C memory object and value semantics: the space of de facto and ISO standards

[This revises and extends WG14 N2013]

Revision: 1571 2016-03-17

David Chisnall Justus Matthiesen Kayvan Memarian Peter Sewell Robert N. M. Watson
University of Cambridge
http://www.cl.cam.ac.uk/~pes20/cerberus/

Contents

1. Introduction

In this note we discuss the semantics of memory in C, focussing on the non-concurrent aspects: the semantics of pointers, casts, effective types, unspecified values, and so on. These make up what we call the *memory object model*, to distinguish it from the *memory concurrency model* that addresses the relaxed-memory semantics of C; the two are largely but not completely orthogonal, and together they give a complete semantics of C memory. This is a part of our larger Cerberus C semantics project.

We are concerned principally with the de facto standards of C as it is used in practice: the existing usage of C, especially in systems code, and the behaviour of the dominant compiler implementations and the idioms they support. We also discuss C as specified in the ISO C11 standard. The ISO and de facto standards can differ in important ways, and in reality neither of them are singular: the C11 standard is prose text, open to interpretation, and there are multiple distinct de facto standards in different contexts (some specific to particular compilers or compiler flags). We are developing a formal model intended to capture one reasonable view of the de facto standards, though, given the real conflicts seen between different views, this is intended only as a precise reference point for discussion; no single model can currently be acceptable to all parts of the C community. We may later equip it with switches to express particular views of de facto and/or ISO standards. We also discuss the intended behaviour of CHERI C [14], with its hardware support for capabilities [55, 56].

In the longer term, this analysis may be helpful to understand what a well-designed language for systems programming would have to support.

One can look at the de facto semantics from several different perspectives:

1. the languages implemented by mainstream compilers (GCC, Clang, ICC, MSVC, etc.), including the assumptions their optimisation passes make about user code and how these change with certain flags (e.g. GCC's `-fno-strict-aliasing` and `-fno-strict-overflow`);
2. the idioms used in the corpus of mainstream systems code out there, especially in specific large-scale systems (Linux, FreeBSD, Xen, Apache, etc.);
3. the language that systems programmers believe they are writing in, i.e., the assumptions they make about what behaviour they can rely on;
4. the issues that arise in making C code portable between different compilers and architectures; and
5. the behaviour assumed, implicitly or explicitly, by code analysis tools.

We focus throughout on current mainstream C implementations: commonly used compilers and hardware platforms. One could instead consider the set of all current or historical C implementations, or even all conceivable implementations, but that (apart from being even harder to investigate) would lead to a semantics which is significantly different from the one used by the corpus of code we are concerned with, which does make more assumptions about C than that would permit. Our goals are thus rather different from those of the C standard committee, at least as expressed in this from the C99 Rationale v5.10: *"Beyond this two-level scheme [conforming hosted vs freestanding implementations], no additional subsetting is defined for C, since the C89 Committee felt strongly that too many levels dilutes the effectiveness of a standard."*. Our impression is that mainstream usage and implementations are using a significantly different language from that defined by the standard; this divergence makes the standard less relevant and leaves practice on an uncertain footing.

The main body of this note is a collection of 85 specific questions about the semantics of C, each stated reasonably precisely in prose and most supported by one or more test-case examples and by discussion of the ISO and de facto standards. Each particular view of C will have its own answers (or be unclear) for each of these questions; for some questions all views will agree on the answer, while for other questions different views have quite different answers. The answers for a particular view thus locate that view within an 85-dimensional space of conceivable Cs.

Our questions and test cases were developed in an iterative process of reading the literature (the ISO standards, defect reports, academic papers, and blog posts); building candidate models; writing tests; experimenting with those on particular compilers; writing the surveys we discuss below; analysing our survey results; and discussions with experts. We have tried to address all the important issues in the semantics of C memory object models, but there may well be others (as there is no well-defined space of "conceivable C semantics", this cannot be complete in any precise sense); we would be happy to learn of others that we should add.

Our test cases are typically written to illustrate a particular semantic question as concisely as possible. Some are "natural" examples, of desirable C code that one might find in the wild, but many are testing corner cases, e.g. to explore just where the defined/undefined-behaviour boundary is, and would be considered pathological if they occurred in the form given in real code.

Making the tests concise to illustrate semantic questions also means that most are not written to trigger interesting compiler behaviour, which might only occur in a larger context that permits some analysis or optimisation pass to take effect. Moreover, following the spirit of C, compilers do not report all instances of undefined behaviour. Hence, only in some cases is there anything to be learned from the experimental compiler behaviour. For any executable semantics, on the other hand, running all of them should be instructive.

Direct investigation of (1) and (2) is challenging. For (1), the behaviour of mainstream compilers is really defined only by their implementations; it is not documented in sufficient detail to answer all the important questions. Those are very large bodies of code, and particular behaviour of analysis and optimisation passes may only be triggered on relatively complex examples. We include experimental data for all our tests nonetheless, for various C implementations; in some cases this is instructive.

Given a complete candidate model we could conceivably do random testing against existing implementations, but that is challenging in itself. One of our main concerns is the border between defined and undefined behaviour, but (a) we do not have a good random test generator for programs on that border (the existing Csmith test generator by Yang et al. [57] is intended to only produce programs without undefined behaviour, according to its authors' interpretation), and (b) mainstream C implementations are not designed to report all instances of undefined behaviour; they instead assume its absence to justify optimisations.

For (2), it is hard to determine what assumptions a body of C code relies on. We draw on data from the ASPLOS 2015 paper by Chisnall et al. [14], both from instrumenting LLVM and trying to port a number of C programs to a more-than-usually restrictive implementation, their CHERI platform.

We can investigate (3) by asking the community of expert C programmers what properties they think they assume of the language in practice, which we have done with two surveys (to the best of our knowledge, this is a novel approach to investigating the de facto semantics of a widely used language). The first version, in early 2013, had 42 questions, with concrete code examples and subquestions about the de facto and ISO standards. We targeted this at a small number of experts, including multiple contributors to the ISO C or C++ standards committees, C analysis tool developers, experts in C formal semantics, compiler writers, and systems programmers. The results were very instructive, but this survey demanded a lot from the respondents; it was best done by discussing the questions with them in person over several hours. The concrete code examples helped make the questions precise, but they also created confusion: being designed to probe semantic questions about the language, many are not natural idiomatic code, but many readers tried to interpret them as such. Our second version (in mid 2015), was simplified, making it feasible to collect responses from a wider community. We designed 15 questions, focussed on some of the most interesting issues, asked only about the de facto standard (typically asking (a) whether some idiom would work in normal C compilers and (b) whether it was used in practice), and omitted the concrete code examples. Aiming for a modest-scale but technically expert audience, we distributed the survey among our local systems research group, at EuroLLVM 2015, via technical mailing lists: gcc, llvmdev, cfe-dev, libc-alpha, xorg, freebsd-developers, xen-devel, and Google C user and compiler lists, and via John Regehr's blog, widely read by C experts. There were around 323 responses, including around 100 printed pages of textual comments. Most respondents reported expertise in C systems programming (255) and many reported expertise in compiler internals (64) and in the C standard (70). The results are available on the web[1]; we refer to them where appropriate but do not include them here.

1.1 Experimental Testing

The examples are compiled and run with a range of tools:

- GCC 4.8, 4.9, and 5.3, and clang 33-37, all at O0, O2, and O2 with `-fno-strict-aliasing`, on x86 on FreeBSD, e.g.

```
gcc48 -O2 -std=c11 -pedantic -Wall -Wextra
  -Wno-unused-variable -pthread
```

- clang37 with address, memory, and undefined-behaviour sanitisers, e.g.

```
clang37 -fsanitize=address -std=c11 -pedantic
```

---

[1] www.cl.cam.ac.uk/~pes20/cerberus/

```
  -Wall -Wextra -Wno-unused-variable -pthread
```

- CHERI clang at O0, O2, and O2 with `-fno-strict-aliasing`, e.g.

```
clang -O2 -std=c11 -target=cheri-unknown-freebsd
  -mcpu=mips3 -pedantic -Wall -Wextra -mabi=sandbox
  -Wno-unused-variable -lc -lmalloc_simple
```

- The CHERI CPU running pure MIPS code, e.g.:

```
clang -O2 -std=c11 -target=mips64-unknown-freebsd
  -mcpu=mips3 -pedantic -Wall -Wextra
  -Wno-unused-variable
```

- the TrustInSoft `tis-interpreter` tool, version Magnesium-20151002+dev
- the KCC tool, in the evaluation version RV-Match v0.1 distributed by Runtime Verification Inc. at `https://runtimeverification.com/match/download/`, downloaded 2016-03-11.

Some tests rely on address coincidences for the interesting execution; for these we include multiple variants, tuned to the allocation behaviour in the implementations we consider. Running the tests on other platforms may need additional variants to be added.

The tests are run using a test harness, `charon`, that generates individual test instances from JSON files describing the tests and tools; `charon` logs all the compile and execution output (together with the test itself and information about the host) to another JSON file for analysis. The tests and harness can be packaged up in a single tarball that can be run easily. `charon` also supports cross-compilation, to let the CHERI tests be compiled on a normal host and executed on the CHERI FPGA-based hardware. Selected data from the combined log files is automatically included in this document.

1.2 Summary of answers

For each question we give multiple answers, as below. These should be treated with caution: given the complex and conflicted state of C, many are subject to interpretation or to revision, e.g. as we learn more about the de facto standards.

- iso: the ISO C11 standard
- defacto-usage: the de facto standard of usage in practice
- defacto-impl: the de facto standard of mainstream current implementations
- cerberus-defacto: the intended behaviour of our candidate de facto formal model
- cheri: the intended behaviour of CHERI
- tis: the observed behaviour of the TrustInSoft `tis-interpreter`
- kcc: the observed behaviour of the KCC tool

Note that the last two are inferences from the single data points (and, for tis, some discussion with the developers); they should be treated with caution.

Of the 85 questions,

- for 39 the ISO standard is unclear;
- for 27 the de facto standards are unclear, in some cases with significant differences between usage and implementation; and
- for 27 there are significant differences between the ISO and the de facto standards.

We discuss related work in some detail in §6.

Acknowledgements We thank all those who have provided responses to our C surveys, without which this work would not have been possible, especially Hans Boehm and Paul McKenney. We thank John Regehr and Pascal Cuoq for running `tis-interpreter` tests and discussion of the results, Jean Pichon-Pharabod for other testing assistance, and Colin Rothwell, Jon Woodruff, Mike Roe, and Simon Moore for their work on the CHERI ISA and CHERI C. We acknowledge funding from EPSRC grants EP/H005633 (Leadership Fellowship, Sewell) and EP/K008528 (REMS Programme Grant), and a Gates Cambridge Scholarship (Nienhuis). This work is also part of the CTSRD projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

2. Abstract Pointers

The most important and subtle questions are about the extent to which C values (especially pointers, but also unspecified values, structures, and unions) are abstract, as opposed to being simple bit-vector-represented quantities.

2.1 Pointer Provenance

It might be tempting to think that a C pointer is completely concrete, simply a machine address, but things are not that simple, either in the de facto or ISO standards.

2.1.1 Q1. Must the pointer used for a memory access have the right provenance, i.e. be derived from the pointer to the original allocation (with undefined behaviour otherwise)? (This lets compilers do provenance-based alias analysis)

ISO: yes DEFACTO-IMPL: yes DEFACTO-USAGE: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: example not supported (`memcmp` of pointer representations) KCC: Execution failed (unclear why)

Consider the following pathological code (adapted from the WG14 Defect Report DR260[2] and its committee response), first from the mainstream-implementation point of view.

Example (`provenance_basic_global_yx.c`):

```
#include <stdio.h>
#include <string.h>
int  y=2, x=1;
int main() {
  int *p = &x + 1;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  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);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600bd4 q=0x600bd4
x=1 y=2 *p=11 *q=2
```

ISO: `undefined behaviour`

DEFACTO: `undefined behaviour`

Depending on the implementation, `x` and `y` might happen to be allocated in adjacent memory, in which case `&x+1` and `&y` will have bitwise-identical representation values, the `memcmp` will succeed, and `p` (derived from a pointer to `x`) will have the same representation value as a pointer to a different object, `y`, at the point of the update `*p=11`. This can occur in practice with GCC -O2. The output of

```
x=1 y=2 *p=11 *q=2
```

suggests that the compiler is reasoning that `*p` does not alias with `y` or `*q`, and hence that the initial value of `y=2` can be propagated to the final `printf`.

This outcome would not be correct with respect to a naive concrete semantics, and so to make the compiler sound it is necessary for this program to be deemed to have *undefined behaviour* (which in C terms means that the compiler is allowed to do anything at all). GCC does not report a compile- or run-time warning or error for this example, but that is not required by the standard for programs with undefined behaviour. Note that this example does not involve type-based alias analysis, and the outcome is not affected by GCC's `-fno-strict-aliasing` flag. One might ask whether the mere formation of the pointer `&x+1` is legal. We return to such questions later, but this case is explicitly permitted by the ISO standard.

Clang and GCC -O0 allocate differently, so one has to interchange the declarations of `x` and `y` to make `p` and `q` happen to hold bitwise identical values, but then the outcome does not exhibit the effects of similar analysis and optimisation. One has to treat such negative results with caution, of course: it does not follow that this version of the compiler will not optimise similar examples, as the negative result could be simply because the test is not complex enough to cause particular optimisations to fire.

Example (`provenance_basic_global_xy.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600bd8 q=0x600bd0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600ad0 q=0x600ad0
x=1 y=11 *p=11 *q=11
```

On the other hand, ICC on this version gives `x=1 y=2 *p=11 *q=11`, so also definitely needs this to be an undefined-behaviour program to be sound.

Clang37-UBSAN does not detect this undefined behaviour. The clang37-ASAN execution does not have the address coincidence needed to make the test result meaningful. CHERI C behaves just like x86 Clang here because linker support (which is needed to provide provenance to pointers to globals) is not yet implemented.

For reference, consider similar examples but with two malloc'd regions rather than global statically allocated objects, e.g. `provenance_basic_malloc_offset+2.c` and `provenance_basic_malloc_offset+12.c`. Here according to the ISO standard it is illegal to form the pointer required to get from one to the other (as it is not one-past). We return to whether that is allowed in the de facto standard in §2.13 (p.31). Here GCC 4.8 appears not to assume a lack of aliasing; the Clang behaviour is the same as the previous example.

The current CHERI implementation treats globals and variables with automatic storage duration differently (pending improvements to the linker implementation). Accordingly, we include variants of the first test with automatic storage duration.

Example (`provenance_basic_auto_yx.c`):

```
#include <stdio.h>
#include <string.h>
int main() {
  int  y=2, x=1;
  int *p = &x + 1;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  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);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x7fffffffe9f0 q=0x7fffffffe9e8
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x7fffffffe9fc q=0x7fffffffe9fc
x=1 y=11 *p=11 *q=11
```

ISO: `undefined behaviour`

---

[2] http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm

DEFACTO: `undefined behaviour`

EXAMPLE (`provenance_basic_auto_xy.c`):

```
#include <stdio.h>
#include <string.h>
int main() {
  int  x=1, y=2;
  int *p = &x + 1;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  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);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x7fffffffe9ec q=0x7fffffffe9ec
x=1 y=11 *p=11 *q=11
```

ISO: `undefined behaviour`

DEFACTO: `undefined behaviour`

From the ISO-standard point of view, the committee response to Defect Report #260 appears to be regarded as definitive, though it has not been folded into the standard text. It takes the position that the provenance of a pointer value is significant, writing *"[an implementation] may also treat pointers based on different origins as distinct even though they are bitwise identical"*. The pointer addition in `&x + 1` is legal[3] but DR260 implies that the write `*p = 11` gives rise to undefined behaviour, meaning that programmers should not write this code and the ISO standard does not constrain how compilers have to treat it. This licenses use of an analysis and optimisation that would otherwise be unsound.

Our de facto and ISO standard semantics should both deem this program to have undefined behaviour, to be sound w.r.t. GCC and ICC.

2.1.2 Q2. Can equality testing on pointers be affected by pointer provenance information?

ISO: yes (from DR260 CR) DEFACTO-USAGE: unknown DEFACTO-IMPL: yes, nondeterministically at each occurrence CERBERUS-DEFACTO: yes, nondeterministically at each occurrence CHERI: nondet TIS: Such pointer comparison is a source of nondeterminism which tis intentionally flags (with `pointer_comparable`) KCC: unclear (the printed addresses are not concrete values)

[Question 4/15 of our *What is C in practice? (Cerberus survey v2)*[4] relates to this.]

The above example shows that C compilers have to be allowed to do static alias analysis and optimisation based on pointer provenance, but one would not expect a conventional C implementation to keep provenance information at runtime (unconventional and more defensive implementations such as Softbound [40], Hardbound [17], or CHERI might do that). To see this in practice, we form pointers `p` and `q` as above, with different provenance but identical representations, and then test their equality with `==` (instead of their representation equality with `memcmp`). The result is variously true or false depending on the context.

In this first example the equality result is false in GCC -O2 (even though the two pointers print the same):

EXAMPLE (`provenance_equality_global_yx.c`):

```
#include <stdio.h>
#include <string.h>
int  y=2, x=1;
int main() {
  int *p = &x + 1;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  _Bool b = (p==q);
  // can this be false even with identical addresses?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600b4c q=0x600b4c
(p==q) = false
```

ISO: `nondeterministically true or false`

DEFACTO: `nondeterministically true or false`

The same holds (perhaps surprisingly) if the test is pulled out into another function (`provenance_equality_global_fn_yx.c`), but if that function is put into a separate compilation unit (`provenance_equality_global_cu_yx_a.c` and `provenance_equality_global_cu_yx_b.c`) the comparison gives true:

```
p=0x601024 q=0x601024
(p==q) = true
```

For Clang, again flipping the order of `x` and `y`, we see just `true` for all these tests where the addresses print the same.

EXAMPLE (`provenance_equality_global_xy.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600b50 q=0x600b48
(p==q) = false
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600ab0 q=0x600ab0
(p==q) = true
```

---

[3] The addition is licensed by 6.5.6 "Additive operators", where: 6.5.6p7 says *"For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type."*, and 6.5.6p8 says *"[...] Moreover, if the expression P points to the last element of an array object, the expression (P)+1 points one past the last element of the array object [...]"*.

[4] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

EXAMPLE (`provenance_equality_global_fn_xy.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600b90 q=0x600b88
(p==q) = false
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600b08 q=0x600b08
(p==q) = true
```

and `provenance_equality_global_cu_xy_a.c` / `provenance_equality_global_cu_xy_b.c`.

For CHERI, we again give a version of the example using automatic storage location variables.

EXAMPLE (`provenance_equality_auto_yx.c`):

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x7fffffffe9ec q=0x7fffffffe9ec
(p==q) = true
```

EXAMPLE (`provenance_equality_auto_fn_yx.c`):

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x7fffffffe9dc q=0x7fffffffe9dc
(p==q) = true
```

and `provenance_equality_auto_cu_yx_a.c` / `provenance_equality_auto_cu_yx_b.c`.

To allow this variation, our candidate de facto model and any ISO standard semantics should both allow pointer comparison to either use provenance-aware or provenance-oblivious comparison nondeterministically. In many cases the two will give identical results (for performance of the executable semantics, for those one might choose not to make an explicit nondeterministic choice).

2.1.3 GCC and ISO C11 differ on the result of a == comparison on a one-past pointer

This arises from the preceeding examples: a defect in the ISO standard text, in which the DR260 position has not been consistently incorporated.

From the ISO standard point of view, the standard is clear that in general pointers to different objects of compatible type can be compared with == (in contrast to relational operators, where such comparison gives undefined behaviour).[5] But the text of C11 and DR260 seem inconsistent w.r.t. the result of the comparison. In the former, it is specified by 6.5.9p6: *"Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.109)"*

Footnote 109: *"Two objects may be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior."*

The last clause of 6.5.9p6 is surprising: given *"a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space"* the standard *requires* them to compare equal rather than merely permitting them to compare equal. This seems to conflict with the spirit of DR260, which allows the pointer provenance to be taken into account. The variation in experimental results can be licensed by the *may* in the DR260 *"[an implementation] may also treat pointers based on different origins as distinct even though they are bitwise identical"*.

The `provenance_equality_global_yx.c` behaviour is arguably a bug in GCC, violating 6.5.9p6, as we reported (see Fig. 1). The developer comments disagree, arguing that pointers need not have stable numerical values (we think that implausible, as it would break lots of code; we return to stability in §2.9, p.26). But probably the behaviour should be allowed in any case, and the standard should have something better than the if-and-only-if in 6.5.9p6. The proposal above to nondeterministically choose provenance-aware or concrete comparison relaxes the if-and-only-if (taking DR260 to have precedence over the C11 text).

2.2 Pointer provenance via integer types

In practice it seems to be routine to convert from a pointer type to a sufficiently wide integer type and back, e.g. to use unused bits of the pointer to store tag bits. The interaction between that and provenance is interesting.

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

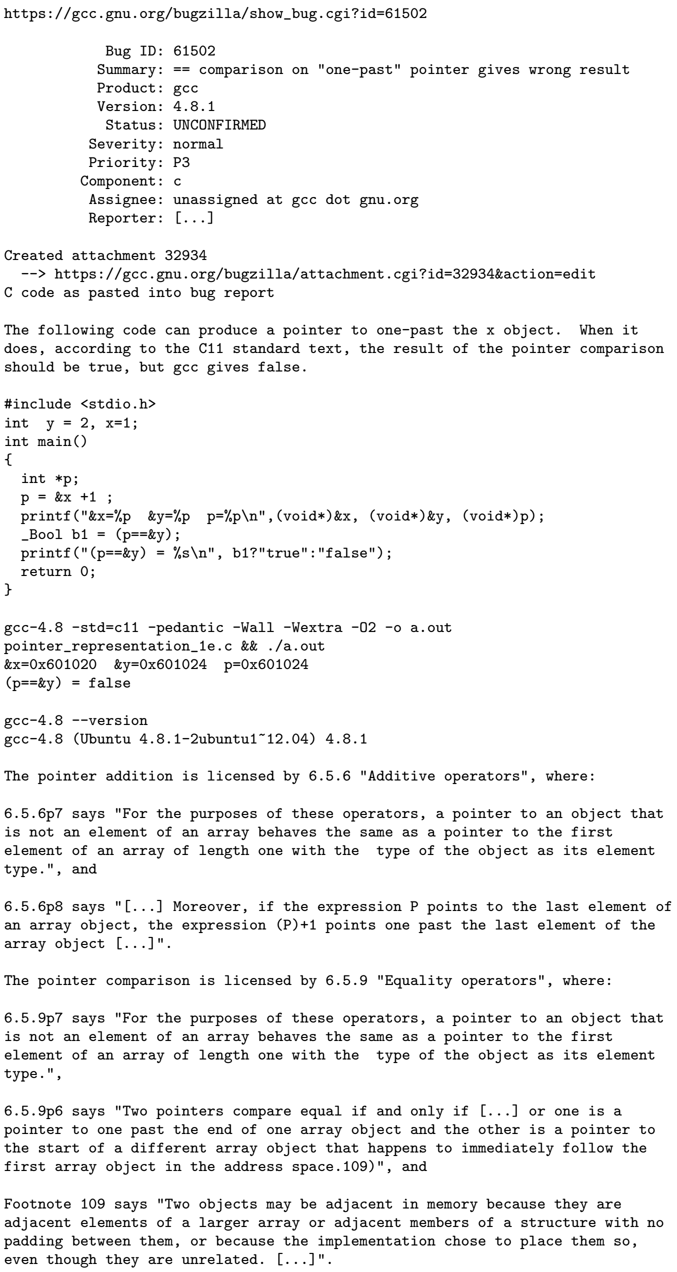
ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes

2.2.2 Q4. Can one make a usable pointer via casts to `unsigned long` and back?

ISO: implementation-defined DEFACTO-USAGE: yes (normally) DEFACTO-IMPL: yes (normally) CERBERUS-DEFACTO: yes (if `unsigned long` is wide enough) CHERI: no TIS: yes KCC: yes

---

[5] The use of == to compare the two pointers is licensed by 6.5.9 *Equality operators*, which allows the case in which *"both operands are pointers to qualified or unqualified versions of compatible types;"*.



```
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61502

            Bug ID: 61502
           Summary: == comparison on "one-past" pointer gives wrong result
           Product: gcc
           Version: 4.8.1
            Status: UNCONFIRMED
          Severity: normal
          Priority: P3
         Component: c
          Assignee: unassigned at gcc dot gnu.org
          Reporter: [...]

Created attachment 32934
  --> https://gcc.gnu.org/bugzilla/attachment.cgi?id=32934&action=edit
C code as pasted into bug report

The following code can produce a pointer to one-past the x object.  When it
does, according to the C11 standard text, the result of the pointer comparison
should be true, but gcc gives false.

#include <stdio.h>
int  y = 2, x=1;
int main()
{
  int *p;
  p = &x +1 ;
  printf("&x=%p  &y=%p  p=%p\n",(void*)&x, (void*)&y, (void*)p);
  _Bool b1 = (p==&y);
  printf("(p==&y) = %s\n", b1?"true":"false");
  return 0;
}

gcc-4.8 -std=c11 -pedantic -Wall -Wextra -O2 -o a.out
pointer_representation_1e.c && ./a.out
&x=0x601020  &y=0x601024  p=0x601024
(p==&y) = false

gcc-4.8 --version
gcc-4.8 (Ubuntu 4.8.1-2ubuntu1~12.04) 4.8.1

The pointer addition is licensed by 6.5.6 "Additive operators", where:

6.5.6p7 says "For the purposes of these operators, a pointer to an object that
is not an element of an array behaves the same as a pointer to the first
element of an array of length one with the  type of the object as its element
type.", and

6.5.6p8 says "[...] Moreover, if the expression P points to the last element of
an array object, the expression (P)+1 points one past the last element of the
array object [...]".

The pointer comparison is licensed by 6.5.9 "Equality operators", where:

6.5.9p7 says "For the purposes of these operators, a pointer to an object that
is not an element of an array behaves the same as a pointer to the first
element of an array of length one with the  type of the object as its element
type.",

6.5.9p6 says "Two pointers compare equal if and only if [...] or one is a
pointer to one past the end of one array object and the other is a pointer to
the start of a different array object that happens to immediately follow the
first array object in the address space.109)", and

Footnote 109 says "Two objects may be adjacent in memory because they are
adjacent elements of a larger array or adjacent members of a structure with no
padding between them, or because the implementation chose to place them so,
even though they are unrelated. [...]".
```

Figure 1. Bug ID: 61502

We first have to consider the basic question of simple roundtrips, casting a pointer to an integer type and back, either via `intptr_t` or `unsigned long`:

EXAMPLE (`provenance_roundtrip_via_intptr_t.c`):

```
#include <stdio.h>
#include <inttypes.h>
int  x=1;
int main() {
  int *p = &x;
  intptr_t i = (intptr_t)p;
  int *q = (int *)i;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
*p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `defined behaviour (if the intptr type is provided)`

DEFACTO: `defined behaviour`

EXAMPLE (`provenance_roundtrip_via_unsigned_long.c`):

```
#include <stdio.h>
int  x=1;
int main() {
  int *p = &x;
  unsigned long i = (unsigned long)p;
  int *q = (int *)i;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
*p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `implementation-defined`

DEFACTO: `defined behaviour`

In the de facto standards this is clearly allowed, both for `intptr_t` and (as in Linux or more generally in Unix) some other integer types (e.g. `unsigned long`). This involves the *Int: storing a pointer in an integer variable in memory* of the CHERI ASPLOS paper, which they observed commonly in practice.

One respondent comments that the 8086 model (up to 80286) had 16-bit near pointers (relying on segment registers for 4 more bits) and longer far pointers, so just copying the former wouldn't be sufficient. CDC6600 had pointers to 60-bit words, so character pointers were complex. Neither are current mainstream C.

The ISO standard leaves conversions between pointer and integer types almost entirely implementation-defined (except for conversion of integer constant 0 and null pointers), with:

6.3.2.3p5: *"An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned,* *might not point to an entity of the referenced type, and might be a trap representation.67)"*

6.3.2.3p6: *"Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type."*

(Footnote 67 says *"The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment."*; the exact force of this is not clear.)

On the other hand, 7.20 *Integer types* `<stdint.h>` introduces optional types `intptr_t` and `uintptr_t` with roundtrip properties from pointer to integer and back:

7.20.1.4p1 *"The following type designates a signed integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer:* `intptr_t`*". "The following type designates an unsigned integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer:* `uintptr_t`*".*

We presume that this "compare equal" is intended to imply that the result is interchangeable with the original pointer, but, as we have seen examples in which two pointers compare equal but access via one gives undefined behaviour while access via the other does not, this is unfortunate phrasing (it likely antedates DR260) and should be changed. In the CHERI case tags are not visible in memory, so there also a pointer and an integer might compare equal but not be equiusable.

Note that these examples do not involve function pointers; things might be different there.

2.2.3 Q5. Must provenance information be tracked via casts to integer types and integer arithmetic?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: tis-interpreter sees the possibility of signed arithmetic overflow (correctly so, if one assumes nothing about memory layout) KCC: Execution failed (unclear why)

Should one be allowed to use `intptr_t` (or `uintptr_t`) arithmetic to work around provenance limitations? The next example (also pathological code) is a variant of the §2.1.1 (p.7) `provenance_basic_global_yx.c` in which we use integer arithmetic (and casts to and from `intptr_t`) instead of pointer arithmetic. The arithmetic again just happens (in these implementations) to be the right offset between the two global variables.

EXAMPLE (`provenance_basic_using_intptr_t_global_yx.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int  y = 2, x = 1;
int main() {
  intptr_t ux = (intptr_t)&x;
  intptr_t uy = (intptr_t)&y;
  intptr_t offset = 4;
  int *p = (int *)(ux + offset);
  int *q = &y;
  printf("Addresses: &x=%"PRIiPTR" p=%p &y=%"PRIiPTR\
         "\n",ux,(void*)p,uy);
  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);
  }
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=6294512 p=0x600bf4 &y=6294516
x=1 y=2 *p=11 *q=2
```

ISO: `undefined behaviour`

DEFACTO: `undefined behaviour`

As before, we see that GCC seems to be assuming that this cannot occur, by making an optimisation that would be unsound if this program does not have undefined behaviour.

This is consistent with the GCC documentation, which says: *"When casting from pointer to integer and back again, the resulting pointer must reference the same object as the original pointer, otherwise the behavior is undefined. That is, one may not use integer arithmetic to avoid the undefined behavior of pointer arithmetic as proscribed in C99 and C11 6.5.6/8."*[6]

Note that this GCC text presumes that there is an obvious "original pointer" associated with any integer value which is cast back to a pointer; as we discuss in §2.3 (p.15), that is not always the case.

As before, for this version of Clang we don't see the optimisation for the analogous example with the two allocations flipped, so this is uninformative.

EXAMPLE (`provenance_basic_using_intptr_t_global_xy.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=6294516 p=0x600bf8 &y=6294512
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: &x=6294236 p=0x600ae0 &y=6294240
x=1 y=11 *p=11 *q=11
```

EXAMPLE (`provenance_basic_using_intptr_t_global_xy_offset64.c`):

TIS-INTERPRETER:

```
[value] Analyzing a complete application starting at
main
[value] Computing initial state
```

---

[6] https://gcc.gnu.org/onlinedocs/gcc/Arrays-and-pointers-implementation.html

```
[value] Initial
state computed
provenance_basic_using_intptr_t_global_xy
_offset64.c:10:[kernel] warning: signed overflow. assert
-9223372036854775808  ux+offset;

stack: main
provenance_basic_using_intptr_t_global_xy_of
fset64.c:10:[kernel] warning: signed overflow. assert
ux+offset  9223372036854775807;

stack: main
[value] Stopping at nth alarm
[value] user
error: Degeneration occurred:

results are not correct for lines of code that can be
reached from the degeneration point.
```

For CHERI we include a variant with automatic storage duration variables:

EXAMPLE (`provenance_basic_using_intptr_t_auto_yx.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int main() {
  int  y = 2, x = 1;
  intptr_t ux = (intptr_t)&x;
  intptr_t uy = (intptr_t)&y;
  intptr_t offset = 4;
  int *p = (int *)(ux + offset);
  int *q = &y;
  printf("Addresses: &x=%"PRIiPTR" p=%p &y=%"PRIiPTR\
         "\n",ux,(void*)p,uy);
  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);
  }
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=140737488349644 p=0x7fffffffe9d0
&y=140737488349640
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: &x=140737488349656 p=0x7fffffffe9dc
&y=140737488349660
x=1 y=11 *p=11 *q=11
```

ISO: `undefined behaviour`

DEFACTO: `undefined behaviour`

EXAMPLE (`provenance_basic_using_intptr_t_auto_yx_offset-16.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int main() {
  int  y = 2, x = 1;
  intptr_t ux = (intptr_t)&x;
  intptr_t uy = (intptr_t)&y;
  intptr_t offset = -16;
  int *p = (int *)(ux + offset);
  int *q = &y;
  printf("Addresses: &x=%"PRIiPTR" p=%p &y=%"PRIiPTR\
         "\n",ux,(void*)p,uy);
  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);
  }
}
```

TIS-INTERPRETER:

```
[value] Analyzing a complete application starting at
main
[value] Computing initial state
[value] Initial
state computed
provenance_basic_using_intptr_t_auto_yx_o
ffset-16.c:10:[kernel] warning: signed overflow. assert
-9223372036854775808  ux+offset;

stack: main
provenance_basic_using_intptr_t_auto_yx_offs
et-16.c:10:[kernel] warning: signed overflow. assert
ux+offset  9223372036854775807;

stack: main
[value] Stopping at nth alarm
[value] user
error: Degeneration occurred:

results are not correct for lines of code that can be
reached from the degeneration point.
```

EXAMPLE (`provenance_basic_using_intptr_t_auto_xy.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int main() {
  int  x = 1, y = 2;
  intptr_t ux = (intptr_t)&x;
  intptr_t uy = (intptr_t)&y;
  intptr_t offset = 4;
  int *p = (int *)(ux + offset);
  int *q = &y;
  printf("Addresses: &x=%"PRIiPTR" p=%p &y=%"PRIiPTR\
         "\n",ux,(void*)p,uy);
  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);
  }
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=140737488349640 p=0x7fffffffe9cc
&y=140737488349644
```

```
x=1 y=11 *p=11 *q=11
```

For reference, for a similar example using two malloc'd regions and a constant offset we also see similar GCC and Clang results as before: GCC sometimes assumes the two pointers do not alias (interestingly, only with GCC 4.9 -O2, not GCC 4.8 -O2), while these versions of Clang do not:

EXAMPLE (`provenance_basic_using_intptr_t_malloc_offset_8.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <inttypes.h>
int main() {
  int *xp=malloc(sizeof(int));
  int *yp=malloc(sizeof(int));
  *xp=1;
  *yp=2;
  int *p = (int*) (((uintptr_t)xp) + 8);
  int *q = yp;
  printf("Addresses: xp=%p p=%p q=%p\n",
         (void*)xp,(void*)p,(void*)q);
  //  if (p == q) {
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("*xp=%d *yp=%d *p=%d *q=%d\n",*xp,*yp,*p,*q);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: xp=0x801417058 p=0x801417060 q=0x801417060
*xp=1 *yp=2 *p=11 *q=2
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: xp=0x801417058 p=0x801417060 q=0x801417060
*xp=1 *yp=11 *p=11 *q=11
```

ISO: `undefined behaviour`

DEFACTO: `undefined behaviour`

This matches the `provenance_basic_malloc_offset+8.c` example of §2.1.1 (p.7), which did the arithmetic directly on pointers instead of at `uintptr_t`, and for which the optimisation was observed in GCC.

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

U:ISO

ISO: unclear – implementation-defined? DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: test not supported (`_Alignof`) KCC: Execution failed (unclear why)

Now we extend the first example of §2.2.1 (p.10), that cast a pointer to `intptr_t` and back, to use logical operations on the integer value to store some tag bits. The following code exhibits a strong form of this, storing the address and tag bit combination as a pointer (which thereby creates a misaligned pointer value, though one not used for accesses); a weaker form would store the combined value only as an integer.

EXAMPLE (`provenance_tag_bits_via_uintptr_t_1.c`):

```
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
int x=1;
int main() {
  int *p = &x;
  // cast &x to an integer
  uintptr_t i = (uintptr_t) p;
  // 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;
  // are r and p now equivalent?
  *r = 11;           //  defined behaviour?
  _Bool b = (r==p);
  printf("x=%i *r=%i (r==p)=%s\n",x,*r,b?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
x=11 *r=11 (r==p)=true
```

CLANG36-O2-NO-STRICT-ALIASING: ... as above

DEFACTO: `defined behaviour`

ISO: `unclear - implementation-defined?`

This idiom seems to be widely relied on in practice, and so our de facto standard semantics should allow it, for any integer type of the right width. It is the *Mask: simple masking of pointers* idiom of the CHERI ASPLOS paper, widely observed in practice.

Beyond just manipulating the low-order bits, Linux has "buddy allocators" in which one XORs some particular pointer bits to move inside a tree structure, within some allocated region (though perhaps not made by malloc).

In this example there is still an obvious unique provenance that one can track through the integer computation; in the next section we consider cases where that is not the case.

For mismatching widths, the GCC documentation[7] gives a concrete algorithm for converting between integers and pointers which gives the identity on their bit representations in this case: *"A cast from pointer to integer discards most-significant bits if the pointer representation is larger than the integer type, sign-extends [Footnote 1: Future versions of GCC may zero-extend, or use a target-defined `ptr_extend` pattern. Do not rely on sign extension.] if the pointer representation is smaller than the integer type, otherwise the bits are unchanged."* and *"A cast from integer to pointer discards most-significant bits if the pointer representation is smaller than the integer type, extends according to the signedness of*

---

[7] Section 4.7 *Arrays and pointers* of *C Implementation-defined behavior*, `http://gcc.gnu.org/onlinedocs/gcc/C-Implementation.html`

the integer type if the pointer representation is larger than the integer type, otherwise the bits are unchanged.”.

It does not comment on provenance, and it also leaves open the question of whether the implementation might use the low-order bits for its own purposes (making the `assert((i & 3u) == 0u)` of the example false). We take this to be an omission in the GCC documentation, and assume implementations do not (otherwise much existing code would break). Really, the set of unused bits of pointers of each alignment should be explicitly implementation-defined in the standard.

For mismatching widths a de facto semantic model has to choose whether to follow this GCC documentation (loosened according to the footnote and strengthened w.r.t. provenance and unused bits), or be more nondeterministic.

This example tells us that at least the specific operations on integers used here should preserve the provenance information. The simplest proposal would be to have all integer operations preserve provenance, but, as we discuss below, that is not always appropriate.

The CHERI behaviour here, failing in the assert, is quite subtle. The `uintptr_t` value `i` is a capability. All arithmetic on it is done on the offset. The assert at the start is failing because `i & 3u` first promotes `3u` to `__intcap_t` (the underlying type that `uintptr_t` is a typedef for), which gives you an untagged capability with base 0 and offset 3. This is then anded with `i`, by getting the offsets of both, anding the result together, and applying the offset to `i`. The result is therefore a capability with the base/length/permissions of `i`, but an offset of 0. This is then compared against a null capability, and the comparison fails (because it is not a null capability).

The assertion seems like something that a reasonable programmer ought to expect to work, so the best design is an open question at present. Without the assert, `provenance_tag_bits_via_uintptr_t_1_no_assert.c`, the test works on CHERI, so, interestingly, it is only code that is defensively written that will experience the problem.

2.2.5 Q7. Can equality testing on integers that are derived from pointer values be affected by their provenance?

U:ISO

ISO: unclear (we suggest no) DEFACTO-USAGE: no? DEFACTO-IMPL: no? (modulo Clang bug?) CERBERUS-DEFACTO: no CHERI: ? TIS: pointer_comparable KCC: Execution failed (unclear why)

EXAMPLE (`provenance_equality_uintptr_t_global_yx.c`):

```
#include <stdio.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  uintptr_t p = (uintptr_t)(&x + 1);
  uintptr_t q = (uintptr_t)&y;
  printf("Addresses: p=%" PRIxPTR " q=%" PRIxPTR "\n",
         p,q);
  _Bool b = (p==q);
  // can this be false even with identical addresses?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=600b64 q=600b64
(p==q) = true
```

ISO: `unclear - should be true when the addresses print equal?`

EXAMPLE (`provenance_equality_uintptr_t_global_xy.c`):

```
#include <stdio.h>
#include <inttypes.h>
int x=1, y=2;
int main() {
  uintptr_t p = (uintptr_t)(&x + 1);
  uintptr_t q = (uintptr_t)&y;
  printf("Addresses: p=%" PRIxPTR " q=%" PRIxPTR "\n",
         p,q);
  _Bool b = (p==q);
  // can this be false even with identical addresses?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=600b68 q=600b60
(p==q) = false
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=600ab8 q=600ab8
(p==q) = true
```

ISO: `unclear - should be true when the addresses print equal?`

Can this print false even when the numeric addresses are identical? This is suggested by an example from Krebbers [29], as discussed in §6.11. The observed Clang ‘false’ behaviour seems to be a compiler bug, similar to the GCC bug reported by them.

2.3 Pointers involving multiple provenances

We now consider examples in which a pointer is constructed using computation based on *multiple* pointer values. How widely this is used is not clear to us. There are at least two important examples in the wild, the Linux and FreeBSD per-CPU allocators, and also the classic XOR linked list implementation (the latter, while much-discussed, appears not to be a currently common idiom, though pointer XOR is apparently used in L4 [48, §6.2]). We discuss both below.

2.3.1 Q8. Should intra-object pointer subtraction give provenance-free integer results?

This is uncontroversial:

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes (third test has `memcmp` errors, as Q1) KCC: first tests ok, later tests not supported, with Execution failed error

We begin with some simple cases. Given two pointers within an array, one should certainly be able to calculate an offset, by subtracting them, that can be used either within the same array or within a different array, e.g.

