C provenance semantics: examples (for PNVI-plain, PNVI address-exposed, PNVI address-exposed user-disambiguation, and PVI models)

PETER SEWELL, University of Cambridge
KAYVAN MEMARIAN, University of Cambridge
VICTOR B. F. GOMES, University of Cambridge
JENS GUSTEDT, Université de Strasbourg, CNRS, Inria, ICube, Strasbourg, France
MARTIN UECKER, University Medical Center, Goettingen

This note discusses the design of provenance semantics for C, looking at a series of examples. We consider three variants of the provenance-not-via-integer (PNVI) model: PNVI plain, PNVI address-exposed (PNVI-ae) and PNVI address-exposed k user-disambiguation (PNVI-ae-udi), and also the provenance-via-integers (PVI) model. The examples include those of *Exploring C Semantics and Pointer Provenance* [POPL 2019] (also available as ISO WG14 N2311 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2311.pdf), with several additions. This is based on recent discussion in the C memory object model study group. It should be read together with the two companion notes, one giving detailed semantics for these variants, and another giving detailed diffs to the C standard text.

Contents

1 INTRODUCTION

The new material for PNVI-address-exposed and PNVI address-exposed user-disambiguation models starts in §3, but first we introduce introduce the problem in general and describe the basic pointer provenance semantics.

The semantics of pointers and memory objects in C has been a vexed question for many years. A priori, one might imagine two language-design extremes: a concrete model that exposes the memory semantics of the underlying hardware, with memory being simply a finite partial map from machine-word addresses to bytes and pointers that are simply machine words, and an abstract model in which the language types enforce hard distinctions, e.g. between numeric types that support arithmetic and pointer types that support dereferencing. C is neither of these. Its values are not abstract: the language intentionally permits manipulation of their underlying representations, via casts between pointer and integer types, `char*` pointers to access representation bytes, and so on, to support low-level systems programming. But C values also cannot be considered to be simple concrete values: at runtime a C pointer will typically just be a machine word, but compiler analysis reasons about abstract notions of the provenance of pointers, and compiler optimisations rely on assumptions about these for soundness. Particularly relevant here, some compiler optimisations rely on alias analysis to deduce that two pointer values do not refer to the same object, which in turn relies on assumptions that the program only constructs pointer values in "reasonable" ways (with other programs regarded as having undefined behaviour, UB). The committee response to Defect Report DR260 [Feather 2004] states that implementations can track the origins (or "provenance") of pointer values, *"the implementation is entitled to take account of the provenance of a pointer value when determining what actions are and are not defined"*, but exactly what this "provenance" means is left undefined, and it has never been incorporated into the standard text. Even what a memory object is is not completely clear in the standard, especially for aggregate types and for objects within heap regions.

Second, in some respects there are significant discrepancies between the ISO standard and the de facto standards, of C as it is implemented and used in practice. Major C codebases typically rely on particular compiler flags, e.g. `-fno-strict-aliasing` or `-fwrapv`, that substantially affect the semantics but which standard does not attempt to describe, and some idioms are UB in ISO C but relied on in practice, e.g. comparing against a pointer

value after the lifetime-end of the object it pointed to. There is also not a unique de facto standard: in reality, one has to consider the expectations of expert C programmers and compiler writers, the behaviours of specific compilers, and the assumptions about the language implementations that the global C codebase relies upon to work correctly (in so far as it does). Our recent surveys [Memarian et al. 2016; Memarian and Sewell 2016b] of the first revealed many discrepancies, with widely conflicting responses to specific questions.

Third, the ISO standard is a prose document, as is typical for industry standards. The lack of mathematical precision, while also typical for industry standards, has surely contributed to the accumulated confusion about C, but, perhaps more importantly, the prose standard is not *executable as a test oracle*. One would like, given small test programs, to be able to automatically compute the sets of their allowed behaviours (including whether they have UB). Instead, one has to do painstaking argument with respect to the text and concepts of the standard, a time-consuming and error-prone task that requires great expertise, and which will sometimes run up against the areas where the standard is unclear or differs with practice. One also cannot use conventional implementations to find the sets of all allowed behaviours, as (a) the standard is a loose specification, while particular compilations will resolve many nondeterministic choices, and (b) conventional implementations cannot detect all sources of undefined behaviour (that is the main point of UB in the standard, to let implementations assume that source programs do not exhibit UB, together with supporting implementation variation beyond the UB boundary). Sanitisers and other tools can detect some UB cases, but not all, and each tool builds in its own more-or-less ad hoc C semantics.

