Cerberus, and the Memory Object Semantics for ISO and De Facto C

Kayvan Memarian Victor B F Gomes Peter Sewell

University of Cambridge

http://www.cl.cam.ac.uk/~pes20/cerberus/

EuroLLVM, Bristol, 16 April 2018

C and C++

Over 45 years old:

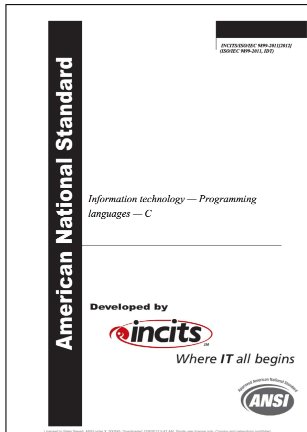
C	1972	K&R	ANSI C/C90	C99	C11			C2x
C++	1985		C++2.0 (89)		C++11	C++14	C++17	C++20

but still ubiquitous, especially for systems and embedded code

survive all attempts to kill them off

Notionally, defined by ISO standards

contracts between compiler writers and programmers, from WG14 and WG21 committees:



"The Committee's overall goal was to develop a **clear, consistent, and unambiguous** *Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments."*

Actually?

ISO C11 is 702 pages *of prose*. C++17 is 1622

Actually?

ISO C11 is 702 pages *of prose*. C++17 is 1622

- inevitably imprecise, incomplete, and ambiguous

Actually?

ISO C11 is 702 pages *of prose*. C++17 is 1622

- inevitably imprecise, incomplete, and ambiguous
- not executable in any form
  - can't automatically compute the set of all allowed behaviour of examples – and whether there is undefined behaviour
  - can't use as a test oracle, for compiler, analysis, or sanitiser testing

Actually?

ISO C11 is 702 pages *of prose*. C++17 is 1622

- inevitably imprecise, incomplete, and ambiguous
- not executable in any form
  - can't automatically compute the set of all allowed behaviour of examples – and whether there is undefined behaviour
  - can't use as a test oracle, for compiler, analysis, or sanitiser testing
- not just prose, but pretty subtle prose – a source of endless debate and confusion, even for experts

Actually?

ISO C11 is 702 pages *of prose*. C++17 is 1622

- inevitably imprecise, incomplete, and ambiguous
- not executable in any form
  - can't automatically compute the set of all allowed behaviour of examples – and whether there is undefined behaviour
  - can't use as a test oracle, for compiler, analysis, or sanitiser testing
- not just prose, but pretty subtle prose – a source of endless debate and confusion, even for experts
- some real disagreements between the standards and *de facto* usage/implementation. Major projects rely on semantics not specified by ISO, both explicitly, eg `-fno-strict-aliasing`, and implicitly, e.g. flat-address-space assumptions

Actually?

a “**clear, consistent, and unambiguous** Standard”?

Actually?

a "**clear, consistent, and unambiguous** Standard"?

no, there are many C's:

- ISO C
- ...modulo interpretation and debate

- each compiler implementation
- ...with various compiler flag choices

- assumptions implicit in codebase

- programmer beliefs

Previously...

Helped clarify C/C++11 concurrency:

- engaged with WG21 and WG14, clarifying proposed model with mathematical definitions, and finding and fixing flaws
- final C/C++11 text in tight correspondence with mathematical model
- proofs of correctness of concurrency mapping w.r.t. hardware models (x86, IBM POWER, ARM)
- `cppmem` exploration tool
- Batty, Owens, Sarkar, Memarian, Sewell – with C++0x concurrency working group: Boehm, Adve, McKenney, Crowl, Nelson, Wong, etc.

[caveats: see later fixes by Batty, Lahav, etc., and the thin-air problem remains – see Batty et al., Pichon-Pharabod, and Hur et al.]

What about the rest of C?

Cerberus

- rigorous semantics for a large fragment of C
- executable as test oracle (for small tests) with web interface
- closely follows ISO where that is clear and uncontroversial
- NB: a semantic reference, not an analysis tool/sanitiser

Investigation of *Memory object semantics*

- areas where ISO is unclear and/or differs with practice
  ...pointer provenance, uninitialised values, etc.
- identifying questions, making examples, surveying experts (EuroLLVM 2015?), implementing candidate models in Cerberus, engaging with WG14

Caveat: all work in progress!