```
&x([0]) + (&(x[1])-&(x[0]))
&x([0]) + (&(y[1])-&(y[0]))
```

and in full:

EXAMPLE (`provenance_multiple_1_global.c`):

```
#include <stdio.h>
int y[2], x[2];
int main() {
  int *p = &(x[0]) + (&(x[1])-&(x[0]));
  *p = 11;  // is this free of undefined behaviour?
  printf("x[1]=%d *p=%d\n",x[1],*p);
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
x[1]=11 *p=11
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (x[1]=11 *p=11)`

ISO: `defined behaviour (x[1]=11 *p=11)`

EXAMPLE (`provenance_multiple_2_global.c`):

```
#include <stdio.h>
int y[2], x[2];
int main() {
  int *p = &(x[0]) + (&(y[1])-&(y[0]));
  *p = 11;  // is this free of undefined behaviour?
  printf("x[1]=%d *p=%d\n",x[1],*p);
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
x[1]=11 *p=11
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (x[1]=11 *p=11)`

ISO: `defined behaviour (x[1]=11 *p=11)`

However, an offset constructed by intra-object subtraction within one object should not, when added to a pointer to a distinct object, license its use to access the first: in the examples below, the following should not be allowed to be used to access `y[0]`, and we observe GCC optimising based on that assumption.

```
&x[1] + (&y[1]-&y[1]) + 1
&x[1] + (&y[1]-&y[0]) + 0
```

In full:

EXAMPLE (`provenance_multiple_3_global_yx.c`):

```
#include <stdio.h>
#include <string.h>
int  y[2], x[2];
int main() {
  int *p = &x[1] + (&y[1]-&y[1]) + 1;
  int *q = &y[0];
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("y[0]=%d *p=%d *q=%d\n",y[0],*p,*q);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600bf0 q=0x600bf0
y[0]=0 *p=11 *q=0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600ae0 q=0x600ae0
y[0]=11 *p=11 *q=11
```

ISO: `undefined behaviour`

DEFACTO: `undefined behaviour`

EXAMPLE (`provenance_multiple_4_global_yx.c`):

```
#include <stdio.h>
#include <string.h>
int  y[2], x[2];
int main() {
  int *p = &x[1] + (&y[1]-&y[0]) + 0;
  int *q = &y[0];
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("y[0]=%d *p=%d *q=%d\n",y[0],*p,*q);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600bf0 q=0x600bf0
y[0]=0 *p=11 *q=0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600ae0 q=0x600ae0
y[0]=11 *p=11 *q=11
```

ISO: `undefined behaviour`

DEFACTO: `undefined behaviour`

2.3.2 Q9. Can one make a usable offset between two separately allocated objects by inter-object subtraction (using either pointer or integer arithmetic), to make a usable pointer to the second by adding the offset to the first?

U:ISO D:ISO-VS-DEFACTO

ISO: unclear - no? DEFACTO-USAGE: unclear (perhaps Linux/FreeBSD per-CPU variables? perhaps in sqlite?) DEFACTO-IMPL: compilers apparently assume no CERBERUS-DEFACTO: no CHERI: no TIS: no (fails with `signed overflow` (correctly so, if one assumes nothing about memory layout) KCC: no – flags UB

[Question 3/15 of our *What is C in practice? (Cerberus survey v2)*[8] relates to this.]

---

[8] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

This is a variant of the §2.2.3 (p.12) `provenance_basic_using_intptr_t_global_yx.c` in which the constant offset is replaced by a subtraction (here after casting from pointer to integer type).

EXAMPLE (`pointer_offset_from_subtraction_1_global.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int  y = 2, x=1;
int main() {
  intptr_t ux = (intptr_t)&x;
  intptr_t uy = (intptr_t)&y;
  intptr_t offset = uy - ux;
  printf("Addresses: &x=%"PRIiPTR" &y=%"PRIiPTR\
         " offset=%"PRIiPTR" \n",ux,uy,offset);
  int *p = (int *)(ux + offset);
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11; // is this free of undefined behaviour?
    printf("x=%d  y=%d  *p=%d  *q=%d\n",x,y,*p,*q);
  }
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=6294520 &y=6294524 offset=4
x=1 y=11 *p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING: ...as above (modulo addresses)

ISO: `unclear - no?`

DEFACTO: `used in practice but not supported in general`

And again in an automatic-storage-duration version:

EXAMPLE (`pointer_offset_from_subtraction_1_auto.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=140737488349640 &y=140737488349644
offset=4
x=1 y=11 *p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING: ...as above (modulo addresses)

We do not see the analysis and optimisation consequences seen for the previous example, so this experimental data does not force us to make this program have undefined behaviour.

None of the ISO standard text, DR260, and the GCC documentation discuss multiple-provenance pointers explicitly. They are consistent either with a multiple-provenance semantics or an aggressively single-provenance semantics that would regard this program as having undefined behaviour.

In practice this idiom is used in Linux and in FreeBSD for access to variables allocated by the per-CPU allocators[9]. The latter precomputes partially constructed pointers for CPU-local variables. The linker creates a region for CPU 0's copy of the kernel per-CPU variables `x`, `y`, ... A corresponding region for each other CPU is created early in the boot process, before CPU bringup. Say these start at addresses `&x_N` for each CPU `N`. Then an array `dpcpu_off[N]` is initialised with `&x_N - &x_0`, and to access a per-CPU variable `&y_N`. we add `dpcpu_off[N]` and `&y_0` to get `&x_N`. The point here is to optimise access to these variables. There are not very many of them, but they are often used in critical paths, e.g. in scheduler context switching.

The following example does essentially this, and is very similar to `pointer_offset_from_subtraction_1_global.c` above. It differs in using malloc'd regions rather than global variables and in doing the subtraction at `unsigned char *` type rather than after casting to an integer type.

EXAMPLE (`pointer_offset_from_subtraction_1_malloc.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
int main() {
  void *xp=malloc(sizeof(int)); // allocation P
  void *yp=malloc(sizeof(int)); // allocation Q
  *((int*)xp)=1;
  *((int*)yp)=2;
  ptrdiff_t offset=(unsigned char*)yp-(unsigned char*)xp;
    // provenance ?
  unsigned char *p1 = (unsigned char*)xp;// provenance P
  unsigned char *p2 = p1 + offset;       // provenance ?
  int *p = (int*)p2;
  int *q = (int*)yp;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // is this free of undefined behaviour?
    printf("*xp=%d *yp=%d *p=%d *q=%d\n",
           *(int*)xp,*(int*)yp,*(int*)p,*(int*)q);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x801417060 q=0x801417060
*xp=1 *yp=11 *p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING: ...as above

As before, we do not see an alias-analysis-based optimisation here. In previous tests we did see that for a version with a constant offset, but in this dataset we do not, as below. As usual, one should (of course) be cautious not to read too much into a lack of optimisation.

EXAMPLE (`pointer_offset_constant_8_malloc.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
int main() {
  void *xp=malloc(sizeof(int)); // allocation P
  void *yp=malloc(sizeof(int)); // allocation Q
  *((int*)xp)=1;
  *((int*)yp)=2;
  ptrdiff_t offset = 8;
    // (unsigned char*)yp - (unsigned char*)xp;
  unsigned char *p1 = (unsigned char*)xp;// provenance P
  unsigned char *p2 = p1 + offset;
  int *p = (int*)p2;
  int *q = (int*)yp;
```

---

[9] FreeBSD: `_DPCPU_PTR`, https://github.com/freebsd/freebsd/blob/master/sys/sys/pcpu.h

```
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // is this free of undefined behaviour?
    printf("*xp=%d *yp=%d *p=%d *q=%d\n",
           *(int*)xp,*(int*)yp,*(int*)p,*(int*)q);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x801417060 q=0x801417060
*xp=1 *yp=2 *p=11 *q=2
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x801417060 q=0x801417060
*xp=1 *yp=11 *p=11 *q=11
```

2.3.3 Q10. Presuming that one can have valid pointers with multiple provenances, does an inter-object pointer subtraction give a value with explicitly-unknown provenance or something more specific?

U:ISO

ISO: unclear – arguably N/A as the premise is false for ISO? DEFACTO-USAGE: unknown (not significant in normal code?) DEFACTO-IMPL: n/a (multiple-provenance not supported anyway?) CERBERUS-DEFACTO: no CHERI: no TIS: fails with `signed overflow` KCC: no – flags UB

The following example partly discriminates between the choices for the provenance of the result of an inter-object pointer subtraction (if such programs are not deemed to have undefined behaviour): either treating it as a value with explicitly-unknown provenance or one of the other two options. It uses an offset calculated between `z` and `w` to move from a pointer to `x` to a pointer to `y`. GCC does seem to assume that `p` and `q` cannot alias, suggesting that it isn't using the explicitly-unknown provenance and might be consistent with the left-provenance or union-of-provenances model here.

EXAMPLE (`pointer_offset_from_subtraction_2_global.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>
#include <inttypes.h>
int  w=4, z=3, y = 2, x=1;
int main() {
  intptr_t ux = (intptr_t)&x;
  intptr_t uy = (intptr_t)&y;
  intptr_t offsetxy = uy - ux;
  intptr_t uz = (intptr_t)&z;
  intptr_t uw = (intptr_t)&w;
  intptr_t offsetzw = uw - uz;
  printf("Addresses: &x=%"PRIiPTR" &y=%"PRIiPTR\
         " offsetxy=%"PRIiPTR" \n",ux,uy,offsetxy);
  printf("Addresses: &z=%"PRIiPTR" &w=%"PRIiPTR\
         " offsetzw=%"PRIiPTR" \n",uz,uw,offsetzw);
  assert(offsetzw==offsetxy);
  int *p = (int *)(ux + offsetzw);
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11; // is this free of undefined behaviour?
    printf("x=%d  y=%d  *p=%d  *q=%d\n",x,y,*p,*q);
  }
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=6294848 &y=6294852 offsetxy=4
Addresses: &z=6294856 &w=6294860 offsetzw=4
x=1 y=11
*p=11 *q=11
```

ISO: `unclear - undefined behaviour?`

In this dataset none of the compilers appear to optimise based on reasoning about a lack of aliasing, though earlier experiments (with GCC 4.6.3-14 and 4.7.2-5) did.

An automatic storage-duration analogue:

EXAMPLE (`pointer_offset_from_subtraction_2_auto.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=140737488349612 &y=140737488349608
offsetxy=-4
Addresses: &z=140737488349604
&w=140737488349600 offsetzw=-4
x=1 y=11 *p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING: ...as above (modulo addresses)

2.3.4 Q11. Is the XOR linked list idiom supported?

U:ISO U:DEFACTO

ISO: unclear – no? DEFACTO-USAGE: unclear (not really used in practice?) DEFACTO-IMPL: unclear CERBERUS-DEFACTO: no CHERI: no TIS: no (fails at the pointer XOR) KCC: Execution failed (unclear why)

The classic XOR linked list algorithm (implementing a doubly linked list with only one pointer per node, by storing the XOR of two pointers) also makes essential use of multiple-provenance pointers. In this example we XOR the integer values from two pointers and XOR the result again with one of them.

EXAMPLE (`pointer_offset_xor_global.c`):

```
#include <stdio.h>
#include <inttypes.h>
int x=1;
int y=2;
int main() {
  int *p = &x;
  int *q = &y;
  uintptr_t i = (uintptr_t) p;
  uintptr_t j = (uintptr_t) q;
  uintptr_t k = i ^ j;
  uintptr_t l = k ^ i;
  int *r = (int *)l;
  // are r and q now equivalent?
  *r = 11;     // does this have defined behaviour?
  _Bool b = (r==q);
  printf("x=%i y=%i *r=%i (r==p)=%s\n",x,y,*r,
         b?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`x=1 y=11 *r=11 (r==p)=true`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `unclear - undefined behaviour?`

DEFACTO: `unclear - not really used in practice? Could be defined behaviour in a multiple-provenance semantics`

It is unclear whether this algorithm is important in modern practice. One respondent remarks that the XOR list implementation interacts badly with modern pipelines and the space saving is not a big win.

An automatic storage duration analogue:

EXAMPLE (`pointer_offset_xor_auto.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

`x=1 y=11 *r=11 (r==p)=true`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

2.3.5 Q12. For arithmetic over provenanced integer values, is the provenance of the result invariant under plus/minus associativity?

U:ISO U:DEFACTO

ISO: unclear – we suggest yes? DEFACTO-USAGE: unclear - presume yes DEFACTO-IMPL: unclear - presume yes CERBERUS-DEFACTO: yes CHERI: yes for CHERI256; not always for CHERI128 TIS: no (first test ok; second test fails at the addition of pointers cast to `uintptr_t`) KCC: test not supported (Translation failed; unclear why)

Normal integer arithmetic or modular arithmetic satisfies various algebraic laws, e.g. $a+(b-c) = (a+b)-c$ (which we call “plus/minus associativity”, in the absence of a standard name). Does that still hold for provenanced values? For C pointer arithmetic, addition of two pointers is a type error so there is no re-parenthesised variant of the §2.3.1 (p.15) examples with, e.g.

`(&x([0]) + &(y[1]))-&(y[0])`

(in full: `pointer_arith_algebraic_properties_1_global.c`). But in semantics in which integer values also carry provenance data of some kind, we have the same question for analogous examples that do the arithmetic at `uintptr_t` type, e.g. asking whether the following two programs behave the same:

EXAMPLE (`pointer_arith_algebraic_properties_2_global.c`):

```
#include <stdio.h>
#include <inttypes.h>
int y[2], x[2];
int main() {
  int *p=(int*)(((uintptr_t)&(x[0])) +
    (((uintptr_t)&(y[1]))-((uintptr_t)&(y[0]))));
  *p = 11;  // is this free of undefined behaviour?
  printf("x[1]=%d *p=%d\n",x[1],*p);
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`x[1]=11 *p=11`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (x[1]=11 *p=11)`

ISO: `defined behaviour (x[1]=11 *p=11)`

EXAMPLE (`pointer_arith_algebraic_properties_3_global.c`):

```
#include <stdio.h>
#include <inttypes.h>
int y[2], x[2];
int main() {
  int *p=(int*)(
    (((uintptr_t)&(x[0])) + ((uintptr_t)&(y[1])))
    -((uintptr_t)&(y[0])) );
  *p = 11;  // is this free of undefined behaviour?
  //(equivalent to the &x[0]+(&(y[1])-&(y[0])) version?)
  printf("x[1]=%d *p=%d\n",x[1],*p);
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`x[1]=11 *p=11`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `unclear`

ISO: `unclear`

Analogues with automatic storage duration: `pointer_arith_algebraic_properties_2_auto.c`and `pointer_arith_algebraic_properties_3_auto.c`.

2.3.6 Multiple provenance semantics summarised

2.4 Pointer provenance via pointer representation copying

C permits the representation bytes of objects to be accessed, via `unsigned char` pointers, so whenever we introduce abstract values we have to consider the semantics of reading and writing of the associated representation bytes. In particular, we have to consider when manipulation of pointer value representations produces usable pointers, and with what attached provenance.

2.4.1 Q13. Can one make a usable copy of a pointer by copying its representation bytes using the library memcpy?

ISO: yes (not made explicit in ISO, but surely intended to be yes) DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: Execution failed (unclear why)

EXAMPLE (`pointer_copy_memcpy.c`):

```
#include <stdio.h>
#include <string.h>
int x=1;
int main() {
  int *p = &x;
  int *q;
  memcpy (&q, &p, sizeof p);
```

```
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`*p=11 *q=11`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (*p=11 *q=11)`

ISO: `defined behaviour (*p=11 *q=11)`

This should be allowed in both de facto and ISO semantics.

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

U:ISO

ISO: not explicitly addressed in ISO – we suggest yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: not always TIS: yes KCC: Execution failed (unclear why)

EXAMPLE (`pointer_copy_user_dataflow_direct_bytewise.c`):

```
#include <stdio.h>
#include <string.h>
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?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`*p=11 *q=11`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (*p=11 *q=11)`

ISO: `defined behaviour (*p=11 *q=11)`

This should also certainly be allowed in the de facto semantics. People do reimplement `memcpy`, and we believe this works on most compilers and hardware.

The exceptions we are aware of are capability machines such as CHERI or IBM system 38 and descendents. In CHERI you have to copy pointers at pointer types for it to work properly, but capability loads and stores can operate generically, because the capability registers have tag bits. There is also some new tagged memory support for Oracle Sparc, to find invalid pointers.

Real `memcpy` implementations can be more complex. The `glibc memcpy`[10] involves copying byte-by-byte, as above, and also word-by-word and, using virtual memory manipulation, page-by-page. Word-by-word copying is not permitted by the ISO standard, as it violates the effective type rules, but should be permitted by our de facto semantics. Virtual memory manipulation is outside our scope at present.

2.4.3 Q15. Can one make a usable copy of a pointer by copying its representation bytes by user code that indirectly computes the identity function on those bytes?

U:ISO D:ISO-VS-DEFACTO

ISO: unclear DEFACTO-USAGE: yes DEFACTO-IMPL: yes (presumably...) CERBERUS-DEFACTO: yes CHERI: no TIS: no (fails at the XOR of a pointer representation byte) KCC: Execution failed (unclear why)

[Question 5/15 of our *What is C in practice? (Cerberus survey v2)*[11] relates to this.]

For example, suppose one reads the bytes of a pointer representation pointing to some object, encrypts them, decrypts them, store them as the representation of another pointer value, and tries to access the object. The following code is a simplified version of this, just using a XOR twice; one should imagine a more complex transform, with the transform and its inverse separated in the code and in time so that the compiler cannot analyse them.

EXAMPLE (`pointer_copy_user_dataflow_indirect_bytewise.c`):

```
#include <stdio.h>
#include <string.h>
int  x=1;
void user_memcpy2(unsigned char* dest,
                  unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = ((*src) ^ 1) ^ 1;
    src += 1;
    dest += 1;
    n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;
  user_memcpy2((unsigned char*)&q, (unsigned char*)&p,
             sizeof(p));
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`*p=11 *q=11`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `unclear (*p=11 *q=11)`

---

[10] `https://sourceware.org/git/?p=glibc.git;a=blob;f=string/memcpy.c;hb=HEAD`

[11] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

ISO: `unclear (probably undefined behaviour?)`

It is unclear whether this needs to be or can be allowed. Pages can and do get encrypted and compressed to disc, and a C semantics that dealt with virtual memory would have to support that, but it is not visible from normal C. One would not do this by tracking provenance via the disc, in any case, but instead more like our pointer IO semantics (§2.6, p.24): arbitrary (legal...) pointer values can be read in, and the point is that the compiler has to know that it does not know anything about them. People do sometimes do user-space paging, e.g. in user-space collection classes, but it is not mainstream.

In CHERI you cannot copy pointers in this way, and they haven't yet found code that does this. (If you were copying int-by-int, it would be using the capability-aware instructions, so it would work.) This suggests that we could deem this undefined in the de facto standard, though they have not tried very much code yet.

As for the ISO standard semantics, DR260 is reasonably clear that the first of the three examples is allowed, writing *"Note that using assignment or bitwise copying via `memcpy` or `memmove` of a determinate value makes the destination acquire the same determinate value."*. For the second and third, DR260 is ambiguous: one could read its special treatment of `memcpy` and `memmove`, coupled with its *"[an implementation] may also treat pointers based on different origins as distinct even though they are bitwise identical"* as implying that these have undefined behaviour. On the other hand, the standard's 6.5p6 text on effective types suggests that at least `user_memcpy` (though perhaps not `user_memcpy2`) can copy values of any effective type, including pointers: *"[...] If a value is copied into an object having no declared type using memcpy or memmove,* ***or is copied as an array of character type****, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. [...]"* (bold emphasis added).

2.4.4 Q16. Can one carry provenance through dataflow alone or also through control flow?

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: unclear (not used in normal code?) DEFACTO-IMPL: unclear CERBERUS-DEFACTO: no CHERI: no TIS: no (fails at the switch on a pointer representation byte or bit access – intentionally so, given that this introduces nondeterminism) KCC: Execution failed (unclear why)

Our provenance examples so far have all only involved dataflow; we also have to ask if a usable pointer can be constructed via non-dataflow control-flow paths.

For example, consider a version of the previous indirect `memcpy` example (§2.4.3, p.20) with a control-flow choice on the value of the bytes:

EXAMPLE (pointer_copy_user_ctrlflow_bytewise.c):

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <limits.h>
int  x=1;
unsigned char control_flow_copy(unsigned char c) {
  assert(UCHAR_MAX==255);
  switch (c) {
  case 0: return(0);
  case 1: return(1);
  case 2: return(2);
  ...
  case 255: return(255);
  }
}
void user_memcpy2(unsigned char* dest,
                  unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = control_flow_copy(*src);
    src += 1;
    dest += 1;
    n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;
  user_memcpy2((unsigned char*)&q, (unsigned char*)&p,
            sizeof(p));
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
*p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING:

```
pointer_copy_user_ctrlflow_bytewise.c:266:1: warning:
control may reach end of non-void function
[-Wreturn-type]
}
^
1 warning generated.
*p=11 *q=11
```

DEFACTO: `undefined behaviour`

ISO: `unclear (probably undefined behaviour?)`

Similarly, one can imagine copying a pointer via `uintptr_t` bit-by-bit via a control-flow choice for each bit (adapting `provenance_basic_using_intptr_t_global_yx.c` from §2.2.3 (p.12)):

EXAMPLE (pointer_copy_user_ctrlflow_bitwise.c):

```
#include <stdio.h>
#include <inttypes.h>
#include <limits.h>
int  x=1;
int main() {
  int *p = &x;
  uintptr_t i = (uintptr_t)p;
  int uintptr_t_width = sizeof(uintptr_t) * CHAR_BIT;
  uintptr_t bit, j;
  int k;
  j=0;
  for (k=0; k<uintptr_t_width; k++) {
    bit = (i & (((uintptr_t)1) << k)) >> k;
```

```
    if (bit == 1)
      j = j | ((uintptr_t)1 << k);
    else
      j = j;
  }
  int *q = (int *)j;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
*p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING:

```
pointer_copy_user_ctrlflow_bitwise.c:17:9: warning:
explicitly assigning value of variable of type
'uintptr_t' (aka 'unsigned long') to itself
[-Wself-assign]
 j = j;
   ^
1 warning
generated.
*p=11 *q=11
```

DEFACTO: `undefined behaviour`

ISO: `unclear (probably undefined behaviour?)`

as opposed to a similar bitwise example with a dataflow path for each bit:

EXAMPLE (`pointer_copy_user_dataflow_direct_bitwise.c`):

```
#include <stdio.h>
#include <inttypes.h>
#include <limits.h>
int  x=1;
int main() {
  int *p = &x;
  uintptr_t i = (uintptr_t)p;
  int uintptr_t_width = sizeof(uintptr_t) * CHAR_BIT;
  uintptr_t bit, j;
  int k;
  j=0;
  for (k=0; k<uintptr_t_width; k++) {
    bit = (i & (((uintptr_t)1) << k)) >> k;
    j = j | (bit << k);
  }
  int *q = (int *)j;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
*p=11 *q=11
```

CLANG36-O2-NO-STRICT-ALIASING: ... as above

DEFACTO: `defined behaviour`

ISO: `unclear (probably undefined behaviour?)`

Finally, contrasting with the first two examples above, that recover all the concrete value information of the original pointer, we can consider a variant of the §2.1.1 (p.7) `provenance_basic_using_intptr_t_global_yx.c` example in which there is a control-flow choice based on partial information of the intended target pointer (here just whether q is null) and the concrete value information is obtained otherwise:

EXAMPLE (`provenance_basic_mixed_global_offset+4.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int  y = 2, x=1;
int main() {
  intptr_t ux = (intptr_t)&x;
  intptr_t uy = (intptr_t)&y;
  intptr_t offset = 4;
  printf("Addresses: &x=%"PRIiPTR" &y=%"PRIiPTR\
         "\n",ux,uy);
  int *q = &y;
  if (q != NULL) {
    int *p = (int *)(ux + offset);
    if (memcmp(&p, &q, sizeof(p)) == 0) {
      *p = 11; // is this free of undefined behaviour?
      printf("x=%d  y=%d  *p=%d  *q=%d\n",x,y,*p,*q);
    }
  }
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=6294488 &y=6294492
x=1 y=2 *p=11 *q=2
```

DEFACTO: `undefined behaviour`

ISO: `unclear (probably undefined behaviour?)`

EXAMPLE (`provenance_basic_mixed_global_offset-4.c`):

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int  y = 2, x=1;
int main() {
  intptr_t ux = (intptr_t)&x;
  intptr_t uy = (intptr_t)&y;
  intptr_t offset = -4;
  printf("Addresses: &x=%"PRIiPTR" &y=%"PRIiPTR\
         "\n",ux,uy);
  int *q = &y;
  if (q != NULL) {
    int *p = (int *)(ux + offset);
    if (memcmp(&p, &q, sizeof(p)) == 0) {
      *p = 11; // is this free of undefined behaviour?
      printf("x=%d  y=%d  *p=%d  *q=%d\n",x,y,*p,*q);
    }
  }
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: &x=6294488 &y=6294492
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: &x=6294232 &y=6294228
x=1 y=11 *p=11 *q=11
```

The test suite also includes variant `provenance_basic_mixed_global_offset-64.c` and, with automatic storage duration: `provenance_basic_mixed_auto_offset+4.c`, `provenance_basic_mixed_auto_offset-4.c`, and `provenance_basic_mixed_auto_offset-64.c`.

2.5 Pointer provenance and union type punning

Type punning via unions, as discused in §2.15.4 (p.38), gives an additional way of constructing pointer values, and so we have to consider how that interacts with the pointer provenance semantics.

2.5.1 Q17. Is type punning between integer and pointer values allowed?

U:ISO U:DEFACTO
ISO: unclear DEFACTO-USAGE: unclear – impl-def or yes? DEFACTO-IMPL: unclear – impl-def or yes? CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes

The following example (analogous to the roundtrip-via-`uintptr_t` example `provenance_roundtrip_via_intptr_t.c` of §2.2.1 (p.10)) constructs a pointer by casting a pointer to `uintptr_t`, storing that in a member of a union of that type, and then reading from a member of the union of pointer type.

EXAMPLE (`provenance_union_punning_1_global.c`):

```
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
int x=1;
typedef union { uintptr_t ui; int *p; } un;
int main() {
  un u;
  int *px = &x;
  uintptr_t i = (uintptr_t)px;
  u.ui = i;
  int *p = u.p;
  printf("Addresses: p=%p &x=%p\n",(void*)p,(void*)&x);
  *p = 11;  // is this free of undefined behaviour?
  printf("x=%d *p=%d\n",x,*p);
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600b40 &x=0x600b40
x=11 *p=11
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above (modulo addresses)

DEFACTO: `implementation-defined`

ISO: `unclear`

It is unclear whether this should be guaranteed to work. The ISO standard (see §2.15.4, p.38) says *"the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type"*, but says little about that reinterpretation. In GCC and Clang it appears to: the above prints `x=11 *p=11` suggesting that there the two types do have compatible representations, at least. What alias analysis might be assuming about this situation is unclear to us.

One systems researcher said that it is fairly common for implementations to satisfy this and for programmers to exploit it, though more hygienic C would include an explicit cast.

2.5.2 Q18. Does type punning between integer and pointer values preserve provenance?

U:ISO
ISO: unclear DEFACTO-USAGE: presume yes DEFACTO-IMPL: presume yes CERBERUS-DEFACTO: yes CHERI: yes TIS: example not supported (`memcmp` of pointer representations) KCC: Execution failed (unclear why)

For consistency with the rest of the provenance-tracking semantics, we imagine that at least the following example (analogous to the pathological `provenance_basic_global_yx.c` of §2.1.1 (p.7) but indirected via type punning) should have undefined behaviour:

EXAMPLE (`provenance_union_punning_2_global_yx.c`):

```
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
int  y=2, x=1;
typedef union { uintptr_t ui; int *p; } un;
int main() {
  un u;
  int *px = &x;
  uintptr_t i = (uintptr_t)px;
  i = i + sizeof(int);
  u.ui = i;
  int *p = u.p;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  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);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600bd4 q=0x600bd4
x=1 y=2 *p=11 *q=2
```

ISO: `unclear`

DEFACTO: `undefined behaviour`

EXAMPLE (`provenance_union_punning_2_global_xy.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600bd8 q=0x600bd0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600ad0 q=0x600ad0
x=1 y=11 *p=11 *q=11
```

EXAMPLE (`provenance_union_punning_2_auto_xy.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600bd8 q=0x600bd0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600ad0 q=0x600ad0
x=1 y=11 *p=11 *q=11
```

A semantics that tracks provenance on integer values in memory will naturally do that.

Here GCC exhibits the otherwise-unsound optimisation, printing `x=1 y=2 *p=11 *q=2`.

2.6 Pointer provenance via IO

2.6.1 Q19. Can one make a usable pointer via IO?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: no TIS: test not supported (`fopen` library call) KCC: Execution failed (unclear why)

We now consider the extreme example of pointer provenance flowing via IO, if one writes the address of an object to a file and reads it back in. We give three versions: one using `fprintf`/`fscanf` and the `%p` format, one using `fwrite`/`fread` on the pointer representation bytes, and one converting the pointer to and from `uintptr_t` and using `fprintf`/`fscanf` on that value with the `PRIuPTR`/`SCNuPTR` formats. The first gives a syntactic indication of a potentially escaping pointer value, while the others (after preprocessing) do not.

EXAMPLE (`provenance_via_io_percentp_global.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  FILE *f = fopen(
    "provenance_via_io_percentp_global.tmp","w+b");
  printf("Addresses: p=%p\n",(void*)p);
  // print pointer address to a file
  fprintf(f,"%p\n",(void*)p);
  rewind(f);
  void *rv;
  int n = fscanf(f,"%p\n",&rv);
  int *r = (int *)rv;
  if (n != 1) exit(EXIT_FAILURE);
  printf("Addresses: r=%p\n",(void*)r);
  // are r and p now equivalent?
  *r=12; // is this free of undefined behaviour?
  _Bool b1 = (r==p); // do they compare equal?
  _Bool b2 = (0==memcmp(&r,&p,sizeof(r)));//same reps?
  printf("x=%i *r=%i b1=%s b2=%s\n",x,*r,
         b1?"true":"false",b2?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600df0
Addresses: r=0x600df0
x=12 *r=12 b1=true b2=true
```

CLANG36-O2-NO-STRICT-ALIASING: ...as above (modulo addresses)

ISO: `defined behaviour`

EXAMPLE (`provenance_via_io_bytewise_global.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  FILE *f = fopen(
    "provenance_via_io_bytewise_global.tmp","w+b");
  printf("Addresses: p=%p\n",(void*)p);
  // output pointer address to a file
  int nw = fwrite(&p, 1, sizeof(int *), f);
  if (nw != sizeof(int *)) exit(EXIT_FAILURE);
  rewind(f);
  int *r;
  int nr = fread(&r, 1, sizeof(int *), f);
  if (nr != sizeof(int *)) exit(EXIT_FAILURE);
  printf("Addresses: r=%p\n",(void*)r);
  // are r and p now equivalent?
  *r=12; // is this free of undefined behaviour?
  _Bool b1 = (r==p); // do they compare equal?
  _Bool b2 = (0==memcmp(&r,&p,sizeof(r)));//same reps?
  printf("x=%i *r=%i b1=%s b2=%s\n",x,*r,
         b1?"true":"false",b2?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600e08
Addresses: r=0x600e08
x=12 *r=12 b1=true b2=true
```

CLANG36-O2-NO-STRICT-ALIASING: ...as above (modulo addresses)

ISO: `defined behaviour`

EXAMPLE (`provenance_via_io_uintptr_t_global.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  uintptr_t i = (uintptr_t) p;
  FILE *f = fopen(
    "provenance_via_io_uintptr_t_global.tmp","w+b");
  printf("Addresses: i=%"PRIuPTR" \n",i);
  // print pointer address to a file
  fprintf(f,"%"PRIuPTR"\n",i);
  rewind(f);
  uintptr_t k;
  // read a pointer address from the file
  int n = fscanf(f,"%"SCNuPTR"\n",&k);
  if (n != 1) exit(EXIT_FAILURE);
  printf("Addresses: k=%"PRIuPTR"\n",k);
  int *r = (int *)k;
  // are r and q now equivalent?
  *r=12; // is this free of undefined behaviour?
  _Bool b1 = (r==p); // do they compare equal?
  _Bool b2 = (0==memcmp(&r,&p,sizeof(r)));//same reps?
  printf("x=%i *r=%i b1=%s b2=%s\n",x,*r,
         b1?"true":"false",b2?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: i=6295040
Addresses: k=6295040
x=12 *r=12 b1=true b2=true
```

CLANG36-O2-NO-STRICT-ALIASING: …as above (modulo addresses)
ISO: `defined behaviour`

This is used in practice: in graphics code for marshalling/unmarshalling, at least using `%p`, and `SCNuPTR` and suchlike are used in xlib. Debuggers do this kind of thing too.

