $(Conf, Obs, \Rightarrow)$ where*

$$\cdot \stackrel{\cdot}{\Rightarrow} \cdot \subseteq Conf \times Obs \times (Conf + \{\texttt{undef}\})$$

The definition of the transition relation gives rise[1] *to an interpretation function*

[1] To be more precise, $\mathcal{O}_{\mathcal{A}} \triangleq \mathrm{fix}(\Phi_{\mathcal{A}})$, the unique fixed point of $\Phi_{\mathcal{A}}$, where $\Phi_{\mathcal{A}} : \mathrm{Sem}_{\mathcal{A}} \to \mathrm{Sem}_{\mathcal{A}}$ (with $\mathrm{Sem}_{\mathcal{A}} \triangleq \mathrm{Conf} \to \mathcal{P}(\mathrm{Obs}^*) \times \mathcal{P}_{\geq 1}(\mathrm{Obs}^\infty)$) and

$$\Phi_{\mathcal{A}}(S)(c) \triangleq \{\alpha \mid c \stackrel{\alpha}{\Rightarrow} \texttt{undef}\} \cup \bigcup \{\alpha \cdot S(c') \mid c \stackrel{\alpha}{\Rightarrow} c'\}.$$

See Bakker and Vink [4] for details. $\mathcal{O}_{\mathcal{X}}$ below would also be defined in this way.

$\mathcal{O}_{\mathcal{A}} : \mathit{Conf} \to \mathcal{P}(\mathit{Obs}^*) \times \mathcal{P}_{\geq 1}(\mathit{Obs}^\infty)$ *where*

$$\mathcal{O}_{\mathcal{A}}(c) = \Big(\big\{\alpha_1 \cdots \alpha_n \;\big|\; c \stackrel{\alpha_1}{\Longrightarrow} \cdots \stackrel{\alpha_n}{\Longrightarrow} \mathtt{undef}\big\}, \big\{\alpha_1 \cdots \alpha_n \;\big|\; c \stackrel{\alpha_1}{\Longrightarrow} \cdots \stackrel{\alpha_n}{\Longrightarrow} c_n\big\} \bigcup \big\{\alpha_1\alpha_2 \cdots \;\big|\; c \stackrel{\alpha_1}{\Longrightarrow} c_1 \stackrel{\alpha_2}{\Longrightarrow} \cdots\big\}\Big)$$

and $\mathcal{P}_{\geq 1}(\mathit{Obs}^\infty)$ *stands for the powerdomain of all nonempty subsets of* Obs^∞ *and* $\mathit{Obs}^\infty \triangleq \mathit{Obs}^* \cup \mathit{Obs}^\omega$*, the union of all finite and infinite sequences* $\alpha_1\alpha_2\cdots$*, respectively, such that* $\alpha_1, \alpha_2, \ldots \in \mathit{Obs}$*.*

The extra `undef` state mirrors how our operational semantics of the standard models undefinedness: as an extra "stuck" state from which there is no transition.

We define an implementation semantics just as an abstract semantics, except implementations can never enter the `undef` state. That is, implementations must actually define the behavior the standard calls undefined:

Definition 2 (Implementation)**.** *An implementation* $\mathcal{X}$ *of* C11 *is a finitely-branching, non-deterministic transition system* $(\mathit{Conf}, \mathit{Obs}, \to)$ *where*

$$\cdot \stackrel{\cdot}{\to} \cdot \subseteq \mathit{Conf} \times \mathit{Obs} \times \mathit{Conf}$$

The definition of the transition relation gives rise to an interpretation function $\mathcal{O}_{\mathcal{X}} : \mathit{Conf} \to \mathcal{P}_{\geq 1}(\mathit{Obs}^\infty)$ *where*

$$\mathcal{O}_{\mathcal{X}}(c) = \big\{\alpha_1 \cdots \alpha_n \;\big|\; c \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} c_n\big\} \bigcup \big\{\alpha_1\alpha_2 \cdots \;\big|\; c \xrightarrow{\alpha_1} c_1 \xrightarrow{\alpha_2} \cdots\big\}.$$

Now that we have a formal notion of the semantics of the standard and the semantics of a particular implementation, we can describe what the standard actually requires of implementations when the abstract machine encounters undefined behavior. In particular, our goal is to define what it means for a program to be *strictly conforming* (i.e., not rely on undefined or implementation-specific behavior) and what it means for an implementation to conform to the standard. Using the above definitions, the former is straightforward:

Definition 3 (Strictly conforming)**.** *The strictly conforming programs are given by*

$$\{p \in \mathit{Conf} \mid \mathcal{O}_{\mathcal{A}_1}(p) = \cdots = \mathcal{O}_{\mathcal{A}_n}(p) \subseteq \emptyset \times \mathit{Obs}^\infty\}.$$

A program is strictly conforming if and only if (1) evaluating it according to all abstract semantics results in the same set of traces and (2) it is well-defined on all traces according to all abstract semantics.

However, there is little consensus on what the standard requires of implementations where undefined behavior is concerned, so we will give three interpretations for the definition of a conforming implementation. We can describe what it means for an implementation to conform to the standard in terms of a prune function and an undef-*closure*. The prune function shortens traces terminating in undefined behavior based on our understanding of how much of the trace must be preserved, whereas cl_{undef} prepends the resulting trace-prefix onto Obs^∞ to capture the idea that anything is allowed after encountering undefined behavior.

Definition 4 (Conforming implementation)**.** *An implementation $\mathcal{X}$ is conforming if and only if there exists some abstract semantics $\mathcal{A}$ such that, for all programs p,*

$$\mathcal{O}_{\mathcal{X}}(p) \subseteq (cl_{\mathsf{undef}} \circ \mathsf{prune} \circ \mathcal{O}_{\mathcal{A}})(p).$$

Definition 5 (`undef`-closure)**.** *Define $cl_{\mathsf{undef}} : \mathcal{P}(Obs^*) \times \mathcal{P}_{\geq 1}(Obs^\infty) \to \mathcal{P}_{\geq 1}(Obs^\infty)$ as*

$$cl_{\mathsf{undef}}(U, S) \triangleq S \cup \bigcup_{u \in U} u \cdot Obs^\infty$$

The definition of `prune` captures an interpretation of how much of a trace that eventually encounters undefined behavior should be retained according to the standard. For example, consider the somewhat canonical case of retroactive undefinedness involving loop optimizations in Figure 5.1. An optimizing compiler will notice `5 / d` is invariant and possibly choose to move it to before the loop, causing an exception when `d` is zero before the `printf` can be evaluated. Despite the undefinedness not actually occurring (in a sense) until the division expression is evaluated, it still can affect the evaluation of preceding statements. We offer three possible definitions for `prune`, the last two of which might allow this sort of optimization:

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

Figure 5.1. Example of a program with undefined behavior that might move backwards through a trace due to optimization.

Definition 6 (prune)**.** $\mathsf{prune} : \mathcal{P}(Obs^*) \times \mathcal{P}_{\geq 1}(Obs^\infty) \to \mathcal{P}(Obs^*) \times \mathcal{P}_{\geq 1}(Obs^\infty)$ *is given by one of*

$$\mathsf{prune}_\bot(U, S) \triangleq (U, S)$$

$$\mathsf{prune}_+(U, S) \triangleq \Big(\big\{u[n] \mid u \in U, n = \max\{k \geq 0 \mid s \in S,\ u[k] = s[k]\}\big\}, S\Big),$$

where $s'[n']$ is the sequence s' truncated to the first n' elements.

$$\mathsf{prune}_\top(U, S) \triangleq \begin{cases} (\emptyset, S) & \text{if } U = \emptyset, \\ (\{\epsilon\}, S) & \text{otherwise.} \end{cases}$$

Lemma 1. *If* $S \not\subseteq \emptyset \times Obs^{\infty}$*, then*

$$(cl_{\mathsf{undef}} \circ \mathsf{prune}_{\top})(S) = Obs^{\infty}.$$

Proof. By the definition of $\mathsf{prune}_{\top}$, the result of $\mathsf{prune}_{\top}(S)$ will be $(\{\epsilon\}, \pi_2(S))$, and $cl_{\mathsf{undef}}(\{\epsilon\}, \pi_2(S)) = \pi_2(S) \cup \epsilon \cdot \mathrm{Obs}^{\infty} = \mathrm{Obs}^{\infty}$. □

That is, if S has any traces at all that end in undefined behavior, then according the interpretation represented by $\mathsf{prune}_{\top}$, a conforming implementation might do anything.

Lemma 2. *If* $S \subseteq \emptyset \times Obs^{\infty}$*, then*

$$(cl_{\mathsf{undef}} \circ \mathsf{prune}_{\top})(S) = (cl_{\mathsf{undef}} \circ \mathsf{prune}_{+})(S) = (cl_{\mathsf{undef}} \circ \mathsf{prune}_{\bot})(S).$$

Proof. If $\pi_1(S) = \emptyset$, then according to the various definitions for prune, $\mathsf{prune}_{\top}(S) = \mathsf{prune}_{+}(S) = \mathsf{prune}_{\bot}(S) = S$. Therefore $(cl_{\mathsf{undef}} \circ \mathsf{prune}_{\top})(S) = (cl_{\mathsf{undef}} \circ \mathsf{prune}_{+})(S) = (cl_{\mathsf{undef}} \circ \mathsf{prune}_{\bot})(S)$. □

If S has no traces that end in undefined behavior, then all definitions for prune given above leave the set of traces unchanged.