Goal

make C (and C++?)

- have a clear unambiguous definition
- ...that's usable as a test oracle
- ...and supports the language that systems code – OS kernels etc – actually uses (with options, attributes, etc. as needed)

- without unduly constraining compiler optimisation
- ...and with clear relationship to compiler intermediate languages ...c.f. Lopes, Hur, Regehr et al.

- provide smooth migration path to safer C, with fewer UB gotchas, where desired

...for today/tomorrow:

get a clear picture of what you think the source-language semantics for some of these issues is / should be

...see if plausible consensus

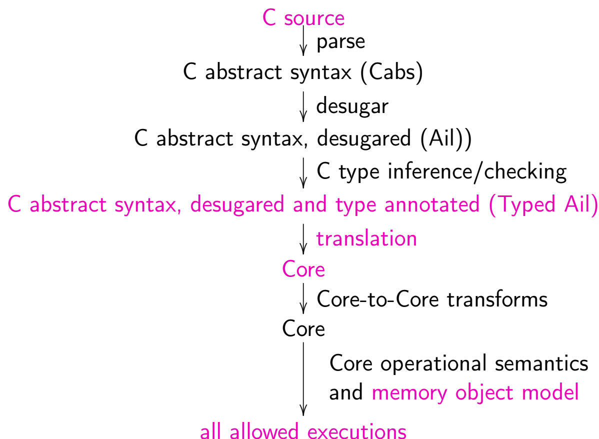
off to WG14 again next week

Cerberus

How to express C semantics cleanly?

- Translate into a purpose-built simpler language, Core
- Define Core with an operational-semantics interpreter combined with a memory object model

Cerberus pipeline:



One clause of the C↦Core translation: left-shift e_1 << e_2

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 **E1**$\times 2^{\mathbf{E2}}$, reduced modulo one more than the maximum value representable in the result type. If **E1** has a signed type and nonnegative value, and **E1**$\times 2^{\mathbf{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 ... *similarly for* **E1 >> E2** ...

```
⟦ e1<< e2⟧ =
  sym_e1 := E.fresh_symbol;  sym_e2 := E.fresh_symbol;
  sym_res := E.fresh_symbol;
  core_e1 := ⟦e1⟧;
  core_e2 := ⟦e2⟧;
  E.return (
    let weak (sym_e1, sym_e2) = [core_e1 || core_e2] in
      pure (
        if is_unspec_value(sym_e2) then
          undef Exceptional_condition else
        if is_unspec_value(sym_e1) then
          V_unspec result_ty else
        if sym_e2 < 0 then
          undef Negative_shift else
        if (ctype_width [result_ty] < sym_e2) ∨
           (ctype_width [result_ty] = sym_e2) then
          undef Shift_too_large else
        IF is_unsigned_integer_type (ctype_of e1) THEN
        (sym_e1*(2^sym_e2))%(ivmax(result_ty)+1)
        ELSE
        if sym_e2 < 0 then
          undef Negative_left_shift else
        let sym_res = sym_e1*(2^sym_e2) in
        if is_representable [sym_res; result_ty] then
          sym_res else
          undef Negative_left_shift))
```

Demo 1

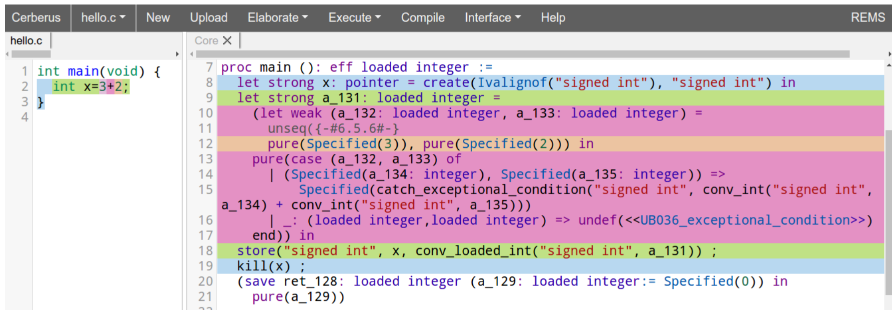
```
int main(void) {
  int x=3+2;
}
```

looks simple – but already a lot going on in C: lifetime of `x`, unsequenced evaluation of arguments to `+`, integer conversions, UB-on-overflow