In the ISO standard, the standard text for `fprintf` and `scanf` for `%p` say that this should work: *"If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the* `%p` *conversion is undefined."* (modulo the usual remarks about "compare equal"), and the text for `uintptr_t` and the presence of `SCNuPTR` in `inttypes.h` implies the same there.

2.7 Q20. Can one make a usable pointer from a concrete address (of device memory)?

U:ISO
ISO: unclear DEFACTO-USAGE: yes (at least in embedded) DEFACTO-IMPL: yes (at least in embedded) CERBERUS-DEFACTO: yes (for implementation-defined device-memory addresses) CHERI: no TIS: test not informative (but correctly detects UB for the out-of-bounds write) KCC: Segmentation fault

C programs should normally not form pointers from particular concrete addresses. For example, the following should normally be considered to have undefined behaviour, as address `0xABC` might not be mapped or, if it is, might alias with other data used by the runtime. By the ISO standard it does have undefined behaviour. Cyclone did not aim to support it (this example is adapted from [19, Ch. 2]). Note that our experimental data is (as usual) for execution in a user-space process in a system with virtual memory, for which that address is presumably not mapped to anything sensible, so one would not expect it to work; they just illustrate how and where the failure is detected.

EXAMPLE (`pointer_from_concrete_address_1.c`):

```
int main() {
  // on systems where 0xABC is not a legal non-stack/heap
  // address, does this have undefined behaviour?
  *((int *)0xABC) = 123;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:
CLANG36-O2-NO-STRICT-ALIASING: …as above
ISO: `undefined behaviour`
DEFACTO: `implementation-defined whether undefined-behaviour or not`

But in some circumstances it is idiomatic to use concrete addresses in C to access memory-mapped devices. For example, ARM documentation[12] states *"In most ARM embedded systems, peripherals are located at specific addresses in memory. It is often convenient to map a C variable onto each register of a memory-mapped peripheral, and then read/write the register via a pointer. [...] The simplest way to implement memory-mapped variables is to use pointers to fixed addresses. If the memory is changeable by 'external factors' (for example, by some hardware), it must be labelled as volatile."* with an example similar to the following.

EXAMPLE (`pointer_from_concrete_address_2.c`):

```
#define PORTBASE 0x40000000
unsigned int volatile * const port =
  (unsigned int *) PORTBASE;
int main() {
  unsigned int value = 0;
  // on systems where PORTBASE is a legal non-stack/heap
  // address, does this have defined behaviour?
  *port = value; /* write to port */
  value = *port; /* read from port */
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:
CLANG36-O2-NO-STRICT-ALIASING: …as above
ISO: `undefined behaviour`
DEFACTO: `implementation-defined whether undefined-behaviour or not`

2.8 Pointer provenance for other allocators

ISO C has a distinguished `malloc`, but operating system kernels have multiple allocators, e.g. the FreeBSD and Linux per-CPU allocators mentioned earlier. GCC has a function attribute `__attribute__((malloc))` documented with:

"This tells the compiler that a function is `malloc`*-like, i.e., that the pointer P returned by the function cannot alias any other pointer valid when the function returns, and moreover no pointers to valid objects occur in any storage addressed by P. Using this attribute can improve optimization. Functions like* `malloc` *and* `calloc` *have this property because they return a pointer to uninitialized or zeroed-out storage. However, functions like* `realloc` *do not have this property, as they can return a pointer to storage containing pointers."* (`https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html`).

Ideally a de facto semantics would be able to treat all `malloc`-like functions uniformly; we do not currently support this. Do compilers special-case `malloc` in any way beyond what that text says?

---

[12] *Placing C variables at specific addresses to access memory-mapped peripherals*, ARM Technical Support Knowledge Articles, `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3750.html`

2.9 Stability of pointer values

2.9.1 Q21. Are pointer values stable?

ISO: yes (modulo GCC debate) DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: Execution failed (unclear why)

We assume, in both de facto and ISO standard semantics, that pointer values are stable over time, as are the results of comparisons of them (modulo nondeterministic choices as to whether their provenance is taken into account in those comparisons).

This follows our understanding of normal implementations and our reading of the ISO standard, which says (6.2.4p2): *"[...] An object exists, has a constant address, 33) and retains its last-stored value throughout its lifetime. [...]"* where footnote 33 is: *"The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal. The address may be different during two different executions of the same program."*. Though note that this is contrary to one interpretation of the standard in a response to the GCC bug report mentioned above. It rules out C implementations using a moving garbage collector.

For example, we believe the following should be guaranteed to print `true`:

EXAMPLE (`pointer_stability_1.c`):

```
#include <stdio.h>
#include <inttypes.h>
int main() {
  int x=1;
  uintptr_t i = (uintptr_t) &x;
  uintptr_t j = (uintptr_t) &x;
  // is this guaranteed to be true?
  _Bool b = (i==j);
  printf("(i==j)=%s\n",b?"true":"false");
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
(i==j)=true
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour ((i==j)=true)`

ISO: `defined behaviour ((i==j)=true) (though debated)`

(`pointer_stability_2.c` and `pointer_stability_3.c` are similar but with the equality at pointer type and with a pointer representation equality, respectively.)

2.10 Pointer Equality Comparison (with == and `!=`)

There are several notions of pointer equality which would coincide in a completely concrete semantics but which in a provenance-aware semantics can differ:

(a) comparison with ==

(b) comparison of their representations, e.g. with `memcmp`

(c) accessing the same memory

(d) giving rise to equally defined or undefined behaviour

(e) equivalent as far as alias analysis is concerned

As we note elsewhere, the standard appears to use *"compare equal"* to imply that the pointers are equally usable, but that is not the case. Our first examples show cases where two pointers are `memcmp`-equal but ==-unequal, and where they are `memcmp`- or ==-equal but accessing them is not equally defined.

Jones [24] mentions some architectures, now more-or-less exotic, in which (b) may not hold.

We say that two pointer values are *equivalent* if they are interchangeable, satisfying all of (a–e). And we say that a pointer value is *usable* if accesses using it access the right memory and do not give rise to undefined behaviour.

2.10.1 Q22. Can one do == comparison between pointers to objects of non-compatible types?

U:DEFACTO D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: unclear – should be impl-def? DEFACTO-IMPL: unclear – should be impl-def? CERBERUS-DEFACTO: yes CHERI: under debate TIS: yes KCC: yes

[Question 6/15 of our *What is C in practice? (Cerberus survey v2)*[13] relates to this.]

As we noted in §2.1.3 (p.10), the ISO standard explicitly permits == comparison between pointers to different objects of compatible types. 6.5.9 *Equality operators* allows comparison between any two pointers if

- *"both operands are pointers to qualified or unqualified versions of compatible types;"*
- *"one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of void; or"*
- *"one operand is a pointer and the other is a null pointer constant."*

As we saw in §2.1.2 (p.9), pointer comparison with == should be nondeterministically allowed to be provenance-aware or not.

It is not clear whether the restriction to compatible types is needed for typical modern implementations. It is also not clear whether == comparison between pointers to non-compatible types is used in practice, and similarly below for relational comparison with < etc.

For the following, GCC and Clang both give warnings; GCC says that this comparison without a cast is enabled by default, perhaps suggesting that it is used in the de facto standard corpus of code and hence that our de facto standard semantics should allow it.

---

[13] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

EXAMPLE (pointer_comparison_eq_1_global.c):

```
#include <stdio.h>
#include <string.h>
int  x=1;
float f=1.0;
int main() {
  int *p = &x;
  float *q = &f;
  _Bool b = (p == q); // free of undefined behaviour?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_1_global.c: In function 'main':
pointer_comparison_eq_1_global.c:8:16: warning:
comparison of distinct pointer types lacks a cast

_Bool b = (p == q); // free of undefined behaviour?

 ^
(p==q) = false
```

CLANG36-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_1_global.c:8:16: warning:
comparison of distinct pointer types ('int *' and 'float
*') [-Wcompare-distinct-pointer-types]
 _Bool b = (p ==
q); // free of undefined behaviour?
   ^

1 warning generated.
(p==q) = false
```

DEFACTO: `implementation-defined`

ISO: `undefined behaviour`

EXAMPLE (pointer_comparison_eq_1_auto.c):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_1_auto.c: In function 'main':
pointer_comparison_eq_1_auto.c:8:16: warning: comparison
of distinct pointer types lacks a cast
 _Bool b = (p
== q); // free of undefined behaviour?

^
(p==q) = false
```

CLANG36-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_1_auto.c:8:16: warning: comparison
of distinct pointer types ('int *' and 'float *')
[-Wcompare-distinct-pointer-types]
 _Bool b = (p == q);
// free of undefined behaviour?
   ^
1
warning generated.
(p==q) = false
```

DEFACTO: `implementation-defined`

ISO: `undefined behaviour`

Compilers might conceivably optimise such comparisons (between pointers of non-compatible type) to `false`, but the following example shows that (at least in this case) GCC does not:

EXAMPLE (pointer_comparison_eq_2_global.c):

```
#include <stdio.h>
#include <string.h>
int  x=1;
float f=1.0;
int main() {
  int *p = (int *)&f;
  float *q = &f;
  _Bool b = (p == q); // free of undefined behaviour?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_2_global.c: In function 'main':
pointer_comparison_eq_2_global.c:8:16: warning:
comparison of distinct pointer types lacks a cast

_Bool b = (p == q); // free of undefined behaviour?

 ^
(p==q) = true
```

CLANG36-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_2_global.c:8:16: warning:
comparison of distinct pointer types ('int *' and 'float
*') [-Wcompare-distinct-pointer-types]
 _Bool b = (p ==
q); // free of undefined behaviour?
   ^

1 warning generated.
(p==q) = true
```

DEFACTO: `implementation-defined`

ISO: `undefined behaviour`

EXAMPLE (pointer_comparison_eq_2_auto.c):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_2_auto.c: In function 'main':
pointer_comparison_eq_2_auto.c:8:16: warning: comparison
of distinct pointer types lacks a cast
 _Bool b = (p
== q); // free of undefined behaviour?

^
(p==q) = true
```

CLANG36-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_2_auto.c:8:16: warning: comparison
of distinct pointer types ('int *' and 'float *')
[-Wcompare-distinct-pointer-types]
 _Bool b = (p == q);
```

```
// free of undefined behaviour?
   ^
1
warning generated.
(p==q) = true
```

DEFACTO: `implementation-defined`

ISO: `undefined behaviour`

2.10.2 Q23. Can one do == comparison between pointers (to objects of compatible types) with different provenances that are not strictly within their original allocations?

ISO: yes DEFACTO-USAGE: unclear how much this is used DEFACTO-IMPL: yes (modulo §2.1.3 discussion) CERBERUS-DEFACTO: yes CHERI: ? TIS: fails with pointer_comparable, as expected KCC: yes

EXAMPLE (`klw-itp14-2.c`):

```
#include <stdio.h>
int  x=1, y=2;
int main() {
  int *p = &x + 1;
  int *q = &y;
  _Bool b = (p == q); // free of undefined behaviour?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
(p==q) = false
```

CLANG36-O2-NO-STRICT-ALIASING:

```
(p==q) = true
```

This example is from Krebbers et al. [32], as we discuss in §6.7. Their model forbids this, while our candidate de facto model should allow arbitrary pointer comparison.

2.10.3 Q24. Can one do == comparison of a pointer and `(void*)-1`?

U:ISO

ISO: unclear DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: ? TIS: fails with pointer_comparable (but needed for sqlite?) KCC: yes

EXAMPLE (`besson_blazy_wilke_6.2.c`):

```
#include <stdlib.h>
int main() {
  void *p = malloc(sizeof(int));
  _Bool b = (p == (void*)-1); // defined behaviour?
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `unclear`

This is from Besson et al. [9], as we discuss in §6.8. Their §6.2 notes that system calls such as `mmap` return `-1` on error, and so one must be able to compare pointers against `-1`. Our test uses `malloc` as the source of the pointer, just to avoid dependence on `sys/mman.h`, even though `malloc` should not return `-1`. Their model permits the `mmap` analogue of this, apparently by building in the fact that `mmap` should return aligned values.

John Regehr observes that sqlite also compares against -2 and other error codes.

In a semantics in which == might respect provenance, both -1 values should be constructed in a provenance-free fashion, otherwise such a comparison might mistakenly give false.

2.11 Pointer Relational Comparison (with <, >, <=, or >=)

Here the ISO standard seems to be significantly more restrictive than common practice. First, there is a type constraint, as for ==: 6.5.8p2 *"both operands are pointers to qualified or unqualified versions of compatible object types."*.

Then 6.5.8p5 allows comparison of pointers only to the same object (or one-past) or to members of the same array, structure, or union: 6.5.8p5 *"When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If two pointers to object types both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression `P` points to an element of an array object and the expression `Q` points to the last element of the same array object, the pointer expression `Q+1` compares greater than `P`. In all other cases, the behavior is undefined."*

(Similarly to 6.5.6p7 for pointer arithmetic, 6.5.8p4 treats all non-array element objects as arrays of size one for this: 6.5.8p4 *"For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type."*)

This rules out the following comparisons, between pointers to two separately allocated objects and between a pointer to a structure member and one to a sub-member of another member, but some of these seem to be relied upon in practice.

2.11.1 Q25. Can one do relational comparison (with <, >, <=, or >=) of two pointers to separately allocated objects (of compatible object types)?

D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: impl-def or yes? DEFACTO-IMPL: impl-def or yes? CERBERUS-DEFACTO: yes

CHERI: yes TIS: no (fails with `pointer_comparable`, intentionally) KCC: no (flags UB)

[Question 7/15 of our *What is C in practice? (Cerberus survey v2)*[14] relates to this.]

Example (`pointer_comparison_rel_1_global.c`):

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

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600b68 q=0x600b6c
(p<q) = true (p>q) = false
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x600ad0 q=0x600acc
(p<q) = false (p>q) = true
```

DEFACTO: `defined behaviour`

ISO: `undefined behaviour`

And with automatic storage duration:

Example (`pointer_comparison_rel_1_auto.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x7fffffffe9ec q=0x7fffffffe9e8
(p<q) = false (p>q) = true
```

CLANG36-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x7fffffffe9e8 q=0x7fffffffe9ec
(p<q) = true (p>q) = false
```

DEFACTO: `defined behaviour`

ISO: `undefined behaviour`

In practice, comparison of pointers to different objects seems to be used heavily, e.g. in memory allocators and for a lock order in Linux, and we believe the de facto semantics should allow it, leaving aside segmented architectures. Though one respondent reported for `pointer_comparison_rel_1_global.c`: *“May produce inconsistent results in practice if p and q straddle the exact middle of the address space. We’ve run into practical problems with this. Cast to `intptr_t` first in the rare case you really need it.”*.

2.11.2 Q26. Can one do relational comparison (with <, >, <=, or >=) of a pointer to a structure member and one to a sub-member of another member, of compatible object types?

U:ISO D:ISO-VS-DEFACTO

ISO: unclear - no? (subject to interpretation) DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: Execution failed (unclear why)

Example (`pointer_comparison_rel_substruct.c`):

```
#include <stdio.h>
typedef struct { int i1; float f1; } st1;
typedef struct { int i2; st1 s2; } st2;
int main() {
  st2 s = {.i2=2, .s2={.i1=1, .f1=1.0 } };
  int *p = &(s.i2), *q = &(s.s2.i1);
  _Bool b = (p < q); // does this have defined behaviour?
  printf("Addresses: p=%p  q=%p\n",(void*)p,(void*)q);
  printf("(p<q) = %s\n", b?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p=0x7fffffffe9e0 q=0x7fffffffe9e4
(p<q) = true
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (true)`

ISO: `undefined behaviour?`

Whether this is allowed in the ISO standard depends on one’s interpretation of 6.5.8p5 *“If the objects pointed to are members of the same aggregate object”*. A literal reading suggests that it is not, as the object pointed to by q is not a member of the struct, but merely a part of a member of it.

2.11.3 Q27. Can one do relational comparison (with <, >, <=, or >=) of pointers to two members of a structure that have incompatible types?

U:DEFACTO D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: unclear - should be impl-def? DEFACTO-IMPL: unclear - should be impl-def? CERBERUS-DEFACTO: yes CHERI: under debate TIS: yes KCC: Execution failed (unclear why)

The ISO standard constraint also rules out comparison of pointers to two members of a structure with different types:

Example (`pointer_comparison_rel_different_type_members.c`):

```
#include <stdio.h>
typedef struct { int i; float f; } st;
int main() {
  st s = {.i=1, .f=1.0 };
  int *p = &(s.i);
  float *q = &(s.f);
  _Bool b = (p < q); // does this have defined behaviour?
  printf("Addresses: p=%p  q=%p\n",(void*)p,(void*)q);
  printf("(p<q) = %s\n", b?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
pointer_comparison_rel_different_type_members.c: In
function ’main’:
pointer_comparison_rel_different_type_m
embers.c:7:16: warning: comparison of distinct pointer
types lacks a cast
 _Bool b = (p < q); // does this
have defined behaviour?
 ^
```

---

[14] www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html

```
Addresses: p=0x7fffffffe9d0 q=0x7fffffffe9d4
(p<q) = true
```

CLANG36-O2-NO-STRICT-ALIASING:

```
pointer_comparison_rel_different_type_members.c:7:16:
warning: comparison of distinct pointer types ('int *'
and 'float *') [-Wcompare-distinct-pointer-types]

_Bool b = (p < q); // does this have defined behaviour?

   ^
1 warning generated.
Addresses: p=0x7fffffffe9d8 q=0x7fffffffe9dc
(p<q) = true
```

DEFACTO: `implementation-defined`
ISO: `undefined behaviour`

As for == comparison (`pointer_comparison_eq_1_global.c`, §2.10.1, p.26), this is presumably to let implementations use different representations for pointers to different types. In practice GCC gives the same warning, `comparison of distinct pointer types lacks a cast [enabled by default]`, which weakly implies that this is used in practice and that our de facto semantics should allow it.

2.12 Null pointers

2.12.1 Q28. Can one make a null pointer by casting from a non-constant integer expression?

D:ISO-VS-DEFACTO
ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: yes (modulo segmented or multiple-address-space architectures) CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes

[Question 12/15 of our *What is C in practice? (Cerberus survey v2)*[15] relates to this.]

The standard permits the construction of null pointers by casting from integer constant zero expressions, but not from other integer values that happen to be zero (6.3.2.3p3): *"An integer constant expression with the value 0, or such an expression cast to type `void *`, is called a* null pointer constant.*66) If a null pointer constant is converted to a pointer type, the resulting pointer, called a* null pointer*, is guaranteed to compare unequal to a pointer to any object or function. 66) The macro NULL is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.19."*

EXAMPLE (`null_pointer_1.c`):

```
#include <stdio.h>
#include <stddef.h>
#include <assert.h>
int y=0;
int main() {
  assert(sizeof(long)==sizeof(int*));
  long x=0;
  int *p = (int *)x;
  // is the value of p a null pointer?
  _Bool b1 = (p == NULL);// guaranteed to be true?
  _Bool b2 = (p == &y);  // guaranteed to be false?
  printf("(p==NULL)=%s  (p==&y)=%s\n", b1?"true":"false",
         b2?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
(p==NULL)=true (p==&y)=false
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above
DEFACTO: `implementation-defined (typically true/false)`
ISO: `defined behaviour (nondeterministic results)?`

The situation in practice is not completely clear. The CHERI ASPLOS paper observes that "this distinction is difficult to support in modern compilers" and points to an LLVM mailing list thread[16] that suggests that lots of code depends on being able to form null pointers from non-constant zero expressions. The `comp.lang.c` FAQ[17] has an example claimed to show that in some cases the compiler will get it wrong if not given an explicit cast, but this is essentially just telling the compiler the right type. It would be useful to know of any current platforms in which the NULL pointer isn't represented with a zero value (perhaps embedded systems?).

2.12.2 Q29. Can one assume that all null pointers have the same representation?

D:ISO-VS-DEFACTO
ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: yes (modulo segmented or multiple-address-space architectures) CERBERUS-DEFACTO: iff the implementation-defined set of null pointer values is a singleton CHERI: yes? TIS: yes KCC: Execution failed (unclear why)

6.3.2.3p3 says this for == comparison: *"Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal."* but leaves open whether they have the same representation bytes.

EXAMPLE (`null_pointer_2.c`):

```
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <assert.h>
int y=0;
int main() {
  assert(sizeof(int*)==sizeof(char*));
  int *p = NULL;
  char *q = NULL;
  // are two null pointers guaranteed to have the
  //  same representation?
  _Bool b = (memcmp(&p, &q, sizeof(p))==0);
  printf("p=%p q=%p\n",(void*)p,(void*)q);
  printf("%s\n",b?"equal":"unequal");
}
```

---

[15] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

[16] `http://lists.cs.uiuc.edu/pipermail/llvmdev/2015-January/080288.html`

[17] `http://c-faq.com/null/null2.html`

GCC-5.3-O2-NO-STRICT-ALIASING:

```
p=0x0 q=0x0
equal
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `implementation-defined (typically equal)`

ISO: `defined behaviour but nondeterministic results? Should be an implementation-defined set of null-pointer representations`

A de facto semantics could base this on the implementation-defined set of null-pointer values. Or, even more simply and consistent with the desire for `calloc` to initialise memory that will be used as pointer values to the representation of NULL, just fix on zero.

2.12.3 Q30. Can null pointers be assumed to have all-zero representation bytes?

D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: yes (modulo segmented or multiple-address-space architectures) CERBERUS-DEFACTO: iff the implementation-defined set of null pointer values contains just zero CHERI: yes TIS: yes KCC: Execution failed (unclear why)

[Question 13/15 of our *What is C in practice? (Cerberus survey v2)*[18] relates to this.]

EXAMPLE (`null_pointer_3.c`):

```
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <stdlib.h>
int y=0;
int main() {
  int *p = NULL;
  int **q = (int **) calloc(1,sizeof(int*));
  // is this guaranteed to be true?
  _Bool b = (memcmp(&p, q, sizeof(p))==0);
  printf("%s\n",b?"zero":"nonzero");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
zero
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `implementation-defined (typically zero)`

ISO: `defined behaviour but nondeterministic results`

2.13 Pointer Arithmetic

The ISO standard permits only very limited pointer arithmetic, restricting the formation of pointer values.

First, there is arithmetic within an array: 6.5.6 *Additive operators* (6.5.6p{8,9}) permits one to add a pointer and integer (or subtract an integer from a pointer) only within the start and one past the end of an array object, inclusive. 6.5.6p7 adds *"For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type."*. Subtraction of two pointers is permitted only if both are in a similar range (and only if the result is representable in the result type).

Second, 6.3.2.3p7 says that one can do pointer arithmetic on character-type pointers to access representation bytes: *"[...] When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object."*.

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

U:DEFACTO D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: yes sometimes DEFACTO-IMPL: yes sometimes but not in general CERBERUS-DEFACTO: yes CHERI: yes in 256-bit CHERI, not always in 128-bit CHERI TIS: yes for first test; correctly found a bug in mis-edited second test KCC: no (flags UB at pointer arithmetic)

[Question 9/15 of our *What is C in practice? (Cerberus survey v2)*[19] relates to this.]

In practice it seems to be common to transiently construct out-of-bounds pointer values, e.g. with `(px +11) -10` rather than `px + (11-10)`, as below, and we are not aware of examples where this will go wrong in standard implementations, at least for small deltas. There are cases where pointer arithmetic subtraction can overflow[20]. There might conceivably be an issue on some platforms if the transient value is not aligned and only aligned values are representable at the particular pointer type, or if the hardware is doing bounds checking, but both of those seem exotic at present. There are also cases where pointer arithmetic might wrap at values less than the obvious word size, e.g. for "near" or "huge" pointers on 8086 [53, §2.4], but it is not clear if any of these are current. We give examples involving pointers to an integer array and to representation bytes, and with both addition and subtraction.

EXAMPLE (`cheri_03_ii.c`):

```
#include <stdio.h>
int main() {
  int x[2];
  int *p = &x[0];
  //is this free of undefined behaviour?
  int *q = p + 11;
  q = q - 10;
  *q = 1;
  printf("x[1]=%i  *q=%i\n",x[1],*q);
```

---

[18] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

[19] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

[20] `http://sourceforge.net/p/png-mng/mailman/png-mng-implement/?viewmonth=201511`

```
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
x[1]=1 *q=1
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour`

ISO: `undefined behaviour`

EXAMPLE (`cheri_03_ii_char.c`):

```
#include <stdio.h>
int main() {
  unsigned char x;
  unsigned char *p = &x;
  //is this free of undefined behaviour?
  unsigned char *q = p + 11;
  q = q - 10;
  *q = 1;
  printf("x=0x%x  *p=0x%x  *q=0x%x\n",x,*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
cheri_03_ii_char.c: In function 'main':
cheri_03_ii_char.c:9:3: warning: 'x' is used
uninitialized in this function [-Wuninitialized]

printf("x=0x%x *p=0x%x *q=0x%x\n",x,*p,*q);
 ^
x=0x0 *p=0x0 *q=0x1
```

CLANG36-O2-NO-STRICT-ALIASING:

```
x=0x0 *p=0x0 *q=0x0
```

DEFACTO: `defined behaviour`

ISO: `undefined behaviour`

This is the II *invalid intermediate* idiom of the CHERI ASPLOS paper; the second example also involves the Sub *pointer subtraction* idiom and perhaps the IA *performing integer arithmetic on pointers* idiom (it's not clear exactly what that is). All are widely observed in practice.

2.13.2 Q32. Can one form pointer values by pointer addition that overflows (without undefined behaviour)?

D:ISO-VS-DEFACTO

ISO: no   DEFACTO-USAGE: yes sometimes   DEFACTO-IMPL: yes sometimes but not in general   CERBERUS-DEFACTO: yes?   CHERI: ? yes in 256-bit CHERI, not always in 128-bit CHERI   TIS: yes   KCC: no (flags UB at pointer arithmetic)

EXAMPLE (`pointer_add_wrap_1.c`):

```
#include <stdio.h>
int main() {
  unsigned char x;
  unsigned char *p = &x;
  unsigned long long h = ( 1ull << 63 );
  //are the following free of undefined behaviour?
  unsigned char *q1 = p + h;
  unsigned char *q2 = q1 + h;
  printf("Addresses: p =%p  q1=%p\n",
         (void*)p,(void*)q1);
  printf("Addresses: q2=%p  h =0x%llx\n",
         (void*)q2,h);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p =0x7fffffffea0f q1=0x80007fffffffea0f
Addresses: q2=0x7fffffffea0f h =0x8000000000000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above (modulo addresses)

ISO: `undefined behaviour`

Obviously this presumes that constructing an out-of-bounds (by more than one) pointer value by pointer arithmetic, as per §2.13.1 (p.31), is itself allowed.

2.13.3 Q33. Can one assume pointer addition wraps on overflow?

U:DEFACTO

ISO: no   DEFACTO-USAGE: ?   DEFACTO-IMPL: ?   CERBERUS-DEFACTO: ?   CHERI: ?   TIS: no (or, if so, tis is not assuming a 64-bit address space). Unclear?   KCC: no (flags UB at pointer arithmetic)

EXAMPLE (`pointer_add_wrap_2.c`):

```
#include <stdio.h>
int main() {
  unsigned char x;
  unsigned char *p = &x;
  unsigned long long h = ( 1ull << 63 );
  //are the following free of undefined behaviour?
  unsigned char *q1 = p + h;
  unsigned char *q2 = q1 + h;
  *q2 = 1;
  printf("Addresses: p =%p  q1=%p\n",
         (void*)p,(void*)q1);
  printf("Addresses: q2=%p  h =0x%llx\n",
         (void*)q2,h);
  printf("x=0x%x  *p=0x%x  *q2=0x%x\n",x,*p,*q2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: p =0x7fffffffea0f q1=0x80007fffffffea0f
Addresses: q2=0x7fffffffea0f h
=0x8000000000000000
x=0x1 *p=0x1 *q2=0x1
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above (modulo addresses)

ISO: `undefined behaviour`

This presumes that the previous question is allowed.

2.13.4 Q34. Can one move among the members of a struct using representation-pointer arithmetic and casts?

U:ISO D:ISO-VS-DEFACTO

ISO: unclear – impl-def?   DEFACTO-USAGE: yes   DEFACTO-IMPL: yes   CERBERUS-DEFACTO: yes

CHERI: yes TIS: yes KCC: no ((mistakenly) detects UB: A pointer (or array subscript) outside the bounds of an object)

The standard is ambiguous on the interaction between the allowable pointer arithmetic (on `unsigned char*` representation pointers) and subobjects. For example, consider:

EXAMPLE (`cast_struct_inter_member_1.c`):

```
#include <stdio.h>
#include <stddef.h>
typedef struct { float f; int i; } st;
int main() {
  st s = {.f=1.0, .i=1};
  int *pi = &(s.i);
  unsigned char *pci = ((unsigned char *)pi);
  unsigned char *pcf = (pci - offsetof(st,i))
    + offsetof(st,f);
  float *pf = (float *)pcf;
  *pf = 2.0;  // is this free of undefined behaviour?
  printf("s.f=%f *pf=%f  s.i=%i\n",s.f,*pf,s.i);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
s.f=2.000000 *pf=2.000000 s.i=1
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour`

ISO: `unclear`

This forms an `unsigned char*` pointer to the second member (`i`) of a struct, does arithmetic on that using `offsetof` to form an `unsigned char*` pointer to the first member, casts that into a pointer to the type of the first member (`f`), and uses that to write.

In practice we believe that this is all supported by most compilers and it is used in practice, e.g. as in the Container idiom of the CHERI ASPLOS paper, where they discuss container macros that take a pointer to a structure member and compute a pointer to the structure as a whole. They see it heavily used by one of the example programs they studied. We are told that Intel's MPX compiler does not support the container macro idiom, while Linux, FreeBSD, and Windows all rely on it.

The standard says (6.3.2.3p7): *"...When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object."*. This licenses the construction of the `unsigned char*` pointer `pci` to the start of the representation of `s.i` (presuming that a structure member is itself an "object", which itself is ambiguous in the standard), but allows it to be used only to access the representation of `s.i`.

The `offsetof` definition in `stddef.h`, 7.19p3, *"[...] `offsetof(type, member-designator)` which expands to an integer constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by member-designator), from the beginning of its structure (designated by type). [...]"*, implies that the calculation of `pcf` gets the correct numerical address, but does not say that it can be used, e.g. to access the representation of `s.f`. As we saw in the discussion of provenance, the mere fact that a pointer has the correct address does not necessarily mean that it can be used to access that memory without giving rise to undefined behaviour.

Finally, if one deems `pcf` to be a legitimate `char*` pointer to the representation of `s.f`, then the standard says that it can be converted to a pointer to any object type if sufficiently aligned, which for `float*` it will be. 6.3.2.3p7: *"A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned 68) for the referenced type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer..."*. But whether that pointer has the right value and is usable to access memory is left unclear.

2.13.5 Q35. Can one move between subobjects of the members of a struct using pointer arithmetic?

U:ISO D:ISO-VS-DEFACTO

ISO: unclear DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: ? TIS: guess yes, but tis appears not to support `%td` format KCC: no (detects UB at the pointer arithmetic)

EXAMPLE (`struct_inter_submember_1.c`):

```
#include <stdio.h>
#include <stddef.h>
struct S { int a[3]; int b[3]; } s;
int main() {
  s.b[2]=10;
  ptrdiff_t d;
  d = &(s.b[2]) - &(s.a[0]);  // defined behaviour?
  int *p;
  p = &(s.a[0]) + d;          // defined behaviour?
  *p = 11;                    // defined behaviour?
  printf("d=%td  s.b[2]=%d  *p=%d\n",d,s.b[2],*p);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
d=5 s.b[2]=11 *p=11
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `unclear`

This is inspired by an example from Krebbers [29], as discussed in §6.11.

2.13.6 Q36. Can one implement `offsetof` using the addresses of members of a NULL struct pointer?

U:ISO

ISO: unclear DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: ? TIS: unclear (the print seems to stop at the %p) KCC: no (flags a null-dereference UB)

EXAMPLE (`ubc_addr_null_1.c`):

```
#include <stddef.h>
#include <inttypes.h>
```

```
#include <stdio.h>
struct s { uint8_t a; uint8_t b; };
int main () {
  struct s *f = NULL;
  uint8_t *p = &(f->b); // free of undefined behaviour?
  // and equal to the offsetof result?
  printf("p=%p  offsetof(struct s,b)=0x%zx\n",
         (void*)p,offsetof(struct s, b));
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
p=0x1 offsetof(struct s,b)=0x1
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `unclear`

This seems to be a common idiom in practice. The test is inspired by examples from Regehr's UB Canaries, as discussed in §6.18.

If one views `p->x` as syntactic sugar for `(*p).x` (as stated by Jones [24, p.982], but, interestingly, not the ISO standard) then this is undefined behaviour when `p` is null. CompCert seems to do this, while GCC seems to keep the `->` at least as far as GIMPLE.

2.14 Casts between pointer types

Standard The standard (6.3.2.3p{1–4,7,8}) identifies various circumstances in which conversion between pointer types is legal, with some rather weak constraints on the results:

1 *"A pointer to void may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer."*

2 *"For any qualifier q, a pointer to a non-q-qualified type may be converted to a pointer to the q-qualified version of the type; the values stored in the original and converted pointers shall compare equal."*

7 *"A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned 68) for the referenced type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer. When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object."*

8 *"A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined."*

Paragraphs 3 and 4 relate to null pointers, as discussed in §2.12 (p.30). Paragraphs 5 and 6 relate to casts between pointer and integer types, as discussed in §2.2 (p.10). Footnote 68 just says that "correctly aligned" should be transitive.

This raises several questions. First, this "compare equal" is probably supposed to mean the the pointers are (in our sense discussed in §2.10, p.26) equivalent: that they not only compare equal with `==` but also are equally usable to access (the same) memory and have equal representations. We imagine that this is pre-DR260 text, when these concepts arguably coincided.

Second, the standard only covers roundtrips of size two, via one other pointer type and back. This seems curiously irregular: there seems to be no reason not to give a roundtrip property for longer roundtrips via multiple pointer types, and both our ISO and de facto standard semantics should allow that.

Third, (7) gives undefined behaviour for a conversion between object types where the result value is not aligned for the new type, while (1) allows such a conversion via `(void *)`, albeit with no guarantee on the result.

Fourth, it gives no guarantees for the usability of pointers constructed by a combination of casts and arithmetic, as discussed in §2.13.4 (p.32).

Additionally, 6.7.2.1 *Structure and union specifiers* licenses conversions (in both directions) between pointers to structures and their initial members, and between unions and their members.

The Friendly C proposal (Point 4) by Cuoq et al., discussed in §6.17, has a link[21] which points to C committee discussion[22] in which they considered interconvertability of object and function pointers. POSIX apparently requires it, for `dlsym`.

2.14.1 Q37. Are usable pointers to a struct and to its first member interconvertable?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes

A Linux kernel developer says that they rely on this, and also that they use offsetof to move between members. If offsetof is not available, it is faked up (with subtraction between address-of a member reference off the null pointer).

EXAMPLE (`cast_struct_and_first_member_1.c`):

```
#include <stdio.h>
typedef struct { int  i; float f; } st;
int main() {
  st s = {.i = 1, .f = 1.0};
  int *pi = &(s.i);
  st* p = (st*) pi; // free of undefined behaviour?
```

---

[21] `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2605.pdf`

[22] Defect Report 195 in `http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html`

```
  p->f = 2.0;       // and this?
  printf("s.f=%f  p->f=%f\n",s.f,p->f);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`s.f=2.000000 p->f=2.000000`

CLANG36-O2-NO-STRICT-ALIASING: ... as above

DEFACTO: `defined behaviour`

ISO: `defined behaviour`

This is allowed in the standard: 6.7.2.1p15 *"Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared.* ***A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa.*** *There may be unnamed padding within a structure object, but not at its beginning."* (bold emphasis added).

2.14.2 Q38. Are usable pointers to a union and to its current member interconvertable?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes

EXAMPLE (`cast_union_and_member_1.c`):

```
#include <stdio.h>
typedef union { int  i; float f; } un;
int main() {
  un u = {.i = 1};
  int *pi = &(u.i);
  un* p = (un*) pi; // free of undefined behaviour?
  p->f = 2.0;       // and this?
  printf("u.f=%f  p->f=%f\n",u.f,p->f);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`u.f=2.000000 p->f=2.000000`

CLANG36-O2-NO-STRICT-ALIASING: ... as above

DEFACTO: `defined behaviour`

ISO: `defined behaviour`

The standard says: 6.7.2.1p16 *"The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time.* ***A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa."*** (bold emphasis added).

This is likewise allowed in practice and in the standard.

2.15 Accesses to related structure and union types

If one only accesses structures via assignment and member projections, the standard treats structure types abstractly. Type declarations create new types:

- 6.7.2.1p8 *"The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit. [...]"*
- 6.7.2.3p5 *"Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type."*;

accessing a structure member requires the name of a member of the type:

- 6.5.2.3p1 *"The first operand of the* `.` *operator shall have an atomic, qualified, or unqualified structure or union type, and the second operand shall name a member of that type."*
- 6.5.2.3p2 *"The first operand of the* `->` *operator shall have type "pointer to atomic, qualified, or unqualified structure" or "pointer to atomic, qualified, or unqualified union", and the second operand shall name a member of the type pointed to."*;

and assignment requires the left and right-hand-side types to be *compatible*:

- 6.5.16.1p1b2 *"the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right;"*
- 6.5.16.1p1b3 *"the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;"*,

where (6.2.7p1) for two structure types to be compatible they have to be either the same or (if declared in separate translation units) very similar: broadly, with the same ordering, names, and compatible types of members.

But the standard permits several ways to break this type abstraction: conversion between pointers to object types, reading from a union of structures sharing a common initial sequence, and type punning by writing and reading different union members.

Most simply, one can initialise a structure by initialising its individual members at their underlying types:

EXAMPLE (`struct_initialise_members.c`):

```
#include <stdio.h>
void f(char* cp, float*fp) {
  *cp='A';
  *fp=1.0;
}
typedef struct { char c; float f; } st;
int main() {
  st s1;
  f(&s1.c, &s1.f);
  st s2;
  s2 = s1;
```

```
  printf("s2.c=0x%x  s2.f=%f\n",s2.c,s2.f);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
s2.c=0x41 s2.f=1.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour`

ISO: `defined behaviour`

This suggests that isomorphic structs could be interchangeable as memory objects, at least if one can cast from one pointer type to the other. This is reasonable in the de facto semantics, but the standard's effective types (discussed in §4, p.61) make it false in the standard.