Lemma 3. *For all S,*

$$\pi_2(S) \subseteq (cl_{\mathsf{undef}} \circ \mathsf{prune}_{\top})(S)$$
$$\pi_2(S) \subseteq (cl_{\mathsf{undef}} \circ \mathsf{prune}_{+})(S)$$
$$\pi_2(S) \subseteq (cl_{\mathsf{undef}} \circ \mathsf{prune}_{\bot})(S).$$

Proof. This follows from the definitions of prune, which leave the well-defined traces $\pi_2(S)$ unchanged and from the definition of cl_{undef}, where $cl_{\mathsf{undef}}(S) = \pi_2(S) \cup \cdots$. □

That is, the well-defined traces are always included in the result of $cl_{\mathsf{undef}} \circ$ prune for the definitions of prune given above.

Lemma 4. *For all programs p and interpretations $\mathcal{O}_{\mathcal{A}} : Conf \to \mathcal{P}(Obs^*) \times \mathcal{P}_{\geq 1}(Obs^\infty)$,*

$$(cl_{\mathsf{undef}} \circ \mathsf{prune}_{\bot} \circ \mathcal{O}_{\mathcal{A}})(p)$$
$$\subseteq (cl_{\mathsf{undef}} \circ \mathsf{prune}_{+} \circ \mathcal{O}_{\mathcal{A}})(p)$$
$$\subseteq (cl_{\mathsf{undef}} \circ \mathsf{prune}_{\top} \circ \mathcal{O}_{\mathcal{A}})(p).$$

Proof. Let $S = \mathcal{O}_{\mathcal{A}}(p)$. Two cases:

1. $\pi_1(S) = \emptyset$. Follows from Lemma 2.

2. $\pi_1(S) \neq \emptyset$. By the definitions of prune, this reduces to showing that $cl_{\mathsf{undef}}(S) \subseteq (cl_{\mathsf{undef}} \circ \mathsf{prune}_{+})(S)$. If $\pi_1(S) = \{s_1, \cdots, s_n\}$, then by the

definition of prune_+, there exist $s'_1, \cdots, s'_n$ such that $\pi_1(\mathsf{prune}_+(S)) = \{s'_1, \cdots, s'_n\}$ and $s' = s[i]$ for some $i \leq |s|$.

Therefore, $cl_{\mathsf{undef}}(S) \subseteq (cl_{\mathsf{undef}} \circ \mathsf{prune}_+)(S)$ iff $s_1 \cdot \text{Obs}^\infty \cup \cdots \cup s_n \cdot \text{Obs}^\infty \cup \pi_2(S) \subseteq s'_1 \cdot \text{Obs}^\infty \cup \cdots \cup s'_n \cdot \text{Obs}^\infty \cup \pi_2(\mathsf{prune}_+(S))$ iff (by Lemma 3) $s_1 \cdot \text{Obs}^\infty \cup \cdots \cup s_n \cdot \text{Obs}^\infty \subseteq s'_1 \cdot \text{Obs}^\infty \cup \cdots \cup s'_n \cdot \text{Obs}^\infty$. This follows because $s_i \cdot \text{Obs}^\infty \subseteq s'_i \cdot \text{Obs}^\infty$ for each $i \in \{1, \cdots, n\}$.

□

$\mathsf{prune}_\perp$ (prefix-preserving)

In this interpretation, the implementation must preserve the observable behavior on the entire trace that encounters undefinedness, but anything is allowed after the abstract machine encounters undefinedness. This would not allow the above loop optimization because it would require that the complete trace that encounters the undefined behavior on the abstract semantics be preserved in all conforming implementations.

prune_+ (prefix-eliding)

This interpretation attempts to capture the conventional reasoning that leads to the above loop optimization. Under this interpretation, a trace that eventually encounters undefined behavior must be preserved by implementations as long as it shares a prefix with another trace that never encounters undefined behavior.

prune$_\top$ (minimal)

As a sort of degenerate case, one could argue that the C11 standard requires nothing of implementations when any trace at all encounters undefined behavior. This would trivially allow the above optimization (just as it would allow optimizing the whole program to do nothing).

prune$_\bot$ appears to go against the conventional understanding that allows optimizations like the one above, while prune$_\top$, in contrast, would impose no requirements at all on an implementation for programs that might somewhere encounter undefined behavior, effectively rendering the standard inapplicable to most C programs over a few hundred lines (given the prevalence of undefined behavior). The accepted definition for prune appears to be somewhere between these two, and prune$_+$ is an attempt at this sort of compromise.

Example

Consider a program that calls the function `f()` defined above:

```
1 #include <stdio.h>
2 int main() {
3       int x = getchar();
4       printf("%d\n", x);
5       f(x);
6 }
```

- Under $\texttt{prune}_{\bot}$: if `x` is `0`, then conforming implementations must print `0` twice, once for the `printf` in `main` and again for the `printf` in `f`. After printing the second `0`, the program might do anything.
- Under $\texttt{prune}_{+}$: if `x` is `0`, then conforming implementations must print `0` once, after which the program might do anything.
- Under $\texttt{prune}_{\top}$: the program might do anything on conforming implementations.

In the next sections, we apply this idea to operational and monadic-style denotational semantics for two example languages, each capturing different aspects of the full C language. The first covers the unspecified evaluation order of operands and sequence points in expressions and the second covers statements, with a goal of describing allowed loop optimizations under a particular interpretation of undefined behavior.

5.3 The $\textsf{ELSE}_{\textsf{undef}}$ language

The semantics of C expressions is notoriously complicated. The standard leaves the evaluation order for most operands unspecified, with the exception that all side-effects must be applied by the time an implementation encounters a *sequence point* during evaluation. Sequence points are specified as occurring in several places, such as between statements and between the evaluation of a function call

and its arguments. This underspecified evaluation order is usually modeled by non-determinism in an operational semantics.

Papaspyrou and Maćoš [42] give a semantics for ELSE, a simple language of C-like expressions with assignment and sequence points, also modeling the unspecified evaluation order of operands as non-determinism, which we will consider here.

5.3.1 Syntax

$$
\begin{array}{l}
e \in \textbf{Expr} ::= n \mid x \mid x = e \mid e_1 \texttt{ + } e_2 \mid e_1 \texttt{ , } e_2 \mid \langle\ e\ \rangle \\
\qquad\qquad \mid \texttt{\# } e \\
n \in \textbf{Num},\ x \in \textbf{Ide}
\end{array}
$$

These expressions are interpreted in a usual way:

- n and I are integer literals and identifiers;
- $\langle\,\rangle$ evaluates an expression as a single atomic step, allowing no interleaving;
- the operands of $+$ are evaluated in an unspecified order, modeled here as non-determinism with possible interleaving;
- $\#$ inserts a sequence point before and after evaluation of its operand, modeling the effect of a function call in C;
- and the remaining operators have the usual meaning, with the comma evaluating its operands in left-to-right order and assignment updating the store.

5.3.2 Denotational semantics in Haskell

This language is interpreted by Papaspyrou and Maćoš [42] in a resumption monad with non-determinism (presented here in Haskell pidgin, where $\mathcal{E}$ is defined for well-typed expressions of type t):

$$\mathcal{E}[\![- : t]\!] : \mathbf{Expr} \to \texttt{R (StateT S []}_{\geq 1}\texttt{)}\ t$$

Where `StateT` and `R` are monad transformers. The former captures observable behavior as updates to a store mapping variables to values. The latter, a resumption monad, models interleaving in expression evaluation as a sequence of continuations. `R` has the following definition in Haskell:

```
data R m a = Computed a
           | Resume (m (R m a))
instance Monad m ⇒ Monad (R m) where
  return = Computed
  Computed a >>= f = f a
  Resume m   >>= f = Resume (m >>= λ r → return (r >>= f))
```

To capture undefined behavior, our only addition to this scheme is to

1. make a distinction between the semantics of an "abstract" and "concrete" semantics,

2. extend the resumption monad in the domain of abstract semantics with an extra `Undef` constructor, and

3. invoke undefined behavior in the abstract semantics of $\mathsf{ELSE}_{\mathsf{undef}}$.

First, we extend ELSE with undefinedness (and call the result $\mathsf{ELSE}_{\mathsf{undef}}$):

$$
\begin{aligned}
e \;&\in\; \textbf{Expr} \;::=\; n \mid x \mid x = e \mid e_1 \texttt{ + } e_2 \mid e_1 \texttt{ , } e_2 \mid \langle\ e\ \rangle \\
&\qquad\quad \mid \texttt{\# } e \mid e \texttt{ ?} \\
n \;&\in\; \textbf{Num},\; x \;\in\; \textbf{Ide}
\end{aligned}
$$

where e? is interpreted as evaluating to the value of e when e is non-zero, and invoking undefined behavior otherwise.