Demo 1

```
int main(void) {
  int x=3+2;
}
```

looks simple – but already a lot going on in C: lifetime of x, unsequenced evaluation of arguments to +, integer conversions, UB-on-overflow



Demo 2

Cerberus defines, and can identify, all sources of undefined behaviour (for small enough examples):

```
int f(int n) {
  return 20 << n;
}

int g(void){
  int ret = 3;
  int i = 4;
  while (i--) ret--;
  return ret;
}

int main() {
  f(g());
}
```

Exhaustive execution: { Undefined UB051a_negative_shift }

Demo 3

How can we explore all allowed executions without trying all possible allocations?

Optionally, use symbolic allocation, accumulating constraints and using Z3 to solve them

```
#include <stdint.h>
int main(void) {
  int x,y;
  return (uintptr_t)&x < (uintptr_t)&y;
}
```

Exhaustive execution: { Defined Specified(0), Defined Specified(1) }

(or, use concrete model, with specific allocator – faster)

Demo 4

```
#include <stdint.h>
int main(void) {
  int x,y,z;
  if ((uintptr_t)&x < (uintptr_t)&y && (uintptr_t)&y < (uintptr_t)&z)
    return (uintptr_t)&x < (uintptr_t)&z;
  else
    return 22;
}
```

Exhaustive execution: { Defined Specified(1), Defined Specified(22) }

Pointer Provenance

What's a pointer value?

A simple numeric address?

What's a pointer value?

A simple numeric address?

At runtime, normally yes. But WG14 DR260 CR (2001):

> *"Implementations are permitted to track the origins of a bit-pattern [...] They may also treat pointers based on different origins as distinct even though they are bitwise identical."*

What's a pointer value?

A simple numeric address?

At runtime, normally yes. But WG14 DR260 CR (2001):

> *"Implementations are permitted to track the origins of a bit-pattern [...] They may also treat pointers based on different origins as distinct even though they are bitwise identical."*

Exploited in practice by some compilers: alias analysis uses provenance to reason that two pointers are distinct (even if they might have the same runtime numeric address), with optimisations assuming that is correct.

What's a pointer value?

A simple numeric address?

At runtime, normally yes. But WG14 DR260 CR (2001):

> *"Implementations are permitted to track the origins of a bit-pattern [...] They may also treat pointers based on different origins as distinct even though they are bitwise identical."*

Exploited in practice by some compilers: alias analysis uses provenance to reason that two pointers are distinct (even if they might have the same runtime numeric address), with optimisations assuming that is correct.

From ISO p.o.v.: using a pointer "originally" to one object to access another object (after some pointer arithmetic) is UB, so this is ok.

What's a pointer value?

A simple numeric address?

At runtime, normally yes. But WG14 DR260 CR (2001):

> *"Implementations are permitted to track the origins of a bit-pattern [...] They may also treat pointers based on different origins as distinct even though they are bitwise identical."*

Exploited in practice by some compilers: alias analysis uses provenance to reason that two pointers are distinct (even if they might have the same runtime numeric address), with optimisations assuming that is correct.

From ISO p.o.v.: using a pointer "originally" to one object to access another object (after some pointer arithmetic) is UB, so this is ok.

But what does DR260CR really mean? It was never incorporated into the standard, and there are lots of choices – which determine what code is legal, and what alias analysis & optimisation is allowed to do.

Provenance: Proposed semantics

Associate a *provenance* to every pointer and integer value:

- a single provenance ID, freshly chosen at each allocation
- the "empty" provenance

(in the C abstract machine, not at runtime in normal impls!)

Check provenance on all accesses, with UB otherwise:

- access via a pointer value with a single provenance ID must be within the memory footprint of the corresponding original allocation
- access via a pointer value with empty provenance is undefined behaviour (except device memory)
- ...plus escape hatches, TBD, for exotic code

Now take care which operations preserve provenance....

