

Defining the Undefinedness of C

Chris Hathhorn, Univ. of Missouri

Chucky Ellison, Univ. of Illinois at Urbana-Champaign

Grigore Roşu, Univ. of Illinois at Urbana-Champaign



June 16, 2015
Portland, OR

Agenda

A semantics of C capturing undefined behavior.

- Intro to the C semantics.
- Intro to undefined behavior in C.
- Our experience in dealing with undefinedness.
- Our kcc tool.

Framework C semantics

Feature	Definition						
	GH	CCR	CR	No	Pa	BL	ER
Bitfields	●	◐	○	○	◐	○	●
Enums	◐	●	○	○	●	○	●
Floats	○	○	○	○	○	●	●
String Literal	○	●	○	○	●	●	●
Struct as Value	○	○	○	●	○	○	●
Arithmetic	◐	●	●	○	●	●	●
Bitwise	○	●	○	○	●	●	●
Casts	◐	◐	○	◐	◐	●	●
Functions	●	●	◐	●	●	●	●
Exp. Side Effects	●	●	○	●	●	○	●
Break/Continue	◐	●	◐	●	●	●	●
Goto	◐	○	○	○	●	○	●
Switch	◐	●	○	○	●	◐	●
Longjmp	○	○	○	○	○	○	●
Malloc	○	○	○	○	○	○	●
Variadic Funcs.	○	○	○	○	○	○	●
Feature	GH	CCR	CR	No	Pa	BL	ER
●: Fully Described ◐: Partially Described ○: Not Described							

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

- The most complete operational semantics of C.
- Interpreter passed 99% of 776 GCC torture tests. Also, a model checker and state-space search tool.
- But focused on the semantics of defined programs.

[Source: Chucky Ellison and Grigore Roşu. *An Executable Formal Semantics of C with Applications*. POPL'12.]

Framework C semantics

$$\langle \cdots \langle K \rangle_k \langle Map \rangle_{env} \langle Map \rangle_{mem} \cdots \rangle_T$$

RULE

$$\left\langle \frac{*X}{V} \cdots \right\rangle_k \quad \langle \cdots X \mapsto L \cdots \rangle_{env} \quad \langle \cdots L \mapsto V \cdots \rangle_{mem}$$

Conventional version of this same rule:

$$\begin{aligned} &\langle *X \curvearrowright \kappa \rangle_k \quad \langle \rho_1, X \mapsto L, \rho_2 \rangle_{env} \quad \langle \sigma_1, L \mapsto V, \sigma_2 \rangle_{mem} \\ &\quad \Rightarrow \langle V \curvearrowright \kappa \rangle_k \quad \langle \rho_1, X \mapsto L, \rho_2 \rangle_{env} \quad \langle \sigma_1, L \mapsto V, \sigma_2 \rangle_{mem} \end{aligned}$$

Framework C semantics

- Our current semantics comprises > 2300 rewrite rules, up from 1163 in the original semantics.
- > 150 cells in the configuration. (62 in the translation semantics, 99 in the execution semantics).
- Catching undefined behavior in C takes a lot of extra effort—e.g., assuming well-defined input programs means type qualifiers can be ignored, pointers will all be valid, declarations compatible, etc.

Undefined behavior

[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**. [C11, §3.4.3:1]

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). [C11, §3.4.3:2]

Undefined behavior

This will *not* cause GCC, Clang, nor ICC to generate a program that raises an error when run:

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

And this won't "at least" print 0 in Clang and ICC:

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

Undefined behavior

- Not just dereferencing invalid pointers, division by zero, and signed overflow. Undefinedness permeates the standard.
- Type qualifiers like `const` and `restrict` are described by the standard in terms of the undefinedness they invoke.
- Undefinedness invalidates the entire execution on which it occurs—compilers assume programs are free of undefinedness when reordering code during optimization.
- All undefinedness should be considered a bug and a potential security risk.
- Our goal is to detect *strictly-conforming*, maximally-portable, undefinedness-free programs.