Now, we make a distinction between the abstract and concrete semantics. In comparison to the interpretation function given by Papaspyrou and Maćoš [42] mentioned above, the domain of the interpretation function for our abstract semantics has an extra `Undef` value, which captures traces that terminate in undefined behavior (in Haskell pidgin):

$$\mathcal{E}_{\mathsf{abs}}[\![- : t]\!] : \textbf{Expr} \to \mathsf{R}_{\mathsf{undef}}\,\texttt{(StateT S []}_{\geq 1}\texttt{)}\ t$$

Where the new definition of the $\mathsf{R}_{\mathsf{undef}}$ resumption includes an extra `Undef` constructor:

```
data R_undef m a = Computed a | Undef
                 | Resume (m (R_undef m a))
instance Monad m ⇒ Monad (R m) where
  return = Computed
  Undef      >>= _ = Undef
  Computed a >>= f = f a
  Resume m   >>= f = Resume (m >>= λ r → return (r >>= f))
```

Below is the definition of $\mathcal{E}_{\mathsf{abs}}$. The semantics remain unchanged for the parts of $\mathsf{ELSE}_{\mathsf{undef}}$ which are the same as ELSE. The `stepR`, `runR`, `+:+`, and `+|+` are all operations on resumption monads defined Papaspyrou and Maćoš [42].

```
ℰabs :: Expr → Rundef (StateT S []≥1) Int
ℰabs⟦ n ⟧ = return n
ℰabs⟦ e1 + e2 ⟧ = do
  (n1, n2) ← ℰabs⟦ e1 ⟧ +:+ ℰabs⟦ e2 ⟧
  return (n1 + n2)
ℰabs⟦ -e ⟧ = do
  n ← ℰabs⟦ e ⟧
  return (-n)
ℰabs⟦ e1 , e2 ⟧ = do
  ℰabs⟦ e1 ⟧
  seqpt
  ℰabs⟦ e2 ⟧
ℰabs⟦ ⟨ e ⟩ ⟧ = stepR (runR ℰabs⟦ e ⟧)
ℰabs⟦ # e ⟧ = do
  seqpt
  n ← ℰabs⟦ e ⟧
  seqpt
  return n
```

We make a minor change in our version by returning undef when unsequenced side effects occur on a variable (i.e., when `load` or `store` returns `Nothing`) instead of crashing:

```
⟦ x ⟧ = do
  s ← get
  case load x s of
    Nothing → Undef
    Just n  → return n
⟦ x = e ⟧ = do
  n ← ⟦ e ⟧
  s ← get
  case store x n s of
    Nothing → Undef
    Just s' → put s' >> return n
```

Finally, we give semantics to our new ? operator, invoking `undef` when the operand evaluates to zero:

```
⟦ e? ⟧ = do
  n ← ⟦ e ⟧
  if n == 0 then Undef else return n
```

A concrete implementation, in contrast, never invokes undefined behavior:

$$\mathcal{E}_{\text{conc}}[\![- : t]\!] : \mathbf{Expr} \to \texttt{R (StateT S []}_{\geq 1}\texttt{)}\ t$$

The definition of $\mathcal{E}_{\text{conc}}[\![-]\!]$ is the same as $\mathcal{E}_{\text{abs}}[\![-]\!]$ except that it will never invoke `undef`.

```
ℰconc :: Expr → R (StateT S []≥1) Int
ℰconc⟦ x ⟧ = do
  s ← get
  case load x s of
    Nothing → return 0
    Just n  → return n
ℰconc⟦ x = e ⟧ = do
  n ← ℰconc⟦ e ⟧
  s ← get
  case store x n s of
    Nothing → return n
    Just s′ → put s′ >> return n
ℰconc⟦ x? ⟧ = ℰconc⟦ x ⟧
```

Note that $\mathcal{E}_{\mathsf{conc}}$ is still non-deterministic in its choice of evaluation order for the operands of the "+" operator, but we might have instead chose to fix the evaluation order in the concrete semantics:

```
ℰ′conc⟦ e1 + e2 ⟧ = do
  n1 ← ℰ′conc⟦ e1 ⟧
  n2 ← ℰ′conc⟦ e2 ⟧
  return (n1 + n2)
```

Does $\mathcal{E}_{\mathsf{conc}}$ conform to the semantics of the specification given by $\mathcal{E}_{\mathsf{abs}}$? To answer this question, we will apply the idea presented in Definition 4, but we

represent this slightly differently with the Haskell, denotational-style semantics given here. $\mathcal{E}_{\mathsf{conc}}[\![-]\!]$ is conforming if and only if, for all programs p,

$$(\mathtt{proj} \circ \mathcal{E}_{\mathsf{conc}})[\![p]\!] \sqsubseteq (\mathtt{prune} \circ \mathtt{proj} \circ \mathcal{E}_{\mathsf{abs}})[\![p]\!]$$

This is the same idea as above, except that instead of $cl_{\mathtt{undef}}$, which would be impractical to implement, we define $\sqsubseteq$ in a way that accepts anything in the left operand when the right is undefined. Also, instead of sets, `prune` and $\sqsubseteq$ are defined in terms of trees, which `proj` extracts from the resumption structure returned by the evaluation function.

The `proj` function itself is defined in terms of `unfold`, `shrink`, and `merge`:

```
proj :: Ord a ⇒ R (StateT S []≥1) a → Tree S a
proj = merge ∘ shrink ∘ unfold s0
```

The `unfold` function translates the resumption structure into an isomorphic `Tree S a` structure, where a trace prefix of observable behavior makes up the interior nodes and the final results of computation make up the leaves:

```
unfold :: S → R (StateT S []≥1) a → Tree S a
unfold s r = case r of
  Resume p → Br s (map (uncurry (flip unfold)) (runStateT p s))
  Computed a → Br s [Lf a]
```

```
data Tree s a = Br s [Tree s a]≥1 | Lf a
```

The `shrink` and `merge` functions simplify and normalize the resulting `Tree`. The former shortens branches in the tree when the observable state is unchanged from parent to child:

```
shrink :: Ord a ⇒ Tree S a → Tree S a
shrink (Lf n)    = Lf n
shrink (Br s bs) = Br s (shrink′ s (Br s bs))
  where
    shrink′ s t = case t of
      Br s′ bs | s == s′ → join (map (shrink′ s′) bs)
      Br s′ bs → return (Br s′ (join (map (shrink s′) bs)))
      Lf n     → return (Lf n)
```

And the `merge` function merges subtrees with a common observable prefix:

```
merge :: Ord a ⇒ Tree S a → Tree S a
merge (Lf n)    = Lf n
merge (Br s bs) = Br s (map merge′ (groupWith lbl (sort bs)))
  where
   merge′ (Br s bs : bs′) = merge (Br s (bs++join (map ch bs′)))
   merge′ (b : _)         = b
   lbl (Br s _) = Left s
   lbl (Lf a)   = Right a
   ch (Br _ bs) = bs
   ch (Lf _)    = []
```

Now we have three possible definitions of the prune function depending on our interpretation of undefined behavior, corresponding to the three possible definitions given in Definition 6. The definitions here are in terms of traces represented as trees instead of sets of traces, however:

```
prune⊥ :: Tree s (Maybe a) → Tree s (Maybe a)
prune⊥ t = t


prune+ :: Tree s (Maybe a) → Tree s (Maybe a)
prune+ t = case t of
  Br s bs | any isUB bs → Br s [Lf Nothing]
  Br s bs               → Br s (map prune+ bs)
  _                     → t
  where
    isUB t = case t of
      Lf a     → maybe True (const False) a
      Br _ [b] → isUB b


prune⊤ :: Tree s (Maybe a) → Tree s (Maybe a)
prune⊤ t = if hasUB t then Lf Nothing else t
  where
    hasUB t = case t of
      Lf a    → maybe True (const False) a
      Br _ bs → any hasUB bs
```

Finally, the $\sqsubseteq$ operator is an asymmetric operator on trees instead of sets, where `Nothing` in a leaf node on the right corresponds to undefined behavior and allows anything on the left:

```
(⊑) :: Eq a ⇒ Tree S a → Tree S (Maybe a) → Bool
_       ⊑ Lf Nothing   = True
Lf n    ⊑ Lf (Just n′) = n == n′
Br s bs ⊑ Br s′ bs′    = s == s′ && all (flip any bs′ ∘ (⊑)) bs
_       ⊑ _            = False
```

5.3.3 Example

Let $e =$

$$\texttt{x = 1, (a, x = 0) + (b, } \langle \texttt{\# x?} \rangle \texttt{).}$$

Figure 5.2 sketches the evaluation of this expression according to the abstract semantics $\mathcal{E}_{\mathsf{abs}}[\![e]\!]$ as a tree (such as that produced by `proj`), where branching represents non-determinism. Now consider the evaluation of this expression according to the concrete semantics $\mathcal{E}_{\mathsf{conc}}[\![e]\!]$. According to the possible interpretations of undefined behavior outlined above, $\mathcal{E}_{\mathsf{conc}}$ should conform to one of the sequences of observables given in Figure 5.3.