Even in the de facto semantics, isomorphic struct types are not directly interchangeable. The following example gives a static type error in GCC and Clang, and is clearly forbidden in the standard (for the two struct types to be compatible they have to be almost identical).

EXAMPLE (`use_struct_isomorphic.c`):

```
#include <stdio.h>
typedef struct { int  i1; float f1; } st1;
typedef struct { int  i2; float f2; } st2;
int main() {
  st1 s1 = {.i1 = 1, .f1 = 1.0 };
  st2 s2;
  s2 = s1;
  printf("s2.i2=%i2  s2.f2=%f\n",s2.i2,s2.f2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
use_struct_isomorphic.c: In function 'main':
use_struct_isomorphic.c:7:6: error: incompatible types
when assigning to type 'st2 {aka struct <anonymous>}'
from type 'st1 {aka struct <anonymous>}'
 s2 = s1;

 ^
use_struct_isomorphic.c.gcc-5.3-O2-no-strict-aliasing.ou
t: not found
```

CLANG36-O2-NO-STRICT-ALIASING:

```
use_struct_isomorphic.c:7:6: error: assigning to 'st2'
from incompatible type 'st1'
 s2 = s1;
 ^
1
error generated.
use_struct_isomorphic.c.clang36-O2-no-strict-aliasing.ou
t: not found
```

DEFACTO: `type error`

ISO: `type error`

Most generally, 6.3.2.3p7 says that *"A pointer to an object type may be converted to a pointer to a different object type"*, if *"the resulting pointer is correctly aligned"*, otherwise undefined behaviour results. (6.5.4 *Cast operators* does not add any type restrictions to this.)

There are two interesting cases here: conversion to a `char *` pointer and conversion to a related structure type. In the former, 6.3.2.3p7 (as discussed in §2.14, p.34) goes on to specify enough about the value of the resulting pointer to make it usable for accessing the representation bytes of the original object. In the latter, the standard says little about the resulting value, but it might be used to access related structures without going via a union type:

2.15.1 Q39. Given two different structure types sharing a prefix of members that have compatible types, can one cast a usable pointer to an object of the first to a pointer to the second, that can be used to read and write members of that prefix (with strict-aliasing disabled and without packing variation)?

D:ISO-VS-DEFACTO

ISO: n/a (ISO does not specify semantics with strict aliasing disabled, and effective types forbid this) DEFACTO-USAGE: yes DEFACTO-IMPL: yes (with `-fno-effective-types`, at least) CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes (contrary to ISO effective types)

[Question 10/15 of our *What is C in practice? (Cerberus survey v2)*[23] relates to this.]

First we consider a case with two isomorphic structure types:

EXAMPLE (`cast_struct_isomorphic.c`):

```
#include <stdio.h>
typedef struct { int  i1; float f1; } st1;
typedef struct { int  i2; float f2; } st2;
int main() {
  st1 s1 = {.i1 = 1, .f1 = 1.0 };
  st2 *p2 = (st2 *) (&s1);// is this free of undef.beh.?
  p2->f2=2.0;             // and this?
  printf("s1.f1=%f  p2->f2=%f\n",s1.f1,p2->f2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
s1.f1=2.000000 p2->f2=2.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour`

ISO: `undefined behaviour`

And now with a common prefix but differing after that:

EXAMPLE (`cast_struct_same_prefix.c`):

```
#include <stdio.h>
typedef struct { int  i1; float f1; char c1; double d1; }
  st1;
typedef struct { int  i2; float f2; double d2; char c2; }
  st2;
int main() {
  st1 s1 = {.i1 = 1, .f1 = 1.0, .c1 = 'a', .d1 = 1.0};
  st2 *p2 = (st2 *) (&s1);// is this free of undef.beh.?
  p2->f2=2.0;             // and this?
  printf("s1.f1=%f  p2->f2=%f\n",s1.f1,p2->f2);
```

---

[23] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

```
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
s1.f1=2.000000 p2->f2=2.000000
```

CLANG36-O2-NO-STRICT-ALIASING: ... as above

DEFACTO: `defined behaviour (with effective types switched off)`

ISO: `undefined behaviour`

Several survey respondents reported that this idiom is both used and supported in practice, e.g. in some C object systems and in the Perl interpreter.

For it to work in implementations,

1. the offsets of `f1` and `f2` have to be equal,
2. the code emitted by the compiler for the `f2` access has to be independent of the subsequent members of the structure (in particular, it cannot use an over-wide write that would only hit padding in one structure but hit data in the other). Or we need a more elaborate condition: the last member of the common prefix is only writable if it is aligned and sized such that wide writes will never be used (an implementation-defined property).
3. either the alignments of `st1` and `st2` have to be equal or the code emitted by the compiler for the `f2` access has to be independent of the structure alignment (we imagine that the latter holds in practice), and
4. the compiler has to not be doing some alias analysis that assumes that it is illegal.

For the offsets, the standard implies that within the scope of each compilation, there is a fixed layout for the members of each structure, and that that is available to the programmer via `offsetof(`*type*`, `*member-designator*`)`, *"the offset in bytes, to the structure member (designated by member-designator), from the beginning of its structure (designated by type)."* (7.19p3, in *Common definitions* `<stddef.h>`), and via the the 6.5.3.4 `sizeof` and `_Alignof` operators. The C standard provides only weak constraints for these layout values[24]; it does not guarantee that `st1` and `st2` have the same offsets for `f1` and `f2`.[25]

In practice, however, these values are typically completely determined by the ABI, with constant sizes and alignments for the fundamental types and the algorithm *"Each member is assigned to the lowest available offset with the appropriate alignment."* for structures, from the x86-64 Unix ABI [37]. There is similar text for Power [6], MIPS [45], and Visual Studio [38]. The ARM ABI [5] is an exception in that it does not clearly state this, but the wording suggests that the writers may well have had the same algorithm in mind. This algorithm will guarantee that the offsets are equal.

W.r.t. the (hypothetical) use of wide writes, the situation is unclear to us.

We should recall also that there are various compiler flags and pragmas to control packing, so it can (and does) happen that the same type (and code manipulating it) is compiled with different packing in different compilation units, relying on the programmer to not intermix them. We currently ignore this possibility but it should be relatively straightforward to add the packing flags to the structure name used within the semantics.

If one wanted to argue that this example should be illegal (e.g. to license an otherwise-unsound analysis), one might attempt to do so in terms of the *effective types* of 6.5p{6,7}. The key question here is whether one considers the effective type of a structure member to be simply the type of the member itself or also to involve the structure type that it is part of, which the text (with its ambiguous use of "object") leaves unclear. In the former case the example would be allowed, while in the latter it would not. We return to this in §4 (p.61).

2.15.2 Q40. Can one read from the initial part of a union of structures sharing a common initial sequence via any union member (if the union type is visible)?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes (though they ask whether union type visibility is obscured by `*&`?) KCC: yes

Next we have 6.5.2.3p6, which licenses *reading* from a common initial sequence of two structure types which are members of a union type declaration: *"One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is visible. Two structures share a common initial sequence if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members."*

EXAMPLE (`read_union_same_prefix_visible.c`):

```
#include <stdio.h>
typedef struct { int  i1; float f1; char c1; } st1;
typedef struct { int  i2; float f2; double d2; } st2;
typedef union { st1 m1; st2 m2; } un;
int main() {
  un u = {.m1 = {.i1 = 1, .f1 = 1.0, .c1 = 'a'}};
  int i = u.m2.i2; // is this free of undef.beh.?
  printf("i=%i\n",i);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
i=1
```

---

[24] e.g. that they increase along a structure, per 6.7.2.1p15

[25] DR074CR confirms this: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_074.html`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour`

ISO: `defined behaviour`

2.15.3 Q41. Is writing to the initial part of a union of structures sharing a common initial sequence allowed via any union member (if the union type is visible)?

U:DEFACTO

ISO: no DEFACTO-USAGE: unclear DEFACTO-IMPL: unclear CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes

We presume the above is restricted to reading to avoid the case in which a write to one structure type might overwrite what is padding there but not padding in the other structure type. We return to padding below.

EXAMPLE (`write_union_same_prefix_visible.c`):

```
#include <stdio.h>
typedef struct { int  i1; float f1; char c1; } st1;
typedef struct { int  i2; float f2; double d2; } st2;
typedef union { st1 m1; st2 m2; } un;
int main() {
  un u = {.m1 = {.i1 = 1, .f1 = 1.0, .c1 = 'a'}};
  u.m2.i2 = 2; // is this free of undef.beh.?
  printf("u.m1.i1=%i  u.m2.i2=%i\n",u.m1.i1,u.m2.i2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
u.m1.i1=2 u.m2.i2=2
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (under the ‘more elaborate condition’)`

ISO: `undefined behaviour`

2.15.4 Q42. Is type punning by writing and reading different union members allowed (if the lvalue is syntactically obvious)?

U:DEFACTO D:ISO-VS-DEFACTO

ISO: yes DEFACTO-USAGE: yes (subject to GCC “syntactically obvious” notion) DEFACTO-IMPL: yes (subject to GCC “syntactically obvious” notion) CERBERUS-DEFACTO: yes? CHERI: yes TIS: yes KCC: Execution failed (unclear why)

[Question 15/15 of our *What is C in practice? (Cerberus survey v2)*[26] relates to this.]

And finally, in some cases subsuming the previous clause, 6.5.2.3p3 and Footnote 95 explicitly license much more general type punning for union members, allowing the representation of one member to be reinterpreted as another member.

- 6.5.2.3p3 *“A postfix expression followed by the . operator and an identifier designates a member of a structure or union object. The value is that of the named member,95) and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.”.*
- Footnote 95) *“If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called “type punning”). This might be a trap representation.”*

The GCC documentation[27] suggests that for this to work the union must be somehow syntactically visible in the access, in the construction of the lvalue, or in other words that GCC pays attention to more of the lvalue than just the lvalue type (at least with `-fstrict-aliasing`; without that, it’s not clear):

`-fstrict-aliasing` Allow the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C (and C++), this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an `unsigned int` can alias an `int`, but not a `void*` or a `double`. A character type may alias any other type.

Pay special attention to code like this:

EXAMPLE (`union_punning_gcc_1.c`):

```
// adapted from GCC docs
#include <stdio.h>
union a_union {
  int i;
  double d;
};
int main() {
  union a_union t;
  t.d = 3.1415;
  int j = t.i; // is this defined behaviour?
  printf("j=%d\n",j);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
j=-1065151889
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (with implementation-defined value)`

ISO: `defined behaviour (with implementation-defined value)`

The practice of reading from a different union member than the one most recently written to (called “type-punning”) is common. Even with -fstrict-aliasing, type-punning is allowed, provided the memory is accessed through the union

---

[26] www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html

[27] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Type-punning

type. So, the code above works as expected. See Structures unions enumerations and bit-fields implementation. However, this code might not:

EXAMPLE (`union_punning_gcc_2.c`):

```
// adapted from GCC docs
#include <stdio.h>
union a_union {
  int i;
  double d;
};
int main() {
  union a_union t;
  int* ip;
  t.d = 3.1415;
  ip = &t.i;   // is this defined behaviour?
  int j = *ip; // is this defined behaviour?
  printf("j=%d\n",j);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
j=-1065151889
```

CLANG36-O2-NO-STRICT-ALIASING: ... as above

DEFACTO: `undefined behaviour`

ISO: `unclear (perhaps defined behaviour with implementation-defined value?)`

See also the LLVM mailing list thread on the same topic: `http://lists.cs.uiuc.edu/pipermail/cfe-dev/2015-March/042034.html`

Hence one should presumably regard both of these as giving undefined behaviour in the a facto semantics. The ISO standard text is unclear about whether it is allowed in the standard or not.

For reference: a GCC mailing list post[28] observes that upcasts from int to union can go wrong in practice, and another[29] says that GCC conforms to TC3 with respect to type punning through union accesses.

2.16 Pointer lifetime end

After the end of the lifetime of an object[30], one can ask whether pointers to that object retain their values, or, in more detail, whether:

1. they can be compared (with `==` and `!=`) against other pointers,
2. they can be compared (with <, >, <=, or >=) against other pointers,
3. their representation bytes can be inspected and still contain their address values,
4. pointer arithmetic and member offset calculations can be performed,
5. they can be used to access a newer object that happens to be allocated at the same address, or
6. they can be used to access the memory that was used for the lifetime-ended object.

The ISO standard is clear that these are not allowed in a useful way: 6.2.4 *Storage durations of objects* says (6.4.2p2) *"If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime."*. More precisely, the first sentence makes 6 and 5 undefined behaviour. The second sentence means that 1, 2, 3, and 4 are not guaranteed to have useful results, but (in our reading, and in the absence of trap representations) the standard text does not make these operations undefined behaviour. Other authors differ on this point.

This side-effect of lifetime end on *all* pointer values that point to the object, wherever they may be in the abstract-machine state, is an unusual aspect of C when compared with other programming language definitions.

Note that there is no analogue of this "lifetime-end zap" in the standard text for pointers to objects stored within a `malloc`'d region when those objects are overwritten (with a strong update) with something of a different type; the lifetime end zap is not sufficient to maintain the invariant that all extant pointer values point to something live of the appropriate type.

In practice the situation is less clear:

1. some debugging environments null out the pointer being freed (though presumably not other pointers to the same object)
2. one respondent notes *"After a pointer is freed, its value is undefined. A fairly common optimisation is to reuse the stack slot used for a pointer in between it being freed and it having a defined value assigned to it."* though it is not clear whether this actually happens.

On the other hand, several respondents suggest that checking equality (with == or `!=`) against a pointer to an object whose lifetime has ended is used and is supported by implementations. One remarks that whether the object has gone out of scope or been free'd may be significant here, and so we give an example below for each.

In a TrustInSoft blog post[31], Julian Cretin gives examples showing GCC giving surprising results for comparisons between lifetime-ended pointers. He argues that those pointers have indeterminate values and hence that any uses of them, even in a == comparison, give undefined behaviour. The first is clear in the ISO standard; the second is not, at least in our reading – especially in implementations where there are no trap representations at pointer types. The behaviour he ob-

---

[28] `https://gcc.gnu.org/ml/gcc/2010-01/msg00013.html`

[29] `https://gcc.gnu.org/ml/gcc/2010-01/msg00027.html`

[30] For an object of thread storage duration, the lifetime ends at the termination of the thread (6.2.4p4). For an object of automatic storage duration (leaving aside those that "have a variable length array type" for the moment), the lifetime ends when *"execution of that block ends in any way"* (6.2.4p6). For an object of allocated storage duration, the lifetime ends at the deallocation of an associated `free` or `realloc` call (7.22.3p1).

[31] `http://trust-in-soft.com/dangling-pointer-indeterminate/`

serves for pointer comparison could also be explained by the semantics we envision that nondeterministically takes pointer provenance into account, without requiring an appeal to undefined behaviour. The behaviour of the corresponding integers (cast from pointers to `uintptr_t`) is less clear, but that could arguably be a compiler bug.

2.16.1 Q43. Can one inspect the value, (e.g. by testing equality with ==) of a pointer to an object whose lifetime has ended (either at a `free()` or block exit)?

D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: yes (except in debugging environments) CERBERUS-DEFACTO: yes CHERI: yes TIS: no (warning of access to escaping addresses) KCC: no (flags UB)

[Question 8/15 of our *What is C in practice? (Cerberus survey v2)*[32] relates to this.]

EXAMPLE (`pointer_comparison_eq_zombie_1.c`):

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  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?
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
pointer_comparison_eq_zombie_1.c: In function 'main':
pointer_comparison_eq_zombie_1.c:8:3: warning: attempt
to free a non-heap object 'i' [-Wfree-nonheap-object]


free(pj);
 ^
(&i==pj)=false
(&i==pj)=false
```

CLANG36-O2-NO-STRICT-ALIASING:

```
(&i==pj)=false
(&i==pj)=false
```

DEFACTO: `switchable`

ISO: `unclear -- nondeterministic or undefined behaviour`

Here the comparison against `pj` after the `free()` is undefined behaviour according to the ISO standard. GCC -O2 gives a misleading warning about the `free()` itself (the warning goes away if one omits either `printf()` or with `-O0`); that might be a GCC bug.

EXAMPLE (`pointer_comparison_eq_zombie_2.c`):

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

GCC-5.3-O2-NO-STRICT-ALIASING:

```
(&i==pj)=false
(&i==pj)=false
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `switchable`

ISO: `unclear -- nondeterministic or undefined behaviour`

One could construct similar examples for rest of the first four items above (relational comparison, access to representation bytes, and pointer arithmetic). We do not expect the last two of the six (access to newly allocated objects or to now-deallocated memory) are used in practice, at least in non-malicious code.

2.16.2 Q44. Is the dynamic reuse of allocation addresses permitted?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes? CHERI: ? TIS: test not supported (tis fails with escaping address, even though it's cast to `intptr_t` – perhaps intentionally due to nondeterminism?) KCC: mistakenly flags reference to an object outside its lifetime

EXAMPLE (`compcertTSO-2.c`):

```
#include <stdio.h>
#include <inttypes.h>
uintptr_t f() {
  int a;
  return (uintptr_t)&a; }
uintptr_t g() {
  int a;
  return (uintptr_t)&a; }
int main() {
  _Bool b = (f() == g()); // can this be true?
  printf("(f()==g())=%s\n",b?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
compcertTSO-2.c: In function 'f':
compcertTSO-2.c:5:10: warning: function returns address
of local variable [-Wreturn-local-addr]
 return
(uintptr_t)&a; }
 ^

compcertTSO-2.c: In
function 'g':
compcertTSO-2.c:8:10: warning: function
```

---

[32] www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html

```
returns address of local variable [-Wreturn-local-addr]

 return (uintptr_t)&a; }
 ^
(f()==g())=true
```

CLANG36-O2-NO-STRICT-ALIASING:

```
(f()==g())=false
```

This example based on one from CompCertTSO, as discussed in §6.5. This version casts to `uintptr_t` to make the out-of-lifetime == comparison permitted (at least w.r.t. our reading of ISO), though GCC 4.8 -O2 still warns that the functions return addresses of local variables. One could write analogous tests using other constructs that expose the concrete address of a pointer value, e.g. casting to an integer type, examining the pointer representation bytes, or using `printf` with `%p`. The CompCertTSO example `compcertTSO-1.c` uses == on the pointer values directly because (as in CompCert 1.5) none of those are supported there, while CompCertTSO does allow that comparison.

2.17 Invalid Accesses

In the ISO standard, reads and writes to invalid pointers give undefined behaviour, and likewise in typical implementations. For a conventional C implementation, undefined behaviour for general invalid writes is essentially forced, given that they might (e.g.) write over return addresses on the stack. But accesses to NULL pointers and reads from an invalid pointer could conceivably be strengthened, as in the following two questions.

2.17.1 Q45. Can accesses via a null pointer be assumed to give runtime errors, rather than give rise to undefined behaviour?

ISO: no DEFACTO-USAGE: no? DEFACTO-IMPL: no? CERBERUS-DEFACTO: should flag UB CHERI: ? TIS: flags UB KCC: flags UB

EXAMPLE (`null_pointer_4.c`):

```
#include <stdio.h>
int main() {
  int x;
  // is this guaranteed to trap (rather than be
  // undefined behaviour)?
  x = *(int*)NULL;
  printf("x=%i\n",x);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

CLANG36-O2-NO-STRICT-ALIASING:

```
null_pointer_4.c:6:7: warning: indirection of
non-volatile null pointer will be deleted, not trap
[-Wnull-dereference]
 x = *(int*)NULL;

 ^
null_pointer_4.c:6:7: note: consider using
__builtin_trap() or qualifying pointer with 'volatile'
1
warning generated.
x=-5512
```

ISO: `undefined behaviour`

This is inspired by the fifth example of Wang et al. [53], discussed in §6.14.

2.17.2 Q46. Can reads via invalid pointers be assumed to give runtime errors or unspecified values, rather than undefined behaviour?

ISO: no DEFACTO-USAGE: no DEFACTO-IMPL: no CERBERUS-DEFACTO: no CHERI: ? TIS: flags UB KCC: reads some value, mistakenly not flagging UB

EXAMPLE (`read_via_invalid_1.c`):

```
#include <stdio.h>
int main() {
  int x;
  // is this free of undefined behaviour?
  x = *(int*)0x654321;
  printf("x=%i\n",x);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `undefined behaviour`

This is from the Friendly C proposal (Point 4) by Cuoq et al., discussed in §6.17. For such a semantics one would nonetheless want to identify a (different, not expressed in terms of undefined behaviour) sense in which such reads indicate programmer errors.

3. Abstract Unspecified Values

[Question 2/15 of our *What is C in practice? (Cerberus survey v2)*[33] relates to uninitialised values.]

The ISO standard introduces:

- *indeterminate values* which are *"either an unspecified value or a trap representation"* (3.19.2),
- *unspecified values*, saying *"valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance. 2 NOTE An unspecified value cannot be a trap representation."* (3.19.3), and
- *trap representations*, *"an object representation that need not represent a value of the object type"* (3.19.4).

In the standard text, reading uninitialised values can give rise to undefined behaviour in two ways, either

---

[33] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

1. if the type being read does have some trap representations in the particular implementation being used, or
2. if the last sentence of 6.3.2.1p2 applies (c.f. the DR338 CR[34]): *"If the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined."*. This makes reading such lvalues undefined behaviour irrespective of the existence of trap representations.

For the de facto standard, as far as we can tell, trap representations can be neglected, and the last sentence of 6.3.2.1p2 has debatable force.

3.1 Trap Representations

In the ISO standard, trap representations are object representations that do not represent values of the object type, for which reading a trap representation, except by an lvalue of character type, is undefined behaviour. Note that this gives undefined behaviour to programs that merely read such a representation, even if they do not operate on it. Note also that this need not give rise to a hardware trap[35]; trap representations might simply licence some compiler optimisation, by imposing an obligation on the programmer to avoid them.

6.2.6.1p5 *"Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.50) Such a representation is called a trap representation."*. Footnote 50: *"Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it."*.

However, it is not clear that trap representations are significant in practice for current mainstream C implementations. For integer types it appears not:

- 6.2.6.1p5 makes clear that trap representations are particular concrete bit patterns, and in the most common integer type implementations there are no spare bits for integer types (See DR338 for similar reasoning), and
- the GCC documentation states *"GCC supports only two's complement integer types, and all bit patterns are ordinary values."*[36]. (This resolves 6.2.6.2p2 *"Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value."*.)

It is sometimes suggested that trap representations exist to model Itanium's NaT (*"not a thing"*) flag, e.g. in a stackoverflow discussion[37]: *"Such variables are treated specially because there are architectures that have real CPU registers that have a sort of extra state that is "uninitialized" and that doesn't correspond to a value in the type domain."* and *"Itanium CPUs have a NaT (Not a Thing) flag for each integer register. The NaT Flag is used to control speculative execution and may linger in registers which aren't properly initialized before usage."*. But that is at odds with this 6.2.6.1p5 text that makes clear that trap representations are storable concrete bit patterns.

If it were not for this 6.2.6.1p5 text, one might deem there to be shadow semantic state determining whether any value is a trap representation, analogous to the pointer provenance data discussed earlier, but we see no reason to introduce that.

For pointer types, one can imagine machines that check well-formedness of a pointer value when an address is loaded (e.g. into a particular kind of register), but this doesn't occur in the most common current hardware. We would be interested to hear of any cases where it does, or where a compiler internally uses an analysis about trap representations.

There is also the case of floating point *Signalling NaN's*. One respondent remarks that in general we wouldn't expect to get a trap by reading an uninitialised value unless the FP settings enable signalling NaNs, and that Intel FPUs can do that but Clang doesn't support them, and so arranges for there to never be signalling NaNs.

3.1.1 Q47. Can one reasonably assume that no types have trap representations?

U:DEFACTO D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: yes for most integer types; debatable for `_Bool`, float, and pointer types CERBERUS-DEFACTO: yes? CHERI: yes TIS: yes KCC: no (flags UB indeterminate value used in expression)

The following example has undefined behaviour in the ISO standard if and only if the implementation has a trap representation for type `int`; one can also consider similar examples for any other object type (the address of `i` is taken, so the last sentence of 6.3.2.1p2 does not apply here).

EXAMPLE (`trap_representation_1.c`):

```
int main() {
  int i;
  int *p = &i;
  int j=i;   // is this free of undefined behaviour?
  // note that i is read but the value is not used
```

---

[34] `http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_338.htm`

[35] 3.19.5 Footnote 2 *"[...] Note that fetching a trap representation might perform a trap but is not required to [...]"*

[36] `https://gcc.gnu.org/onlinedocs/gcc/Integers-implementation.html#Integers-implementation`

[37] `http://stackoverflow.com/questions/11962457/why-is-using-an-uninitialized-variable-undefined-behavior-in-c`

```
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
trap_representation_1.c: In function ’main’:
trap_representation_1.c:4:7: warning: ’i’ is used
uninitialized in this function [-Wuninitialized]
 int
j=i; // is this free of undefined behaviour?
 ^
```

CLANG36-O2-NO-STRICT-ALIASING:

DEFACTO: `defined behaviour`

ISO: `defined or undefined behaviour depending on implementation-defined presence of trap representations at this type`

Do any current C implementations rely on concrete trap representations that are representable as bit patterns? The only possible case we are aware of is “signalling NaNs”. Supposedly definitely not for Clang. Do any current C implementations rely on semantic shadow-state trap “representations”?

3.1.2 Q48. Does reading an uninitialised object give rise to undefined behaviour?

U:DEFACTO D:ISO-VS-DEFACTO

ISO: in some cases, depending on trap representations and whether the address is taken DEFACTO-USAGE: no DEFACTO-IMPL: unclear – perhaps for `_Bool` and some float types, and on Itanium? CERBERUS-DEFACTO: no CHERI: no more than the base Clang implementation TIS: no for some tests, yes for others (guess that reading uninitialised is not flagged as UB, but branching on one is, as nondeterministic) KCC: yes (flags UB Indeterminate value used in an expression)

The real question is then whether compiler writers assume that reading an uninitialised value gives rise to undefined behaviour (not merely an unspecified value), and rely on that to permit optimisation.

EXAMPLE (trap_representation_2.c):

```
int main() {
  int i;
  int j=i;   // does this have undefined behaviour?
  // note that i is read but the value is not used
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
trap_representation_2.c: In function ’main’:
trap_representation_2.c:3:7: warning: ’i’ is used
uninitialized in this function [-Wuninitialized]
 int
j=i; // does this have undefined behaviour?
 ^
```

CLANG36-O2-NO-STRICT-ALIASING:

```
trap_representation_2.c:3:9: warning: variable ’i’ is
uninitialized when used here [-Wuninitialized]
 int
j=i; // does this have undefined behaviour?

^
trap_representation_2.c:2:8: note: initialize the
variable ’i’ to silence this warning
 int i;
 ^


 = 0
1 warning generated.
```

DEFACTO: `defined behaviour`

ISO: `undefined behaviour`

In practice we suspect that this would be at odds with too much extant code. For example, it would mean that a partly initialised struct could not be copied by a function that reads and writes all its members.

Uninitialised memory is sometimes intentionally read as a source of entropy, e.g. in openSSL, but whether this happens at non-character type is unclear, and it is now widely agreed to be undesirable in any case (see the Xi Wang blog post[38] which notes the problems involved).

On the other hand, Chris Lattner’s *What Every C Programmer Should Know About Undefined Behavior #1/3* blog post[39] says without qualification that “use of an uninitialized variable” is undefined behaviour (though this is in an introductory section which might have been simplified for exposition). Looking at the LLVM IR generated from

EXAMPLE (trap_representation_3.c):

```
int f() {
  int i,j;
  j=i;
  //  int* ip=&i;
  return j;
}
```

the front-end of Clang doesn’t seem to be assuming undefined behaviour.

Besson et al. [9] seem to interpret the standard to mean that reading an uninitialised variable always gives rise to undefined behaviour, but it’s not clear why.

A Frama-C blog post by Pascal Cuoq[40] gives examples which it argues show that GCC has to be considered at treating reads of an uninitialised `int` as undefined behaviour, not unspecified behaviour, and (in the second example below) even if its address is taken:

EXAMPLE (frama-c-2013-03-13-2.c):

```
#include <stdio.h>
```

---

[38] http://kqueue.org/blog/2012/06/25/more-randomness-or-less/

[39] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

[40] http://blog.frama-c.com/index.php?post/2013/03/13/indeterminate-undefined

```
int main(int c, char **v)
{
  unsigned int j;
  if (c==4)
    j = 1;
  else
    j *= 2;
  // does this have undefined behaviour for c != 4 ?
  printf("j:%u ",j);
  printf("c:%d\n",c);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
frama-c-2013-03-13-2.c: In function ’main’:
frama-c-2013-03-13-2.c:3:24: warning: unused parameter
’v’ [-Wunused-parameter]
 int main(int c, char **v)

 ˆ
frama-c-2013-03-13-2.c:9:7:
warning: ’j’ may be used uninitialized in this function
[-Wmaybe-uninitialized]
 j *= 2;
 ˆ
j:0 c:1
```

CLANG36-O2-NO-STRICT-ALIASING:

```
frama-c-2013-03-13-2.c:3:24: warning: unused parameter
’v’ [-Wunused-parameter]
int main(int c, char **v)

 ˆ
frama-c-2013-03-13-2.c:9:5: warning:
variable ’j’ is uninitialized when used here
[-Wuninitialized]
 j *= 2;

ˆ
frama-c-2013-03-13-2.c:5:17: note: initialize the
variable ’j’ to silence this warning
 unsigned int j;

 ˆ
 = 0
2 warnings
generated.
j:0 c:1
```

DEFACTO: nondeterministic value for j

ISO: undefined behaviour

EXAMPLE (frama-c-2013-03-13-3.c):

```
#include <stdio.h>

int main(int c, char **v)
{
  unsigned int j;
  unsigned int *p = &j;
  if (c==4)
    j = 1;
  else
    j *= 2;
  // does this have undefined behaviour for c != 4 ?
  printf("j:%u ",j);
  printf("c:%d\n",c);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
frama-c-2013-03-13-3.c: In function ’main’:
frama-c-2013-03-13-3.c:3:24: warning: unused parameter
’v’ [-Wunused-parameter]
 int main(int c, char **v)

 ˆ
frama-c-2013-03-13-3.c:10:7:
warning: ’j’ may be used uninitialized in this function
[-Wmaybe-uninitialized]
 j *= 2;
 ˆ
j:0 c:1
```

CLANG36-O2-NO-STRICT-ALIASING:

```
frama-c-2013-03-13-3.c:3:24: warning: unused parameter
’v’ [-Wunused-parameter]
int main(int c, char **v)

 ˆ
1 warning generated.
j:0 c:1
```

DEFACTO: nondeterministic value for j

ISO: nondeterministic value for j

The same happens using `unsigned char` instead of `int`[41]. But this behaviour is still consistent with a semantics that treats reads of uninitialised variables as giving a symbolic undefined value which arithmetic operations are strict in, which is a possible semantics not discussed in that blog post; it does not force a semantics giving global undefined behaviour.

Returning to the last sentence of 6.3.2.1p2, it is restricted in two ways: to objects of automatic storage duration, and moreover to those whose address is not taken. That makes the above `trap_representation_2.c` have undefined behaviour but the following example just read an unspecified value (presuming that `int` has no trap representations).

EXAMPLE (`trap_representation_1.c`):

```
int main() {
  int i;
  int *p = &i;
  int j=i;   // is this free of undefined behaviour?
  // note that i is read but the value is not used
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
trap_representation_1.c: In function 'main':
trap_representation_1.c:4:7: warning: 'i' is used
uninitialized in this function [-Wuninitialized]
 int
j=i; // is this free of undefined behaviour?
 ^
```

CLANG36-O2-NO-STRICT-ALIASING:

DEFACTO: `defined behaviour`

ISO: `defined or undefined behaviour depending on implementation-defined presence of trap representations`

```
at this type
```

3.2 Unspecified Values

Standard Unspecified values are introduced in the standard principally:

1. for otherwise-uninitialized objects with automatic storage duration (6.2.4p6 and 6.7.9p10), and
2. for the values of padding bytes on writes to structures or unions (6.2.6.1p6 *"When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.51) [...]"* with Footnote 51: *"Thus, for example, structure assignment need not copy any padding bits."*).

In principle those two could have different semantics, but so far we see no reason to distinguish them.

The behaviour of an unspecified value is described as: *"[...] valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance. [...]"* (3.19.3).

Semantics That standard text leaves several quite different semantic interpretations of unspecified values open:

1. the semantics could choose a concrete value nondeterministically (from among the set of valid values) for each unspecified value, at the time of the initialization or store (and keeping that concrete value stable thereafter), or
2. the semantics could include a symbolic constant representing an abstract unspecified value, allow that to occur in memory writes, and either
   (a) choose a concrete value nondeterministically each time such a constant is read from, or
   (b) propagate the abstract unspecified value through arithmetic, regarding all operations as strict (giving the unspecified-value result if any of their arguments are unspecified values). Then on a control-flow choice based on an unspecified value, it could either
      i. nondeterministically branch or
      ii. give undefined behaviour.

      And on any library call (or perhaps better any I/O system call?) involving an unspecified-value argument, it could either:

      A. nondeterministically choose a concrete value, or
      B. give undefined behaviour.

   Or it could have a per-representation-bit undefined-value constant rather than a per-abstract-value undefined-value constant (with the same sub-choices)
3. Or (as per Besson et al. [9]) pick a fresh symbolic value (per bit, byte, or value) and allow computation on that.

---

[41]

EXAMPLE (`frama-c-2013-03-13-3-uc.c`):
GCC-5.3-O2-NO-STRICT-ALIASING:

```
frama-c-2013-03-13-3-uc.c: In function 'main':
frama-c-2013-03-13-3-uc.c:2:24: warning: unused
parameter 'v' [-Wunused-parameter]
 int main(int c, char
**v) {

^
frama-c-2013-03-13-3-uc.c:8:7: warning: 'j' may be
used uninitialized in this function
[-Wmaybe-uninitialized]
 j *= 2;
 ^
j:0 c:1
```

CLANG36-O2-NO-STRICT-ALIASING:

```
frama-c-2013-03-13-3-uc.c:2:24: warning: unused
parameter 'v' [-Wunused-parameter]
int main(int c, char
**v) {
 ^
1 warning generated.
j:0 c:1
```

DEFACTO: `nondeterministic value for j`
ISO: `nondeterministic value for j`

The following examples explore what one can assume about the behaviour of uninitialised variables. We use `unsigned char` in these examples so that there is no question of trap representations being involved. We take unspecified values directly from uninitialised variables with automatic storage duration, so the compiler can easily see that they are uninitialised, but they could equally be taken from reads of a computed pointer that happens to end up pointing at a structure padding byte. We also take the address of the uninitialised variable in each example to ensure the last sentence of 6.3.2.1p2 does not apply, though in our de facto semantics that makes no difference.

See the LLVM discussion of its `undef` and `poison` [42]. And this LLVM thread about "poison": `http://lists.cs.uiuc.edu/pipermail/llvmdev/2015-January/081310.html`

Chris Lattner's *What Every C Programmer Should Know About Undefined Behavior #3/3* blog post[43] says that *"Arithmetic that operates on undefined values is considered to produce a undefined value instead of producing undefined behavior."* and *"Arithmetic that dynamically executes an undefined operation (such as a signed integer overflow) generates a logical trap value which poisons any computation based on it, but that does not destroy your entire program. This means that logic downstream from the undefined operation may be affected, but that your entire program isn't destroyed. This is why the optimizer ends up deleting code that operates on uninitialized variables, for example."*.

It also says *"The optimizer does go to some effort to "do the right thing" when it is obvious what the programmer meant (such as code that does "*(int*)P" when P is a pointer to float). This helps in many common cases, but you really don't want to rely on this, and there are lots of examples that you might think are "obvious" that aren't after a long series of transformations have been applied to your code."*, which suggests that it's a bit more liberal than one might imagine for type-based alias analysis?

3.2.1 Q49. Can library calls with unspecified-value arguments be assumed to execute with an arbitrary choice of a concrete value (not necessarily giving rise to undefined behaviour)?

U:ISO D:ISO-VS-DEFACTO

ISO: unclear (unless one follows DR451) DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: no more than the base Clang implementation TIS: no (warning `unspecified_value_libary_call_argument`) KCC: Execution failed (unclear why)

We start with this so that `printf` can be used in later examples.

EXAMPLE (`unspecified_value_library_call_argument.c`):