Undefined behavior

Two more kinds of behavior left up to the implementation:

- **Implementation-defined behavior.** Unspecified behavior where each implementation documents how the choice is made.
- **Unspecified behavior.** Use of an unspecified value, or other behavior [with] two or more possibilities and [...] no further requirements on which is chosen in any instance. [C11, §3.4]

Undefined behavior

Our classification	No.	CERT undef. behavior ids.
Early translation	24	#2, 3, 6, 7, 27–31, 34, 90–99, 101, 102, 104, 107.
lexical	11	
macros	13	
Core language	77	#4, 8–26, 32, 33, 35–89.
compile time	24	
link time	8	
run time	45	
Library	101	#5, 100, 103, 105, 106, 108–203.
compile time	18	
run time	83	
Other	1	#1. “Shall” or “shall not” violated.
Total	203	

Undefined behavior

Our classification	No.	CERT undef. behavior ids.
Early translation	24	#2, 3, 6, 7, 27–31, 34, 90–99,
lexical	11	101, 102, 104, 107.
macros	13	
Core language	77	#4, 8–26, 32, 33, 35–89.
compile time	24	
link time	8	
run time	45	
Library	101	#5, 100, 103, 105, 106,
compile time	18	108–203.
run time	83	
Other	1	#1. “Shall” or “shall not” violated.
Total	203	

Some examples

- Unsequenced side-effects.
- Strict aliasing.
- Type qualifiers.
- Invalid pointers.
- Pointer provenance.
- Translation phase.

Unsequenced side-effects.

This returns 3 in Clang and ICC, but 4 in GCC:

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


UB #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 [C11, §6.5].

Unsequenced side-effects.

This returns 3 in Clang and ICC, but 4 in GCC:

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

UB #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 [C11, §6.5].

 Locations written to between sequence points must be stored. Reads and writes must be checked against this set.

Strict aliasing


```
int main() {  
    int x = 0;  
    long *p = (long*)&x;  
    *p;  
}
```

UB #37. An object has its stored value accessed other than by an lvalue of an allowable type [C11, §6.5].

Strict aliasing

```
int main() {  
    int x = 0;  
    long *p = (long*)&x;  
    *p;  
}
```

UB #37. An object has its stored value accessed other than by an lvalue of an allowable type [C11, §6.5].

 Store the effective type of objects along with their object representation.

const qualifier

const can be dropped by casting:

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

UB #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 [C11, §6.7.3].

const qualifier

const can be dropped by casting:

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

UB #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 [C11, §6.7.3].



Store the effective type of objects along with their object representation.

Type qualifiers: `restrict`

These global-scope declarations

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

assert, according to the standard, that “if an object is accessed using one of `a`, `b`, or `c`, and that object is modified anywhere (at all) in the program, then it is not also accessed through the other two.” [C11, §6.7.3.1]

Type qualifiers: `restrict`

UB #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 [C11, §6.7.3.1].

UB #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 [C11, §6.7.3.1].

Invalid pointers


```
int main() {  
    int *p;  
    { int x = 0; p = &x; }  
    p;  
}
```

UB #10. The value of a pointer to an object whose lifetime has ended is used [C11, §6.2.4].

Invalid pointers

```
int main() {  
    int *p;  
    { int x = 0; p = &x; }  
    p;  
}
```

UB #10. The value of a pointer to an object whose lifetime has ended is used [C11, §6.2.4].

 Check that pointers are valid when they appear in an expression.

Pointer provenance

Inspired by Defect Report #260.

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

Pointer provenance

```
Provenance ::= fromArray(Int, Int)
             | basedOn(SymBase, BlockTag)
             | fromUnion(SymLoc, CId, FieldInfo)
             | align(Int)
```


Pointer provenance

```
Provenance ::= fromArray(Int, Int)
              | basedOn(SymBase, BlockTag)
              | fromUnion(SymLoc, CId, FieldInfo)
              | align(Int)
```

- ...tracks pointers based on arrays. Tagged when an array decays into a pointer to its first element.

Pointer provenance

```
Provenance ::= fromArray(Int, Int)
              | basedOn(SymBase, BlockTag)
              | fromUnion(SymLoc, CId, FieldInfo)
              | align(Int)
```

- ...tracks pointers based on arrays. Tagged when an array decays into a pointer to its first element.
- ...tracks pointers based on restrict-qualified pointers.