That is, if $\mathcal{E}_{\mathsf{conc}}$ conforms to $\mathcal{E}_{\mathsf{abs}}$, then it should be the case that

$$(\mathsf{proj} \circ \mathcal{E}_{\mathsf{conc}})[\![e]\!] \sqsubseteq (\mathsf{prune} \circ \mathsf{proj} \circ \mathcal{E}_{\mathsf{abs}})[\![e]\!]$$

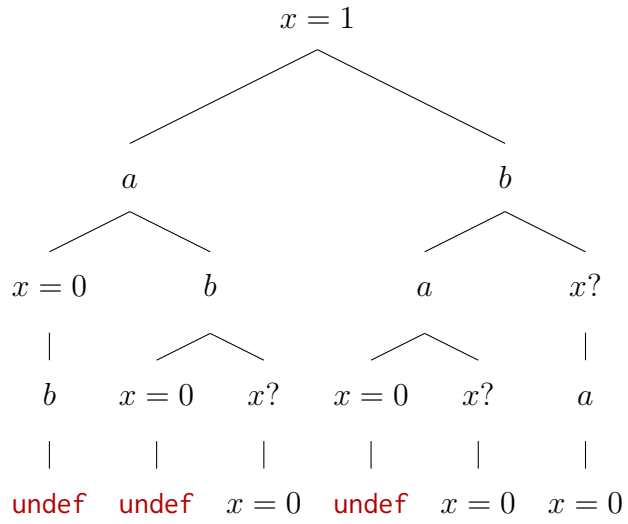


Figure 5.2. Possible evaluations of the expression `x = 1, (a, x = 0) + (b, ⟨# x?⟩)` according to the abstract semantics.

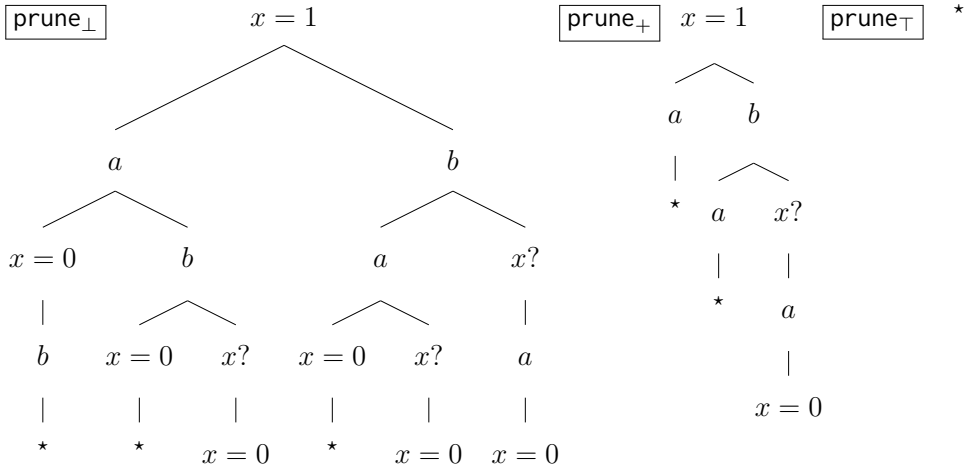


Figure 5.3. Possible evaluations of the expression `x = 1, (a, x = 0) + (b, ⟨# x?⟩)` according to the concrete semantics.

for one of the definitions of prune given above.

5.4 The IMP$_{\texttt{undef}}$ language

In the semantics of expressions given above, non-determinism represented unspecified evaluation order, but statements in IMP$_{\texttt{undef}}$ have a well-defined evaluation order. In this language, we introduce non-determinism with the $\oplus$ operator, which evaluates its operands non-deterministically, and we introduce undefined behavior with the undef statement. Below, we give the syntax for the language followed by an operational and denotational semantics. Our goal with this language is to explore a loop optimization in the context of undefined behavior.

5.4.1 Syntax

$$
\begin{array}{rcl}
s & \in & \textbf{Stmt} ::= x \leftarrow e \mid \texttt{while (} e \texttt{) } s \mid \texttt{if (} e \texttt{) } s \\
& & \quad \mid s_1 \texttt{ ; } s_2 \mid \texttt{obs } e \mid \texttt{undef} \\
e & \in & \textbf{Expr} ::= e_1 \texttt{ + } e_2 \mid e_1 \texttt{ - } e_2 \mid e_1 \vee e_2 \mid e_1 \wedge e_2 \mid \neg\, e \\
& & \quad \mid n \mid x \mid e_1 \oplus e_2 \\
n & \in & \textbf{Num}, \; x \in \textbf{Ide}
\end{array}
$$

This is a simple imperative language extended with an undef statement, which is a statement that intentionally has no semantics. As such, the semantics of this language are not novel. Some notes on the intended meaning of this language:

- The semi-colon sequences statements.

- As in $\mathsf{ELSE}_{\texttt{undef}}$ above, this language uses a store, where variables are assigned integer values in the assignment statement ($x \leftarrow e$).
- Non-zero integers are interpreted as the logical value true and zero as false for the purposes of evaluating the logical operators as well as the condition of the `while` and `if` statements.
- The operands of $\oplus$ are evaluated non-deterministically.
- The `undef` statement unconditionally invokes undefined behavior.

Below, we present an operational semantics for this language in addition to a denotational semantics in Haskell like that given for $\mathsf{ELSE}_{\texttt{undef}}$ in Section 5.3 above.

5.4.2 Operational semantics

Here we give a $\mathbb{K}$-style operational semantics of this language (not including heating/cooling rules for strictness). Note that no rule rewrites the `undef` statement:

```
rule I1 + I2 => I1 +Int I2

rule I1 - I2 => I1 -Int I2

rule ¬ 0 => 1

rule ¬ N => 0 requires N =/=Int 0

rule N ∧ E => E requires N =/=Int 0

rule 0 ∧ _ => 0

rule N ∨ _ => 1 requires N =/=Int 0

rule 0 ∨ E => E

rule E ⊕ _ => E [transition]

rule _ ⊕ E => E [transition]

rule ⟨ X : Id / V ⟩k ⟨ ··· X ↦ V : Int ··· ⟩sto

rule ⟨ X : Id ← V : Int / •K ⟩k ⟨ Sto : Map / Sto[X ← V] ⟩sto

rule S1 ; S2 => S1 ↷ S2

rule if ( 0 ) _ => •K

rule if ( N ) S => S requires N =/=Int 0

rule while ( E ) S => if ( E ) S ; while ( E ) S

rule ⟨ obs V : Int / •K ⟩k ⟨ ··· •List / V ⟩obs
```

We also give a more traditional small-step operational semantics for this language below in terms of resumptions. A resumption is a syntactic class given by

$$r \in \textbf{Res} ::= \texttt{Undef} \mid \texttt{E} \mid s \curvearrowright r.$$

where E represents normal termination and Undef that the undef statement was encountered.

Our operational semantics is in terms of a transition system $\mathcal{T}_{\texttt{undef}} = (\text{Res} \times \Sigma, \mathbb{N}, \Rightarrow, \text{Spec})$. The transitions of $\mathcal{T}_{\texttt{undef}}$ are triples of the form $([r_1, \sigma_1], n, [r_2, \sigma_2])$, or, in terms of $\Rightarrow$

$$[r_1, \sigma_1] \overset{n}{\Rightarrow} [r_2, \sigma_2]$$

Spec is given by

$$[(x \leftarrow e) \curvearrowright r, \sigma] \overset{\epsilon}{\Rightarrow} [r, \sigma\{\beta/x\}] \qquad \text{where } \beta \in \mathcal{O}_{\text{Expr}}(e)(\sigma)$$

$$[\texttt{while } (e)\ s \curvearrowright r, \sigma] \overset{\epsilon}{\Rightarrow} [\texttt{if } (e)\ (s; \texttt{while } (e)\ s) \curvearrowright r, \sigma]$$

$$[\texttt{if } (e)\ s \curvearrowright r, \sigma] \overset{\epsilon}{\Rightarrow} [s \curvearrowright r, \sigma] \qquad \text{if } \exists x \in \mathcal{O}_{\text{Expr}}(e)(\sigma) \wedge x \neq 0$$

$$[\texttt{if } (e)\ s \curvearrowright r, \sigma] \overset{\epsilon}{\Rightarrow} [r, \sigma] \qquad \text{if } 0 \in \mathcal{O}_{\text{Expr}}(e)(\sigma)$$

$$[s_1; s_2 \curvearrowright r, \sigma] \overset{\epsilon}{\Rightarrow} [s_1 \curvearrowright s_2 \curvearrowright r, \sigma]$$

$$[\texttt{obs } e \curvearrowright r, \sigma] \overset{\alpha}{\Rightarrow} [r, \sigma] \qquad \text{where } \alpha \in \mathcal{O}_{\text{Expr}}(e)(\sigma)$$

$$[\texttt{undef} \curvearrowright r, \sigma] \overset{\epsilon}{\Rightarrow} [\texttt{Undef}, \sigma]$$