Q1′. Must the pointer used for a memory access have the right provenance, i.e. be derived from the pointer to the original allocation (UB otherwise)? Yes

Example `provenance_basic_global_xy.c`:

```
int x=1, y=2;
...
int *p = &x + 1;
int *q = &y;
if (memcmp(&p, &q, sizeof(p)) == 0) {
  *p = 11;  // does this have undefined behaviour?
  printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
}
```

(NB these are tests of tricky edge-cases – not necessarily sensible code!)

`p` has the provenance of the `x` allocation, but is used to access memory outside that, so the access gives UB

...licensing alias analysis and optimisations that assume `p` and `q` don't alias ✓

Q3 Can one make a usable pointer via casts to `intptr_t` and back? Yes

Example `provenance_roundtrip_via_intptr_t.c`

```
int  x=1;
...
int *p     = &x;
intptr_t i = (intptr_t)p;
int *q     = (int *)i;
*q = 11; // is this free of undefined behaviour?
```

So make casts between pointer and integer values preserve provenance ✓

Q9′ Can one make a usable pointer to one allocation from a pointer to another allocation by arithmetic? No

Example `pointer_offset_from_subtraction_1_global.c`

```
int  y = 2, x=1;
...
intptr_t ix = (intptr_t)&x;
intptr_t iy = (intptr_t)&y;
intptr_t offset = iy - ix;

int *p = (int *)(ix + offset);
int *q = &y;
if (memcmp(&p, &q, sizeof(p)) == 0) {
  *p = 11; // is this free of undefined behaviour?
}
```

To forbid this, make binary subtraction of mixed-provenance values have empty-provenance result. Then `p` has provenance of `x` but address of `y`, and `*p=11` is UB.✓

This also forbids XOR linked-list idiom – do we care?
Systems code might need inter-object arithmetic, rarely – add escape hatch?

Q6 Can one use bit manipulation and integer casts to store information in unused bits of pointers? Yes

Example `provenance_tag_bits_via_uintptr_t_1.c`

```
int x=1;
int *p = &x;
uintptr_t i = (uintptr_t) p;    // cast &x to an integer
// check the bottom two bits of an int* are not used
assert(_Alignof(int) >= 4);
assert((i & 3u) == 0u);
// construct an integer like &x with low-order bit set
i = i | 1u;
// cast back to a pointer
int *q = (int *) i; // defined behaviour?
// cast to integer and mask out the low-order two bits
uintptr_t j = ((uintptr_t)q) & ~((uintptr_t)3u);
// cast back to a pointer
int *r = (int *) j;
*r = 11;           //  defined behaviour?
```

Systems code relies on this, but ISO leaves it unclear.

To support it, define those operations so provenance is preserved here, and make set of unused bits implementation-defined ✓

Q14 Can one make a usable copy of a pointer by copying its representation bytes (unchanged) in user code? Yes

Example `pointer_copy_user_dataflow_direct_bytewise.c`

```
int  x=1;
void user_memcpy(unsigned char* dest, unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = *src;
    src += 1;
    dest += 1;
    n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;
  user_memcpy((unsigned char*)&q, (unsigned char*)&p, sizeof(p));
  *q = 11; // is this free of undefined behaviour?
}
```

Our proposal makes it legal: each representation byte (as an integer value) has the provenance of the original pointer, and the result pointer, being composed of representation bytes all with that provenance, does too

Can one make a usable copy of a pointer by a non-bytewise copy, e.g. via encryption and decryption? No

...need escape hatch, TBD

Q19 Can one make a usable pointer via IO? Yes

This is used in practice: in graphics code for serialisation/unserialisation, using `%p`, in xlib, using `SCNuPTR`, and in debuggers.

Either use escape hatch or record (in abstract machine) the escaped pointers and find the right provenance on reconstruction.

Can `intptr_t` arithmetic be used to mimic pointer arithmetic? Debatable

Example adapted from Martin Sebor mail (WG14):

```
int a[2];
int *p = &a[0];
uintptr_t i = (uintptr_t)p;
uintptr_t j = i + sizeof (int);
int *q = (int *)j;        // defined behaviour?
*q=11;                    // defined behaviour?
```

Our proposal allows this if the implementation-defined conversions are reasonable. Should it be forbidden?