This is not just an academic problem: disagreements over exactly what is or should be permitted in C have caused considerable tensions, e.g. between OS kernel and compiler developers, as increasingly aggressive optimisations can break code that worked on earlier compiler implementations.

This note continues an exploration of the design space and two candidate semantics for pointers and memory objects in C, taking both ISO and de facto C into account. We earlier [Chisnall et al. 2016; Memarian et al. 2016] identified many design questions. We focus here on the questions concerning pointer provenance, which we revise and extend. We develop two main coherent proposals that reconcile many design concerns; both are broadly consistent with the provenance intuitions of practitioners and ISO DR260, while still reasonably simple. We highlight their pros and cons and various outstanding open questions. These proposals cover many of the interactions between abstract and concrete views in C: casts between pointers and integers, access to the byte representations of values, etc.

2 BASIC POINTER PROVENANCE

C pointer values are typically represented at runtime as simple concrete numeric values, but mainstream compilers routinely exploit information about the *provenance* of pointers to reason that they cannot alias, and hence to justify optimisations. In this section we develop a provenance semantics for simple cases of the construction and use of pointers,

For example, consider the classic test [Chisnall et al. 2016; Feather 2004; Krebbers and Wiedijk 2012; Krebbers 2015; Memarian et al. 2016] on the right (note that this and many of the examples below are edge-cases, exploring the boundaries of what different semantic choices allow, and sometimes what behaviour existing compilers exhibit; they are not all intended as desirable code idioms).

```
// provenance_basic_global_yx.c (and an xy variant)
#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);
  }
}
```

Depending on the implementation, `x` and `y` might in some executions 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, e.g. with GCC 8.1 -O2 on some platforms. Its 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`. ICC, e.g. ICC 19 -O2, also optimises here (for a variant with `x` and `y` swapped), producing `x=1 y=2 *p=11 *q=11`. In contrast, Clang 6.0 -O2 just outputs the `x=1 y=11 *p=11 *q=11` that one might expect from a concrete semantics. Note that this example does not involve type-based alias analysis, and the outcome is not affected by GCC or ICC's `-fno-strict-aliasing` flag. Note also that the mere formation of the `&x+1` one-past pointer is explicitly permitted by the ISO standard, and, because the `*p=11` access is guarded by the

`memcmp` conditional check on the representation bytes of the pointer, it will not be attempted (and hence flag UB) in executions in which the two storage instances are not adjacent.

These GCC and ICC outcomes would not be correct with respect to a concrete semantics, and so to make the existing compiler behaviour sound it is necessary for this program to be deemed to have undefined behaviour.

The current ISO standard text does not explicitly speak to this, but the 2004 ISO WG14 C standards committee response to Defect Report 260 (DR260 CR) [Feather 2004] hints at a notion of provenance associated to values that keeps track of their "origins":

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

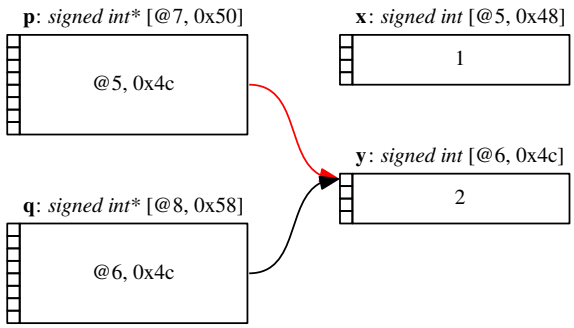
However, DR260 CR has never been incorporated in the standard text, and it gives no more detail. This leaves many specific questions unclear: it is ambiguous whether some programming idioms are allowed or not, and exactly what compiler alias analysis and optimisation are allowed to do.