```
#include <stdio.h>
int main()
{
  unsigned char c;
  unsigned char *p = &c;
  printf("char 0x%x\n",(unsigned int)c);
  // does this have defined behaviour?
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_library_call_argument.c: In function
'main':
unspecified_value_library_call_argument.c:6:3:
warning: 'c' is used uninitialized in this function
[-Wuninitialized]
 printf("char 0x%x\n",(unsigned
int)c);
 ^

char 0x0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
char 0x0
```

DEFACTO: `nondeterministic value`

ISO: `unclear - nondeterministic value or (from DR451CR) undefined behaviour`

GCC and Clang both print a zero value.

The CR to DR451, below (§3.2.3, p.47), implies that calling library functions on indeterminate values is undefined behaviour, but that seems too restrictive, e.g. preventing serialising a struct that contains padding or uninitialised members by printing it (byte-by-byte or member-by-member). And we don't see how it is exploitable by compilers.

We also have to consider library calls with unspecified-value arguments of pointer type; they should give undefined behaviour if the pointer is used for access, and perhaps could be deemed to give undefined behaviour whether or not the pointer is used.

3.2.2 Q50. Can control-flow choices based on unspecified values be assumed to make an arbitrary choice (not giving rise to undefined behaviour)?

U:ISO U:DEFACTO

ISO: unclear - yes? DEFACTO-USAGE: yes DEFACTO-IMPL: unclear - yes? CERBERUS-DEFACTO: yes CHERI: yes TIS: no KCC: yes

EXAMPLE (`unspecified_value_control_flow_choice.c`):

```
#include <stdio.h>
int main()
{
  unsigned char c;
  unsigned char *p = &c;
  if (c == 'a')
    printf("equal\n");
  else
    printf("nonequal\n");
  // does this have defined behaviour?
```

---

[42] `http://llvm.org/docs/LangRef.html#undefined-values`

[43] `http://blog.llvm.org/2011/05/what-every-c-programmer-should-know` `21.html`

```
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_control_flow_choice.c: In function
'main':
unspecified_value_control_flow_choice.c:6:9:
warning: 'c' is used uninitialized in this function
[-Wuninitialized]
 if (c == 'a')
 ^
nonequal
```

CLANG36-O2-NO-STRICT-ALIASING:

```
nonequal
```

DEFACTO: `defined behaviour (printing a nondeterministic value)`

ISO: `defined behaviour (printing a nondeterministic value)`

One respondent remarks that Clang decides `c` is definitely not equal to `'a'`; GCC appears to do the same. This is consistent with the docmentation for the Clang internal `undef`: *"undefined 'select' (and conditional branch) conditions can go either way, but they have to come from one of the two operands."*[44].

An example from Joseph Myers, with a switch derived from several uninitialised `_Bool` values, suggests that compilers could do wild jumps if the values are not in $\{0, 1\}$, but he didn't observe GCC actually do that. If they do, and if such values are not regarded as trap representations (in which case the program would already have undefined behaviour due to the loads), then this question would have to be 'no'.

In the de facto standards this example seems to be permitted. The ISO standard does not address the question explicitly, but the value of `c` is unambigously an unspecified value w.r.t. the standard, and 3.19.3p1 *"unspecified value: valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance"* implies that one should be able to make a comparison and branch based on it.

3.2.3 Q51. In the absence of any writes, is an unspecified value potentially unstable, i.e., can multiple usages of it give different values?

U:ISO

ISO: unclear - yes? DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: test not supported – it seems printing the uninitialised value makes tis flag an error KCC: flags UB indeterminate value in expression (also reports error for printing signed `int` with `%x`)

EXAMPLE (`unspecified_value_stability.c`):

```
#include <stdio.h>
int main() {
  // assume here that int has no trap representations and
  // that printing an unspecified value is not itself
  // undefined behaviour
  int i;
  int *p = &i;
  // can the following print different values?
  printf("i=0x%x\n",i);
  printf("i=0x%x\n",i);
  printf("i=0x%x\n",i);
  printf("i=0x%x\n",i);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_stability.c: In function 'main':
unspecified_value_stability.c:9:3: warning: 'i' is used
uninitialized in this function [-Wuninitialized]

printf("i=0x%x\n",i);
 ^
i=0x0
i=0x0
i=0x0
i=0x0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
i=0xffffea60
i=0x4007cd
i=0x4007cd
i=0x4007cd
```

DEFACTO: `defined behaviour (printing nondeterministic values)`

ISO: `unclear - nondeterministic value or (from DR451CR) undefined behaviour`

If we assume that printing an unspecified value is not itself undefined behaviour, we can test with this example. Note that in a semantics (like our Cerberus candidate de facto model) with a symbolic unspecified value, and in which operations are strict in unspecified-value-ness, this question only really makes sense for external library calls, as other (data-flow) uses of an unspecified value will result in the (unique) symbolic unspecified value, not in a nondeterministic choice of concrete values.

Both GCC and Clang warn that `i` is used uninitialized; Clang sometimes prints distinct values. That is the first time that we've seen instability in practice; it (under the above assumption) rules out (1).

This is consistent with the Clang internal `undef` documentation: *"an 'undef' "variable" can arbitrarily change its value"*[45].

DR 451 by Freek Wiedijk and Robbert Krebbers[46] asks about stability of uninitialised variables with automatic storage duration, and also about library calls with indeterminate values. Their questions and the committee responses are:

---

[44] `http://llvm.org/docs/LangRef.html#undefined-values`

[45] `http://llvm.org/docs/LangRef.html#undefined-values`

[46] `http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_451.htm`

1 *“Can an uninitialized variable with automatic storage duration (of a type that does not have trap values, whose address has been taken so 6.3.2.1p2 does not apply, and which is not volatile) change its value without direct action of the program?”*. CR: *yes*

2 *“If the answer to question 1 is ”yes”, then how far can this kind of ”instability” propagate?”* CR: *any operation performed on indeterminate values will have an indeterminate value as a result.*

Note that this strong strictness is stronger than Clang’s documented behaviour, as we discuss in §3.2.4 (p.48).

3 *“If “unstable” values can propagate through function arguments into a called function, can calling a C standard library function exhibit undefined behavior because of this?”* CR: *“library functions will exhibit undefined behavior when used on indeterminate values”*.

Note that this means one cannot print an uninitialised value or padding byte. For our de facto semantics, we argue otherwise (c.f. §3.2.1, p.46).

The CR also says *“ The committee agrees that this area would benefit from a new definition of something akin to a “wobbly” value and that this should be considered in any subsequent revision of this standard. The committee also notes that padding bytes within structures are possibly a distinct form of “wobbly” representation. ”*

The unspecified values of our de facto semantics seem to be serving the same role as those “wobbly” values.

See also §3.3.2 (p.56) for the question of whether padding bytes *intrinsically* hold unspecified values (even if concrete values are written over the top), and whether that varies between structs in malloc’d regions and those with automatic, static, and thread storage durations.

The observed behaviour forces this to be “yes”, and rules out the unspecified-value semantics in which a concrete value is chosen nondeterministically at allocation time.

The ISO semantics similarly has nondeterministic prints (unless one follows the DR451CR notion that a print of an unspecified value immediately gives undefined behaviour, which we do not).

3.2.4 Q52. Do operations on unspecified values result in unspecified values?

U:ISO U:DEFACTO
ISO: unclear - yes? DEFACTO-USAGE: unclear - yes? (though see some cases in which the LLVM docs give stronger guarantees, and [9]) DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: test not supported (fails either on first read of uninitialised value or on the arithmetic) KCC: flags UB indeterminate value in expression

EXAMPLE (`unspecified_value_strictness_int.c`):

```
#include <stdio.h>
int main() {
  int i;
  int *p = &i;
  int j = (i-i);    // is this an unspecified value?
  _Bool b = (j==j); // can this be false?
  printf("b=%s\n",b?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
b=true
```

CLANG36-O2-NO-STRICT-ALIASING:

```
unspecified_value_strictness_int.c:6:15: warning:
self-comparison always evaluates to true
[-Wtautological-compare]
 _Bool b = (j==j); // can this
be false?
 ^
1 warning generated.
b=false
```

ISO: `unclear`

GCC gives `true` and Clang gives `false` (despite the Clang warning that a self-comparison always gives true, presumably a bug in Clang). This could be explained by taking subtraction on one or more unspecified values to give an unspecified value which can then be instantiated to any valid value.

For an `unsigned char` variant, both GCC and Clang give `true`:

EXAMPLE (`unspecified_value_strictness_unsigned_char.c`):

```
#include <stdio.h>
int main() {
  unsigned char c;
  unsigned char *p=&c;
  int j = (c-c);    // is this an unspecified value?
  _Bool b = (j==j); // can this be false?
  printf("b=%s\n",b?"true":"false");
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
b=true
```

CLANG36-O2-NO-STRICT-ALIASING:

```
unspecified_value_strictness_unsigned_char.c:6:15:
warning: self-comparison always evaluates to true
[-Wtautological-compare]
 _Bool b = (j==j); // can this
be false?
 ^
1 warning generated.
b=true
```

DEFACTO: `defined behaviour (printing nondeterministically true or false)`

ISO: `unclear`

For another test of whether arithmetic operators are strict w.r.t. unspecified values, consider:

EXAMPLE (`unspecified_value_strictness_mod_1.c`):

```
#include <stdio.h>
int main() {
```

```
  unsigned char c;
  unsigned char *p=&c;
  unsigned char c2 = (c % 2);
  // can reading c2 give something other than 0 or 1?
  printf("c=%i  c2=%i\n",(int)c,(int)c2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_strictness_mod_1.c: In function
’main’:
unspecified_value_strictness_mod_1.c:5:17:
warning: ’c’ is used uninitialized in this function
[-Wuninitialized]
 unsigned char c2 = (c % 2);

  ^
c=0 c2=0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
c=0 c2=0
```

DEFACTO: `defined behaviour (printing nondeterministically true or false)`

ISO: `unclear`

GCC and Clang both print `c=0  c2=0` on x86 (though not on non-CHERI MIPS). Making the computation of `c2` more complex by appending a `+(1-c)` makes them both print `c=0 c2=1`, weakly suggesting that they are *not* (in this instance) aggressively propagating unspecifiedness strictly through these arithmetic operators.

EXAMPLE (unspecified_value_strictness_mod_2.c):

```
#include <stdio.h>
int main() {
  unsigned char c;
  unsigned char *p=&c;
  unsigned char c2 = (c % 2) + (1-c);
  // can reading c2 give something other than 0 or 1?
  printf("c=%i  c2=%i\n",(int)c,(int)c2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_strictness_mod_2.c: In function
’main’:
unspecified_value_strictness_mod_2.c:5:17:
warning: ’c’ is used uninitialized in this function
[-Wuninitialized]
 unsigned char c2 = (c % 2) +
(1-c);
  ^
c=0 c2=1
```

CLANG36-O2-NO-STRICT-ALIASING:

```
c=0 c2=1
```

DEFACTO: `defined behaviour (printing nondeterministically true or false)`

ISO: `unclear`

An LLVM developer remarks that different parts of LLVM assume that undef is propagated aggressively or that it represents an unknown particular number.

The Clang `undef` documentation below[47] suggests that their internal `undef` is a per-value not a per-bit entity, and any instance can be regarded as giving any bit pattern, but operations are not simply strict. Instead, if any resulting representation bit is unaffected by the choice of a concrete value for the `undef`s, the text suggests it is guaranteed to hold its "proper" value. Does the fact that they go to this trouble imply that it is needed for code found in the wild? The text does not mention correlations between bits; presumably those are simply lost. And is this affected by any value-range-analysis facts the compiler knows about the non-undef values involved?

```
  %A = add %X, undef
  %B = sub %X, undef
  %C = xor %X, undef
Safe:
  %A = undef
  %B = undef
  %C = undef
This is safe because all of the output bits are affected by the
undef bits. Any output bit can have a zero or one depending on
the input bits.

  %A = or %X, undef
  %B = and %X, undef
Safe:
  %A = -1
  %B = 0
Unsafe:
  %A = undef
  %B = undef

These logical operations have bits that are not always affected
by the input. For example, if %X has a zero bit, then the output
of the ‘and‘ operation will always be a zero for that bit, no
matter what the corresponding bit from the ‘undef‘ is. As such,
it is unsafe to optimize or assume that the result of the ‘and‘
is ‘undef‘. However, it is safe to assume that all bits of the
‘undef‘ could be 0, and optimize the ‘and‘ to 0. Likewise, it is
safe to assume that all the bits of the ‘undef‘ operand to the
‘or‘ could be set, allowing the ‘or‘ to be folded to -1.
```

3.2.5 Q53. Do bitwise operations on unspecified values result in unspecified values?

U:ISO U:DEFACTO

ISO: unclear - yes? DEFACTO-USAGE: unclear - yes? (as for previous question) DEFACTO-IMPL: ? CERBERUS-DEFACTO: yes CHERI: ? TIS: test not supported, similarly KCC: Execution failed (unclear why)

EXAMPLE (unspecified_value_strictness_and_1.c):

```
#include <stdio.h>
int main() {
  unsigned char c;
  unsigned char *p=&c;
  unsigned char c2 = (c | 1);
  unsigned char c3 = (c2 & 1);
  // does c3 hold an unspecified value (not 1)?
  printf("c=%i  c2=%i  c3=%i\n",(int)c,(int)c2,(int)c3);
}
```

TIS-INTERPRETER:

```
[value] Analyzing a complete application starting at
main
[value] Computing initial state
```

---

[47] `http://llvm.org/docs/LangRef.html#undefined-values`

```
[value] Initial
state computed
unspecified_value_strictness_and_1.c:5:[k
ernel] warning: accessing uninitialized left-value:
assert \initialized(&c);
 stack:
main
[value] Stopping at nth alarm
[value] user error:
Degeneration occurred:
 results are
not correct for lines of code that can be reached from
the degeneration point.
```

DEFACTO: `defined behaviour (printing a nondeterministic unsigned char value)`

ISO: `unclear`

Refining the previous question, this tests whether bits of an unspecified value can be set and cleared individually to result in a specified value.

3.2.6 Q54. Must unspecified values be considered daemonically for identification of other possible undefined behaviours?

U:ISO

ISO: unclear – yes? DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: test not supported (any arithmetic on uninitialised values makes it flag an error?) KCC: (flags UB indeterminate value in expression)

EXAMPLE (`unspecified_value_daemonic_1.c`):

```
int main() {
  int i;
  int *p = &i;
  int j = i;
  int k = 1/j; // does this have undefined behaviour?
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_daemonic_1.c: In function 'main':
unspecified_value_daemonic_1.c:4:7: warning: 'i' is used
uninitialized in this function [-Wuninitialized]
 int
j = i;
 ^
```

CLANG36-O2-NO-STRICT-ALIASING:

DEFACTO: `undefined behaviour`

ISO: `unclear, but should be undefined behaviour`

Similarly, division by the Clang internal `undef` is considered to give rise to undefined behaviour[48].

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

U:ISO

ISO: unclear – yes? DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes

This and the following questions investigate whether the property of being an unspecified value is associated with arbitrary (possibly struct) C values, or with "leaf" (non-struct/non-union) values, or with individual bitfields, or with individual representation bytes of values, or with individual representation bits of values (see the later examples and LLVM documentation in §3.2.4 for the last).

It seems intuitively clear (though not specified in the ISO standard) that a structure value as a whole should not be allowed to be an unspecified value; instead one should have a struct containing unspecified values for each of its members (or hereditarily, for nested structs). It's not clear that one can express a test that distinguishes the two in ISO C, however.

Consistent with this, forming a structure value should not be strict in unspecified-value-ness: in the following example, the read of the structure value from `s1` and write to `s2` should both be permitted, and should copy the value of `i1=1`. The read of the uninitialised member should not give rise to undefined behaviour (is this contrary to the last sentence of 6.3.2.1p2, or could the structure not "have been declared with the register storage class" in any case?) . What `s2.i2` holds after the structure copy depends on the rest of the unspecified-value semantics.

EXAMPLE (`unspecified_value_struct_copy.c`):

```
#include <stdio.h>
typedef struct { int i1; int i2; } st;
int main() {
  st s1;
  s1.i1 = 1;
  st s2;
  s2 = s1; // does this have defined behaviour?
  printf("s2.i1=%i\n",s2.i1);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
s2.i1=1
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (s2.i1=1)`

ISO: `unclear, but should be defined behaviour (s2.i1=1)`

Then there is a similar question for unions: can a union value as a whole be an unspecified value? Here there might be a real semantic difference, between an unspecified value as whole and a union that contains a specific member which itself is an unspecified value. However, it's again unclear whethere there is a test in ISO C that distinguishes between them. Consider:

EXAMPLE (`unspecified_value_union_1.c`):

---

[48] `http://llvm.org/docs/LangRef.html#undefined-values`

```
#include <stdio.h>
typedef union { int i; float f; } un;
int main() {
  un u;
  int j;
  u.i = j;
  // does u contain an unspecified union value, or an
  // i member that itself has an unspecified int value?
  int k;
  float g;
  k = *((int*)&u);  //does this have defined behaviour?
  g = *((float*)&u);//does this have undefined behaviour?
}
```

If those are both true, then u does not contain an unspecified union value, but rather it contains an `i` member which contains an unspecified `int` value. Because the two accesses to u are via `int*` and `float*` pointers, not via pointers to the union type, the type punning allowed by Footnote 95[49] does not apply. Then we were hoping that the effective type of the subobject addressed by `(int*)&u` would be `int` and hence that the 65p6 effective type rules would forbid the second access. But in fact 6.5p6 doesn't treat subobjects properly and the effective type is just the union type, and the second load is permitted.

3.2.8 Q56. Given multiple bitfields that may be in the same word, can one be a well-defined value while another is an unspecified value?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: ? TIS: yes KCC: yes

EXAMPLE (`besson_blazy_wilke_bitfields_1u.c`):

```
#include <stdio.h>
struct f {
  unsigned int a0 : 1; unsigned int a1 : 1;
} bf ;
int main() {
  unsigned int a;
  bf.a1 = 1;
  a = bf.a1;
  printf("a=%u\n",a);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
a=1
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `defined behaviour (a=1)`

This example is from Besson et al. [10], discussed in §6.9. The obvious de facto standards semantics answer is "yes", with a per-leaf-value unspecified value. Though Cerberus does not currently support bitfields, so our candidate formal model likely will also not.

The Besson et al. example suggests a per-bit property. The Clang `undef` documentation is a hybrid, with some per-bit reasoning but a per-leaf-value `undef`.

3.2.9 Q57. Are the representation bytes of an unspecified value themselves also unspecified values? (not an arbitrary choice of concrete byte values)

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: unclear DEFACTO-IMPL: unclear CERBERUS-DEFACTO: yes? CHERI: unclear TIS: unclear – either reading or printing a representation byte of an uninitialised value makes it flag an error KCC: (flags indeterminate value used in an expression for this uninitialised unsigned char)

If so, then a bytewise hash or checksum computation involving them would produce an unspecified value (given the other answers above), or (in a more concrete semantics) would produce different results in different invocations, even if the value is not mutated in the meantime. It is not clear whether that is an issue in practice, and similarly for the padding bytes of structs.

EXAMPLE (`unspecified_value_representation_bytes_1.c`):

```
#include <stdio.h>
int main() {
  // assume here that the implementation-defined
  // representation of int has no trap representations
  int i;
  unsigned char c = * ((unsigned char*)(&i));
  // does c now hold an unspecified value?
  printf("i=0x%x  c=0x%x\n",i,(int)c);
  printf("i=0x%x  c=0x%x\n",i,(int)c);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_representation_bytes_1.c: In function
’main’:
unspecified_value_representation_bytes_1.c:8:3:
warning: ’i’ is used uninitialized in this function
[-Wuninitialized]
 printf("i=0x%x
c=0x%x\n",i,(int)c);
  ^
unspecified_value_representati
on_bytes_1.c:6:17: warning: ’i’ is used uninitialized in
this function [-Wuninitialized]
 unsigned char c = *
((unsigned char*)(&i));
  ^
i=0x8 c=0x8
i=0x8 c=0x8
```

CLANG36-O2-NO-STRICT-ALIASING:

```
i=0x0 c=0x0
i=0x0 c=0x0
```

DEFACTO: `defined behaviour (printing nondeterministically true or false)`

---

[49] 95) If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called type punning). This might be a trap representation.

ISO: `unclear`

3.2.10 Q58. If one writes some but not all of the representation bytes of an uninitialized value, do the other representation bytes still hold unspecified values?

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: unclear DEFACTO-IMPL: unclear CERBERUS-DEFACTO: yes CHERI: unclear TIS: yes KCC: (flags indeterminate value used in an expression for this uninitialised unsigned char)

EXAMPLE (`unspecified_value_representation_bytes_4.c`):

```
#include <stdio.h>
int main() {
  // assume here that the implementation-defined
  // representation of int has no trap representations
  int i;
  printf("i=0x%x\n",i);
  printf("i=0x%x\n",i);
  unsigned char *cp = (unsigned char*)(&i);
  *(cp+1) = 0x22;
  // does *cp now hold an unspecified value?
  printf("*cp=0x%x\n",*cp);
  printf("*cp=0x%x\n",*cp);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_representation_bytes_4.c: In function
'main':
unspecified_value_representation_bytes_4.c:6:3:
warning: 'i' is used uninitialized in this function
[-Wuninitialized]
 printf("i=0x%x\n",i);
 ^
i=0x0
i=0x0
*cp=0x0
*cp=0x0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
unspecified_value_representation_bytes_4.c:6:21:
warning: variable 'i' is uninitialized when used here
[-Wuninitialized]
 printf("i=0x%x\n",i);

 ^
unspecified_value_representation_bytes_4.c:5:8:
note: initialize the variable 'i' to silence this
warning
 int i;
 ^
 = 0
1 warning
generated.
i=0x2200
i=0x2200
*cp=0x0
*cp=0x0
```

ISO: `unclear`

3.2.11 Q59. If one writes some but not all of the representation bytes of an uninitialized value, does a read of the whole value still give an unspecified value?

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: unclear DEFACTO-IMPL: unclear CERBERUS-DEFACTO: yes CHERI: unclear TIS: yes KCC: (flags indeterminate value used in an expression)

EXAMPLE (`unspecified_value_representation_bytes_2.c`):

```
#include <stdio.h>
int main() {
  // assume here that the implementation-defined
  // representation of int has no trap representations
  int i;
  printf("i=0x%x\n",i);
  printf("i=0x%x\n",i);
  * (((unsigned char*)(&i))+1) = 0x22;
  // does i now hold an unspecified value?
  printf("i=0x%x\n",i);
  printf("i=0x%x\n",i);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
unspecified_value_representation_bytes_2.c: In function
'main':
unspecified_value_representation_bytes_2.c:6:3:
warning: 'i' is used uninitialized in this function
[-Wuninitialized]
 printf("i=0x%x\n",i);
 ^
i=0x0
i=0x0
i=0x2200
i=0x2200
```

CLANG36-O2-NO-STRICT-ALIASING:

```
unspecified_value_representation_bytes_2.c:6:21:
warning: variable 'i' is uninitialized when used here
[-Wuninitialized]
 printf("i=0x%x\n",i);

 ^
unspecified_value_representation_bytes_2.c:5:8:
note: initialize the variable 'i' to silence this
warning
 int i;
 ^
 = 0
1 warning
generated.
i=0x2200
i=0x2200
i=0x2200
i=0x2200
```

DEFACTO: `defined behaviour (printing nondeterministic values)`

ISO: `unclear`

If one comments out the first two `printf`s, neither give a warning:

EXAMPLE (`unspecified_value_representation_bytes_3.c`):

```
#include <stdio.h>
int main() {
  // assume here that the implementation-defined
  // representation of int has no trap representations
  int i;
  // printf("i=0x%x\n",i);
  // printf("i=0x%x\n",i);
  * (((unsigned char*)(&i))+1) = 0x22;
  // does i now hold an unspecified value?
  printf("i=0x%x\n",i);
  printf("i=0x%x\n",i);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
i=0x2200
i=0x2200
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `unclear`

These two observations weakly suggest that Clang forgets that any part of the `int` is an unspecified value after a write of one of the representation bytes.

3.3 Structure and Union Padding

[Question 1/15 of our *What is C in practice? (Cerberus survey v2)*[50] relates to structure padding]

Standard The standard discusses two quite different kinds of padding: padding bits within the representation of integer types (6.2.6.2), and padding bytes in structures and unions. We focus here on the latter[51].

Padding can be added by an implementation between the members of a structure, or at the end of a structure or union, but not before the first member:

- 6.7.2.1p15 *"[...] There may be unnamed padding within a structure object, but not at its beginning."*
- 6.7.2.1p17 *"There may be unnamed padding at the end of a structure or union."*

Padding might be needed simply to ensure alignment:

(1) for performance, where some machine instructions are significantly faster when used on suitably aligned data than on misaligned data; or

(2) for correctness, where the machine instruction has the right width but must be suitably aligned to operate correctly (e.g. for some synchronisation instructions).

or to ensure that there is some spare space that the implementation is free to overwrite:

(a) for performance, where it is faster to use a wider machine memory access than the actual size of the data, and hence for the wider stores one has to allow spare space (otherwise the implementation would be wrong for concurrent accesses — just reading and writing back adjacent data would be incorrect); or

(b) for correctness, where the machine does not have an instruction that touches just the right width of footprint, and so again one needs spare space (e.g. again for some synchronisation instructions — though some cases of those are dealt with not by padding but by making the size of the relevant atomic type larger than one would expect from its precision).

We call these *alignment padding* and *space padding* respectively. There is also the space between the end of a union's current member and the size of the maximally sized member of its union type. The standard does not refer to this as padding, writing instead (6.2.6.1p7) *"...the bytes of the object representation that do not correspond to that member but do correspond to other members..."*, but it behaves in a similar way; we call it *union member padding*.

It is also conceivable that the compiler would reserve space in a structure or union type for its own purposes, e.g. to store a runtime representation of the name of the most recently written union member, or other bounds-checking or debug information, which would appear to the programmer as padding but which they would have to take care never to overwrite; we call this *metadata padding*.

Usage For the current processors that we are familiar with, we are not aware of any cases of (b) that are not handled by fixing the type size. Simple code with GCC does not seem to exhibit (a) except for struct copying, but we expect that compilers using vector instructions for optimisation might well do so. It's possible that implementations overwrite union member padding in a similar way. We would like more ground-truth data on all this.

Semantics Space padding is semantically more interesting that alignment padding as the semantics has to permit the implementation to overwrite those padding bytes. There are two main options:

(i) regard the padding bytes as holding unspecified values throughout the lifetime of the object, or

---

[50] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

[51] In fact, in the implementations we are most familiar with, there seem to be no integer-type padding bits, and we neglect them in our semantics. The C99 Rationale [2, p.43] refers to a machine that implements a 32-bit signed integer type with two 16-bit signed integers, with one of those two sign bits being deemed a padding bit. That machine is not named, so it is hard to tell whether it still exists.

(ii) write unspecified values to the padding bytes when any member of the object is written (or perhaps (ii′): when an adjacent member is written)

Standard The standard is unclear which of these it chooses. On the one hand, we have:

- 6.2.6.1p6 *"When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.51) [...]"* Footnote 51: *"Thus, for example, structure assignment need not copy any padding bits."*

that suggests (ii), with similar text for object member padding:

- 6.2.6.1p7 *"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 that member but do correspond to other members take unspecified values."*

This is reiterated in J.1 *Unspecified behavior* p1: *"The following are unspecified:"*

...

- *"The value of padding bytes when storing values in structures or unions (6.2.6.1)."*
- *"The values of bytes that correspond to union members other than the one last stored into (6.2.6.1)."*

...

Then the 6.7.9p10 text on initialization says that in some circumstances padding is initialized *"to zero bits"*: 6.7.9p10 *"If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate. If an object that has static or thread storage duration is not initialized explicitly, then:*

- *if it has pointer type, it is initialized to a null pointer;*
- *if it has arithmetic type, it is initialized to (positive or unsigned) zero;*
- *if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;*
- *if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;"*

This suggests that one can sometimes depend on the values of padding bytes, and hence that in the absence of writes to the structure, they are stable.

Note that this text does not say anything about the value of padding for an object (of automatic, static, or thread storage duration) that *is* initialized explicitly. An oversight?

On the other hand, 7.24.4.1 *The `memcmp` function* implies that padding bytes within structures *always* hold unspecified values: Footnote 310 *"The contents of "holes" used as padding for purposes of alignment within structure objects are indeterminate. Strings shorter than their allocated space and unions may also cause problems in comparison."* (even in the standard there are no trap representations here so indeterminate values are unspecified values).

Reading uninitialised local variables one might perhaps take to be undefined behaviour, but reading padding bytes (at least bytewise) surely has to be allowed, even if completely nondeterministic or symbolic-undefined with strict computation. And should that strictness extend to making a structure value an undefined value if one of its members is? Surely not.

3.3.1 Q60. Can structure-copy copy padding?

U:ISO

ISO: unclear DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes? TIS: unclear (the test seems to fail on the first print) KCC: yes (though also reports %x error)

EXAMPLE (`padding_struct_copy_1.c`):

```
#include <stdio.h>
#include <stddef.h>
#include <assert.h>
#include <inttypes.h>
typedef struct { char c; uint16_t u; } st;
int x;
void f(st* s2p, st* s1p) {
  *s2p=*s1p;
}
int main() {
  // check there is a padding byte between c and u
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  assert(offsetof(st,u)>offset_padding);
  st s1 = { .c = ’A’, .u = 0x1234 };
  unsigned char *padding1 =
    (unsigned char*)(&s1) + offset_padding;
  //  printf("*padding1=0x%x\n",(int)*padding1);
  *padding1 = 0xBA;
  printf("*padding1=0x%x\n",(int)*padding1);
  st s2;
  unsigned char *padding2 =
    (unsigned char*)(&s2) + offset_padding;
  // can this print something other than 0xBA then the
  // last line print 0xBA ?
  printf("*padding2=0x%x\n",(int)*padding2);//warn
  f(&s2,&s1);   //s2 = s1;
  printf("*padding2=0x%x\n",(int)*padding2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
padding_struct_copy_1.c: In function ’main’:
padding_struct_copy_1.c:25:3: warning: ’*((void
*)&s2+1)’ is used uninitialized in this function
[-Wuninitialized]
 printf("*padding2=0x%x\n",(int)*pad
ding2);//warn
 ^
*padding1=0xba
*padding2=0x0
*padding2=0xba
```

CLANG36-O2-NO-STRICT-ALIASING:

```
*padding1=0xba
*padding2=0x0
*padding2=0xba
```

DEFACTO: `defined behaviour (printing 0xBA then two nondeterministic values)`

ISO: `unclear`

(`padding_struct_copy_2.c` is the same with the padding at the end of the struct:

EXAMPLE (`padding_struct_copy_2.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
padding_struct_copy_2.c: In function 'main':
padding_struct_copy_2.c:25:3: warning: '*((void
*)&s2+3)' is used uninitialized in this function
[-Wuninitialized]
 printf("*padding2=0x%x\n",(int)*pad
ding2);//warn
 ^
*padding1=0xba
*padding2=0x0
*padding2=0xba
```

CLANG36-O2-NO-STRICT-ALIASING:

```
*padding1=0xba
*padding2=0x0
*padding2=0xba
```

However, slightly surprisingly, in the following example neither GCC nor Clang appear to recognise that copying the two members of the structure (with one-byte and two-byte instructions) could be optimised to a single four-byte copy:

EXAMPLE (`padding_struct_members_copy.c`):

```
#include <stdio.h>
#include <stddef.h>
#include <assert.h>
#include <inttypes.h>
typedef struct { char c; uint16_t u; } st;
int x;
int main() {
  // check there is a padding byte between c and u
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  assert(offsetof(st,u)>offset_padding);
  st s1 = { .c = 'A', .u = 0x1234 };
  unsigned char *padding1 =
    (unsigned char*)(&s1) + offset_padding;
  //  printf("*padding1=0x%x\n",(int)*padding1);
  *padding1 = 0xBA;
  printf("*padding1=0x%x\n",(int)*padding1);
  st s2;
  unsigned char *padding2 =
    (unsigned char*)(&s2) + offset_padding;
  printf("*padding2=0x%x\n",(int)*padding2);//warn
  s2.c = s1.c;
  s2.u = s1.u;
  printf("*padding2=0x%x\n",(int)*padding2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
padding_struct_members_copy.c: In function 'main':
padding_struct_members_copy.c:20:3: warning: '*((void
*)&s2+1)' is used uninitialized in this function
[-Wuninitialized]
 printf("*padding2=0x%x\n",(int)*pad
ding2);//warn
 ^
*padding1=0xba
*padding2=0x0
*padding2=0x0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
*padding1=0xba
*padding2=0x0
*padding2=0x0
```

DEFACTO: `defined behaviour (printing 0xBA then two nondeterministic values)`

ISO: `unclear`

`padding_struct_copy_3.c` is similar except with the copy in a separate function:

EXAMPLE (`padding_struct_copy_3.c`):

GCC-5.3-O2-NO-STRICT-ALIASING:

```
padding_struct_copy_3.c: In function 'main':
padding_struct_copy_3.c:24:3: warning: '*((void
*)&s2+1)' is used uninitialized in this function
[-Wuninitialized]
 printf("*padding2=0x%x\n",(int)*pad
ding2);//warn
 ^
*padding1=0xba
*padding2=0x0
*padding2=0x0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
*padding1=0xba
*padding2=0x0
*padding2=0x0
```

DEFACTO: `defined behaviour (printing 0xBA then two nondeterministic values)`

ISO: `unclear`

Nonetheless, we presume that a reasonable compiler might combine member writes. And that it might be dependent on inlining and code motion, and so that one cannot tell locally syntactically whether a write is “really” to a single struct member or whether the padding might be affected by combining it with writes of adjacent members?

Similarly, when we think about writing a struct member to a malloc'd region, differentiating between a write of the value qua the struct member and a write of the value simply of its underlying type is problematic, as optimisations inlining might convert the latter to the former? Unclear.

3.3.2 Q61. After an explicit write of a padding byte, does that byte hold a well-defined value? (not an unspecified value)

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: unclear – well-defined assumed for security leak prevention and CAS? DEFACTO-IMPL: unclear – well-defined? CERBERUS-DEFACTO: well-defined CHERI: well-defined? TIS: well-defined (surprisingly so, given the previous test result) KCC: well-defined

EXAMPLE (`padding_unspecified_value_1.c`):

```
#include <stdio.h>
#include <stddef.h>
typedef struct { char c; float f; int i; } st;
int main() {
  // check there is a padding byte between c and f
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  if (offsetof(st,f)>offset_padding) {
      st s;
      unsigned char *p = ((unsigned char*)(&s))
        + offset_padding;
      *p = 'A';
      unsigned char c1 = *p;
      // does c1 hold 'A', not an unspecified value?
      printf("c1=%c\n",c1);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
c1=A
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (printing A)`

ISO: `unclear`

The observations (of `A`) don't constrain the answer to this question.

In the ISO standard, for objects with static, thread, or automatic storage durations, and leaving aside unions, for each byte it's fixed whether it's a padding byte or not for the lifetime of the object, and one could conceivably regard the padding bytes as being unspecified values irrespective of any explicit writes to them (for a union, the padding status of a byte depends on which member the union "currently contains"). But for objects with allocated storage duration, that is at odds with the idea that a malloc'd region can be reused.

In practice we imagine (though without data) that "wide writes" for a single struct member only ever extend over the preceeding and following padding (or perhaps just only the following padding). Then the fact that concurrent access to distinct members is allowed (§3.3.12, p.60) constrains wide writes to not touch other members, at least in the absence of sophisticated analysis. There is again an issue here if `memcmp` or uniform hashing of structure representations is desired; it is debatable what circumstances one might reasonable expect those to work.

There is also a security-relevant issue here: one might want an assurance that potentially secret data does not leak into reads from padding bytes, and hence might (a) explicitly clear those bytes and (b) rely on the compiler not analysing that those bytes contain unspecified values and hence using values that happen to be found in registers in place of reads.

3.3.3 Q62. 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)

U:ISO

ISO: unclear DEFACTO-USAGE: unspecified value DEFACTO-IMPL: unspecified value CERBERUS-DEFACTO: unspecified value CHERI: unspecified value TIS: test not supported (tis bug, reported and fixed) KCC: (reports error for printing signed `int` with `%x`)

EXAMPLE (`padding_unspecified_value_2.c`):

```
#include <stdio.h>
#include <stddef.h>
typedef struct { char c; float f; int i; } st;
int main() {
  // check there is a padding byte between c and f
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  if (offsetof(st,f)>offset_padding) {
      st s;
      unsigned char *p =
        ((unsigned char*)(&s)) + offset_padding;
      *p = 'B';
      s = (st){ .c='E', .f=1.0, .i=1};
      unsigned char c2 = *p;
      // does c2 hold 'B', not an unspecified value?
      printf("c2=0x%x\n",(int)c2);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
c2=0x42
```

CLANG36-O2-NO-STRICT-ALIASING:

```
c2=0x0
```

DEFACTO: `defined behaviour (printing a nondeterministic value)`

ISO: `unclear (printing an unspecified value?)`

Here we see reads both of `B` and of `0x0`.