The operational semantics for expressions is in terms of a transition system $\mathcal{T}_{\text{Expr}} = (\text{Expr}, \Sigma, \Rightarrow, \text{Spec})$. The transitions of $\mathcal{T}_{\text{Expr}}$ are tuples of the form $([e_1, \sigma], e_2)$:

$$\frac{[e_1, \sigma] \Rightarrow \alpha_1 \quad [e_2, \sigma] \Rightarrow \alpha_2}{[e_1 \odot e_2, \sigma] \Rightarrow \alpha_1 \odot \alpha_2} \text{ where } \odot \in \{+, -, \wedge, \vee\}.$$

$$n_1 \wedge n_2 = \begin{cases} 1 & n_1 \neq 0 \text{ and } n_2 \neq 0, \\ 0 & \text{otherwise.} \end{cases} \qquad n_1 \vee n_2 = \begin{cases} 1 & n_1 \neq 0 \text{ or } n_2 \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

$$\frac{[e, \sigma] \Rightarrow \alpha \quad \alpha \neq 0}{[\neg e, \sigma] \Rightarrow 0} \qquad \frac{[e, \sigma] \Rightarrow 0}{[\neg e, \sigma] \Rightarrow 1}$$

$$[n, \sigma] \Rightarrow n \qquad [x, \sigma] \Rightarrow \sigma(x)$$

$$\frac{[e_1, \sigma] \Rightarrow \alpha}{[e_1 \oplus e_2, \sigma] \Rightarrow \alpha} \qquad \frac{[e_2, \sigma] \Rightarrow \alpha}{[e_1 \oplus e_2, \sigma] \Rightarrow \alpha}$$

Now $\mathcal{O}_{\text{Expr}} : \text{Expr} \to \Sigma \to \mathcal{P}_{\geq 1}(\text{Val})$:

$$\mathcal{O}_{\text{Expr}}(e)(\sigma) = \{\alpha \mid [e, \sigma] \Rightarrow \alpha\}$$

Finally, $\mathcal{O}_{\mathsf{undef}} : \text{Stmt} \to \mathcal{P}_{\geq 1}(\mathbb{N}^\infty)$:

$$\begin{aligned} \mathcal{O}_{\mathsf{undef}}(p) = &\left\{\alpha_1 \cdots \alpha_n \;\middle|\; [p \curvearrowright \mathsf{E}, \sigma_0] \overset{\alpha_1}{\Rightarrow} \cdots \overset{\alpha_n}{\Rightarrow} [U, \sigma]\right\} \\ &\bigcup \left\{\alpha_1 \cdots \alpha_n \alpha_n \cdots \;\middle|\; [p \curvearrowright \mathsf{E}, \sigma_0] \overset{\alpha_1}{\Rightarrow} \cdots \overset{\alpha_n}{\Rightarrow} [\mathsf{E}, \sigma]\right\} \\ &\bigcup \left\{\alpha_1 \cdots \;\middle|\; [p \curvearrowright \mathsf{E}, \sigma_0] \overset{\alpha_1}{\Rightarrow} \cdots\right\} \end{aligned}$$

5.4.3 Denotational semantics in Haskell

Here we give a monadic-style denotational semantics of $\mathsf{IMP}_{\mathsf{undef}}$ in terms of a function $\mathcal{W}^{\mathcal{E}}_{\mathsf{abs}}$ for evaluating expressions and $\mathcal{W}_{\mathsf{abs}}[\![\ ]\!]$ for evaluating statements. The definition for $\mathcal{W}^{\mathcal{E}}_{\mathsf{abs}}$ follows:

```
W^E_abs[[ e1 ⊙I e2 ]] = do                -- where ⊙I ∈ {+, -}.
  n1 ← W^E_abs[[ e1 ]]
  n2 ← W^E_abs[[ e2 ]]
  return (n1 ⊙I n2)
W^E_abs[[ e1 ⊙B e2 ]] = do                -- where ⊙B ∈ {∧, ∨}.
  n1 ← W^E_abs[[ e1 ]]
  n2 ← W^E_abs[[ e2 ]]
  return (if (n1 /= 0) ⊙B (n2 /= 0) then 1 else 0)
W^E_abs[[ ¬e ]] = do
  n ← W^E_abs[[ e ]]
  return (if n /= 0 then 0 else 1)
W^E_abs[[ n ]] = return n
W^E_abs[[ x ]] = do
  s ← get
  return (load' x s)
W^E_abs[[ e1 ⊕ e2 ]] = W^E_abs[[ e1 ]] +|+ W^E_abs[[ e2 ]]
```

Monadic-style denotational semantics for the statements of $\mathsf{IMP}_{\mathsf{undef}}$:

```
𝒲abs⟦ x = e ⟧ = do
  n ← 𝒲ᵉabs⟦ e ⟧
  modify (store′ x n)
𝒲abs⟦ undef ⟧ = undef
𝒲abs⟦ obs e ⟧ = do
  n ← 𝒲ᵉabs⟦ e ⟧
  modify (obs (show n))
𝒲abs⟦ while ( e ) s ⟧ =
  𝒲abs⟦ if ( e ) ( s ; while ( e ) s ) ⟧
𝒲abs⟦ if ( e ) s ⟧ = do
  n ← 𝒲ᵉabs⟦ e ⟧
  if (n /= 0) then 𝒲abs⟦ s ⟧ else return ()
𝒲abs⟦ s1 ; s2 ⟧ = do
  𝒲abs⟦ s1 ⟧
  𝒲abs⟦ s2 ⟧
```

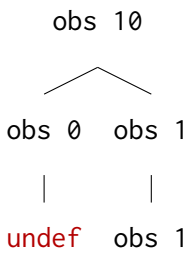
5.4.4 Example

With $\mathsf{IMP}_{\mathsf{undef}}$, we can consider the case of undefined behavior propagating backward through a trace due to loop optimizations given in Figure 5.1. An analogous example in our $\mathsf{IMP}_{\mathsf{undef}}$ language would be a program like that given in Figure 5.4.

Program p_1 =

```
1 n ← 2;
2 x ← 0 ⊕ 1;
3 obs 10;
4 while (n) {
5       obs x;
6       if (¬ x) undef;
7       n ← n - 1
8 }
```

Observable traces through p_1:



Program p_2 =

```
1 n ← 2;
2 x ← 0 ⊕ 1;
3 obs 10;
4 if (¬ x) undef;
5 while (n) {
6       obs x;
7       n ← n - 1
8 }
```

Observable traces through p_2:

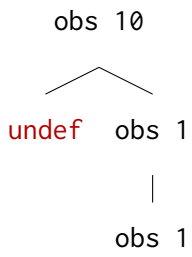


Figure 5.4. A possible loop optimization and its effect on the abstract evaluation of a program.

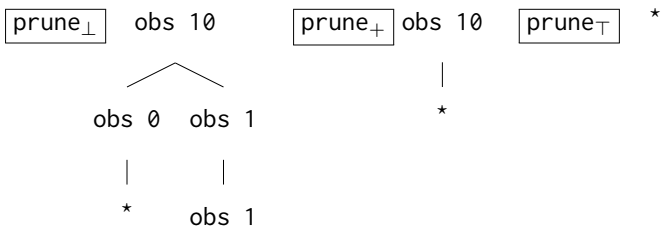


Figure 5.5. The effect of a loop optimization on the concrete evaluation of a program, according to different definitions of prune.

Figure 5.5 sketches the effect of the different versions of prune presented above on the observable behavior of p_1, where the asterisk represents the point at where the trace was pruned.

Now if $\mathcal{W}_{\mathsf{conc}}$ conforms to $\mathcal{W}_{\mathsf{abs}}$, but optimizes p_1 to p_2, then it should be the case that both

$$(\mathtt{proj} \circ \mathcal{W}_{\mathsf{conc}})[\![p_1]\!] \sqsubseteq (\mathtt{prune} \circ \mathtt{proj} \circ \mathcal{W}_{\mathsf{abs}})[\![p_1]\!]$$

and

$$(\mathtt{proj} \circ \mathcal{W}_{\mathsf{conc}})[\![p_2]\!] \sqsubseteq (\mathtt{prune} \circ \mathtt{proj} \circ \mathcal{W}_{\mathsf{abs}})[\![p_1]\!],$$

for one of the definitions of prune given above. This holds for $\mathtt{prune}_+$ and $\mathtt{prune}_\bot$, but not for $\mathtt{prune}_\top$. That is, if our interpretation of undefined behavior allows this sort of optimization, then we would want to choose a definition

of conformance that does not require a complete observable trace that ends in undefined behavior to be preserved.

Future work. Producing an operational semantics of the $\mathsf{ELSE}_{\mathtt{undef}}$ language is future work. Additionally, we would like to prove a correspondence between the operational and denotational interpretations presented above as well as the theorems about the conformance of the concrete semantics to the abstract semantics of both $\mathsf{ELSE}_{\mathtt{undef}}$ and $\mathsf{IMP}_{\mathtt{undef}}$.

Chapter 6

Conclusion

We have investigated undefined behaviors in C and how to capture these behaviors semantically. We have used these techniques to develop a semantics-based analysis tool, which we tested against other popular analysis tools. We compared the tools on our own (publicly-available) test suite, as well as third-party test suites. We believe this demonstrates the viability of our technique. Although there exist many tools specialized in detecting various kinds of undefinedness, no other tool manages this sort of comprehensivness.

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

Future work. Future efforts include more comprehensive support for threading (either POSIX threads or the `threads.h` of the standard library). We have demonstrated support simple cases (including simple cases with many threads [18]), but we have little support for many of the features of these threading libraries.

Extending the C semantics with semantics for C++ (and thereby catching C++ undefined behavior as well) is an ongoing effort.

