Cerberus: towards an Executable Semantics for Sequential and Concurrent C11

Kayvan Memarian, Kyndylan Nienhuis, Justus Matthiesen,
James Lingard, Peter Sewell

May 5, 2015

1 - What is C?

Different possible answers:

- what is described in the **standard**
  ⇒ ISO/IEC 9899-2011 + various defect reports
- various “de facto” definitions:
  - what compilers assume
  - what programmers believe
  - what different corpuses of C code actually require to run
- what analysis and verification tools assume (e.g. CompCertC)

2 - What we want

A formalisation of C11 which:

1. focuses on the various **“de-facto” Cs** (to study existing C codes)
2. is **parametric** (independence from impl. choices/interpretations)
3. supports **C/C++11 concurrency**
4. could be recognisable by those familiar with the C standard

3 - Why C is hard

C11 expressions hide a lot their complexity:

- loose and intricate ordering (**sequence-before relation**)
- hidden occurrence of memory operations (**boundary of object lifetime**)
- implicit type conversions (**usual arithmetic conv; integer promotions**)
- partiality (**undefined behaviour**)
- parametricity (**implementation-defined choices**)

- real code uses features outside of ISO C

Direct formalisation of a complex language leads to **heavy and quickly intractable semantic states**.

4 - Semantics by elaboration

$$\mathbf{C11} \xrightarrow{\textit{elaboration}} \mathbf{Core}$$

- the elaboration function *explains* each C11 construct
- Core has *simple constructs* (albeit quite a few)
- the elaboration produces verbose Core programs but with *explicit behaviour*

4 - Semantics by elaboration

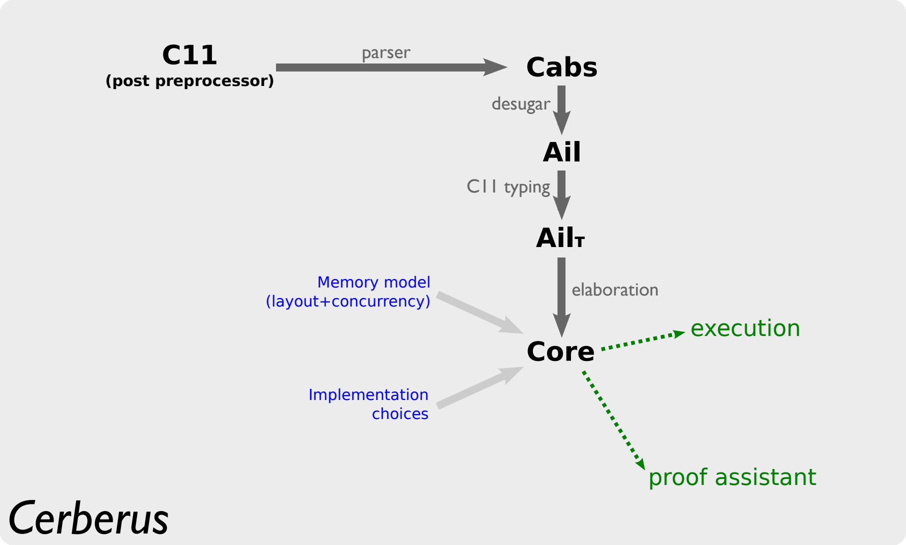
```c
int main(void) {
  int x = 0, y = 1;
  return x+y;
}
```

4 - Semantics by elaboration

```
proc main(): eff integer :=
  let strong (x, y) = unseq(create(<alignof>("signed int"), "signed int"),
                            create(<alignof>("signed int"), "signed int")) in
  let strong ()     = store("signed int", x, conv_int("signed int", 0)) in
  let strong ()     = store("signed int", y, conv_int("signed int", 1)) in

  let strong a3 =
    let weak (a1, a2) = unseq(load("signed int", x), load("signed int", y)) in
    overflow("signed int", conv_int("signed int", a1) + conv_int("signed int", a2))
  in
  let strong () = unseq(kill(x), kill(y)) in
  return (conv_int("signed int", a3))
```

5 - Cerberus' structure



6 - The Core language: *some features*

- functional style with first-order recursive functions
- simple discrimination between pure and effectful computations
- explicit memory actions (including object lifetime boundary)
- a calculus to express the *sequenced-before* relation:

  let weak (a_1,a_2) = **unseq**(E_1, E_2) **in**
  store(τ, `ptr`, a_1 + a_2)

- C types and implementation-defined constants as values:

  `<Integer.conv_nonrepresentable_signed_integer>("signed int", n)`

- explicit undefined behaviours and dynamic-runtime errors:

  if a_2 = 0 **then undef**(`Division_by_zero`) **else** a_1 / a_2

- others: continuation operators, non-determinism, I/O, ...