Basic provenance semantics for pointer values For simple cases of the construction and use of pointers, capturing the basic intuition suggested by DR260 CR in a precise semantics is straightforward: we associate a *provenance* with every pointer value, identifying the original storage instance the pointer is derived from. In more detail:

- We take abstract-machine pointer values to be pairs (π, a), adding a *provenance* π, either $@i$ where i is a storage instance ID, or the *empty* provenance `@empty`, to their concrete address a.
- On every storage instance (of objects with static, thread, automatic, and allocated storage duration), the abstract machine nondeterministically chooses a fresh storage instance ID i (unique across the entire execution), and the resulting pointer value carries that single storage instance ID as its provenance $@i$.
- Provenance is preserved by pointer arithmetic that adds or subtracts an integer to a pointer.
- At any access via a pointer value, its numeric address must be consistent with its provenance, with undefined behaviour otherwise. In particular:
  - access via a pointer value which has provenance a single storage instance ID $@i$ must be within the memory footprint of the corresponding original storage instance, which must still be live.
  - all other accesses, including those via a pointer value with empty provenance, are undefined behaviour.

  This undefined behaviour is what justifies optimisation based on provenance alias analysis.

On the right is a provenance-semantics memory-state snapshot (from the Cerberus GUI) for `provenance_basic_global_xy.c`, just before the invalid access via `p`, showing how the provenance mismatch makes it UB: at the attempted access via `p`, its pointer-value address 0x4c is not within the storage instance with the ID `@5` of the provenance of `p`.



All this is for the *C abstract machine* as defined in the standard: compilers might rely on provenance in their alias analysis and optimisation, but one would not expect normal implementations to record or manipulate provenance at runtime (though dynamic or static analysis tools might, as might non-standard implementations such as CHERI C). Provenances therefore do not have program-accessible runtime representations in the abstract machine.

Even for the basic provenance semantics, there are some open design questions, which we now discuss.

Can one construct out-of-bounds (by more than one) pointer values by pointer arithmetic? Consider the example below, where `q` is transiently (more than one-past) out of bounds but brought back into bounds before being used for access. In ISO C, constructing such a pointer value is clearly stated to be undefined behaviour [WG14 2018, 6.5.6p8]. This can be captured using the provenance of the pointer value to determine the relevant bounds. There are cases where such pointer arithmetic would go wrong on some platforms (some now exotic), e.g. where pointer arithmetic subtraction overflows, or if the transient value is not aligned and only aligned values are representable at the particular pointer type, or for hardware that does bounds checking, or where pointer arithmetic might wrap at values less than the obvious word size (e.g. "near" or "huge" 8086 pointers). However, transiently out-of-bounds pointer construction seems to be common in practice. It may be desirable to make it implementation-defined whether such pointer construction is allowed. That would continue to permit implementations in which it would

```
// cheri_03_ii.c
int x[2];
int *p = &x[0];
int *q = p + 11; // defined behaviour?
q = q - 10;
*q = 1;
```

go wrong to forbid it, but give a clear way for other implementations to document that they do not exploit this UB in compiler optimisations that may be surprising to programmers.

Inter-object pointer arithmetic The first example in this section relied on guessing (and then checking) the offset between two storage instances. What if one instead calculates the offset, with pointer subtraction; should that let one move between objects, as below? In ISO C18, the `q-p` is UB (as it is a pointer subtraction between pointers to different objects, which in some abstract-machine executions are not one-past-related). In a variant semantics that allows construction of more-than-one-past pointers (which allows the evaluation of `p + offset`), one would have to to choose whether the `*r=11` access is UB or not. The basic provenance semantics will forbid it, because `r` will retain the provenance of the `x` storage instance, but its address is not in bounds for that. This is probably the most desirable semantics: we have found very few example idioms that intentionally use inter-object pointer arithmetic, and the freedom that forbidding it gives to alias analysis and optimisation seems significant.

```
// pointer_offset_from_ptr_subtraction_global_xy.c
#include <stdio.h>
#include <string.h>
#include <stddef.h>
int x=1, y=2;
int main() {
  int *p = &x;
  int *q = &y;
  ptrdiff_t offset = q - p;
  int *r = p + offset;
  if (memcmp(&r, &q, sizeof(r)) == 0) {
    *r = 11; // is this free of UB?
    printf("y=%d *q=%d *r=%d\n",y,*q,*r);
  }
}
```