Appendices

Appendix A

Incomplete list of C11 undefined behaviors

Below we include the list of undefined behaviors from Annex J of the C11 standard [24]. The numbers are the same as appear in the CERT C Secure Coding Standard [45] and correspond to the order in which each behavior appears in Annex J. That annex is not normative, but each behavior includes a citation to the (perhaps differently-worded) relevant part of the main body of the standard.

Beside each item, we include a current estimate of our ability to catch each behavior with kcc:

- ○ kcc does not currently catch instances of this behavior.
- ◐ kcc currently catches some instances of this behavior.
- ● kcc currently catches all or most instances of this behavior.

	Other
1	◐ A “shall” or “shall not” requirement that appears outside of a constraint is violated (clause 4).
	Core language: translation time
8	● The same identifier has both internal and external linkage in the same translation unit (6.2.2).
16	● A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).
20	● A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).
23	● An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to void) is applied to a void expression (6.3.2.2).
32	● The identifier `__func__` is explicitly declared (6.4.2.2).
44	● A pointer is converted to other than an integer or pointer type (6.5.4).

55 ● An expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, `_Alignof` expressions, or immediately-cast floating constants; or contains casts (outside operands to sizeof and `_Alignof` operators) other than conversions of arithmetic types to integer types (6.6).

56 ● A constant expression in an initializer is not, or does not evaluate to, one of the following: an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for a complete object type plus or minus an integer constant expression (6.6).

57 ● An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, or `_Alignof` expressions; or contains casts (outside operands to sizeof or `_Alignof` operators) other than conversions of arithmetic types to arithmetic types (6.6).

58 ● The value of an object is accessed by an array-subscript `[]`, member-access `.` or `->`, address `&`, or indirection `*` operator or a pointer cast in creating an address constant (6.6).

59 ● An identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.7).

60 ● A function is declared at block scope with an explicit storage-class specifier other than `extern` (6.7.1).

61 ● A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1).

63 ● When the complete type is needed, an incomplete structure or union type is not completed in the same scope by another declaration of the tag that defines the content (6.7.2.3).

66 ● The specification of a function type includes any type qualifiers (6.7.3).

70 ● A function with external linkage is declared with an `inline` function specifier, but is not also defined in the same translation unit (6.7.4).

78 ● A storage-class specifier or type qualifier modifies the keyword `void` as a function parameter type list (6.7.6.3).

81 ● The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.9).

82 ● The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.9).

83 ● The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.9).

85 ● A function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list (6.9.1).

86 ● An adjusted parameter type in a function definition is not a complete object type (6.9.1).

89 ● An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).

Core language: link time

4 ● A program in a hosted environment does not define a function named main using one of the specified forms (5.1.2.2.1).

15 ● Two declarations of the same object or function specify types that are not compatible (6.2.7).

67 ● Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).

72 ● The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5).

73 ● Declarations of an object in different translation units have different alignment specifiers (6.7.5).

74 ● Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.6.1).

79 ● In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipsis terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list or when one type is specified by a function definition with an identifier list) (6.7.6.3).

84 ● An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9).

Core language: execution time

9 ● An object is referred to outside of its lifetime (6.2.4).

10 ● The value of a pointer to an object whose lifetime has ended is used (6.2.4).

11 ● The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.9, 6.8).

12 ● A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).

13 ● A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).

14 ● The operands to certain operators are such that they could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).

17 ● Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

18 ● Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).

19 ● An lvalue does not designate an object when evaluated (6.3.2.1).

21 ● An lvalue designating an object of automatic storage duration that could have been declared with the register storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1).

22 ● An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).

24 ● Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).

25 ● Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).

26 ● A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).

33 ● The program attempts to modify a string literal (6.4.5).

35 ● 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 (6.5).

36 ● An exceptional condition occurs during the evaluation of an expression (6.5).

37 ● An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

38 ● For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).

39 ● For call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after promotion are not compatible with the types of the parameters (6.5.2.2).

40 ● For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2).

41 ● A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

42 ● A member of an atomic structure or union is accessed (6.5.2.3).

43 ● The operand of the unary * operator has an invalid value (6.5.3.2).

45 ● The value of the second operand of the / or % operator is zero (6.5.5).

46 ● Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

47 ● Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

48 ● Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

49 ● An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`) (6.5.6).

50 ● The result of subtracting two pointers is not representable in an object of type `ptrdiff_t` (6.5.6).

51 ● An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

52 ● An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would be not be representable in the promoted type (6.5.7).

53 ● Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).

54 ● An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).

62 ● An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).

64 ● An attempt is made to modify an object defined with a `const`-qualified type through use of an lvalue with non-`const`-qualified type (6.7.3).

65 ● An attempt is made to refer to an object defined with a `volatile`-qualified type through use of an lvalue with non-`volatile`-qualified type (6.7.3).

68 ● An object which has been modified is accessed through a restrict-qualified pointer to a `const`-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).

69 ● A restrict-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).

71 ● A function declared with a `_Noreturn` function specifier returns to its caller (6.7.4).

75 ● The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.6.2).

76 ● In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.6.2).

77 ● A declaration of an array parameter includes the keyword static within the `[` and `]` and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.6.3).

80 ● The value of an unnamed member of a structure or union is used (6.7.9).

87 ● A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.9.1).

88 ● The `}` that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).

2 ○ A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).

3 ○ Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).

6 ○ A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).

7 ○ An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).

27 ○ An unmatched ' or " character is encountered on a logical source line during tokenization (6.4).

28 ○ A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).

29 ○ A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).

30 ○ The initial character of an identifier is a universal character name designating a digit (6.4.2.1).

31 ○ Two identifiers differ only in nonsignificant characters (6.4.2.1).

34 ○ The characters ', \, ", //, or /* occur in the sequence between the < and > delimiters, or the characters ', \, //, or /* occur in the sequence between the " delimiters, in a header name preprocessing token (6.4.7).

101 ○ A file with the same name as one of the standard headers, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files (7.1.2).

Early translation: preprocessor

---

90 ○ The token defined is generated during the expansion of a `#if` or `#elif` preprocessing directive, or the use of the defined unary operator does not match one of the two specified forms prior to macro replacement (6.10.1).

91 ○ The `#include` preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).

92 ○ The character sequence in an `#include` preprocessing directive does not start with a letter (6.10.2).

93 ○ There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.3).

94 ○ The result of the preprocessing operator # is not a valid character string literal (6.10.3.2).

95 ○ The result of the preprocessing operator ## is not a valid preprocessing token (6.10.3.3).

96 ○ The `#line` preprocessing directive that results after expansion does not match one of the two well-defined forms, or its digit sequence specifies zero or a number greater than 2147483647 (6.10.4).

97 ○ A non-STDC `#pragma` preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.6).

98 ○ A `#pragma` STDC preprocessing directive does not match one of the well-defined forms (6.10.6).

99 ○ The name of a predefined macro, or the identifier defined, is the subject of a `#define` or `#undef` preprocessing directive (6.10.8).

102 ○ A header is included within an external declaration or definition (7.1.2).

104 ○ A standard header is included while a macro is defined with the same name as a keyword (7.1.2).

107 ○ The program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3).

Standard library

---

5 ○ The execution of a program contains a data race (5.1.2.4).

100 ◐ An attempt is made to copy an object to an overlapping object by use of a library function, other than as explicitly allowed (e.g., `memmove`) (clause 7).

103 ● A function, object, type, or macro that is specified as being declared or defined by some standard header is used before any header that declares or defines it is included (7.1.2).

105 ● The program attempts to declare a library function itself, rather than via a standard header, but the declaration does not have external linkage (7.1.2).

106 ◐ The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).

108 ◐ An argument to a library function has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).

109 ◐ The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

110 ○ The macro definition of assert is suppressed in order to access an actual function (7.2).

111 ○ The argument to the assert macro does not have a scalar type (7.2).

112 ○ The `CX_LIMITED_RANGE`, `FENV_ACCESS`, or `FP_CONTRACT` pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.3.4, 7.6.1, 7.12.2).

113 ○ The value of an argument to a character handling function is neither equal to the value of EOF nor representable as an unsigned char (7.4).

114 ○ A macro definition of errno is suppressed in order to access an actual object, or the program defines an identifier with the name errno (7.5).

115 ○ Part of the program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the `FENV_ACCESS` pragma "off" (7.6.1).

116 ○ The exception-mask argument for one of the functions that provide access to the floating-point status flags has a nonzero value not obtained by bitwise OR of the floating-point exception macros (7.6.2).

117 ○ The `fesetexceptflag` function is used to set floating-point status flags that were not specified in the call to the `fegetexceptflag` function that provided the value of the corresponding `fexcept_t` object (7.6.2.4).

118 ○ The argument to fesetenv or feupdateenv is neither an object set by a call to fegetenv or feholdexcept, nor is it an environment macro (7.6.4.3, 7.6.4.4).

119 ○ The value of the result of an integer arithmetic or conversion function cannot be represented (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.22.6.1, 7.22.6.2, 7.22.1).

120 ● The program modifies the string pointed to by the value returned by the `setlocale` function (7.11.1.1).

121 ● The program modifies the structure pointed to by the value returned by the `localeconv` function (7.11.2.1).