By recording the original address at pointer-to-integer casts, not just the provenance?

Are provenance checks only on a per-allocation granularity, or per-subobject?

Per-allocation?

```
typedef struct { int x; int y; } st;
st s;
...
int *p = &s.x + 1;
int *q = &s.y;
if (memcmp(&p, &q, sizeof(p)) == 0) {
  *p = 11;  // does this have undefined behaviour?
}
```

A priori, we don't care whether this example is allowed or not.

Our proposal allows it – and to forbid it would need a more complex provenance semantics.

Provenance Summary

More questions, examples, and a proposed C standard diff in:

Clarifying Pointer Provenance v4 (a revision of WG14 N2219)
Kayvan Memarian, Victor Gomes, Peter Sewell. University of Cambridge
2018-05-04
`http://www.cl.cam.ac.uk/~pes20/cerberus/clarifying-provenance-v4.html`

A few things wrong there (esp. the proposed diff), and some others subject to debate – but hopefully basically uncontroversial.

Uninitialised Values

Uninitialised reads: Survey data

C semantics web survey, 2015

15 questions

323 responses

aiming for experts, not random C hackers

Uninitialised reads: Survey data

Is reading an uninitialised variable or struct member (with a current mainstream compiler):

a) undefined behaviour (meaning that the compiler is free to arbitrarily miscompile the program, with or without a warning)
b) going to make the result of any expression involving that value unpredictable
c) going to give an arbitrary and unstable value (maybe with a different value if you read again)
d) going to give an arbitrary but stable value (with the same value if you read again)

Uninitialised reads: Survey data

Is reading an uninitialised variable or struct member (with a current mainstream compiler):

a) undefined behaviour (meaning that the compiler is free to arbitrarily miscompile the program, with or without a warning) : 139 (43%)
b) going to make the result of any expression involving that value unpredictable : 42 (13%)
c) going to give an arbitrary and unstable value (maybe with a different value if you read again) : 21 (6%)
d) going to give an arbitrary but stable value (with the same value if you read again) : 112 (35%)

Uninitialised reads in practice

- consenus: entirely-uninitialised reads are almost always programmer errors – except integers used for incrementally initialised bits?
- current compilers do implicitly exploit current UB, giving unstable results for repeated reads
- ...but it's not necessary (for compilation) or useful (for programmers) to have arbitrary behaviour in those cases. Even unstable results are confusing for debugging

Uninitialised reads in practice

- consenus: entirely-uninitialised reads are almost always programmer errors – except integers used for incrementally initialised bits?
- current compilers do implicitly exploit current UB, giving unstable results for repeated reads
- ...but it's not necessary (for compilation) or useful (for programmers) to have arbitrary behaviour in those cases. Even unstable results are confusing for debugging

C/C++ notion of UB is too crude – it conflates:

- programmer errors that should be compile-time detected where possible
- cases where the behaviour of a conventional implementation can't be sensibly constrained, e.g. wild writes
- cases where for optimisation / portability we want to assume the programmer doesn't do something – but actually we could bound the potential bad behaviour

Uninitialised reads in ISO text

Confused

Uninitialised reads in ISO text

Confused

Trap representations are particular object representations that do not represent values of the object type, for which merely reading them (except at character type), is UB (3.19.4, 6.2.6.1p5, 6.2.6.2p2, DR338).

- might but does not have to trap
- but common impls don't *have* unused representations at most types
- only exception(?): `_Bool`

Uninitialised reads in ISO text

Confused

Trap representations are particular object representations that do not represent values of the object type, for which merely reading them (except at character type), is UB (3.19.4, 6.2.6.1p5, 6.2.6.2p2, DR338).

- might but does not have to trap
- but common impls don't *have* unused representations at most types
- only exception(?): `_Bool`

For types without trap representations, uninitialised reads are UB iff *"the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken)"*

- confused attempt to handle Itanium NaT (Not-a-Thing) behaviour?

Uninitialised reads in ISO text

Confused

Trap representations are particular object representations that do not represent values of the object type, for which merely reading them (except at character type), is UB (3.19.4, 6.2.6.1p5, 6.2.6.2p2, DR338).