Changing the example to one in which the compiler might naturally use a 4-byte copy, we sometimes see an overwrite of the padding byte on the write of the struct value:

EXAMPLE (`padding_unspecified_value_3.c`):

```
#include <stdio.h>
#include <stddef.h>
#include <inttypes.h>
#include <assert.h>
typedef struct { char c; uint16_t u; } st;
int main() {
  // check there is a padding byte between c and u
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  assert(offsetof(st,u)>offset_padding);
  st s;
```

```
  unsigned char *p =
    ((unsigned char*)(&s)) + offset_padding;
  *p = 'B';
  s = (st){ .c='E', .u=1};
  unsigned char c = *p;
  // does c hold 'B', not an unspecified value?
  printf("c=0x%x\n",(int)c);
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
c=0x42
```

CLANG36-O2-NO-STRICT-ALIASING:

```
c=0x0
```

DEFACTO: `defined behaviour (printing a nondeterministic value)`

ISO: `unclear (printing an unspecified value?)`

and again here, copying another struct value on top as a whole:

EXAMPLE (`padding_unspecified_value_4.c`):

```
#include <stdio.h>
#include <stddef.h>
#include <inttypes.h>
#include <assert.h>
typedef struct { char c; uint16_t u; } st;
int main() {
  // check there is a padding byte between c and u
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  assert(offsetof(st,u)>offset_padding);
  st s;
  unsigned char *p =
    ((unsigned char*)(&s)) + offset_padding;
  *p = 'B';
  st s2 = { .c='E', .u=1};
  s = s2;
  unsigned char c = *p;
  // does c hold 'B', not an unspecified value?
  printf("c=0x%x\n",(int)c);
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
padding_unspecified_value_4.c: In function 'main':
padding_unspecified_value_4.c:15:5: warning: '*((void
*)&s+1)' is used uninitialized in this function
[-Wuninitialized]
 s = s2;
 ^
c=0x0
```

CLANG36-O2-NO-STRICT-ALIASING:

```
c=0x0
```

DEFACTO: `defined behaviour (printing a nondeterministic value)`

ISO: `unclear (printing an unspecified value?)`

3.3.4 Q63. 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)

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: unclear – unspecified value? DEFACTO-IMPL: unclear – unspecified value? CERBERUS-DEFACTO: unspecified value CHERI: unclear – unspecified value? TIS: well-defined value KCC: well-defined value

EXAMPLE (`padding_unspecified_value_7.c`):

```
#include <stdio.h>
#include <stddef.h>
typedef struct { char c; float f; int i; } st;
int main() {
  // check there is a padding byte between c and f
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  if (offsetof(st,f)>offset_padding) {
      st s;
      unsigned char *p =
        ((unsigned char*)(&s)) + offset_padding;
      *p = 'C';
      s.c = 'A';
      s.f = 1.0;
      s.i = 42;
      unsigned char c3 = *p;
      // does c3 hold 'C', not an unspecified value?
      printf("c3=%c\n",c3);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
c3=C
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `unspecified value`

ISO: `unclear (printing an unspecified value?)`

3.3.5 Q64. 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)

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: unclear DEFACTO-IMPL: unclear CERBERUS-DEFACTO: unspecified value CHERI: unspecified value TIS: well-defined zero KCC: well-defined zero (though also reports %x error)

EXAMPLE (`padding_unspecified_value_8.c`):

```
#include <stdio.h>
#include <stddef.h>
typedef struct { char c; float f; int i; } st;
int main() {
  // check there is a padding byte between c and f
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  if (offsetof(st,f)>offset_padding) {
      st s;
      unsigned char *p =
        ((unsigned char*)(&s)) + offset_padding;
```

```
      *p = 0;
      s.c = 'A';
      s.f = 1.0;
      s.i = 42;
      unsigned char c3 = *p;
      // does c3 hold 0, not an unspecified value?
      printf("c3=0x%x\n",c3);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
c3=0x0
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `unspecified value`

ISO: `unclear (printing an unspecified value?)`

(There was a typo `c` in an earlier version of this test.)

This is perhaps the most relevant of these cases in practice, covering the case where the whole footprint of the struct has been filled with zero before use, and also covering the case where all members of the struct have been written (and hence where compilers might coalesce the writes). By requiring the explicit write to be of zero, compilers could implement this either by preserving the in-memory padding byte value or by writing a constant zero to it. Whether that would be sound w.r.t. actual practice is unclear.

3.3.6 Q65. 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)

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: well-defined value? DEFACTO-IMPL: well-defined value? CERBERUS-DEFACTO: well-defined value CHERI: well-defined value? TIS: well-defined value KCC: well-defined value

EXAMPLE (`padding_unspecified_value_5.c`):

```
#include <stdio.h>
#include <stddef.h>
typedef struct { char c; float f; int i; } st;
int main() {
  // check there is a padding byte between c and f
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  if (offsetof(st,f)>offset_padding) {
      st s;
      unsigned char *p =
        ((unsigned char*)(&s)) + offset_padding;
      *p = 'C';
      s.i = 42;
      unsigned char c3 = *p;
      // does c3 hold 'C', not an unspecified value?
      printf("c3=%c\n",c3);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
c3=C
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (printing C)`

ISO: `unclear (printing an unspecified value?)`

These observations (of `C`) don't constrain the answer to this question.

3.3.7 Q66. 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)

U:ISO U:DEFACTO

ISO: unclear DEFACTO-USAGE: well-defined value? DEFACTO-IMPL: well-defined value? CERBERUS-DEFACTO: well-defined value CHERI: well-defined value? TIS: well-defined value KCC: well-defined value

EXAMPLE (`padding_unspecified_value_6.c`):

```
#include <stdio.h>
#include <stddef.h>
void g(char *c, float *f) {
  *c='A';
  *f=1.0;
}
typedef struct { char c; float f; int i; } st;
int main() {
  // check there is a padding byte between c and f
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  if (offsetof(st,f)>offset_padding) {
      st s;
      unsigned char *p =
        ((unsigned char*)(&s)) + offset_padding;
      *p = 'D';
      g(&s.c, &s.f);
      unsigned char c4 = *p;
      // does c4 hold 'D', not an unspecified value?
      printf("c4=%c\n",c4);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
c4=D
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (printing D)`

ISO: `unclear (printing an unspecified value?)`

These observations (of `D`) don't constrain the answer to this question.

3.3.8 Q67. Can one use a malloc'd region for a union that is just big enough to hold the subset of members that will be used?

U:ISO U:DEFACTO D:ISO-VS-DEFACTO

ISO: unclear – no? DEFACTO-USAGE: yes? DEFACTO-IMPL: yes? CERBERUS-DEFACTO: no CHERI: unclear? TIS: yes KCC: no (flags UB Trying to write outside the bounds of an object)

One of our respondents remarks that it is an acceptable idiom, if one has a union but knows that only some of the members will be used, to malloc something only big enough for those members.

EXAMPLE (`padding_subunion_1.c`):

```
#include <stdio.h>
#include <stdlib.h>
typedef struct { char c1; } st1;
typedef struct { float f2; } st2;
typedef union { st1 s1; st2 s2; } un;
int main() {
  // is this free of undefined behaviour?
  un* u = (un*)malloc(sizeof(st1));
  u->s1.c1 = 'a';
  printf("u->s1.c1=0x%x\n",(int)u->s1.c1);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
u->s1.c1=0x61
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `undefined behaviour`

ISO: `unclear - undefined behaviour?`

If that is supported, then presumably one can rely on the compiler, for a union member write, not writing beyond the footprint of that member:

EXAMPLE (`padding_subunion_2.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
typedef struct { char c1; } st1;
typedef struct { float f2; } st2;
typedef union { st1 s1; st2 s2; } un;
int main() {
  // check that st2 is bigger than st1
  // (otherwise the test is uninteresting)
  assert(sizeof(st2) > sizeof(st1));
  // is this free of undefined behaviour?
  unsigned char* p = malloc(sizeof(st1)+sizeof(int));
  un* pu = (un*)p;
  char *pc = (char*)(p + sizeof(st1));
  *pc='B';
  pu->s1.c1 = 'A';
  // is this guaranteed to read 'B'?
  unsigned char c = *pc;
  printf("c=0x%x\n",(int)c);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
c=0x42
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (printing a nondeterministic value)`

ISO: `unclear`

But that is at odds with the idea that after writing a union member, the footprint of the union holds unspecified values beyond the footprint of that member.

If one does want this to be allowed, should be be allowed only when the lvalue is manifestly part of the union, or is it just a fact about struct writes, that they are never widened (very much or at all)?

3.3.9 More remarks on padding

One respondent remarks:

- The C frontend of Clang will make packed structs with `i8` members wherever padding is needed (because the IR is too underspecified). So the mid-level optimisers don't know what's padding and what's not
- A struct copy might really emit particular loads and stores for a small struct (rather than a `memcpy`); in that case it wouldn't copy the padding.
- Doing wide writes to narrow members was mostly an alpha thing? Not sure on x86 if there are shorter encodings that do that. Something in LLVM "scalar evolution" optimisation might do this, but probably only when they know they're working over a bunch of members.
- He hasn't actually seen generic hash-all-the-bytes-of-a-struct code. Maybe for deduplication and content-addressable stores? Also for encrypting structs and doing CRCs. But the only code he knows care about this use byte arrays or packed structs. Another respondent remarks he thinks he has seen code that does something like this - in one of the SPEC CPU2006 benchmarks.

With respect to the semantic options outlined earlier, with (i), continuously unspecified values for padding bytes, `c1` gets an unspecified value despite the fact that `'A'` was just written to the address that `c1` is read from. And `c2`, `c3`, and `c4` are likewise all unspecified values.

With (ii), `c1` is guaranteed to get `'A'`, but `c2` gets an unspecified value, as the structure members are all written to after the write of `*p='B'`. `c3` similarly gets an unspecified value due to the intervening write of `s.i`, despite the fact that `i` is not adjacent to the padding pointed to by `p`.

With (ii′), `c2` gets an unspecified value but `c3` is guaranteed to get `'C'`.

Finally, with either (ii) or (ii′), we believe that `c4` should be guaranteed to get `'D'`, unaffected by the writes *within* members of `s` that are performed by `f` (which might be in a different compilation unit).

For union member padding, we presume that the standard semantics should synthesise explicit writes of undefined values whenever a short member is written. But if compilers don't walk over that space, the concrete semantics need not and both can leave it stable inbetween.

If compilers ever do write to structure padding, then this interacts with the use of a pointer to access a structure with a similar prefix, illustrated in Example `cast_struct_same_prefix.c` of §2.15.1 (p.36). The most plausible case seems to be for a compiler to make a wider-than-expected write starting at the base address of the member representation but continuing strictly beyond it, but the padding after a structure member is determined (in the common ABIs, as discussed above) by the alignment requirement of the *subsequent* member, so the structures would have to have similar prefixes up to one member past the last one used for write accesses.

There is also an interaction between padding and the definition of data races: should a programmer access to padding be regarded as racing with a non-happens-before-related write to any member of the structure, or to an adjacent (or preceding) member of the structure?

Padding also relates to `memcmp` and to related functions, e.g. hash functions that hash all the representation bytes of a structure. The 7.24.4.1 `memcmp` text quoted above suggests that `memcmp` over structures that contain padding is not useful, and with (i), in our symbolic, strict interpretation of unspecified values (2b of §3.2, p.45) it (and hash functions) will return the unspecified value for all such. But it appears that in at least some cases in practice one relies on the padding have been initialised and not overwritten.

3.3.10 Q68. 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?

ISO: yes? (though not made explicit) DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: (fails with a mistaken OOB pointer UB)

We also have to ask whether the compiler can use padding bytes for its own purposes, e.g. to hold some array bounds information or dynamic representations of union tags. In other words, is it legal to copy a structure or union by copying just the representation bytes of its member(s), and writing junk into the padding bytes?

EXAMPLE (`padding_struct_copy_of_representation_bytes.c`):

```
#include <stdio.h>
#include <stddef.h>
#include <string.h>
typedef struct { char c; float f; } st;
int main() {
  st s1 = {.c = ’A’, .f = 1.0 };
  st s2;
  memcpy(&(s2.c), &(s1.c), sizeof(char));
  memset(&(s2.c)+sizeof(char),’X’,
    offsetof(st,f)-offsetof(st,c)-sizeof(char));
  memcpy(&(s2.f), &(s1.f), sizeof(float));
  //memset(&(s2.f)+sizeof(float),’Y’,
  //  sizeof(st)-offsetof(st,f)-sizeof(float));
  // is s2 now a copy of s1?
  printf("s2.c=%c s2.f=%f\n",s2.c,s2.f);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
s2.c=A s2.f=1.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour (s2.c=A s2.f=1.000000)`

ISO: `defined behaviour (s2.c=A s2.f=1.000000)`

We are not aware of any implementations that use padding bytes in that way, and for a de facto sematics it should be legal to copy a structure or union by just copying the member representation bytes.

3.3.11 Q69. Can one read an object as aligned words without regard for the fact that the object’s extent may not include all of the last word?

D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: no? CHERI: ? TIS: no (flags OOB read) KCC: flags UB for a pointer conversion alignment (arguably correctly), UB for an effective type error (debatable), and an OOB read (mistaken)

[Question 14/15 of our *What is C in practice? (Cerberus survey v2)*[52] relates to this.]

This is a question from the CHERI ASPLOS paper, where they write: *“This is used as an optimization for `strlen()` in FreeBSD libc. While this is undefined behavior in C, it works in systems with pagebased memory protection mechanisms, but not in CHERI where objects have byte granularity. We have found this idiom only in FreeBSD’s libc, as reported by valgrind.”*

EXAMPLE (`cheri_08_last_word.c`):

```
#include <assert.h>
#include <stdio.h>
#include <inttypes.h>
char c[5];
int main() {
  char *cp = &(c[0]);
  assert(sizeof(uint32_t) == 4);
  uint32_t x0 = *((uint32_t *)cp);
  // does this have defined behaviour?
  uint32_t x1 = *((uint32_t *)(cp+4));
  printf("x0=%x  x1=%x\n",x0,x1);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
x0=0 x1=0
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `undefined behaviour`

3.3.12 Q70. Does concurrent access to two (non-bitfield) distinct members of a structure constitute a data race?

ISO: no DEFACTO-USAGE: no DEFACTO-IMPL: no CERBERUS-DEFACTO: no CHERI: no TIS: no concurrency support

This is part of the C11 concurrency model.

---

[52] `www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`

It puts an upper bound on the “wide writes” that a compiler might do for a struct member write: they cannot overlap any other members.

3.3.13 Q71. Does concurrent access to a structure member and a padding byte of that structure constitute a data race?

U:ISO U:DEFACTO
ISO: unclear DEFACTO-USAGE: unclear DEFACTO-IMPL: unclear CERBERUS-DEFACTO: unclear CHERI: unclear TIS: no concurrency support

It is hard to imagine that this will matter for any reasonable code, but any semantics will have to decide one way or the other, and it will impact the design of race detectors that aim to be complete.

3.3.14 Q72. Does concurrent (read or write) access to an unspecified value constitute a data race?

U:ISO U:DEFACTO
ISO: unclear DEFACTO-USAGE: unclear DEFACTO-IMPL: unclear CERBERUS-DEFACTO: unclear CHERI: unclear TIS: no concurrency support

One might conceivably want to allow this, to allow concurrent accesses to adjacent members of a struct to write unspecified values to padding without creating a bogus data race. It could be restricted to just padding bytes, but it is simpler to allow races on all unspecified-value accesses.

(Note that you don’t see those accesses in a naive source semantics, but in a semantics in which writes to a member also write unspecified values to the adjacent padding on both sides, it matters, and in Core and the memory model those writes have to be there.)

4. Effective Types

Paragraphs 6.5p{6,7} of the standard introduce *effective types*. These were added to C in C99 to permit compilers to do optimisations driven by type-based alias analysis, by ruling out programs involving unannotated aliasing of references to different types (regarding them as having undefined behaviour). This is one of the less clear, less well-understood, and more controversial aspects of the standard, as one can see from various GCC and Linux Kernel mailing list threads[53][54] [55] and blog postings[56][57][58][59][60]. The type-based aliasing question of our preliminary survey was the only one which received a unanimous response: “don’t know”.

Several major systems software projects, including the Linux Kernel, the FreeBSD Kernel, and PostgreSQL (though not Apache) disable type-based alias analyis with the `-fno-strict-aliasing` compiler flag [53]. Our de facto standard semantics should either simply follow that or have a corresponding switch; for the moment we go for the former.

Standard *“6.5p6 The* effective type *of an object for an access to its stored value is the declared type of the object, if any.87) If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using* `memcpy` *or* `memmove`*, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.*

6.5p7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:88)

- *a type compatible with the effective type of the object,*
- *a qualified version of a type compatible with the effective type of the object,*
- *a type that is the signed or unsigned type corresponding to the effective type of the object,*
- *a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,*
- *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*
- *a character type.*

Footnote 87) Allocated objects have no declared type.

Footnote 88) The intent of this list is to specify those circumstances in which an object may or may not be aliased.”

As Footnote 87 says, allocated objects (from `malloc`, `calloc`, and presumably any fresh space from `realloc`) have no declared type, whereas objects with static, thread, or automatic storage durations have some declared type.

For the latter, 6.5p{6,7} say that the effective types are fixed and that their values can only be accessed by an lvalue that is similar (“compatible”, modulo signedness and qual-

---

[53] https://gcc.gnu.org/ml/gcc/2010-01/msg00013.html

[54] https://lkml.org/lkml/2003/2/26/158

[55] http://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg01647.html

[56] http://blog.regehr.org/archives/959

[57] http://cellperformance.beyond3d.com/articles/2006/06/understanding-strict-aliasing.html

[58] http://davmac.wordpress.com/2010/02/26/c99-revisited/

[59] http://dbp-consulting.com/tutorials/StrictAliasing.html

[60] http://stackoverflow.com/questions/2958633/gcc-strict-aliasing-and-horror-stories

ifiers), an aggregate or union containing such a type, or (to access its representation) a character type.

For the former, the effective type is determined by the type of the last write, or, if that is done by a `memcpy`, `memmove`, or user-code char array copy, the effective type of the source.

4.1 Basic effective types

4.1.1 Q73. Can one do type punning between arbitrary types?

ISO: no DEFACTO-USAGE: yes, with `-fno-strict-aliasting` DEFACTO-IMPL: yes, with `-fno-strict-aliasting` CERBERUS-DEFACTO: ? CHERI: ? TIS: yes KCC: no (flags effective-type UB)

EXAMPLE (`effective_type_1.c`):

```
#include <stdio.h>
#include <inttypes.h>
#include <assert.h>
void f(uint32_t *p1, float *p2) {
  *p1 = 2;
  *p2 = 3.0; // does this have defined behaviour?
  printf("f: *p1 = %" PRIu32 "\n",*p1);
}
int main() {
  assert(sizeof(uint32_t)==sizeof(float));
  uint32_t i = 1;
  uint32_t *p1 = &i;
  float *p2;
  p2 = (float *)p1;
  f(p1, p2);
  printf("i=%" PRIu32 "  *p1=%" PRIu32
         "  *p2=%f\n",i,*p1,*p2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
f: *p1 = 1077936128
i=1077936128 *p1=1077936128 *p2=3.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour iff -no-strict-aliasing, with implementation-defined value for the first three prints`

ISO: `undefined behaviour`

With `-fstrict-aliasing` (the default for GCC here), GCC assumes in the body of `f` that the write to `*p2` cannot affect the value of `*p1`, printing 2 (instead of the integer value of the representation of `3.0` that would the most recent write in a concrete semantics):

```
gcc-4.8 -O2 -fstrict-aliasing -std=c11 -pedantic -Wall
  -Wextra -pthread   effective_types_13.c  && ./a.out
f: *p1 = 2
i=1077936128  *p1=1077936128  *p2=3.000000
```

while with `-fno-strict-aliasing` (as used in the Linux kernel, among other places) it does not assume that:

```
gcc-4.8 -O2 -fno-strict-aliasing -std=c11 -pedantic -Wall
  -Wextra -pthread effective_types_13.c  && ./a.out
f: *p1 = 1077936128
i=1077936128  *p1=1077936128  *p2=3.000000
```

The former behaviour can be explained by regarding the program as having undefined behaviour, due to the write of the `uint32_t i` with a `float*` lvalue.

We give another basic effective type example below, here just involving integer types and without the function call.

EXAMPLE (`effective_type_10.c`):

```
#include <stdio.h>
#include <stdint.h>
int main() {
  int32_t x;
  uint16_t y;
  x = 0x44332211;
  y = *(uint16_t *)&x; // defined behaviour?
  printf("x=%i  y=0x%x\n",x,y);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
x=1144201745 y=0x2211
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `undefined behaviour`

4.1.2 Q74. Can one do type punning between distinct but isomorphic structure types?

ISO: no DEFACTO-USAGE: yes, with `-fno-strict-aliasting` DEFACTO-IMPL: yes, with `-fno-strict-aliasting` CERBERUS-DEFACTO: ? CHERI: ? TIS: yes KCC: yes (contrary to ISO)

Similar compiler behaviour occurs with pointers to two distinct but isomorphic structure types:

EXAMPLE (`effective_type_2.c`):

```
#include <stdio.h>
typedef struct { int  i1; } st1;
typedef struct { int  i2; } st2;
void f(st1* s1p, st2* s2p) {
  s1p->i1 = 2;
  s2p->i2 = 3;
  printf("f: s1p->i1 = %i\n",s1p->i1);
}
int main() {
  st1 s = {.i1 = 1};
  st1 * s1p = &s;
  st2 * s2p;
  s2p = (st2*)s1p;
  f(s1p, s2p); // defined behaviour?
  printf("s.i1=%i  s1p->i1=%i  s2p->i2=%i\n",
         s.i1,s1p->i1,s2p->i2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
f: s1p->i1 = 3
s.i1=3 s1p->i1=3 s2p->i2=3
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour iff -no-strict-aliasing`

ISO: `undefined behaviour`

```
gcc-4.8 -O2 -fstrict-aliasing -std=c11 -pedantic -Wall
  -Wextra -pthread   effective_types_12.c  && ./a.out
f: s1p->i1 = 2
s.i1=3  s1p->i1=3  s2p->i2=3
```

```
gcc-4.8 -O2 -fno-strict-aliasing -std=c11 -pedantic -Wall
  -Wextra -pthread effective_types_12.c  && ./a.out
f: s1p->i1 = 3
s.i1=3  s1p->i1=3  s2p->i2=3
```

4.2 Effective types and character arrays

4.2.1 Q75. Can an unsigned character array with static or automatic storage duration be used (in the same way as a `malloc`'d region) to hold values of other types?

D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: no (w.r.t. compiler respondents) CERBERUS-DEFACTO: yes (for `-fno-strict-aliasing`) CHERI: yes TIS: yes? test not supported – fails to find `stdalign.h` KCC: no (flags alignment and effective type errors – though the `Alignas` makes the former incorrect)

[Question 11/15 of our *What is C in practice? (Cerberus survey v2)*[61] relates to this.]

A literal reading of the effective type rules prevents the use of an unsigned character array as a buffer to hold values of other types (as if it were an allocated region of storage). For example, the following has undefined behaviour due to a violation of 6.5p7 at the access to `*fp`[62].

EXAMPLE (`effective_type_3.c`):

```
#include <stdio.h>
#include <stdalign.h>
int main() {
  _Alignas(float) unsigned char c[sizeof(float)];
  float *fp = (float *)c;
  *fp=1.0; // does this have defined behaviour?
  printf("*fp=%f\n",*fp);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`*fp=1.000000`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour iff -no-strict-aliasing`

ISO: `undefined behaviour`

In the de facto semantics we imagine this should be allowed.

Even bytewise copying of a value via such a buffer leads to unusable results in the standard:

EXAMPLE (`effective_type_4.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdalign.h>
int main() {
  _Alignas(float) unsigned char c[sizeof(float)];
  // c has effective type char array
  float f=1.0;
  memcpy((void*)c, (const void*)(&f), sizeof(float));
  // c still has effective type char array
  float *fp = (float *) malloc(sizeof(float));
  // the malloc'd region initially has no effective type
  memcpy((void*)fp, (const void*)c, sizeof(float));
  // does the following have defined behaviour?
  // (the ISO text says the malloc'd region has effective type
  // unsigned char array, not float, and hence that
  // the following read has undefined behaviour)
  float g = *fp;
  printf("g=%f\n",g);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

`g=1.000000`

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour iff -no-strict-aliasing`

ISO: `undefined behaviour`

This seems to be unsupportable for a systems programming language: a character array and malloc'd region should be interchangeably usable, and this too should be allowed in the de facto standard semantics.

4.3 Effective types and subobjects

Another difficulty with the standard text relates to the treatment of subobjects: members of structures and unions written into allocated regions. Suppose we write a single member of a structure into a fresh allocated region, then does

(i) the footprint of the member take on an effective type as the type of that struct member, or

(ii) the footprint of the member take on an effective type of the type of that structure member annotated as coming from that member of that structure type, or

(iii) the footprint of the whole structure take on the structure type as its effective type?

4.3.1 Q76. After writing a structure to a malloc'd region, can its members can be accessed via pointers of the individual member types?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: yes TIS: yes KCC: yes

This is uncontroversial.

EXAMPLE (`effective_type_5.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
int main() {
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  st1 s1 = { .c1='A', .f1=1.0};
  *((st1 *)p) = s1;
  float *pf = &(((st1 *)p)->f1);
  // is this free of undefined behaviour?
  float f = *pf;
  printf("f=%f\n",f);
```

---

[61] www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html

[62] This reasoning presumes that the conversion of the `(float *)c` cast gives a usable result — the conversion is permitted by 6.3.2.3p7 but the standard text only guarantees a roundtrip property.

```
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
f=1.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour`

ISO: `defined behaviour`

4.3.2 Q77. Can a non-character value be read from an uninitialised malloc'd region?

D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: yes (for `-fno-strict-aliasing`) DEFACTO-IMPL: yes (for `-fno-strict-aliasing`) CERBERUS-DEFACTO: yes (for `-fno-strict-aliasing`) CHERI: yes (for `-fno-strict-aliasing`) TIS: no KCC: no (looks like you can read but not print – flags UB Indeterminate value used in an expression)

EXAMPLE (`effective_type_6.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
int main() {
  void *p = malloc(sizeof(float)); assert (p != NULL);
  // is this free of undefined behaviour?
  float f = *((float *)p);
  printf("f=%f\n",f);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
f=0.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour iff -no-strict-aliasing, reading an unspecified value`

ISO: `undefined behaviour`

The effective type rules seem to deem this undefined behaviour.

4.3.3 Q78. After writing one member of a structure to a malloc'd region, can its other members be read?

D:ISO-VS-DEFACTO

ISO: no DEFACTO-USAGE: yes (for `-fno-strict-aliasing`) DEFACTO-IMPL: yes (for `-fno-strict-aliasing`) CERBERUS-DEFACTO: yes (for `-fno-strict-aliasing`) CHERI: yes (for `-fno-strict-aliasing`) TIS: no (similarly?) KCC: no (flags UB Indeterminate value used in an expression)

EXAMPLE (`effective_type_7.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
int main() {
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  ((st1 *)p)->c1 = 'A';
  // is this free of undefined behaviour?
  float f = ((st1 *)p)->f1;
  printf("f=%f\n",f);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
f=0.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour iff -no-strict-aliasing`

ISO: `undefined behaviour`

If the write should be considered as affecting the effective type of the footprint of the entire structure, then it would change the answer to `effective_type_5.c` here. It seems unlikely but not impossible that such an interpretation is desirable.

4.3.4 Q79. After writing one member of a structure to a malloc'd region, can a member of another structure, with footprint overlapping that of the first structure, be written?

U:ISO D:ISO-VS-DEFACTO

ISO: unclear DEFACTO-USAGE: yes (for `-fno-strict-aliasing`) DEFACTO-IMPL: yes (for `-fno-strict-aliasing`) CERBERUS-DEFACTO: yes (for `-fno-strict-aliasing`) CHERI: yes TIS: yes KCC: yes

EXAMPLE (`effective_type_8.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
typedef struct { char c2; float f2; } st2;
int main() {
  assert(sizeof(st1)==sizeof(st2));
  assert(offsetof(st1,c1)==offsetof(st2,c2));
  assert(offsetof(st1,f1)==offsetof(st2,f2));
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  ((st1 *)p)->c1 = 'A';
  // is this free of undefined behaviour?
  ((st2 *)p)->f2 = 1.0;
  printf("((st2 *)p)->f2=%f\n",((st2 *)p)->f2);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
((st2 *)p)->f2=1.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour`

ISO: `unclear`

Again this is exploring the effective type of the footprint of the structure type used to form the lvalue.

4.3.5 Q80. After writing a structure to a malloc’d region, can its members be accessed via a pointer to a different structure type that has the same leaf member type at the same offset?

D:ISO-VS-DEFACTO
ISO: no DEFACTO-USAGE: yes (for `-fno-strict-aliasing`) DEFACTO-IMPL: yes (for `-fno-strict-aliasing`) CERBERUS-DEFACTO: yes (for `-fno-strict-aliasing`) CHERI: yes iff `-fno-strict-aliasing`) TIS: yes KCC: yes

EXAMPLE (`effective_type_9.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
typedef struct { char c1; float f1; } st1;
typedef struct { char c2; float f2; } st2;
int main() {
  assert(sizeof(st1)==sizeof(st2));
  assert(offsetof(st1,c1)==offsetof(st2,c2));
  assert(offsetof(st1,f1)==offsetof(st2,f2));
  void *p = malloc(sizeof(st1)); assert (p != NULL);
  st1 s1 = { .c1=’A’, .f1=1.0};
  *((st1 *)p) = s1;
  // is this free of undefined behaviour?
  float f = ((st2 *)p)->f2;
  printf("f=%f\n",f);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
f=1.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour iff -no-strict-aliasing`

ISO: `undefined behaviour`

The standard seems to deem this undefined behaviour.

4.3.6 Q81. Can one access two objects, within a malloc’d region, that have overlapping but non-identical footprint?

U:ISO D:ISO-VS-DEFACTO
ISO: unclear - no? DEFACTO-USAGE: yes (for `-fno-strict-aliasing`) DEFACTO-IMPL: yes (for `-fno-strict-aliasing`; no without) CERBERUS-DEFACTO: yes (for `-fno-strict-aliasing`) CHERI: yes iff `-fno-strict-aliasing`) TIS: yes KCC: yes

Robbert Krebbers asks on the GCC list[63] whether *“GCC uses 6.5.16.1p3 of the C11 standard as a license to perform certain optimizations. If so, could anyone provide me an example program. In particular, I am interested about the “then the overlap shall be exact” part of 6.5.16.1p3: “If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.” ”*. Richard Biener replies with this example (rewritten here to print the result), saying that it will be optimised to print 1 and that this is basically effective-type reasoning.

EXAMPLE (`krebbers_biener_1.c`):

```
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
struct X { int i; int j; };
int foo (struct X *p, struct X *q) {
  // does this have defined behaviour?
  q->j = 1;
  p->i = 0;
  return q->j;
}
int main() {
  assert(sizeof(struct X) == 2 * sizeof(int));
  unsigned char *p = malloc(3 * sizeof(int));
  printf("%i\n", foo ((struct X*)(p + sizeof(int)),
                      (struct X*)p));
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
0
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `unclear`

5. Other Questions

5.1 Q82. Given a const-qualified pointer to an object defined with a non-const-qualified type, can the pointer be cast to a non-const-qualified pointer and used to mutate the object?

ISO: yes DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes CHERI: no TIS: yes KCC: yes

This is the *Deconst* idiom from the CHERI ASPLOS paper, where they write: *“Deconst refers to programs that remove the const qualifier from a pointer. This will break with any implementation that enforces the const at run time. 6.7.3.4 states: If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with nonconst-qualified type, the behavior is undefined. This means that such removal is permitted unless the object identified by the pointer is declared const, but this guarantee is very hard to make statically and the removal can violate programmer intent. We would like to be able to make a const pointer a guarantee that nothing that receives the pointer may write to the resulting memory. This allows const pointers to be passed across security-domain boundaries.”*

The current standard text is 6.7.3p6 *“If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. If 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, the behavior is undefined.133)”* and, in Appendix L, *“All undefined behavior shall be limited to bounded undefined behavior, except for*

---

[63] https://gcc.gnu.org/ml/gcc/2015-03/msg00083.html

the following which are permitted to result in critical undefined behavior: [...] 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).”