6 - The Core language (1/2)

values:

v ::= **unit** | **null** | **true** | **false** | $[v_1, \ldots, v_n]$ | $(v_1, \ldots, v_n)$
| τ | **unspec**(τ) | $n \in \mathbb{Z}$ | *<ptr>* | *<array-const>*
| *<struct-const>* | *<union-const>*

pure expressions:

$\mathtt{pe}$ ::= **undef** | **error** | v | a | *<impl-name>* | **cons**($\mathtt{pe}_1$, $\mathtt{pe}_2$)
| **case_list**($\mathtt{pe}_1$, $\mathtt{pe}_2$, $\mathtt{nm}$) | **case_ctype**($\mathtt{pe}_1$, $\mathtt{pe}_2$, $\mathtt{nm}_1, \ldots, \mathtt{nm}_8$)
| **shift**($\mathtt{pe}$, *<shift-path>*) | **not**($\mathtt{pe}$) | $\mathtt{pe}_1 \circ \mathtt{pe}_2$
| *<mem-op>*($\mathtt{pe}_1 \ldots \mathtt{pe}_n$) | $(\mathtt{pe}_1, \ldots, \mathtt{pe}_n)$ | $\mathtt{nm}(\mathtt{pe}_1 \ldots \mathtt{pe}_n)$
| **let** $a = \mathtt{pe}_1$ **in** $\mathtt{pe}_2$ | **if** $\mathtt{pe}_1$ **then** $\mathtt{pe}_2$ **else** $\mathtt{pe}_3$

nm ::= SYMBOL | ‘‘impl-def function names’’

<shift-path> ::= $\{\ (\tau_1, \mathtt{pe}_1), \ldots, (\tau_n, \mathtt{pe}_n)\ \}$

6 - The Core language (2/2)

E `::=` pe | **let** $a = \mathtt{pe}_1$ **in** E_2 | **if** $\mathtt{pe}_1$ **then** E_2 **else** E_3 | $\mathtt{nm}\{\mathtt{pe}_1, \ldots, \mathtt{pe}_n\}$
| **nd**$(E_1, \ldots, E_n)$ | **return**$(\mathtt{pe})$
| A | $\overline{A}$ | **skip**
| **unseq**$(E_1, \ldots, E_n)$ | **let weak** $(a_1, \ldots, a_n)$ = E_1 **in** E_2
| **let strong** $(a_1, \ldots, a_n)$ = E_1 **in** E_2 | **let atomic** a = A_1 **in** P
| **indet**$_i(E)$ | **bound**$_i(E)$
| **save** $\delta(a_1 : \tau_1, \ldots, a_n : \tau_n)$ **in** E | **run** $\delta(a_1 : E_1, \ldots, a_n : E_n)$
| **par**$(E_1, \ldots, E_n)$ | **wait**(<*tid*>)
| **raise**(*event-name*) | **register**(*event-name*, $\mathtt{nm}$)

memory actions:

A `::=` **create**$(\mathtt{pe}_\tau, \mathtt{pe}_{\mathtt{align}})$ | **alloc**$(\mathtt{pe}_n)$ | **kill**$(\mathtt{pe}_{\mathtt{ptr}})$
| **store**$(\mathtt{pe}_\tau, \mathtt{pe}_{\mathtt{ptr}}, \mathtt{pe}_n)$ | **load**$(\mathtt{pe}_\tau, \mathtt{pe}_{\mathtt{ptr}})$ | $\cdots$

6 - The Core language: *design motivation*

Very different from C11

Designed for the purpose of expressing the behaviour of C11 programs
⇒ each Core construct is motivated by a C11 behaviour

Suitable for usual semantics techniques

- each construct has only one distinct behaviour
- functional style
- strongly typed
- small-step with continuations stack

6 - The Core language

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

6 - The Core language

```
proc main(): eff integer :=
  let strong (x, y) = unseq(create(<alignof>("signed int"), "signed int"),
                            create(<alignof>("signed int"), "signed int")) in
  let strong ()     = store("signed int", x, conv_int("signed int", 0)) in
  let strong ()     = store("signed int", y, conv_int("signed int", 1)) in

  let strong a3 =
    let weak (a1, a2) = unseq(load("signed int", x), load("signed int", y)) in
    overflow("signed int", conv_int("signed int", a1) + conv_int("signed int", a2))
  in
  let strong () = unseq(kill(x), kill(y)) in
  return (conv_int("signed int", a3))
```