- might but does not have to trap
- but common impls don't *have* unused representations at most types
- only exception(?): `_Bool`

For types without trap representations, uninitialised reads are UB iff *"the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken)"*

- confused attempt to handle Itanium NaT (Not-a-Thing) behaviour?

some of WG14 report their intent was to make uninitialised reads at non-character-type always UB

Uninitialised reads in C2x?

- Non-struct uninitialised reads are essentially always programmer errors – except integers used for incrementally initialised bits?
- Where a compiler can statically detect them, it should

Uninitialised reads in C2x?

- Non-struct uninitialised reads are essentially always programmer errors – except integers used for incrementally initialised bits?
- Where a compiler can statically detect them, it should

Then where it can't, which semantics is most useful?

a) UB
b) the result of any expression involving that value becomes unpredictable
c) an arbitrary value (maybe different if you read again)
d) an arbitrary but stable value (with the same value if you read again) – the actual value from memory
e) zero, for static/thread/automatic; from memory, for allocated
f) zero

Uninitialised reads in C2x?

- Non-struct uninitialised reads are essentially always programmer errors – except integers used for incrementally initialised bits?
- Where a compiler can statically detect them, it should

Then where it can't, which semantics is most useful?

a) UB
b) the result of any expression involving that value becomes unpredictable
c) an arbitrary value (maybe different if you read again)
d) an arbitrary but stable value (with the same value if you read again) – the actual value from memory
e) zero, for static/thread/automatic; from memory, for allocated
f) zero

In N2221 we proposed (b) everywhere
But now?

Q55 Can a structure containing an unspecified-value member can be copied as a whole? Yes?

Example unspecified_value_struct_copy.c

```
typedef struct { int i1; int i2; } st;
int main() {
  st s1;
  s1.i1 = 1;
  st s2;
  s2 = s1; // should this have defined behaviour?
  printf("s2.i1=%i\n",s2.i1);
}
```

Padding

Padding

Usually, code shouldn't care about the contents of padding

But sometimes it matters:

- to prevent leakage of security-relevant information
- for bytewise serialisation, encryption, and suchlike

Then what semantics should we give?

- Can structure-copy copy padding? Y
- After an explicit write of a padding byte, does that byte hold a well-defined value? (not an unspecified value) Y
- After an explicit write of a padding byte followed by a write to the whole structure, does the padding byte hold a well-defined value? (not an unspecified value) N
- After an explicit write of a padding byte followed by a write to adjacent members of the structure, does the padding byte hold a well-defined value? (not an unspecified value) N (or only one side?)
- After an explicit write of zero to a padding byte followed by a write to adjacent members of the structure, does the padding byte hold a well-defined zero value? (not an unspecified value) N
- After an explicit write of a padding byte followed by a write to a non-adjacent member of the whole structure, does the padding byte hold a well-defined value? (not an unspecified value) Y
- After an explicit write of a padding byte followed by a writes to adjacent members of the whole structure, but accessed via pointers to the members rather than via the structure, does the padding byte hold a well-defined value? (not an unspecified value) Y
- Can the user make a copy of a structure or union by copying just the representation bytes of its members and writing junk into the padding bytes? Y

Further Pointer Issues

Q31 Can one construct out-of-bounds (by more than one) pointer values by pointer arithmetic (without undefined behaviour)? Yes?

Example `cheri_03_ii.c`

```
int x[2];
int *p = &x[0];
int *q = p + 11;  // is this free of undefined behaviour?
q = q - 10;
*q = 1;           // and this?
```

ISO: no. Can imagine cases where it would go wrong: hardware bounds checking, or where intermediate unaligned value isn't representation, or near the top of memory, or where pointer subtraction overflows.

But real code seems to rely on it (Chisnall et al., 2015).

Can one identify reasonable cases in which it can be guaranteed to work?

Conclusion

For many years most semantics researchers gave up C as a lost cause (except a few: Norrish, Ellison et al., Krebbers, etc.)

But it's an inescapable part of our infrastructure

...and the current state leaves much to be desired, especially for memory object model issues and from a rigorous semantics pov.

It is possible to improve: both the state of the definition, and the language itself.

Hope to establish some consensus as to what that should be

...in a way that can be precisely related to compiler behaviour and internal languages.