122 ○ A macro definition of `math_errhandling` is suppressed or the program defines an identifier with the name `math_errhandling` (7.12).

123 ○ An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.14).

124 ○ A macro definition of `setjmp` is suppressed in order to access an actual function, or the program defines an external identifier with the name `setjmp` (7.13).

125 ○ An invocation of the `setjmp` macro occurs other than in an allowed context (7.13.2.1).

126 ● The `longjmp` function is invoked to restore a nonexistent environment (7.13.2.1).

127 ○ After a `longjmp`, there is an attempt to access the value of an object of automatic storage duration that does not have `volatile`-qualified type, local to the function containing the invocation of the corresponding `setjmp` macro, that was changed between the `setjmp` invocation and `longjmp` call (7.13.2.1).

128 ● The program specifies an invalid pointer to a signal handler function (7.14.1.1).

129 ● A signal handler returns when the signal corresponded to a computational exception (7.14.1.1).

130 ● A signal handler called in response to `SIGFPE`, `SIGILL`, `SIGSEGV`, or any other implementation-defined value corresponding to a computational exception returns (7.14.1.1).

131 ● A signal occurs as the result of calling the `abort` or `raise` function, and the signal handler calls the `raise` function (7.14.1.1).

132 ○ A signal occurs other than as the result of calling the `abort` or `raise` function, and the signal handler refers to an object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as `volatile sig_atomic_t`, or calls any function in the standard library other than the `abort` function, the `_Exit` function, the `quick_exit` function, or the `signal` function (for the same signal number) (7.14.1.1).

133 ○ The value of errno is referred to after a signal occurred other than as the result of calling the `abort` or `raise` function and the corresponding signal handler obtained a `SIG_ERR` return from a call to the `signal` function (7.14.1.1).

134 ○ A signal is generated by an asynchronous signal handler (7.14.1.1).

135 ● The `signal` function is used in a multi-threaded program (7.14.1.1).

136 ● A function with a variable number of arguments attempts to access its varying arguments other than through a properly declared and initialized `va_list` object, or before the `va_start` macro is invoked (7.16, 7.16.1.1, 7.16.1.4).

137 ○ The macro `va_arg` is invoked using the parameter ap that was passed to a function that invoked the macro `va_arg` with the same parameter (7.16).

138 ○ A macro definition of `va_start`, `va_arg`, `va_copy`, or `va_end` is suppressed in order to access an actual function, or the program defines an external identifier with the name `va_copy` or `va_end` (7.16.1).

139 ● The `va_start` or `va_copy` macro is invoked without a corresponding invocation of the `va_end` macro in the same function, or vice versa (7.16.1, 7.16.1.2, 7.16.1.3, 7.16.1.4).

140 ● The type parameter to the `va_arg` macro is not such that a pointer to an object of that type can be obtained simply by postfixing a * (7.16.1.1).

141 ● The `va_arg` macro is invoked when there is no actual next argument, or with a specified type that is not compatible with the promoted type of the actual next argument, with certain exceptions (7.16.1.1).

142 ● The `va_copy` or `va_start` macro is called to initialize a `va_list` that was previously initialized by either macro without an intervening invocation of the `va_end` macro for the same `va_list` (7.16.1.2, 7.16.1.4).

143 ● The parameter parmN of a `va_start` macro is declared with the register storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions (7.16.1.4).

144 ● The member designator parameter of an offsetof macro is an invalid right operand of the . operator for the type parameter, or designates a bit-field (7.19).

145 ○ The argument in an instance of one of the integer-constant macros is not a decimal, octal, or hexadecimal constant, or it has a value that exceeds the limits for the corresponding type (7.20.4).

146 ○ A byte input/output function is applied to a wide-oriented stream, or a wide character input/output function is applied to a byte-oriented stream (7.21.2).

147 ○ Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.21.2).

148 ● The value of a pointer to a `FILE` object is used after the associated file is closed (7.21.3).

149 ● The stream for the `fflush` function points to an input stream or to an update stream in which the most recent operation was input (7.21.5.2).

150 ● The string pointed to by the mode argument in a call to the `fopen` function does not exactly match one of the specified character sequences (7.21.5.3).

151 ● An output operation on an update stream is followed by an input operation without an intervening call to the `fflush` function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.21.5.3).

152 ● An attempt is made to use the contents of the array that was supplied in a call to the `setvbuf` function (7.21.5.6).

153 ● There are insufficient arguments for the format in a call to one of the formatted input/output functions, or an argument does not have an appropriate type (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

154 ○ The format in a call to one of the formatted input/output functions or to the `strftime` or `wcsftime` function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

155 ○ In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).

156 ● A conversion specification for a formatted output function uses an asterisk to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.21.6.1, 7.29.2.1).

157 ◐ A conversion specification for a formatted output function uses a `#` or `0` flag with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).

158 ● A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

159 ● An `s` conversion specifier is encountered by one of the formatted output functions, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.21.6.1, 7.29.2.1).

160 ◐ An `n` conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

161 ◐ A % conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly

162 ◐ An invalid conversion specification is found in the format for one of the formatted input/output functions, or the `strftime` or `wcsftime` function (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

163 ○ The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than `INT_MAX` (7.21.6.1, 7.29.2.1).

164 ○ The number of input items assigned by a formatted input function is greater than `INT_MAX` (7.21.6.2, 7.29.2.2).

165 ◐ The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.21.6.2, 7.29.2.2).

166 ◐ A `c`, `s`, or `[` conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is s or []) (7.21.6.2, 7.29.2.2).

167 ○ A `c`, `s`, or `[` conversion specifier with an l qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.21.6.2, 7.29.2.2).

168 ◐ The input item for a `%p` conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.21.6.2, 7.29.2.2).

169 ● The `vfprintf`, `vfscanf`, `vprintf`, `vscanf`, `vsnprintf`, `vsprintf`, `vsscanf`, `vfwprintf`, `vfwscanf`, `vswprintf`, `vswscanf`, `vwprintf`, or `vwscanf` function is called with an improperly initialized `va_list` argument, or the argument is used (other than in an invocation of `va_end`) after the function returns (7.21.6.8, 7.21.6.9, 7.21.6.10, 7.21.6.11, 7.21.6.12, 7.21.6.13, 7.21.6.14, 7.29.2.5, 7.29.2.6, 7.29.2.7, 7.29.2.8, 7.29.2.9, 7.29.2.10).

170 ● The contents of the array supplied in a call to the `fgets` or `fgetws` function are used after a read error occurred (7.21.7.2, 7.29.3.2).

171 ○ The file position indicator for a binary stream is used after a call to the `ungetc` function where its value was zero before the call (7.21.7.10).

172 ○ The file position indicator for a stream is used after an error occurred during a call to the `fread` or `fwrite` function (7.21.8.1, 7.21.8.2).

173 ○ A partial element read by a call to the `fread` function is used (7.21.8.1).

174 ○ The `fseek` function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the `ftell` function on a stream associated with the same file or whence is not `SEEK_SET` (7.21.9.2).

175 ● The `fsetpos` function is called to set a position that was not returned by a previous successful call to the `fgetpos` function on a stream associated with the same file (7.21.9.3).

176 ● A non-null pointer returned by a call to the `calloc`, `malloc`, or `realloc` function with a zero requested size is used to access an object (7.22.3).

177 ● The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is used (7.22.3).

178 ● The alignment requested of the `aligned_alloc` function is not valid or not supported by the implementation, or the size requested is not an integral multiple of the alignment (7.22.3.1).

179 ● The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to free or realloc (7.22.3.3, 7.22.3.5).

180 ● The value of the object allocated by the `malloc` function is used (7.22.3.4).

181 ● The value of any bytes in a new object allocated by the `realloc` function beyond the size of the old object are used (7.22.3.5).

182 ● The program calls the exit or `quick_exit` function more than once, or calls both functions (7.22.4.4, 7.22.4.7).

183 ○ During the call to a function registered with the atexit or `at_quick_exit` function, a call is made to the `longjmp` function that would terminate the call to the registered function (7.22.4.4, 7.22.4.7).

184 ● The string set up by the `getenv` or `strerror` function is modified by the program (7.22.4.6, 7.24.6.2).

185 ○ A signal is raised while the `quick_exit` function is executing (7.22.4.7).

186 ○ A command is executed through the `system` function in a way that is documented as causing termination or some other form of undefined behavior (7.22.4.8).

187 ○ A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.22.5).

188 ○ The comparison function called by a searching or sorting utility function alters the contents of the array being searched or sorted, or returns ordering values inconsistently (7.22.5).

189 ○ The array being searched by the `bsearch` function does not have its elements in proper order (7.22.5.1).

190 ○ The current conversion state is used by a multibyte/wide character conversion function after changing the `LC_CTYPE` category (7.22.7).

191 ● A string or wide string utility function is instructed to access an array beyond the end of an object (7.24.1, 7.29.4).

192 ○ A string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.24.1, 7.29.4).

193 ○ The contents of the destination array are used after a call to the `strxfrm`, `strftime`, `wcsxfrm`, or `wcsftime` function in which the specified length was too small to hold the entire null-terminated result (7.24.4.5, 7.27.3.5, 7.29.4.4.4, 7.29.5.1).