Pointer provenance

```
Provenance ::= fromArray(Int, Int)
              | basedOn(SymBase, BlockTag)
              | fromUnion(SymLoc, CId, FieldInfo)
              | align(Int)
```

- ...tracks pointers based on arrays. Tagged when an array decays into a pointer to its first element.
- ...tracks pointers based on `restrict`-qualified pointers.
- ...tracks union-field lvalues for marking the rest of the union unspecified.

Pointer provenance

```
Provenance ::= fromArray(Int, Int)
              | basedOn(SymBase, BlockTag)
              | fromUnion(SymLoc, CId, FieldInfo)
              | align(Int)
```

- ...tracks pointers based on arrays. Tagged when an array decays into a pointer to its first element.
- ...tracks pointers based on `restrict`-qualified pointers.
- ...tracks union-field lvalues for marking the rest of the union unspecified.
- ...tracks pointer alignment.

Translation phase

```
// Trans. unit 1.
```

```
int x;
```

```
int main() { }
```

```
// Trans. unit 2.
```

```
int x = 0;
```

UB #84. [...] there exist multiple external definitions for [an identifier with external linkage] [C11, §6.9].

```
// Trans. unit 1.
```

```
int f(int);
```

```
int main() {
```

```
    return f(1);
```

```
}
```

```
// Trans. unit 2.
```

```
int f(void) {
```

```
    return 1;
```

```
}
```

UB #41. A function is defined with a type that is not compatible with the type [of] the called function [C11, §6.5.2.2].

Translation phase

```
// Trans. unit 1.
```

```
int x;
```

```
int main() { }
```

```
// Trans. unit 2.
```

```
int x = 0;
```

UB #84. [...] there exist multiple external definitions for [an identifier with external linkage] [C11, §6.9].

```
// Trans. unit 1.
```

```
int f(int);
```

```
int main() {
```

```
    return f(1);
```

```
}
```

```
// Trans. unit 2.
```

```
int f(void) {
```

```
    return 1;
```

```
}
```

UB #41. A function is defined with a type that is not compatible with the type [of] the called function [C11, §6.5.2.2].



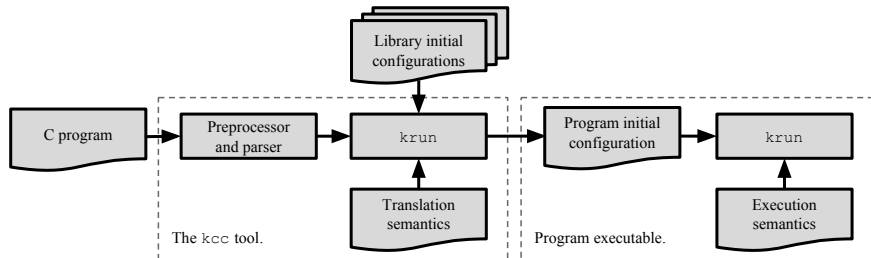
A translation phase!

Translation phase

$$\left\langle \begin{array}{l} \langle \textit{String} \rangle_{\text{tu-id}} \langle \textit{Map} \rangle_{\text{genv}} \langle \textit{Map} \rangle_{\text{gtypes}} \\ \langle \textit{Map} \rangle_{\text{local-statics}} \langle \textit{Map} \rangle_{\text{local-types}} \langle \textit{Map} \rangle_{\text{goto-map}} \end{array} \right\rangle_{\text{tu}}$$

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

The kcc tool



The kcc tool

```
int main() {  
    char a[9][9] = {0};  
    char *p = &a[0][0] + 9;  
    *p;  
}
```

```
$ kcc array.c
```

```
$ ./a.out
```

```
Error: CER4
```

```
Description: Dereferencing a pointer past the end of an array
```

```
Type: Undefined behavior.
```

```
See also: C11 sec. 6.5.6:8
```

```
at main(array.c:4)
```

```
at <file-scope>(<unknown>)
```

```
Error: Execution failed.
```

Our UB test suite

Undefined behavior	No. tests	Tools (% passed)					kcc
		Astrée	CompCert	Valgrind	V. Analysis	old 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

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
<i>all but</i> kcc	17	7	34	58
old kcc	10	2	23	35
kcc	24	8	45	77

Questions?

Check out our work at:

<https://github.com/kframework/c-semantics>