EXAMPLE (`cheri_01_deconst.c`):

```
#include <stdio.h>
int main() {
  int x=0;
  const int *p = (const int *)&x;
  //are the next two lines free of undefined behaviour?
  int *q = (int*)p;
  *q = 1;
  printf("x=%i  *p=%i  *q=%i\n",x,*p,*q);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
x=1 *p=1 *q=1
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

DEFACTO: `defined behaviour`

ISO: `defined behaviour`

5.2 Q83. Can `char` and `unsigned char` be assumed to be 8-bit bytes?

ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes?

5.3 Q84. Can one assume two’s-complement arithmetic?

ISO: no DEFACTO-USAGE: yes DEFACTO-IMPL: yes CERBERUS-DEFACTO: yes?

5.4 Q85. In the absence of floating point, can one assume that no base types have multiple representations of the same value?

U:DEFACTO

ISO: no DEFACTO-USAGE: yes? DEFACTO-IMPL: yes? (or perhaps pointer values?) CERBERUS-DEFACTO: yes?

This is not necessarily true for CHERI pointers, at least.

Where there are multiple representations, one has to consider the extent to which the representation bytes are stable.

6. Related Work

In this section we discuss some of the related work in a moderately in-depth way. For work that involves a model, a verification tool, or an implementation of much of C, a fully detailed comparison would involve going through each of our earlier questions one by one, considering both the intended semantics and any observable results for the test cases. This would require an extended discussion with the authors of each work, which at the time of writing we only just embarked on, though we do have experimental data and have had some limited discussion (including survey responses) for a few systems. Instead, here we consider the related work as it is described in the literature (with a subsection for each paper or group of papers), focussing on the motivating examples they give and checking whether they suggest additional questions.

We first consider several lines of work building memory models for C to support mechanised formal reasoning in a proof assistant. We begin with the fully concrete model used by Norrish, who aimed to make (aspects of) the ISO C90 standard precise:

- *C formalised in HOL; Norrish; PhD thesis 1998* [43], §6.1

Tuch et al. develop a concrete model used for the seL4 verification, aiming to provide a model that is sound for the particular C used in that work (a particular compiler and underlying architecture) rather than a model for either ISO or de facto standards in general.

- *A unified memory model for pointers; Tuch, Klein; LPAR 2005* [50], §6.2
- *Types, bytes, and separation logic; Tuch, Klein, Norrish; POPL 2007* [48], §6.3

Work by several groups on verified compilation has produced a number of models. These too are not trying to exactly capture either the ISO or the de facto standards in general, but rather to provide a semantics for the C-like language of some particular verified compiler, that justifies or eases reasoning about its compiler transformations. Most of these models are abstract, based on a block-ID/offset notion; the later work in this line aims at supporting more low-level programming idioms.

- *Formal verification of a C-like memory model and its uses for verifying program transformations; Leroy and Blazy; JAR 2008* [34], §6.4
- *CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency; Ševčík, Vafeiadis, Zappa Nardelli, Jagannathan, Sewell; POPL 2011, JACM 2013* [51, 52] , §6.5
- *The CompCert Memory Model, Version 2; Leroy, Appel, Blazy, Stewart; INRIA RR-7987 2012* [33], §6.6
- *Formal C semantics: CompCert and the C standard; Krebbers, Leroy, and Wiedijk; ITP 2014* [32], §6.7
- *A Precise and Abstract Memory Model for C using Symbolic Values, Besson, Blazy, and Wilke; APLAS 2014* [9], §6.8
- *A Concrete Memory Model for CompCert; Besson, Blazy, Wilke; ITP 2015* [10], §6.9
- *A formal C memory model supporting integer-pointer casts; Kang, Hur, Mansky, Garbuzov, Zdancewic, Vafeiadis; PLDI 2015* [25], §6.10

Work by Krebbers and by Krebbers and Wiedijk aims at a semantics *"corresponding to a significant part of [...] the C11 standard, as well as technology to enable verification of C programs in a standards compliant and compiler independent way"*:

- *The C standard formalized in Coq; Krebbers; PhD thesis 2015* [29] and also [27, 28, 30, 31], §6.11

Ellison et al. give another semantics for a substantial fragment of C, expressed as a rewrite system in the K framework rather than within an interactive prover:

- *An Executable Formal Semantics of C with Applications; Ellison and Roşu; POPL 2012* [18], and also [21, 22], §6.12

Cohen et al. describe the model used in their VCC system:

- *A precise yet efficient memory model for C; SSV 2009; Cohen, Moskał, Tobies, Schulte* [15], §6.13

A number of papers and blog posts look at undefined behaviour in C (much but not all of which concerns the memory and pointer behaviour we focus on here) from a systems point of view, without mathematical models:

- *Undefined Behavior: What Happened to My Code?; Wang, Chen, Cheung, Jia, Zeldovich, Kaashoek; APSys 2012* [53] and *Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. Wang, Zeldovich, Kaashoek, Solar-Lezama; SOSP 13* [54], §6.14
- *Beyond the PDP-11: Architectural support for a memory-safe C abstract machine; Chisnall et al.; ASPLOS 2015* [14], §6.15
- *What every C programmer should know about undefined behavior; Lattner; Blog post 2011*, §6.16
- *Proposal for a Friendly Dialect of C; Cuoq, Flatt, Regehr; Blog post 2014*, §6.17
- *UB Canaries; Regehr; Blog post 2015*, §6.18

For completeness we mention early work on sequential C semantics, by Gurevich and Higgens [20], Cook and Subramanian [16], Papaspyrou [46], Bofinger [13], Black and Windley [11, 12], and Anderson [4].

On the concurrency side, Batty et al. [8] formalised the concurrency aspects of the ISO C/C++11 standards during the standardisation process, with the resulting mathematical models and standard prose in close correspondence; this was later extended and related the IBM POWER hardware model [7, 47], and used for compiler testing by Morisset et al. [39].

Then there are very extensive literatures on static and dynamic analysis, symbolic execution, model-checking, and formal verification for C, and systems-oriented work on bug-finding tools, including tools such as Valgrind [42], the Clang sanitisers, and the Csmith tool of Yang et al. [57], which aims to generate programs that cover a large subset of C while avoiding undefined and unspecified behaviors. Yet another line of related work includes C-like languages that provide additional safety guarantees, such as Cyclone [23], and tools for hardening C execution, such as Softbound [40], and many more. We cannot begin to summarise all of these here, but each implicitly embodies some notion of C semantics.

Our work on Cerberus began with Justus Matthiesen's undergraduate and MPhil project dissertations [35, 36].

6.1 C formalised in HOL; Norrish; PhD thesis 1998

This model [43] (the basis also for the expression determinacy proof of [44]), adopts an almost fully concrete model, in which memory is a map from addresses to concrete 8-bit byte values (together with a map saying which addresses have been initialised). These bit-sequences are interpreted as values when read, including a check that *"the bytes read out of memory constitute a valid value for the given type"* [43, §3.3.2]. Pointer values are allowed to point one-past any allocated address, but there is no notion of provenance.

6.2 A unified memory model for pointers; Tuch, Klein; LPAR 2005

This paper [50] aims at a *"heap abstraction that allows for effective reasoning about both typed and untyped views of the heap and the effects of updates on the heap"*. It describes a model consisting concretely of a map from addresses to concrete bitvector values (`word32`) together with a map from addresses to optional source-language types. The Isabelle/HOL types corresponding to those have to be equipped with maps to and from their concrete representations and with sizes. For programs that respect this type information, a heap abstraction lets the concrete heap be viewed as a collection of heaps, one for each type; this supports formal reasoning that exploits type-based lack-of-aliasing properties.

6.3 Types, bytes, and separation logic; Tuch, Klein, Norrish; POPL 2007

This paper [48] presents a memory model for C intended to support formal verification of C systems code by mechanised interactive proof, following automated program-logic verification condition generation (VCG) for a translation of the C source program and its semantics into a prover (Isabelle/HOL). The paper includes example verifications of a simple list reversal and the L4 kernel memory allocator, and the model was used for the seL4 verification [26]. More details are in Tuch's 2008 PhD thesis [49]. The paper presents *"a formal model of memory that both captures the low-level features of Cs pointers and memory, and that forms the basis for an expressive implementation of separation logic."* However, the work targets C code written for verification in mind, rather than systems code found in the wild, and it targets that code as compiled for a specific architecture. That permits a number of simplifications w.r.t. general C ([26, §4.3],[48, §3]):

- syntactically, expressions are restricted to be largely side-effect-free; this and other restrictions make the evaluation order deterministic;
- the C implementation-defined behaviour choices can be fixed based on the intended compiler and machine architecture; and
- some unspecified behaviours are handled by automatically inserting guards when translating into the prover, covering *"division by zero, dereferencing the null pointer, and dereferencing an improperly aligned pointer"*. Any verification has to show that these hold whenever they are encountered.

The basic memory model is completely concrete, similar to that of Tuch and Klein [50]: a heap memory state is a total function from addresses ($word32$) to bytes ($word8$). Each language type has an associated Isabelle/HOL type in a type class recording its representation functions to and from byte sequences, a type-name tag, and size and alignment information.

There is no allocation ID or other provenance information, and whether the model is sound w.r.t. the behaviour of the specific compiler (GCC) used for seL4 for our test cases involving provenance (if indeed those are supported by their translation into the prover and VCG) is unclear from the paper.

The model does not support structs whose members have their address taken or which involve padding, or local veriables whose address is taken [48, §4.1].

The model also contains a history variable mapping addresses to optional source-language types, with proof annotations updating this added by the verifier. Above this concrete model the paper builds an abstraction of multiple typed heaps and a separation logic.

The first example is a C program with well-defined but nondeterministic behaviour w.r.t. the ISO standard that is excluded by their syntactic restrictions:

EXAMPLE (`tkn-1.c`):

```
#include <stdio.h>
int i = 0, a[2] = {0,0};
int f(void) {
  i++;
  return i; }
/* will print either 0 or 1 */
int main(void) {
  a[i] = f();
  printf("%i\n",a[0]); }
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
1
```

CLANG36-O2-NO-STRICT-ALIASING:

```
0
```

(adapted to print the result rather than return it).

The second and third examples illustrate what can be verified in this system; they also illustrate the specific *"low-level features of C's pointers and memory"* that this model supports. The second is an in-place linked list reverse, for lists which contain no data beyond the link pointer:

EXAMPLE (`tkn-2.c`):

```
#include <stdio.h>
typedef unsigned long  word_t;

word_t reverse(word_t *i) {
  word_t j = 0;
  while (i) {
    word_t *k = (word_t*)*i;
    *i = j;
    j = (word_t)i;
    i = k;
  }
  return j;
}

int main() {
  word_t a[3];
  a[0] = (word_t) &a[1];
  a[1] = (word_t) &a[2];
  a[2] = (word_t) 0;
  word_t b;
  printf("a[0]=%lu a[1]=%lu a[2]=%lu\n",
         a[0],a[1],a[2]);
  b = reverse(a);
  printf("a[0]=%lu a[1]=%lu a[2]=%lu b=%lu\n",
         a[0],a[1],a[2],b);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
a[0]=140737488349720 a[1]=140737488349728 a[2]=0
a[0]=0 a[1]=140737488349712 a[2]=140737488349720
b=140737488349728
```

CLANG36-O2-NO-STRICT-ALIASING:

```
a[0]=140737488349704 a[1]=140737488349712 a[2]=0
a[0]=0 a[1]=140737488349696 a[2]=140737488349704
b=140737488349712
```

adapted with a `typedef` to capture the prose definition of `word_t`, though to `unsigned long` rather than their `unsigned int`, to match the types of the 64-bit machine used to run the example) and with the `main()` usage added.

The third is an allocation function:

EXAMPLE (`tkn-3.c`):

```
#include <stdio.h>
#include <stdlib.h>

typedef unsigned long  word_t;

word_t* kfree_list;

void * alloc(word_t size) {
  word_t *prev, *curr, *tmp;
  word_t i;
  size = size >= 1024 ? size : 1024;
  for (prev = (word_t*) &kfree_list, curr = kfree_list;
       curr;
       prev = curr, curr = (word_t*) *curr) {
    if (!((word_t) curr & (size - 1))) {
      tmp = (word_t*) *curr;
```

```
      for (i = 1; tmp && (i < size / 1024); i++) {
        if ((word_t) tmp != ((word_t) curr + 1024*i)) {
          tmp = 0;
          break;
        };
        tmp = (word_t*) *tmp;
      }
      if (tmp) {
        *prev = (word_t) tmp;
        for (i = 0; i < (size / sizeof(word_t)); i++) {
          curr[i] = 0;
        }
        return curr;
      }
    }
  }
  return 0;
}

void print_free_list(word_t* p) {
  word_t* q = p;
  printf("free list: ");
  while (q != NULL) {
    printf("%p ",(void*)q);
    q = (word_t*) *q;
  }
  printf("%p\n",(void*)q);
}

int main() {
  int n=10; // number of blocks
  void *r = malloc(1024*(n+1));
  // crudely force r to be 1024-byte-aligned
  if (((word_t)r & (1024-1)) != 0)
    r = (void*)((((word_t)r) & ~((word_t)(1024-1)))
                  + (word_t)1024);
  // initialise the internal next-block pointers
  int i;
  for (i=0; i < n-1; i++)
    *((word_t *)((word_t)r+i*1024))
      = (word_t)r+(i+1)*1024;
  *(word_t *)((word_t)r+(n-1)*1024) = 0;
  kfree_list = (word_t *)r;
  // try some allocations
  print_free_list(kfree_list);
  void *a, *b, *c;
  a = alloc(1024); // should succeed
  b = alloc(2048); // should succeed
  c = alloc(65536);// should fail
  printf("a=%p b=%p c=%p\n",a,b,c);
  print_free_list(kfree_list);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
free list: 0x801417000 0x801417400 0x801417800
0x801417c00 0x801418000 0x801418400 0x801418800
0x801418c00 0x801419000 0x801419400 0x0
a=0x801417000
b=0x801417800 c=0x0
free list: 0x801417400 0x801418000
0x801418400 0x801418800 0x801418c00 0x801419000
0x801419400 0x0
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

This is adapted similarly with a specific choice of `word_t` and with a usage example, and also to fix an error: the paper has `word_t* prev, curr, tmp;` which should be `word_t *prev, *curr, *tmp;` (in their supplementary pdf proof document the three declarations are separated out, so this seems to be a typo introduced when they typeset the code).

6.4 Formal verification of a C-like memory model and its uses for verifying program transformations; Leroy and Blazy; JAR 2008

The early CompCert memory model, as described by Leroy and Blazy [34], is rather abstract from our de facto standards point of view.

They present both an axiomatisation and a "concrete model" [34, §4] that satisfies it. The main focus is the establishment of the memory injection machinery used in the CompCert compiler correctness proof to relate memory contents across compilation phases. Their concrete model has a memory state consisting of: a block ID counter; blocks with unique non-reused IDs; a boolean for each block ID saying whether it has been deallocated; the bounds (in $\mathbb{Z}$) for each block, supplied as arguments to the allocation operation; and a optional abstract typed value (`option (memtype val)`) for each block ID. The memory types (`int8signed`, `int8unsigned`, `int16signed`, `int16unsigned`, `int32`, `float32`, `float64`) have sizes and alignment restrictions (in numbers of bytes). The values are *"defined as the discriminated union of 32-bit integers* `int(n)`*, 64-bit double-precision floating-point numbers* `float(f)`*, memory locations* `ptr(b, i)` *where b is a memory block reference and i a byte offset within this block, and the constant* `undef` *representing an undefined value such as the value of an uninitialized variable"*.

In this semantics the IDs are used to give a strong provenance semantics, e.g. with == pointer comparison comparing the IDs, but more concrete manipulations of pointers and memory are not supported. In particular:

- pointer values do not contain anything corresponding to the numeric address of a pointer value in a conventional C implementation. They therefore cannot be meaningfully cast to integer types.
- there is no support for manipulation of the representation bytes of values. For the integer and floating-point types that would need a relatively straightforward adaptation of their `store` function, at least given a fixed implementation-defined representation. But for pointer values, because there is no address information, it would require more radical change.
- there is (correspondingly) no modelling of the layout and padding of C struct and union types.

It is important to note that the CompCert C semantics is intended to be the semantics of a particular implementation (that of the CompCert compiler), rather than a semantics that captures the envelope of all behaviour permitted by any particular version of the ISO or de facto standards; in that sense their goals are quite different from ours.

6.5 CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency; Ševčík, Vafeiadis, Zappa Nardelli, Jagannathan, Sewell; POPL 2011, JACM 2013

CompCertTSO [51, 52] is a verified compiler for a C-like language with x86 TSO concurrency. The development started with that of CompCert 1.5, and the sequential aspects of the behaviour of pointers and memory are broadly as above, but there are some interesting differences.

In the relaxed-memory TSO setting, the lifetime of an allocation becomes a more involved concept, as an allocation or free event may be in the local write buffer of the thread performing it before becoming visible to other threads. To prevent this complicating the compiler correctness proof, CompCertTSO relaxed the ISO-like restriction of pointer == comparison to pointers to live blocks, allowing comparison of arbitrary pointer values. In turn, to be sound w.r.t. the behaviour of a reasonable implementation, which will often reuse memory for allocations that are separated in time, this means the semantics had to permit both true and false for the example below [52, §3.4], which we use in §2.16.2.

EXAMPLE (`compcertTSO-1.c`):

```
#include <stdio.h>
int* f() {
  int a;
  return &a; }
int* g() {
  int a;
  return &a; }
int main() {
  _Bool b = (f() == g()); // can this be true?
  printf("(f()==g())=%s\n",b?"true":"false");
}
```

The CompCertTSO back-end semantics and correctness proof also supported finite memory [52, §3.4], in which *"allocation can fail and in which pointer values in the running machine-code implementation can be numerically equal to their values in the semantics"*, with the back-end allocations all at concrete addresses in a single block (ID 0), but the concrete values and representations of pointers were not exposed in the source language.

6.6 The CompCert Memory Model, Version 2; Leroy, Appel, Blazy, Stewart; INRIA RR-7987 2012

This paper [33] describes an updated memory model for CompCert, introduced in CompCert 1.7 and refined in CompCert 1.11. The principal changes are support for byte-level manipulations of integers and floats (while keeping pointer representations abstract) and the introduction of per-byte permissions on memory.

This paper writes (§3.1) *"The CompCert memory model version 1 correctly models the memory behaviour of C programs that conform to the ISO C99 standard."*, but this is not entirely correct according to our reading of the ISO standards. For example, the C99 and C11 text on effective types [1, 3, 6.5p6] licenses copying values as arrays of character type, e.g. as in our §2.4.2 with example `pointer_copy_user_dataflow_direct_bytewise.c`, but that earlier CompCert memory model does not. Indeed, given that ISO C99 is not defined in a mathematically rigorous way, and the absence of any proof or test-based evaluation, the exact force of the claim is unclear.

Their §3.1 also gives two idioms which the CompCert memory model version 1 permits which they say ISO C99 does not. The first is roundtrip casts of one pointer type to another and back, e.g.

EXAMPLE (`compcertMMv2-1.c`):

```
#include <stdio.h>
int main() {
  int x=3;
  *((int *) (float *)&x) = 4;
  printf("x=%i\n",x);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
x=4
```

CLANG36-O2-NO-STRICT-ALIASING: ... as above

As we discuss in §2.14, this is permitted in C11, and also in C99 [3, 6.3.2.3p7], [1, 6.3.2.3p7], for pointers to object types (as opposed to pointers to function types) if the intermediate value is correctly aligned. In practice it seems reasonably common for implementations to use the same representation for all pointer types; there it could be allowed in general.

The second is a consequence of the use of concrete byte-count offsets for access within a block and of the particular layout algorithm used, which makes the following two examples well-defined.

EXAMPLE (`compcertMMv2-2.c`):

```
#include <stdio.h>
struct { int x, y, z; } s;
int main() {
  s.y = 41;
  ((int *) &s)[1] = 42;
  printf("s.y=%i ",s.y);
  *((int *) ((char *) &s + sizeof(int))) = 43;
  printf("s.y=%i\n",s.y);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
s.y=42 s.y=43
```

CLANG36-O2-NO-STRICT-ALIASING: ... as above

EXAMPLE (`compcertMMv2-3.c`):

```
#include <stdio.h>
union point3d {
  struct { int x, y, z; } s;
  int d[3];
};
int main() {
  union point3d p;
  p.s.y = 42;
  int w;
  w = p.d[1];
  printf("w=%i\n",w);
```

```
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
w=42
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

The first relies (in the last assignment) on the absence of padding between the `ints` in the struct, which may often be true but is certainly not guaranteed by ISO; both examples rely on struct and array layout corresponding, and the second also relies on union type punning, which we discuss in §2.15.4.

Their §3.2 discusses several limitations of the CompCert memory model version 1. The first three involve bytewise access to the representations of integers and floats:

EXAMPLE (`compcertMMv2-4.c`):

```
#include <stdio.h>
unsigned int bswap(unsigned int x) {
  union { unsigned int i; char c[4];} src, dst;
  int n;
  src.i=x;
  dst.c[3]=src.c[0]; dst.c[2]=src.c[1];
  dst.c[1]=src.c[2]; dst.c[0]=src.c[3];
  return dst.i;
}
int main() {
  unsigned int x=0x11223344;
  unsigned int y;
  y = bswap(x);
  printf("y=0x%x\n",y);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
y=0x44332211
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

EXAMPLE (`compcertMMv2-5.c`):

```
#include <stdio.h>
float fabs_single(float x) {
  union { float f; unsigned int i; } u;
  u.f = x;
  u.i = u.i & 0x7FFFFFFF;
  return u.f;
}
int main() {
  float f=-1.0;
  float g;
  g = fabs_single(f);
  printf("g=%f\n",g);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
g=1.000000
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

(we omit the third, which is broadly similar). The model proposed in this Leroy et al. 2012 paper permits these two, essentially by building particular representation choices into the load and store functions and by shifting to a memory state that stores bytes that can be `Undef`, a concrete 8-bit byte value, or the nth byte of an abstract pointer.

The last is a bytewise user `memcpy`

EXAMPLE (`compcertMMv2-6.c`):

```
#include <stdio.h>
void* memcpy(void *dest,const void *src,size_t n) {
  unsigned long i;
  for (i=0; i<n; i++)
    ((char *)dest)[i] = ((const char *) src)[i];
  return dest;
}
int main () {
  int x[2], y[2];
  x[0] = 0; x[1] = 1;
  memcpy(y, x, sizeof(x));
  printf("y[0]=%i y[1]=%i\n",y[0],y[1]);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
y[0]=0 y[1]=1
```

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

similar to the `pointer_copy_user_dataflow_direct_bytewise.c` example we discuss in our §2.4.2; it is not supported by the CompCert memory model version 2.

The paper also adds a fine-grained access control permission mechanism, aimed both at the separation-logic verification of CompCert C programs in Appel's Verified Software Toolchain project and at supporting compiler optimisations for `const` globals.

6.7 Formal C semantics: CompCert and the C standard; Krebbers, Leroy, and Wiedijk; ITP 2014

This paper [32] extends CompCert 1.12 to bring it closer to something that could be soundly described by Krebbers's Formalin C semantics. It adds support (in CompCert 1.13) for:

- comparison with end-of-array pointers, and
- byte-wise pointer copy.

The motivating example is a user-code `memcpy` implementation, essentially the same as our `pointer_copy_user_dataflow_direct_bytewise.c` in §2.4.2:

EXAMPLE (`klw-itp14-1.c`):

```
void my_memcpy(void *dest, void *src, int n) {
  unsigned char *p = dest, *q = src, *end = p + n;
  while (p < end) // end may be end-of-array
    *p++ = *q++;
}
int main() {
  struct S { short x; short *r; } s = { 10, &s.x }, s2;
  my_memcpy(&s2, &s, sizeof(struct S));
  return *(s2.r);
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

CLANG36-O2-NO-STRICT-ALIASING: . . . as above

ISO: `defined behaviour`

Krebbers et al. note *"In CompCert 1.12, this program has undefined behavior, for two reasons: the comparison* `p`

`< end` *that involves an end-of-array pointer, and the bytewise reads of the pointer* `s.r`*”*. They go on to relax those restrictions slightly. For comparison (at least for equality comparison – whether they also mean to include relational comparison is unclear), they write:

- *Comparison of pointers in the same block is defined only if both are weakly valid. A pointer is weakly valid if it is valid or end-of-array*
- *Comparison of pointers with different block identifiers is defined for valid pointers only.*

They argue that this is *“more sensible than the naive reading of the C standard because it is stable under compilation”*.

We agree that this stability property would be desirable, but the downside, that comparison becomes more partial, is potentially significant. It is already somewhat partial in the ISO standard, but arguably not in important de facto standards. Whether code in practice does comparisons of pointers with different provenances that are not strictly within their original allocations is unknown (we guess it is uncommon but does occur). This suggests another question for §2.10, added in §2.10.2 with the following example.

EXAMPLE (`klw-itp14-2.c`):

```
#include <stdio.h>
int  x=1, y=2;
int main() {
  int *p = &x + 1;
  int *q = &y;
  _Bool b = (p == q); // free of undefined behaviour?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

Turning to bytewise reads and writes of pointer values, the CompCert 1.12 they describe [32, §3] stores *“integer and floating point values by sequences of numeric bytes, but pointer values and uninitialized memory by symbolic bytes”*:

```
Inductive memval: Type :=
| Undef: memval
| Byte: byte -> memval
| Pointer: block -> int -> nat -> memval
```

where `Pointer b i n` is the *n*’th byte of pointer with block ID *n* and offset *i*, but pointer values could only be read or written as complete sequences.

They extend CompCert values with a corresponding symbolic pointer byte constructor, `Vptrfrag: block -> int -> nat -> val` (they also needed an additional `memval` constructor, `PointerPad`, to represent the upper bytes of an in-memory representation of a `Vptrfrag`, determined by sign-extension in the implementation).

These two extensions are enough to support user-defined bytewise `memcpy`, but arithmetic on those byte values is given undefined behaviour, so it will not support examples such as our `pointer_copy_user_dataflow_indirect_bytewise.c`, §2.4.3. They remark *“Reading a pointer byte from memory, adding 0 to it, and writing it back remains undefined behavior. It would be tempting give an ad-hoc semantics to such corner cases, but that will result in a loss of algebraic properties like associativity”*.

6.8 A Precise and Abstract Memory Model for C using Symbolic Values, Besson, Blazy, and Wilke; APLAS 2014

This paper [9] aims at a semantics in which reading uninitialised variables and “low-level pointer operations” (by which they mean manipulations of unused pointer bits) have well-defined behaviour, not the undefined behaviour of the ISO standard, without *“resorting to a concrete representation of pointers as machine integers”*.

They give two motivating examples. The first [9, Fig. 1 and §6.3] reads an uninitialised variable, OR’s it with `1` and writes it (we adapt their example to split the calculation of `status` and add the calculation of `b` and `printf`s). They state that this occurs in practice, in an implementation of `memalign`, but do not explain why author of this code wants to preserve some bits of an uninitialised variable.

EXAMPLE (`besson_blazy_wilkie_Fig_1_adapted.c`):

```
#include <stdio.h>
int set(int p, int flag) {
  return p | (1 << flag); }
int isset(int p, int flag) {
  return (p & (1 << flag)) != 0; }
int main() {
  int status;
  printf("status=0x%x\n",status);
  status = set(status,0);
  _Bool b = isset(status,0);
  printf("status=0x%x  b=%s\n",status,b?"true":"false");
  return isset(status,0); }
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
besson_blazy_wilkie_Fig_1_adapted.c: In function ’main’:
besson_blazy_wilkie_Fig_1_adapted.c:8:3: warning:
’status’ is used uninitialized in this function
[-Wuninitialized]
 printf("status=0x%x\n",status);


^
status=0x0
status=0x1 b=true
```

CLANG36-O2-NO-STRICT-ALIASING:

```
besson_blazy_wilkie_Fig_1_adapted.c:8:26: warning:
variable ’status’ is uninitialized when used here
[-Wuninitialized]
 printf("status=0x%x\n",status);


 ^
besson_blazy_wilkie_Fig_1_ada
pted.c:7:13: note: initialize the variable ’status’ to
silence this warning
 int status;
 ^
```

```
 = 0
1 warning generated.
status=0xffffea50
status=0xffffffff b=true
```

DEFACTO: `defined behaviour (printing nondeterministic values)`

ISO: `unclear`

With respect to the ISO standard, Besson et al. write (their §2.1, "Access to Uninitialised Variables"):

> The C standard states that any read access to uninitialised memory triggers undefined behaviours [10, section 6.7.8, §10]: If an object that has automatic storage duration is not initialised explicitly, its value is indeterminate." Here, "indeterminate" means that the behaviour is undefined.

Here their [10] refers to the C99 standard [1], but what they say does not seem to be exactly supported by that text. Appendix J.2 of C99 says that behaviour is undefined if *"The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.8, 6.8)."* but (a) this refers only to objects with automatic storage duration, and (b) Appendix J is "informative" not "normative", and it is not clear how (even for those) the listed subsections imply undefined behaviour. In any case, in C11 the standard text for this has been changed, and is as we describe in §3. In our reading of ISO C11 the example has undefined behaviour by 6.3.2.1p2, because the address of `status` is not taken.

With respect to the de facto standards, this is essentially the question we discuss in §3.2.4, with examples such as `unspecified_value_strictness_int.c`, of whether various operations are strict w.r.t. unspecified values. Our choice for our candidate de facto semantics model is to make all operations strict, which will not permit this idiom but will permit compiler optimisations that propagate `undef` through operations.

Besson et al. give a symbolic semantics that permits this example. Their model has symbolic values (a grammar of unary, binary, conditional, and cast operations), some extra alignment knowledge, and symbolic byte-n-of symbolic values. These are normalised to a concrete value when reading/writing or making a control-flow choice. It has been exercised on the Doug Lea allocator, NaCl crypto, and CompCert benchmarks.

Note that "symbolic" is used in two senses: these are symbolic identifiers that are eventually resolved to concrete values, not to be confused with the symbolic `undef` which is a distinguished single constructor of a value type, (roughly) as used in the LLVM implementation, in the CompCert memory models, and in our de facto standard model.

Note also that a semantics that nondeterministically picks a value at each read of an undefined value would also permit the above motivating example.

Their second motivating example uses the low-order bits of a pointer (to store a hash of the pointer as a hardening technique, apparently based on a technique in Doug Lea's allocator):

EXAMPLE (`besson_blazy_wilkie_Fig_2.c`):

```
#include <inttypes.h>
#include <stdlib.h>
char hash(void *ptr);
char hash(void *ptr) {
  char h=0;
  unsigned int i;
  for (i=0;i<sizeof(ptr);i++)
    h = h ^ *((char *)ptr+i);
  return h; }
int main(){
  int *p = (int *) malloc(sizeof(int));
  *p = 0;
  int *q = (int *) ((uintptr_t) p | (hash(p) & 0xF));
  int *r = (int *) (((uintptr_t) q >> 4) << 4);
  return *r; }
```

(They assume that pointers are 4-byte values and `malloc` returns a 16-byte aligned value.) This is essentially just like our earlier pointer-bitmask examples, e.g. `provenance_tag_bits_via_uintptr_t_1.c`, in §2.2.4.

Their §6.2 notes that system calls such as `mmap` return `-1` on error, and so one must be able to compare pointers against `-1`. We add a question and test for this in §2.10.3.

EXAMPLE (`besson_blazy_wilke_6.2.c`):

```
#include <stdlib.h>
int main() {
  void *p = malloc(sizeof(int));
  _Bool b = (p == (void*)-1); // defined behaviour?
}
```

In §6.4 they give another example that contains a potentially inter-allocation pointer relational comparison, from a `memmove` implementation found in practice:

```
void* memmove(void *s1, const void *s2, size_t n) {
  char * dest = (char *) s1;
  const char * src = (const char *) s2;
  if ( dest <= src )
    while ( n-- ) { *dest++ = *src++; }
  else {
    src += n; dest += n;
    while ( n-- ) { *--dest = *--src; }
  }
  return s1;
}
```

This seems to be an issue for their semantics because there is no way to resolve the conditional control-flow choice. They write *"In other words, a program whose control-flow depends on the memory layout has an undefined behaviour. This dependance on the memory layout (e.g. on the memory allocator) is a portability bug that is detected by our semantics."*. As we discuss in §2.11.1 with `pointer_comparison_rel_1_global.c`, such comparisons are undefined behaviour w.r.t. ISO but should be allowed in many de facto semantics; we believe that for real OS code the above has to be permitted and that it is not really a portability bug.

6.9 A Concrete Memory Model for CompCert; Besson, Blazy, Wilke; ITP 2015

This paper [10] shows that the model of [9] described in the preceding subsection (§6.8) is an abstraction of the CompCert model, and that the CompCert front-end correctness proof (from CompCert C to Cminor) can be adapted to the new model.

As additional motivation, they mention the CompCert treatment of bitfields, which are translated away (to bitwise operations) by a non-verified elaboration pass before the formally verified front-end. Together with the strictness of arithmetic and logical operations w.r.t. the CompCert symbolic `undef`, this means that for structs containing multiple bitfields that the translation represents in the same back-end word, one cannot set one bitfield at a time.

Their example Fig. 1(a) (adapted below to print the result rather than return it) refines an earlier question about whether unspecified-value-ness is a per-leaf-value property, a per-byte property, or a per-bit property; we include this in §3.2.8.

EXAMPLE (`besson_blazy_wilke_bitfields_1u.c`):

```
#include <stdio.h>
struct f {
  unsigned int a0 : 1; unsigned int a1 : 1;
} bf ;
int main() {
  unsigned int a;
  bf.a1 = 1;
  a = bf.a1;
  printf("a=%u\n",a);
}
```

The example above has been adapted in another way from the original version [10, Fig. 1(a)]: the latter had bitfields `a0` and `a1` declared as simple `int`s. In C11 (6.7.2p5, and similarly in C99), for bitfields it is implementation-defined whether `int` designates `signed int` or `unsigned int`. For the `int` version (`besson_blazy_wilke_bitfields_1.c`), GCC-4.8 -O2 warns of `overflow in implicit constant conversion`, as one might expect when storing the value `1` in a signed bitfield of size `1`, and that conversion results in a print of `-1`. We avoid this complexity, which is not relevant for this example, by using `unsigned int` bitfields.

Their discussion of unspecified values implicitly assumes that they should be stable, as they write in their §3.1: *"For instance, consider two uninitialised char variables* `x` *and* `y`*. Expressions* `x-x` *and* `x-y` *both construct the symbolic expression* `undef-undef`*, which does not normalise. However we would like* `x-x` *to normalise to* `0`*, since whatever the value stored in memory for* `x`*, say* `v`*, the result of* `v-v` *should always be* `0`*."*. This is at odds with our understanding of the de facto standards and experimental observations in §3.2.3, where we see unstable uninitialised values in Clang. It may or may not be sound w.r.t. the CompCert optimisations, but other compilers may optimise usages of an `undef` value to uses of values that happen to be left in registers.

Inter-block pointer relational comparison is not supported, which is also at odds with our de facto standards understanding, as we discuss in the previous subsection. They write: *"The normalisation of e w.r.t. a memory m returns a value v if and only if the side-effect free expression e evaluates to v for every concrete mapping $\mathtt{cm} : block \to \mathbb{B}_{32}$ of blocks to concrete 32 bits addresses which are compatible with the block-based memory m"*.

They identified a glitch w.r.t. pointer wraparound in CompCert: in the semantics used for all phases of the verification, successive incrementing of a pointer to an allocated region will never produce something that compares equal to `NULL`, while in the final implementation (compiled in a non-verified way from the CompCert assembly with semantic values) to machine code, it will. They write that this was fixed in the CompCert trunk by making the comparison of a pointer with `NULL` defined behaviour only if the pointer is either within or one-past its allocation. This is tighter than the ISO semantics and our understanding of the de facto semantics, both of which allow such comparisons freely.

Their model supports finite memory and an allocation operation that can fail. CompCertTSO [51, 52] also had those properties (though it is not discussed in [10]). Referring to a fully concrete memory model in which allocations return non-deterministic currently-fresh pointers, they write (§2.3) *"However, this model lacks an essential property of CompCert's semantics: determinism. For instance, with a fully concrete memory model, allocating a memory chunk returns a non-deterministic pointer – one of the many that does not overlap with an already allocated chunk. In CompCert, the allocation returns a block that is computed in a deterministic fashion. Determinism is instrumental for the simulation proofs of the compiler passes and its absence is a show stopper."* The CompCertTSO development shows that allocation nondeterminism can be accommodated in such a proof: its memory model was not fully concrete, but it did have nondeterministic allocation. The proof separated out the threadwise semantics of each thread from its interactions with memory, thus keeping the former deterministic and allowing relatively straightforward adaptations of many CompCert compiler-phase proofs [52, §4.2–4.4].

6.10 A formal C memory model supporting integer-pointer casts; Kang, Hur, Mansky, Garbuzov, Zdancewic, Vafeiadis; PLDI 2015

This paper [25] is also focussed on C compiler verification: it aims to support casts between pointers and integers and arithmetic over them, in the way a fully concrete model does, while simultaneously making a range of compiler optimisations sound and verifiable, in the way that the abstract block-ID/offset models do.

Their motivating example is:

```
int f(void) {         int f(void) {         int f(void) {
  int a = 0;            int a = 0;
  g();          →       g();          →       g();
  return a;             return 0;             return 0;
}                     }                     }
```

where `g` is an unknown external function, for which they would like the compiler to be able to deduce that `a` is not known to `g()` in the code on the left, and hence that constant propagation can soundly convert it to the middle code. This is not sound in a fully concrete model, as `g()` might happen to write to whatever address the semantics chooses for `a`. They would also like to permit the removal of the now-unused allocation of `a` to give the code on the right. This optimisation is also not sound in general in a fully concrete model: in a finite-memory semantics `g()` might attempt to allocate enough memory to exactly exhaust the available memory of the right-hand code, giving a non-error behaviour that the middle code cannot match.

Their approach (their *quasi-concrete model*) adapts the abstract block-ID/offset model: blocks are created as abstract, and only when/if a pointer to a block is first cast to an integer is a concrete address chosen and associated to the block. A memory state is a map from block IDs to blocks. A block (v, p, n, c) has a boolean flag v indicating whether it is valid or had been freed, a natural-number size n, an n-tuple of values c, and a p that is either a concrete $\mathit{int32}$ address or an `undef` indicating that the block is still abstract (this should not be confused with the C unspecified values or LLVM `undef`). Values are a disjoint union of concrete $\mathit{int32}$ values and block-ID/offset pairs.

This justifies the above optimisations: because the address of `a` is not cast to an integer type before the call to `g()`, the block is still abstract. Hence, pointers to it cannot be forged within `g()`, justifying the first optimisation, and it has not yet consumed any of the finite-memory address space, justifying the second.

Our candidate de facto model will treat this rather differently. As discussed in §2.2.3, while we permit meaningful casts between pointers and integers, we associate provenance information with integer values, and one cannot normally forge a valid pointer from an arbitrary integer. That should make the first optimisation sound, but we need to consider two "abnormal" cases.

First, there is access to device memory via concrete addresses (see §2.7). This is simple: the implementation-defined range of device-memory addresses we propose, guaranteed to be disjoint from normal C-accessible memory, means the compiler can still soundly assume a lack of aliasing, with other accesses via concrete unprovenanced addresses giving rise to undefined behaviour.

Second, there are accesses via pointers read in from IO (see §2.6). For IO is done in a controlled fashion, e.g. with the `scanf %p`, we previously proposed tagging such pointer values with a wildcard provenance, indicating that they might alias with any other pointer. That would disallow the first optimisation above (`g()` could read in a concrete address that happened to be equal to that of `a` and then use it to mutate `a`). But one could refine the proposal in the spirit of the Kang et al. paper, but for IO-escape rather than cast-to-integer escape: dynamically marking block IDs (aka provenances) which might have escaped to the outside world, and letting the wildcard-provenance pointers produced by input alias only with those. This tagging would not have to be at pointer-to-integer cast time; it could be as late as the actual IO. For IO done in an uncontrolled bytewise fashion, one could do something similar: for output, dynamically marking block IDs for which any value tagged with that provenance might have escaped, and for use of pointer values obtained from bytewise input (which is essentially the same as casts from arbitrary unprovenanced integers) treating them as having that wildcard provenance. What mainstream compilers currently do in these cases is an interesting question.

Our candidate de facto model, as currently envisaged, will not licence the second optimisation in general: it is a finite-memory model which will nondeterministically allocate memory from the finite address space at each allocation site and free it at each block kill. Kang et al. are essentially arranging for the memory they wish to optimise away to be in a separate and unbounded region (following CompCertTSO in this, as they say). They argue that this will still permit common optimisation cases, but from a mainstream compiler point of view it seems more likely that compilers will do such optimisations whether or not they are sound in the strong sense implied by the example (they may remove allocations even if their addresses are taken and concretely manipulated), and that the real challenge is to understand some more subtle sense in which they are sound.

Their §3.2 has an interesting argument against models in which (in our terms) integer values derived from pointers carry provenance information. They write that this prevents the optimisation below:

```
a = (a - b) + (2 * b - b);
q = (ptr) a;                    →  q = (ptr) a;
*q = 123;                             *q = 123;
```

"*Suppose the variable* `b` *contains an integer with permission to access some valid block l, and* `a` *contains an integer without any permission that is equal to the concrete address of the block l. Then the source program successfully stores 123 into the block l because* `q` *has the relevant permission, whereas the target program fails because* `q` *does not have the permission.*"

To make a concrete test case, we need to construct such a numerically correct but unprovenanced `a` value programmatically. This is difficult, especially if one wishes to avoid the questions of provenance for IO mentioned above, so we simply use a constant value appropriate to one particular implementation and platform.

EXAMPLE (`khmgzv-1.c`):

```
#include <stdio.h>
```

```
#include <string.h>
#include <inttypes.h>
int x=0;
int main() {
  uintptr_t b = (uintptr_t) &x;
  uintptr_t a = 0x60102C;
  printf("Addresses: b=0x%" PRIXPTR " a=0x%" PRIXPTR
         "\n",b,a);
  if (memcmp(&b, &a, sizeof(b)) == 0) {
    a = (a - b) + (2 * b - b);
    int *q = (int *) a;
    *q = 123;   // does this have undefined behaviour?
    printf("*((int*)b=%d  *q=%d\n",*((int*)b),*q);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: b=0x600BE8 a=0x60102C
```

CLANG36-O2-NO-STRICT-ALIASING: ...as above (modulo addresses)

[TODO: RE-EXAMINE THIS (depending on the integer provenance semantics, we might forbid the original behaviour)] (taking the result provenance of binary arithmetic operations of a provenanced and unprovenanced argument to be that of the former) but will forbid the following:

EXAMPLE (`khmgzv-2.c`):

```
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
int x=0;
int main() {
  uintptr_t b = (uintptr_t) &x;
  uintptr_t a = 0x60102C;
  printf("Addresses: b=0x%" PRIXPTR " a=0x%" PRIXPTR
         "\n",b,a);
  if (memcmp(&b, &a, sizeof(b)) == 0) {
    int *q = (int *) a;
    *q = 123;   // does this have undefined behaviour?
    printf("*((int*)b=%d  *q=%d\n",*((int*)b),*q);
  }
  return 0;
}
```

GCC-5.3-O2-NO-STRICT-ALIASING:

```
Addresses: b=0x600BE8 a=0x60102C
```

CLANG36-O2-NO-STRICT-ALIASING: ...as above (modulo addresses)

In §3.5 Kang et al. point out that in an abstract block-ID/offset model, with integer values a disjoint union of honest integers and abstract pointer values, arithmetic optimisations on integers must be limited to exclude examples such as this:

```
                                 t = a + b;
d1 = a + (b - c1);   →           d1 = t - c1;
d2 = a + (b - c2);               d2 = t - c2;
```

where `a` and `b` happen to be abstract pointer values (or, in our candidate de facto model, integer values with nonempty provenance), as the result of the addition on the right will give rise to undefined behaviour (or, in our model, perhaps an unprovenanced value – leaving aside the multiple-provenance possibility of §2.3). This also holds for our candidate model, but (as above) it is more of an issue for compiler verification, using the same model across such optimisation phases, than for a source-language definition.

6.11 The C standard formalized in Coq; Krebbers; PhD thesis 2015

Krebbers, partly in collaboration with Wiedijk, has developed a semantics in Coq for a substantial fragment of C, in their CH_2O project [27–31]. We discuss the version presented in Krebbers' 2015 PhD thesis [29]. Starting with an abstract-syntax representation of C produced by the FrontC parser (based on the version used by CompCert version 2.2, in turn based on the CIL FrontC parser [41]), this work is based on a translation into *CH_2O Core C*, which is equipped with a type system, an operational semantics, an executable version of that, an axiomatic semantics for reasoning about programs, and machinery for refinements, with metatheory proved in Coq relating these.

Krebbers writes: *"The goal of the CH_2O project is to develop a formal version of the non-concurrent fragment of the C11 standard that is usable in proof assistants."* [29, p.5] and *"It makes the standard utterly precise."* [29, p.4], but the reality is more nuanced: for the aspects of C that it covers, CH_2O is more like a maximally strict interpretation of the ISO C11 standard, as discussed in [29, Ch.2]: *"CH_2O errs on the side of caution: it makes certain behaviors undefined that some people deem defined according to the standard"*. It aims thereby to be sound w.r.t. any compiler that conforms to the ISO standard, but at the cost of excluding some programs that others would deem legitimate; it is not attempting to reflect the de facto standards.

The memory model is basically an abstract one, in terms of abstract object identifiers rather than numerical addressesd. These identifiers correspond to the provenances suggested by DR #260, as we discuss in §2.1. However, if (as we imagine) it follows Krebbers, Leroy, and Wiedijk [32] (discussed in §6.7) in making pointer equality comparison defined only for "valid" pointers, not one-past pointers, some of our examples there will have undefined behaviour in this semantics.

Casting pointers to integer types and back (see our §2.2) is not supported: *"The CH_2O semantics uses an abstract memory model with symbolic pointer values and therefore fails to account for pointer to integer casts. Casting a pointer to an integer, and vice versa, has undefined behavior in the CH_2O semantics."* [29, 2.6.2 *Integer representations of pointers*].

It differs from earlier abstract memory models in associating a tree-structured object (corresponding to the C data type structure) rather than a vector of bytes with each object ID, and pointer values therefore include paths through those trees rather than offsets within such vectors.

They aim throughout at a semantics that takes effective types into account (and also using this tree structure for that), while our candidate de facto model aims at C compiled with `-fno-strict-aliasing`.

Pointer manipulation (relational comparison, subtraction, and addition) appears to be permitted only within the same leaf subobject (or one-past for the latter two). For example, [29, p.102]:

```
struct S { int a[3]; int b[3]; } s1, s2;
s1.a == s2.b;
// OK, neither of the two pointers is end-of-array
s1.a == s1.b+3; // OK, same object
s1.a == s2.b+3; // Undefined, different objects,
                //  s2.b+3 end-of-array
s1.a <= s1.b;
// OK, <= into the same object
s1.a <= s2.a;
// Undefined, <= with different objects
```

and [29, p.66]:

```
struct S { int a[3]; int b[3]; } s;
s.a - s.b;
// Undefined, different array objects
(s.a + 3) - s.b; // Undefined, different array objects
(s.a + 3) - s.a; // OK, same array objects
```

Pointer addition has undefined behaviour when it goes more than one-past the (presumably sub)object [29, p.103]. We add a question for the different-subobject-array case in §2.13.5.

Pointer casts give undefined behaviour if they *"break dynamic typing"*, e.g. [29, p.103]:

```
int x;
(short*)(void*)&x;
// Undefined, int* cast to short*
(int*)((unsigned char*)&x + 1); // Undefined, ill-aligned
```

This seems stricter than ISO; see our §2.14 discussion.

There is support for bytewise manipulation of the representation bytes of C values, with symbolic "bit i of pointer value p" values, presumably permitting pointer values to be copied bytewise but not supporting arithmetic on them.

The treatment of type punning and unions [29, §2.5.6 Type-punning] seems to aim at the GCC interpretation, c.f. our discussion in §2.15.4. They make the following example disallowed [29, p.28], following that GCC text, though a literal reading of the ISO text might suggest otherwise.