6 - The Core language

```c
int f(int n) {
  n+1;
}
```

```c
int main(void) {
  int x = 0, y = 1;
  return f(x)+y;
}
```

6 - The Core language

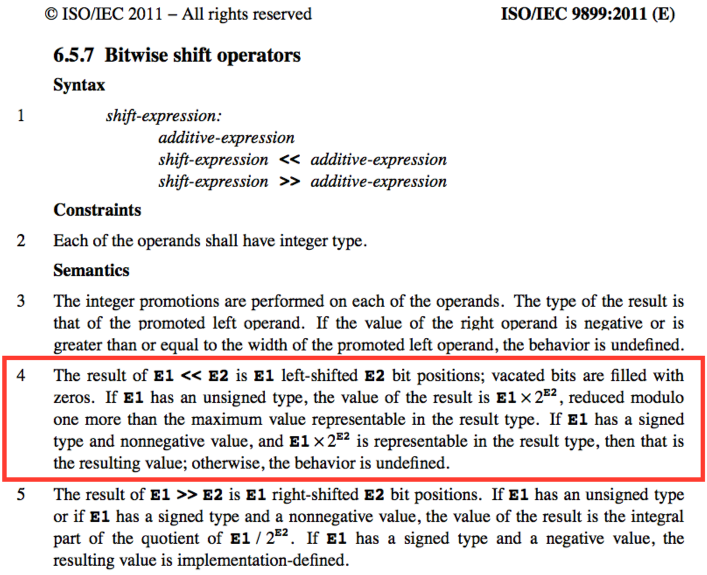
```
proc f(n: pointer): eff integer :=
  let strong a1 =
    let weak (a2, a3) = unseq(load("signed int", n), 1) in
    overflow("signed int", conv_int("signed int", a2) + conv_int("signed int", a3))
  in
  undef("Reached_end_of_function")


proc main(): eff integer :=
  let strong (x, y) = unseq(create(<alignof>("signed int"), "signed int"),
                            create(<alignof>("signed int"), "signed int")) in
  let strong () =
    unseq(store("signed int", y, conv_int("signed int", 1)),
          store("signed int", x, conv_int("signed int", 0))) in

  let strong a1 =
    let weak (a2, a3) = unseq(
      let strong ptr1 = create(<alignof>("signed int"), "signed int") in
      let strong tmp1 = load("signed int", x) in
      let strong ()   = store("signed int", ptr1, tmp1) in
      let strong ret1 = f{ptr1} in
      let strong ()   = kill(ptr1) in
      ret1
      ,
      load("signed int", y)
    ) in
    overflow("signed int", conv_int("signed int", a2) + conv_int("signed int", a3))
  in

  let strong () = unseq(kill(x), kill(y)) in
  return (conv_int("signed int", a1))
```

7 - Elaboration: the left shift operator ($E_1 << E_2$)



© ISO/IEC 2011 – All rights reserved

ISO/IEC 9899:2011 (E)

6.5.7 Bitwise shift operators

Syntax

1 *shift-expression:*
additive-expression
shift-expression **<<** *additive-expression*
shift-expression **>>** *additive-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 The result of **`E1 << E2`** is **`E1`** left-shifted **`E2`** bit positions; vacated bits are filled with zeros. If **`E1`** has an unsigned type, the value of the result is $\mathtt{E1} \times 2^{\mathtt{E2}}$, reduced modulo one more than the maximum value representable in the result type. If **`E1`** has a signed type and nonnegative value, and $\mathtt{E1} \times 2^{\mathtt{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 The result of **`E1 >> E2`** is **`E1`** right-shifted **`E2`** bit positions. If **`E1`** has an unsigned type or if **`E1`** has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $\mathtt{E1} / 2^{\mathtt{E2}}$. If **`E1`** has a signed type and a negative value, the resulting value is implementation-defined.

7 - Elaboration: the left shift operator $(E_1 << E_2)$

$[\![({E_1}^{\mathrm{ty}_1} << {E_2}^{\mathrm{ty}_2})^{\mathrm{ty}}]\!] \rightsquigarrow$

```
let weak ($a_1$, $a_2$) = unseq($[\![E_1]\!]$, $[\![E_2]\!]$) in
let $a_1$ = promote(ty$_1$, ty, $a_1$) in
let $a_2$ = promote(ty$_2$, ty, $a_2$) in
if $a_2$ < 0 then
  undef(Negative_shift)
else if ctype_width(ty) <= $a_2$ then
  undef(Shift_too_large)
else
  let res = ($a_1$ * 2^$a_2$) % (<ctype_max>(ty) + 1) in
IF AilTypesAux.is_unsigned_integer_type(ty1) THEN
  res
ELSE
  if is_representable(res, ty) then
    res
  else
    undef(Non_representable_shift)
```

8 - Memory layout model

The elaboration of C into Core is parametric on semantic choices regarding the memory.