194 ○ The first argument in the very first call to the strtok or wcstok is a null pointer (7.24.5.8, 7.29.4.5.7).

195 ○ The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.25).

196 ○ A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.25).

197 ○ At least one member of the broken-down time passed to asctime contains a value outside its normal range, or the calculated year exceeds four digits or is less than the year 1000 (7.27.3.1).

198 ○ The argument corresponding to an s specifier without an l qualifier in a call to the `fwprintf` function does not point to a valid multibyte character sequence that begins in the initial shift state (7.29.2.11).

199 ○ In a call to the `wcstok` function, the object pointed to by ptr does not have the value stored by the previous call for the same wide string (7.29.4.5.7).

200 ○ An `mbstate_t` object is used inappropriately (7.29.6).

201 ○ The value of an argument of type `wint_t` to a wide character classification or case mapping function is neither equal to the value of WEOF nor representable as a `wchar_t` (7.30.1).

202 ○ The `iswctype` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctype` function that returned the description (7.30.2.2.1).

203 ○ The `towctrans` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctrans` function that returned the description (7.30.3.2.1).

VITA

Chris is a software developer (and C semanticist) at Runtime Verification Inc., working to turn the work described in this thesis into a professional-quality tool to help real programmers write better (and fully-defined) code.

Bibliography

[1] Andrei Ștefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roșu. "Semantics-Based Program Verifiers for All Languages". In: *Proceedings of the 31th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, 2016. DOI: 10.1145/2983990.2984027.

[2] Grigore Roșu and Traian Florin Șerbănuță. "An Overview of the K Semantic Framework". In: *J. Logic and Algebraic Programming* 79.6 (2010), pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

[3] Grigore Roșu and Traian Florin Șerbănuță. "$\mathbb{K}$ Overview and SIMPLE Case Study". In: *Proceedings of International K Workshop (K'11)*. ENTCS. Elsevier, 2014, pp. 3–56. DOI: 10.1016/j.entcs.2014.05.002.

[4] Jaco de Bakker and Erik de Vink. *Control Flow Semantics*. The MIT Press, 1996.

[5] Sandrine Blazy and Xavier Leroy. "Mechanized Semantics for the Clight Subset of the C Language". In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288. DOI: 10.1007/s10817-009-9148-3.

[6] Denis Bogdănaş and Grigore Roşu. "K-Java: A Complete Semantics of Java". In: *POPL*. ACM, 2015, pp. 445–456. DOI: 10.1145/2676726.2676982.

[7] Brian Campbell. "An Executable Semantics for CompCert C". In: *Certified Programs and Proofs*. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 60–75. DOI: 10.1007/978-3-642-35308-6_8.

[8] Géraud Canet, Pascal Cuoq, and Benjamin Monate. "A Value Analysis for C Programs". In: *Conf. on Source Code Analysis and Manipulation (SCAM'09)*. IEEE, 2009, pp. 123–124. DOI: 10.1109/SCAM.2009.22.

[9] Clang. *Clang 3.9 Documentation*. URL: http://clang.llvm.org/docs/index.html.

[10] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTRÉE Analyzer". In: *Programming Languages and Systems*. Vol. 3444. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3.

[11] Philip Daian, Dwight Guth, Chris Hathhorn, Yilong Li, Edgar Pek, Manasvi Saxena, Traian Florin Șerbănuță, and Grigore Roșu. "Runtime verification at work: a tutorial". In: *16th Conf. on Runtime Verification (RV'16)*. 2016. DOI: 10.1007/978-3-319-46982-9_5.

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

[13] Chucky Ellison and Grigore Roșu. "An Executable Formal Semantics of C with Applications". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. 2012, pp. 533–544. DOI: 10.1145/2103656.2103719.

[14] Daniele Filaretti and Sergio Maffeis. "An Executable Formal Semantics of PHP". In: *ECOOP*. Springer-Verlag, 2014, pp. 567–592. DOI: 10.1007/978-3-662-44202-9_23.

[15] GrammaTech. *CodeSonar*. URL: http://grammatech.com/.

[16] Dwight Guth. "A formal semantics of Python 3.3". MA thesis. University of Illinois, 2013. URL: http://hdl.handle.net/2142/45275.

[17] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roșu. "RV-Match: practical semantics-based program analysis". In: *28th Conf. on Computer Aided Verification (CAV'16)*. 2016. DOI: 10.1007/978-3-319-41528-4_24.

[18] Chris Hathhorn, Michela Becchi, William Harrison, and Adam Procter. "Formal semantics of heterogeneous CUDA-C: a modular approach with applications". In: *7th Systems Software Verification Conf. (SSV'12)*. 2012. DOI: 10.4204/EPTCS.102.11.

[19] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the Undefinedness of C". In: *36th Conf. on Programming Language Design and Implementation (PLDI'15)*. 2015. DOI: 10.1145/2813885.2737979.

[20] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Roşu. *KEVM: A Complete Semantics of the Ethereum Virtual Machine*. Tech. rep. University of Illinois, 2017. URL: http://hdl.handle.net/2142/97207.

[21] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. DOI: 10.1145/363235.363259.

[22] ISO/IEC JTC 1, SC 22, WG 14. *Defect report #236*. Tech. rep. 2004. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

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

[24] ISO/IEC JTC 1, SC 22, WG 14. *ISO/IEC 9899:2011: Prog. Languages—C*. Tech. rep. Intl. Org. for Standardization, 2012.

[25] ISO/IEC JTC 1, SC 22, WG 14. *Rationale for the International Standard—Prog. Languages—C*. Tech. rep. 5.10. Intl. Org. for Standardization, 2003. URL: http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf.

[26] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. "Cyclone: A Safe Dialect of C". In: *USENIX Annual Technical Conf. (ATEC'02)*. USENIX Association, 2002, pp. 275–288. URL: http://dl.acm.org/citation.cfm?id=647057.713871.

[27] Robbert Krebbers. "Aliasing Restrictions of C11 Formalized in Coq". In: *Certified Programs and Proofs*. Lecture Notes in Computer Science. 2013. DOI: 10.1007/978-3-319-03545-1_4.

[28] Robbert Krebbers. "An operational and axiomatic semantics for non-determinism and sequence points in C". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. ACM, 2014, pp. 101–112. DOI: 10.1145/2535838.2535878.

[29] Robbert Krebbers. "The C standard formalized in Coq". PhD thesis. Radboud University, 2015.

[30] Chris Lattner. *What Every C Programmer Should Know About Undefined Behavior*. 2011. URL: http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[31] Xavier Leroy. *The CompCert C verified compiler: Documentation and user's manual, version 2.3*. Tech. rep. INRIA Paris-Rocquencourt, 2014.

[32] MathWorks. *Polyspace Bug Finder*. URL: http://www.mathworks.com/products/polyspace-bug-finder.

[33] MathWorks. *Polyspace Code Prover*. URL: http://www.mathworks.com/products/polyspace-code-prover.

[34] José Meseguer. "Conditional rewriting logic as a unified model of concurrency". In: *Theoretical Computer Science* 96.1 (1992), pp. 73–155. DOI: 10.1016/0304-3975(92)90182-F.

[35] MISRA. *MISRA-C: 2004—Guidelines for the use of the C language in critical systems*. Tech. rep. MIRA Ltd., 2004.

[36] MITRE. *The Common Weakness Enumeration (CWE) Initiative*. 2012. URL: http://cwe.mitre.org/.

[37] George C. Necula, Scott McPeak, and Westley Weimer. "CCured: type-safe retrofitting of legacy code". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. ACM, 2002, pp. 128–139. DOI: 10.1145/503272.503286.

[38] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'07)*. ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[39] NIST. *Juliet Test Suite for C/C++, version 1.0*. 2010. URL: http://samate.nist.gov/SRD/testsuite.php.

[40] Michael Norrish. *C formalised in HOL*. Tech. rep. UCAM-CL-TR-453. University of Cambridge, 1998.

[41] Nikolaos S. Papaspyrou. "Denotational semantics of ANSI C". In: *Computer Standards and Interfaces* 23.3 (2001), pp. 169–185. DOI: 10.1016/S0920-5489(01)00059-9.

[42] Nikolaos S. Papaspyrou and Dragan Maćoš. “A study of evaluation order semantics in expressions with side effects”. In: *J. Functional Programming* 10.3 (2000), pp. 227–244. DOI: 10.1017/S095679680000366X.

[43] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. “KJS: A Complete Formal Semantics of JavaScript”. In: *PLDI*. ACM, June 2015, pp. 346–356. DOI: 10.1145/2737924.2737991.

[44] John Regehr. *A Guide to Undefined Behavior in C and C++*. 2010. URL: http://blog.regehr.org/archives/213.

[45] Robert C. Seacord. *The CERT C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems*. 2014. URL: https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard.

[46] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. “Test Suites for Benchmarks of Static Analysis Tools”. In: *Intl. Symposium on Software Reliability Engineering (ISSRE’15)*. 2015. DOI: 10.1109/ISSREW.2015.7392027.

[47] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. “Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior”. In: *ACM Symposium on Operating Systems Principles (SOSP’13)*. ACM, 2013, pp. 260–275. DOI: 10.1145/2517349.2522728.