```
short g(int *p, short *q) {
short z = *q; *p = 10; return z;
}
union int_or_short { int x; short y; } u = { .y = 3 };
int *p = &u.x;
// p points to the x variant of u
short *q = &u.y; // q points to the y variant of u
return g(p, q);  // g is called with aliased pointers p,q
```

Their pointer values include a bit saying whether they can be used for type punning; see [29, p.66,80,81]:

```
union U { int x; short y; } u = { .x = 3 };
short *p = &u.y; // a frozen version of the pointer
                 //  &u.y is stored
printf("%d", *p); // type-punning via a frozen pointer
                  //  -> undefined
```

and

```
union U { int x; short y; } u = { .x = 3 };
printf("%d", u.y);
```

At the end of an object lifetime, they make all pointers to that object indeterminate [29, §2.5.7 Indeterminate memory and pointers, p.30]; see our discussion in §2.16.1. They also assert that *"using an indeterminate pointer in pointer arithmetic and pointer comparisons also yields undefined behavior"*, but the justification of that w.r.t. the ISO text is not clear to us. The [29, p.30] example:

```
int *p = malloc(sizeof(int)); assert (p != NULL);
free(p);
int *q = malloc(sizeof(int)); assert (q != NULL);
if (p == q) {//undefined, p indeterminate due to the free
  *q = 10;
  *p = 14;
  printf("%d\n", *q);//p and q alias, expected to print 14
}
```

seems intended to justify it, but that could be explained in other ways, e.g. by giving a nondeterministic result to such a comparison, coupled with the manifest undefined behaviour of the `*p=14` in a provenance-aware semantics. The C99 Rationale [2, p.49, l.22–33] does introduce the notion of an invalid pointer and says that any of use of it gives rise to undefined behaviour. It justifies this with a *"hypothetical segmented architecture"* in which arrays might be represented using mulitple segments, where pointer comparison involves some metadata that might no longer exist after an object has been deallocated. We would like to know whether such implementations actually exist.

[29, §2.6.1 Integer representations of indeterminate memory] relates to our §3.2.1 and following.

For indeterminate values, they say [29, p.104]: *"Branching on an indeterminate value has undefined behavior."* See our §3.2.2.

Their "implementation environment" specifies sizes and alignments (and hence struct layout in the normal ABI way, see [29, p.138]), with explicit modelling of padding bytes, but *"In our tree based memory model we enforce that padding bytes always have an indeterminate value"* [29, p.27].

[29, §2.5.8 End-of-array pointers] relates to our §2.1.3.

The [29, p.36] example:

```
int x = 30, y = 31;
int *p = &x + 1, *q = &y;
intptr_t i = (intptr_t)p, j = (intptr_t)q;
printf("%ld %ld %d\n", i, j, i == j);
```

(reported by them as a GCC bug, and fixed from 4.7.1 to 4.8) suggests another possible question we add in §2.2.5: Can equality testing on integers, for integers derived from pointer values, be affected by their provenance?

In [29, p.63] they suggest that reading from abstract memory may affect its effective type information, with the example below in which the member of a union is left unresolved until the read [29, p.77]. This is not clearly mandated by the ISO text, by our reading thereof. But as our modelling is aiming at the `-fno-strict-aliasing` case, the point is moot as far as comparison goes.

```
short g(int *p, short *q) {
short z = *q; *p = 10; return z;
}
int main() {
union int_or_short { int x; short y; } u;
// initialize u with zeros, the variant of u remains
//  unspecified
for (size_t i = 0; i < sizeof(u); i++)
  ((unsigned char*)&u)[i] = 0;
return g(&u.x, &u.y);
}
```

In [29, p.194] the discussion of Kang et al. [25] has this amusing example:

```
int x = 0, *p = 0;
for (uintptr_t i = 0; ; i++) {
  if (i == (uintptr_t)&x) {
    p = (int*)i;
    break;
  }
}
*p = 15;
printf("%d\n", x);
```

We ran CH$_2$O on our tests (test run from 2016-02-01, ch2o github checkout 64d98faf7631252524230c859a4fc3bb4767f6e2 from Tue Nov 17 14:10:57 2015). Most tests (all except those for around 11 questions) were not supported in this version, many due to missing features in the CH$_2$O `printf` and standard libraries.

6.12 An Executable Formal Semantics of C with Applications; Ellison and Roşu; POPL 2012

This paper [18] describes a semantics for a substantial fragment of C expressed in the K rewriting logic, explained in more detail in Ellison's 2012 PhD thesis [22] and extended by Hathhorn et al. [21]. The authors claim to give *"the first complete formal semantics of the C programming language"* [22, Abstract], but again the reality is more nuanced.

The memory model is described as a map [18, §4.3] from block IDs to blocks with a size (in bytes) and a sequence of bytes of that size. In the rewriting setting those bytes are not necessarily ground numbers, and pointer values are represented essentially as a pair of a block ID and a numeric offset with the block, encoded e.g. as $sym(B) + O$ where sym seems to be a fresh function symbol, B is a block ID, and O is an offset. Pointer values are themselves represented in memory with symbolic bytes, e.g. as a list $subObject(sym(B)+O), 0), \ldots, subObject(sym(B)+O), 3)$ [22, p.81] (the sym of the paper seems to correspond to the loc of the thesis). This is very broadly similar to the CompCert memory model of Krebbers et al. discussed in §6.7. It is considerably more abstract than either the ISO or de facto standards, e.g. in the fact that pointers are not associated to concrete addresses, and so cannot be meaningfully cast to integer types.

Uninitialised values can be represented in memory with another function symbol, $Unknown(N)$ [22, p.82] (where N is the bitwidth).

Hathhorn et al. [21] extend KCC with additional machinery for detecting undefined behaviour. The basic memory model is as above. They *"use a trap representation wherever the standard allows one to be used"* [21, §3.4], which (as they observe) leads to more undefinedness; it is significantly different from the de facto standards. They also add a record of the last-stored type of memory values, for effective-type checks (though see the experimental data below). Then there is additional provenance-related metadata attached to pointer values:

- *"the union variant a pointer or lvalue expression is based on so we can mark the section of memory not overlapping with the active variant as unspecified"*
- *"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"*
- *"when a pointer can be traced back to the value stored in some restrict-qualified pointer variable"*
- *"a pointers's alignment"*

KCC detected two potential alignment errors in earlier versions of our tests. But it gave 'Execution failed', with no further details, for the tests of 20 of our questions; 'Translation failed' for one; segfaulted at runtime for one; and gave results contrary to our reading of the ISO standard for at least 6: it exhibited a very strict semantics for reading uninitialised values (but not for padding bytes), and permitted some tests that ISO effective types forbid.

6.13 A precise yet efficient memory model for C; SSV 2009; Cohen, Moskal, Tobies, Schulte

Cohen et al. [15] describe a model implicit in their *"Verifying C Compiler"*. This translates annotated C code into BoogiePL; the verification condition generator Boogie takes BoogiePL as input, and feeds the generated verification conditions into the Z3 SMT solver. The main focus of the paper is on capturing type-based aliasing properties, though they do not refer to the C99/C11 effective types; they relate a fully concrete model to one in which memory is a *"collection of typed objects"*. There is no discussion of provenance, of reading uninitialised memory, or of undefined behaviour in general.

		Construct	Sufficient condition	Undefined behavior
Language	(1)	$p + x$	$p_\infty + x_\infty \in [0, 2^n - 1]$	pointer overflow
	(2)	p	$p = \texttt{NULL}$	null pointer dereference
	(3)	$x\, op_s y$	$x_\infty\, op_s y_\infty \in [2^{n-1}, 2^{n-1} - 1]$	signed integer overflow
	(4)	$x/y,\ x\%y$	$y = 0$	division by zero
	(5)	x<<$y,\ x$>>y	$y < 0 \vee y \geq n$	oversized shift
	(6)	$a[x]$	$x < 0 \vee x \geq \text{ARRAY\_SIZE}(a)$	buffer overflow
Library	(7)	$abs(x)$	$x = -2^{n-1}$	absolute value overflow
	(8)	`memcpy`(dst, src, len)	$\lvert dst - src \rvert < len$	overlapping memory copy
	(9)	use q after `free`(p)	$alias(p, q)$	use after free
	(10)	use q after $p :=$ `realloc`$(p, ...)$	$alias(p, q) \wedge p \neq \texttt{NULL}$	use after realloc

A list of sufficient (though not necessary) conditions for undefined behavior in certain C constructs [3, §J.2]. Here p, p , q are n-bit pointers; x, y are n-bit integers; a is an array, the capacity of which is denoted as $\text{ARRAY\_SIZE}(\text{a})$; op_s refers to binary operators `+`, `-`, `*`, `/`, `%` over signed integers; x_∞ means to consider x as infinitely ranged; `NULL` is the null pointer; $alias(p, q)$ predicates whether p and q point to the same object.

Figure 2. Reproduced from Wang et al. [54, Fig. 3]

6.14 Undefined Behavior: What Happened to My Code?; Wang, Chen, Cheung, Jia, Zeldovich, Kaashoek; APSys 2012, and Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. Wang, Zeldovich, Kaashoek, Solar-Lezama; SOSP 13

The first of these two papers [53] *"investigates whether bugs due to programmers using constructs with undefined behavior happen in practice"*. Similarly to our position that the de facto standards differ significantly from the ISO standard, they write *"Our results show that programmers do use undefined behavior in real-world systems, including the Linux kernel and the PostgreSQL database, and that some cases result in serious bugs."*

The investigation consists of a collection of 7 such cases, taken from PostgreSQL, the Linux kernel, and FreeBSD, each with a code snippet, and a preliminary evaluation of the combined cost of three optimisation-limiting compiler flags used by some of these:

```
-fno-strict-overflow
-fno-delete-null-pointer-checks
-fno-strict-aliasing
```

Their first three examples relate to the arithmetic undefined behaviours, which are not our focus in this document: division by zero, oversized shifts, and signed integer overflow.

Their fourth example involves formation of pointers that are (more than one) beyond their original allocation, which can occur in some bounds-checking code. We discuss this in §2.13.

Their fifth example is one where dereferencing a null pointer was expected to cause a kernel oops, but where GCC removes a program-order-later null-pointer check based on such dereferences being undefined behaviour. Our candidate de facto model follows ISO in this respect, but conceivably one could strengthen the behaviour of null-pointer dereferences to definitely trap rather than be undefined behaviour. It is not clear how widely that would be feasible. We add a question to §2.17 for this.

Their sixth example involves integer type aliasing, with a write of a `uint16_t` struct member followed by a read at type `int` (within a Linux-kernel `memcpy`). This is an effective-type question, as we discuss in §4.1.

Their seventh example is an intentional read of uninitialised memory in an attempt to produce entropy, as we discuss in §3.1.2.

The second of these two papers [54] describes a tool, STACK, to identify some instances of what they term "unstable code": *"code that is unexpectedly discarded by compiler optimizations due to undefined behavior in the program"*. They give six motivating examples, where an optimising compiler might remove the body of a conditional, in most cases based on reasoning that it could only be executed in the presence of undefined behaviour:

```
if (p + 100 < p)
{p dereferencable} if (!p)
if (x + 100 < x)
{x non-negative} if (x + 100 < 0)
if (!(1 << x))
if (abs(x) < 0)
```

Their tool detects cases where their (solver-based) optimiser optimises based on ten undefined-behaviour conditions, which we reproduce in Fig. 2. It found significant numbers of bugs in real systems code and many instances of unstable code across a snapshot of all debian packages.

These ten conditions are (as the authors note) sufficient for undefined behaviour but do not characterise it in general; they are very specific. Looking at them in more detail:

- their (1) identifies pointer addition overflow but not the ISO-forbidden more-than-one out-of-bounds pointer arithmetic (this suggests another test, below);
- their (2,6) identify null pointer dereference and out-of-bounds array access but not other illegal pointer dereferences.
- their (3,4,5,7) are arithmetic issues, which are not our focus in this document
- their (8), overlapping memory copy, refers to the ISO `memcpy` text: *"If copying takes place between objects that overlap, the behavior is undefined"* [3, §7.24.2.1]. In Cerberus this library call can be implemented in C except that it needs this explicit undefined-behaviour check.
- their (9,10) identify use-after-free and use-after-realloc cases, which are clearly forbidden in both ISO and de facto standards.

We add two questions following §2.13.1 (p.31), first just forming a pointer value by arithmetic that overflows (on an architecture with 64-bit pointer representations), and then a test that makes an access using such a pointer value.

6.15 Beyond the PDP-11: Architectural support for a memory-safe C abstract machine; Chisnall et al.; ASPLOS 2015

The following examples give simple forms of the "difficult idioms" listed in this paper [14]. The data there shows that most of these idioms occur often in practice and hence that those (mostly) should be allowed in a semantics for a de facto standard C, while a CHERI C semantics will be tighter in some respects.
*"*DECONST *refers to programs that remove the const qualifier from a pointer"*. We used the following example in §5.1.

EXAMPLE (`cheri_01_deconst.c`):

```
#include <stdio.h>
int main() {
  int x=0;
  const int *p = (const int *)&x;
  //are the next two lines free of undefined behaviour?
  int *q = (int*)p;
  *q = 1;
  printf("x=%i  *p=%i  *q=%i\n",x,*p,*q);
}
```

[TODO: fix up the following (cf David's email)] *"*CONTAINER *describes behavior in a macro common in the Linux, BSD, and Windows kernels that, given a pointer to a structure member, returns a pointer to the enclosing structure"*. This is essentially the question of §2.13.4.

EXAMPLE (`cheri_02_container.c`):

```
#include <stdio.h>
#include <stddef.h>
typedef struct { int i; float f; int j; } st;
int main() {
  st s = {.i=1, .f=2.0, .j=3};
  int *pj = &(s.j);
  char *pcj = ((char *)pj);
  char *pcst = (pcj - (offsetof(st,j)-offsetof(st,i)));
  //are these two lines free of undefined behaviour?
  st *ps = (st *)pcst;
  ps->f = 22.0;
  printf("s.i=%i s.f=%f s.j=%i ps->f=%f\n",s.i,s.f,s.j,
         ps->f);
}
```

*"*II *refers to computation of invalid intermediate results. [...] This case refers to pointer arithmetic where the end result is within the bounds of an object, but intermediate results are not"*. We used the next two tests in §2.13.1.

EXAMPLE (`cheri_03_ii.c`):

```
#include <stdio.h>
int main() {
  int x[2];
  int *p = &x[0];
  //is this free of undefined behaviour?
  int *q = p + 11;
  q = q - 10;
  *q = 1;
  printf("x[1]=%i  *q=%i\n",x[1],*q);
}
```

EXAMPLE (`cheri_03_ii_char.c`):

```
#include <stdio.h>
int main() {
  unsigned char x;
  unsigned char *p = &x;
  //is this free of undefined behaviour?
  unsigned char *q = p + 11;
  q = q - 10;
  *q = 1;
  printf("x=0x%x  *p=0x%x  *q=0x%x\n",x,*p,*q);
}
```

*"*INT *refers to storing a pointer in an integer variable in memory — implementation-defined behavior in C. [...] Disallowing this behavior makes accurate garbage collection possible, as the compiler can statically track every pointer use"*. These are the examples we used in §2.2.2:

EXAMPLE (`provenance_roundtrip_via_intptr_t.c`):

```
#include <stdio.h>
#include <inttypes.h>
int  x=1;
int main() {
  int *p = &x;
  intptr_t i = (intptr_t)p;
  int *q = (int *)i;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

EXAMPLE (`provenance_roundtrip_via_unsigned_long.c`):

```
#include <stdio.h>
int  x=1;
```

```
int main() {
  int *p = &x;
  unsigned long i = (unsigned long)p;
  int *q = (int *)i;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

*"*IA *refers to performing integer arithmetic on pointers — such as storing a pointer in an integer value and then performing arbitrary arithmetic on it. This is a more general case of the Int idiom and relies on the same implementation-defined behavior"*. This is essentially a combination of II and Int.

EXAMPLE (`cheri_05_ia.c`):

```
#include <stdio.h>
#include <inttypes.h>
int main() {
  int x=0;
  int *px = &x;
  uintptr_t ql = (uintptr_t)px;
  ql = ql + 287343;
  ql = ql - 287343;
  int *q = (int *)ql;
  *q = 1;
  printf("x=%i  *px=%i  *q=%i\n",x,*px,*q);
}
```

*"*MASK *refers to simple masking of pointers. For example, to store some other data in the low bits"*. This is the test below from §2.2.4.

EXAMPLE (`provenance_tag_bits_via_uintptr_t_1.c`):

```
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
int x=1;
int main() {
  int *p = &x;
  // cast &x to an integer
  uintptr_t i = (uintptr_t) p;
  // 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;
  // are r and p now equivalent?
  *r = 11;           //  defined behaviour?
  _Bool b = (r==p);
  printf("x=%i *r=%i (r==p)=%s\n",x,*r,b?"true":"false");
}
```

*"*WIDE *refers to storing a pointer in an integer variable of a smaller size. This is undefined according to the C specification, but may work if you are able to guarantee that pointers are within a certain range, for example by allocating memory with malloc and the `MAP_32BIT` flag. Code using this idiom is broken by existing implementations, and most likely reflects bugs in the code. We were surprised to see examples of this in programs that we inspected, but fortunately it is sufficiently rare that fixing all of the cases would be easy in these codebases."* This seems sufficiently pathological that we do not include a question for it.

EXAMPLE (`cheri_07_wide.c`):

```
#include <stdio.h>
#include <inttypes.h>
#include <limits.h>
#include <assert.h>
int x=1;
int main() {
  int *p = &x;
  uintptr_t i = (uintptr_t) p;
  assert( i <= UINT_MAX);
  unsigned int j = (unsigned int)i;
  uintptr_t k = (uintptr_t)j;
  int *q = (int *)k;
  *q = 2;
  printf("i=0x%"PRIxPTR" UINT_MAX=0x%x ULONG_MAX=0x%lx\n",
         i,UINT_MAX,ULONG_MAX);
  printf("x=%i  *q=%i\n",x,*q);
}
```

*"*LAST WORD *refers to accessing an object as aligned words without regard for the fact that the objects extent may not include all of the last word. This is used as an optimization for strlen() in FreeBSD libc. While this is undefined behavior in C, it works in systems with pagebased memory protection mechanisms, but not in CHERI where objects have byte granularity. We have found this idiom only in FreeBSDs libc, as reported by valgrind"*. This is the example we used in §3.3.11.

EXAMPLE (`cheri_08_last_word.c`):

```
#include <assert.h>
#include <stdio.h>
#include <inttypes.h>
char c[5];
int main() {
  char *cp = &(c[0]);
  assert(sizeof(uint32_t) == 4);
  uint32_t x0 = *((uint32_t *)cp);
  // does this have defined behaviour?
  uint32_t x1 = *((uint32_t *)(cp+4));
  printf("x0=%x  x1=%x\n",x0,x1);
}
```

6.16 What every C programmer should know about undefined behavior; Lattner; Blog post 2011

Part 1 of this three-part blog post by Chris Lattner[64] discusses how six forms of undefined behaviour permit desirable compiler optimisation:

- *Use of an uninitialized variable*

  As we discuss in §3, in ISO C11 this does not always give rise to undefined behaviour. The motivation given by Lattner for treating this as undefined behaviour would apply

---

[64] `http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html`

equally to a semantics in which reading uninitialised variables gives unspecified values.

- *Signed integer overflow*
- *Oversized Shift Amounts*

  These two are both integer arithmetic undefined behaviours, which are not our focus in this document.
- *Dereferences of Wild Pointers and Out of Bounds Array Accesses*
- *Dereferencing a NULL Pointer*

  These are both discussed in the previous subsection (§6.17, point 4).
- *Violating Type Rules*

  This explains the motivation for type-based alias analysis, but for our candidate de facto memory model we focus on the `-fno-strict-aliasing` case.

Part 3 of this series lists some cases where Clang adopts a stronger semantics than ISO, including:

"2 Arithmetic that operates on undefined values is considered to produce a undefined value instead of producing undefined behavior."

"3 Arithmetic that dynamically executes an undefined operation (such as a signed integer overflow) generates a logical trap value which poisons any computation based on it, but that does not destroy your entire program."

6.17 Proposal for a Friendly Dialect of C; Cuoq, Flatt, Regehr; Blog post 2014

This blog post[65] makes an initial proposal for a more predictable dialect of C. They write: *"As a starting point, we imagine that friendly C is like the current C standard, but replacing many occurrences of 'X has undefined behavior' with 'X results in an unspecified value'. That adjustment alone can produce a much friendlier language. In other cases, we may be forced to refer to machine-specific details that are not features of the C abstract machine, and we are OK with that."* and list 14 features, as below. Many of these relate to integer arithmetic undefined behaviours, which are not our focus in this document. In the other direction, the blog post does not discuss most of our memory-model questions.

1 *The value of a pointer to an object whose lifetime has ended remains the same as it was when the object was alive.*

This would change the ISO "no" to a "yes" for our question in §2.16.1.

2 *Signed integer overflow results in twos complement wrapping behavior at the bitwidth of the promoted type.*

Integer arithmetic UB. This could be accommodated in the Cerberus semantics with an easy change to the elaboration function.

3 *Shift by negative or shift-past-bitwidth produces an unspecified result.*

Integer arithmetic UB. This could be accommodated in the Cerberus semantics with an easy change to the elaboration function.

4 *Reading from an invalid pointer either traps or produces an unspecified value. In particular, all but the most arcane hardware platforms can produce a trap when dereferencing a null pointer, and the compiler should preserve this behavior.*

See §2.17.2.

For null pointers, on many platforms one could require them to definitely give a runtime failure, as per our question in §2.17.1.

5 *Division-related overflows either produce an unspecified result or else a machine-specific trap occurs.*

Integer arithmetic UB. This could be accommodated in the Cerberus semantics with an easy change to the elaboration function.

6 *If possible, we want math- and memory-related traps to be treated as externally visible side-effects that must not be reordered with respect to other externally visible side-effects (much less be assumed to be impossible), but we recognize this may result in significant runtime overhead in some cases.*

The impact of 4–6 on optimisations that involve code motion isn't clear to us.

7 *The result of any signed left-shift is the same as if the left-hand shift argument was cast to unsigned, the shift performed, and the result cast back to the signed type.*

Integer arithmetic UB. This could be accommodated in the Cerberus semantics with an easy change to the elaboration function.

8 *A read from uninitialized storage returns an unspecified value.*

This is our question in §3.1.2. Though exactly how Friendly-C unspecified values should behave, e.g. w.r.t. strictness and our other §3.2 questions, is not stated.

9 *It is permissible to compute out-of-bounds pointer values including performing pointer arithmetic on the null pointer. This works as if the pointers had been cast to `uintptr_t`. However, the translation from pointer math to integer math is not completely straightforward since incrementing a pointer by one is equivalent to incrementing the integer-typed variable by the size of the pointed-to type.*

---

[65] `http://blog.regehr.org/archives/1180` and followup `http://blog.regehr.org/archives/1287`

The first part is our question from §2.13.1. The second is handled in Cerberus by the elaboration.

10 *The strict aliasing rules simply do not exist: the representations of integers, floating-point values and pointers can be accessed with different types.*

This matches our candidate de facto memory model choice to focus on the `-fno-strict-aliasing` behaviour.

11 *A data race results in unspecified behavior. Informally, we expect that the result of a data race is the same as in C99: threads are compiled independently and then data races have a result that is dictated by the details of the underlying scheduler and memory system. Sequentially consistent behavior may not be assumed when data races occur.*

This is rather unclear: what does this usage of "unspecified behaviour" mean?

12 *memcpy() is implemented by memmove(). Additionally, both functions are no-ops when asked to copy zero bytes, regardless of the validity of their pointer arguments.*

This is a library undefined-behaviour issue; we've so far not looked into those.

13 *The compiler is granted no additional optimization power when it is able to infer that a pointer is invalid. In other words, the compiler is obligated to assume that any pointer might be valid at any time, and to generate code accordingly. The compiler retains the ability to optimize away pointer dereferences that it can prove are redundant or otherwise useless.*

The force of this is unclear, especially w.r.t. provenance-based alias analysis.

14 *When a non-void function returns without returning a value, an unspecified result is returned to the caller.*

This is presumably also an easy elaboration change.

6.18 UB Canaries; Regehr; Blog post 2015

This blog post[66] by John Regehr gives *"a collection of canaries for undefined behavior: little test programs that automate the process of determining whether a given compiler configuration is willing to exploit particular UBs."*, together with the results for several versions of GCC and LLVM.

The first two examples (`addr_null_p1.c` and `addr_null_p2.c`) test whether one can use the address of members of a NULL struct pointer in place of `offsetof`. We add an example to §2.13.6 for this.

`array_oob_p1.c` contains a straightforward out-of-bounds array-read undefined behaviour (the question for the canaries is whether compilers aggressively exploit that). `array_oob_p2.c` is similar.

The `dangling_pointer_p1.c`, `dangling_pointer_p2.c`, and `dangling_pointer_p3.c` examples check whether compilers optimise based on an assumption that an out-of-lifetime pointer is distinct from another pointer, after the end of a block scope, a `realloc`, and a `free` respectively. See our §2.16.1, where we give block-end and `free` tests.

The `int_min_mod_minus_1_p1.c` tests `INT_MIN % -1`. Not being a memory object question, this is not in the scope of this note.

`memcpy_overlap_p1.c` tests random `memcpy`'s, presumably to check whether the compiler exploits the [3, §7.24.2.1] statement that overlapping `memcpy`'s (unlike overlapping `memmove`'s) give undefined behaviour. We could add another question, asking whether such a `memcpy` gives a well-defined copy, unspecified values in the target footprint, or undefined behaviour.

`modify_string_literal_p1.c` tries to modify a string literal, undefined behaviour by [3, §6.4.5p7].

`pointer_casts_p1.c` tries to cast away a `const` from a pointer and write using the result.

`pointer_casts_p2.c` tries to use a non-volatile pointer to mutate a `volatile int`; we have not considered `volatile` in this note.

`shift_by_bitwidth_p1.c` tests whether *"it's OK to shift an integer by its bitwidth and the result is 0"*; an arithmetic property we do not consider in this note. `signed_integer_overflow_p1.c`, `signed_integer_overflow_p2.c`, `signed_left_shift_p1.c`, and `signed_left_shift_p2.c` are similarly outside our scope here.

`strict_aliasing_p1.c` is a basic effective-types type punning question, as in our §4.1.1.

`uninitialized_variable_p1.c`, `uninitialized_variable_p2.c`, and `uninitialized_variable_p3.c` involve stability, strictness, and control-flow choices of unspecified values, as in our questions 50, 51, and 52.

`uninitialized_variable_p4.c` asks whether a comparison `x < INT_MIN`, where `x` is uninitialised, is guaranteed false. If all operations on unspecified values give unspecified values (c.f. our Question 52) then the answer to this would be no. `uninitialized_variable_p5.c` is similar but for `>`.

---

[66] http://blog.regehr.org/archives/1234

References

[1] *Programming Languages — C*. December 1999. Second edition. ISO/IEC 9899:1999 (E).

[2] Rationale for international standard – programming languages – C, revision 5.10, April 2003. `www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf`.

[3] *Programming Languages — C*. 2011. ISO/IEC 9899:2011. A non-final but recent version is available at `www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

[4] Lars Ole Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[5] ARM. Procedure call standard for the ARM architecture, November 2012. ARM IHI 0042E, current through ABI release 2.09.

[6] Ryan S. Arnold, Greg Davis, Brian Deitrich, Michael Eager, Emil Medve, Steven J. Munroe, Joseph S. Myers, Steve Papacharalambous, Anmol P. Paralkar, Katherine Stewart, and Edmar Wienskoski. DRAFT: Power Architecture 32-bit Application Binary Interface, Supplement 1.0 - Linux and Embedded, April 2011.

[7] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 509–520, New York, NY, USA, 2012. ACM.

[8] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[9] F. Besson, S. Blazy, and P. Wilke. A precise and abstract memory model for C using symbolic values. In *APLAS*, 2014.

[10] F. Besson, S. Blazy, and P. Wilke. A concrete memory model for CompCert. In *Proc. ITP*, 2015.

[11] Paul E. Black and Phillip J. Windley. Inference rules for programming languages with side effects in expressions. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '96, pages 51–60, London, UK, UK, 1996. Springer-Verlag.

[12] Paul E. Black and Phillip J. Windley. Formal verification of secure programs in the presence of side effects. In *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences - Volume 3*, HICSS '98, pages 327–, Washington, DC, USA, 1998. IEEE Computer Society.

[13] Mark Boffinger. *Reasoning about C programs*. PhD thesis, University of Queensland, 1998.

[14] David Chisnall, Colin Rothwell, Brooks Davis, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Peter G. Neumann, and Michael Roe. Beyond the pdp-11: Processor support for a memory-safe c abstract machine. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2015. ACM.

[15] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electron. Notes Theor. Comput. Sci. (SSV 2009)*, 254:85–103, October 2009.

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

[17] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *Proc. ASPLOS*, 2008.

[18] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proc. POPL*, 2012.

[19] Daniel Joseph Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Ithaca, NY, USA, 2003. AAI3104470.

[20] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In *Proc. CSL '92*, 1993.

[21] C. Hathhorn, C. Ellison, and G. Rosu. Defining the undefinedness of C. In *Proc. PLDI*, 2015.

[22] Charles McEwen Ellison III. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[23] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. USENIX ATC*, 2002.

[24] Derek M. Jones. The new C standard: An economic and cultural commentary. `http://www.coding-guidelines.com/cbook/`. Accessed 2014-06-16.

[25] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A formal C memory model supporting integer-pointer casts. In *Proc. PLDI*, 2015.

[26] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[27] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *Proc. CPP, LNCS 8307*, 2013.

[28] R. Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in C. In *Proc. POPL*, 2014.

[29] R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, December 2015.

[30] R. Krebbers and F. Wiedijk. Separation logic for non-local control flow and block scope variables. In *FoSSaCS*, 2013.

[31] R. Krebbers and F. Wiedijk. A typed C11 semantics for interactive theorem proving. In *Proc. CPP*, 2015.

[32] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C semantics: Compcert and the C standard. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 543–548, 2014.

[33] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012.

[34] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[35] Justus Matthiesen. Mathematizing the C programming language, May 2011. University of Cambridge Computer Science Tripos Part II project dissertation.

[36] Justus Matthiesen. Elaborating C, June 2012. University of Cambridge Computer Science ACS MPhil dissertation.

[37] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell (Eds.). System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.99.6, October 2013.

[38] Microsoft. Visual Studio 2013, Aggregates and Unions. `http://msdn.microsoft.com/en-us/library/9dbwhz68.aspx`, 2013. Accessed 2014-06-16.

[39] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proc. PLDI*, 2013.

[40] S. Nagarakatte, J. Zhao, M. M.K. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proc. PLDI*, 2009.

[41] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, 2002.

[42] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[43] M. Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, U. Cambridge, Computer Laboratory, 1998.

[44] M. Norrish. Deterministic expressions in C. In *ESOP*, 1999.

[45] Santa Cruz Operation. SYSTEM V APPLICATION BINARY INTERFACE, MIPS RISC Processor Supplement, 3rd Edition, February 1996.

[46] N. S Papaspyrou. *A formal semantics for the C programming language*. PhD thesis, National Technical University of Athens, 1998.

[47] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI '12: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 311–322. ACM Press, June 2012.

[48] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proc. POPL*, 2007.

[49] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, UNSW, Sydney, Australia, aug 2008.

[50] Harvey Tuch and Gerwin Klein. A unified memory model for pointers. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 474–488, Montego Bay, Jamaica, dec 2005.

[51] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proceedings of POPL 2011: the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–54, 2011.

[52] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), June 2013.

[53] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: what happened to my code? In *Proc. APSYS*, 2012.

[54] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proc. SOSP*, 2013.

[55] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, and R. Norton. Capability hardware enhanced RISC instructions: CHERI instruction-set architecture. Technical Report UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, November 2015.

[56] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy, SP*, 2015.

[57] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. PLDI*, 2011.

Index