(back to the second slide)

- focuses on the various **"de-facto" Cs** (to study existing C codes)

When it comes to memory related issues, it is common for (system-level) C programs to go outside of ISO C.

⇒ requires an empirical approach:

- we need to learn the assumptions made by programmers and compilers

8 - Memory layout model (litmus tests)

Can one inspect the value of a pointer to an object whose lifetime has ended?

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int i=0;
  int *pj = (int *)(malloc(sizeof(int)));
  *pj=1;
  printf("(&i==pj)=%s\n", (&i==pj)?"true":"false");
  free(pj);
  printf("(&i==pj)=%s\n", (&i==pj)?"true":"false");
  // is the == comparison above defined behaviour?
}
```

Undefined in ISO C

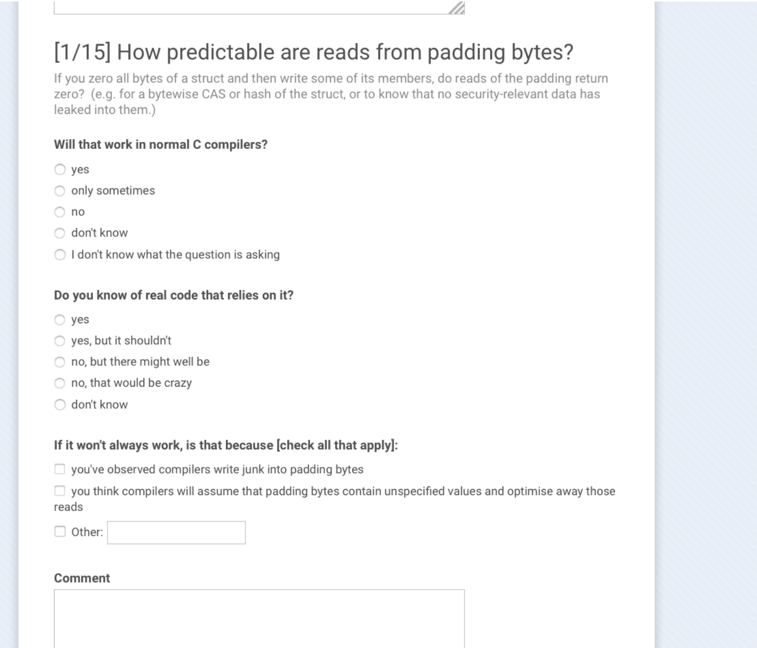
8 - Memory layout model (litmus tests)

Can one do relational comparison (with $<, >, \ldots$) of two pointers to separately allocated objects?

```
#include <stdio.h>
int y=2, x=1;
int main(void) {
  int *p=&x *p=&y;
  _Bool b1 = (p < q); // does this have defined behaviour?
  _Bool b2 = (p > q); // does this have defined behaviour?
  printf("(p<q = %s  (p>q) = %s\n",
         b1?"true":"false", b2?"true":"false");
}
```

Undefined in ISO C

8 - Memory layout model (web survey)



[1/15] How predictable are reads from padding bytes?

If you zero all bytes of a struct and then write some of its members, do reads of the padding return zero? (e.g. for a bytewise CAS or hash of the struct, or to know that no security-relevant data has leaked into them.)

Will that work in normal C compilers?

○ yes
○ only sometimes
○ no
○ don't know
○ I don't know what the question is asking

Do you know of real code that relies on it?

○ yes
○ yes, but it shouldn't
○ no, but there might well be
○ no, that would be crazy
○ don't know

If it won't always work, is that because [check all that apply]:

☐ you've observed compilers write junk into padding bytes
☐ you think compilers will assume that padding bytes contain unspecified values and optimise away those reads
☐ Other:

Comment

9 - C/C++11 concurrency

work of Kyndylan Nienhuis.

⇒ integration with an **operational model** for C/C++11 concurrency.

- enable the incremental exploration of larger concurrent programs
- formally proved equivalent to Batty's C11 axiomatic formalisation (in Isabelle/HOL)

parametricity of Core's dynamics on the memory helped:

- virtually no modification required on the concurrency model
- but relaxed reads required symbolic evaluation

Conclusion