Pointer equality comparison and provenance A priori, pointer equality comparison (with == or `!=`) might be expected to just compare their numeric addresses, but we observe GCC 8.1 -O2 sometimes regarding two pointers with the same address but different provenance as nonequal. Unsurprisingly, this happens in some circumstances but not others, e.g. if the test is pulled into a simple separate function, but not if in a separate compilation unit. To be conservative w.r.t. current compiler behaviour, pointer equality in the semantics should give false if the addresses are not equal, but nondeterministically (at each runtime occurrence) either take provenance into account or not if the addresses are equal – this specification looseness accommodating implementation variation. Alternatively, one could require numeric comparisons, which would be a simpler semantics for programmers but force that GCC behaviour to be regarded as a bug. Cerberus supports both options. One might also imagine making it UB to compare pointers that are not strictly within their original storage instance [Krebbers 2015], but that would break loops that test against a one-past pointer, or requiring equality to *always* take provenance into account, but that would require implementations to track provenance at runtime.

```
// provenance_equality_global_xy.c
#include <stdio.h>
#include <string.h>
int x=1, y=2;
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;
}
```

The current ISO C18 standard text is too strong here unless numeric comparison is required: 6.5.9p6 says *"Two pointers compare equal **if and only if** both are [...] 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"*, which requires such pointers to compare equal – reasonable pre-DR260 CR, but debatable after it.

Pointer equality should not be confused with alias analysis: we could require == to return true for pointers with the same address but different provenance, while still permitting alias analysis to regard the two as distinct by making accesses via pointers with the wrong provenance UB.

Pointer relational comparison and provenance In ISO C (6.5.8p5), inter-object pointer relational comparison (with < etc.) is undefined behaviour. Just as for inter-object pointer subtraction, there are platforms where this would go wrong, but there are also substantial bodies of code that rely on it, e.g. for lock orderings

It may be desirable to make it implementation-defined whether such pointer construction is allowed.

3 REFINING THE BASIC PROVENANCE MODEL TO SUPPORT POINTER CONSTRUCTION VIA CASTS, REPRESENTATION ACCESSES, ETC.

To support low-level systems programming, C provides many other ways to construct and manipulate pointer values:

- casts of pointers to integer types and back, possibly with integer arithmetic, e.g. to force alignment, or to store information in unused bits of pointers;
- copying pointer values with `memcpy`;
- manipulation of the representation bytes of pointers, e.g. via user code that copies them via `char*` or `unsigned char*` accesses;
- type punning between pointer and integer values;
- I/O, using either `fprintf`/`fscanf` and the `%p` format, `fwrite`/`fread` on the pointer representation bytes, or pointer/integer casts and integer I/O;
- copying pointer values with `realloc`;
- constructing pointer values that embody knowledge established from linking, and from constants that represent the addresses of memory-mapped devices.

A satisfactory semantics has to address all these, together with the implications on optimisation. We define and explore several alternatives:

- **PNVI-plain**: a semantics that does not track provenance via integers, but instead, at integer-to-pointer cast points, checks whether the given address points within a live object and, if so, recreates the corresponding provenance. We explain in the next section why this is not as damaging to optimisation as it may sound.
- **PNVI-ae (PNVI exposed-address)**: a variant of PNVI that allows integer-to-pointer casts to recreate provenance only for storage instances that have previously been *exposed*. A storage instance is deemed exposed by a cast of a pointer to it to an integer type, by a read (at non-pointer type) of the representation of the pointer, or by an output of the pointer using `%p`.
- **PNVI-ae-udi (PNVI exposed-address user-disambiguation)**: a further refinement of PNVI-ae that supports roundtrip casts, from pointer to integer and back, of pointers that are one-past a storage instance. This is the currently preferred option in the C memory object model study group.
- **PVI**: a semantics that tracks provenance via integer computation, associating a provenance with all integer values (not just pointer values), preserving provenance through integer/pointer casts, and making some particular choices for the provenance results of integer and pointer +/- integer operations; or

We write PNVI-* for PNVI-plain, PNVI-ae, and PNVI-ae-udi. The PNVI-plain and PVI semantics were described in the POPL 2019/N2311 paper. PNVI-ae and PNVI-ae-udi have emerged from discussions in the C memory object model study group.

We also mention other variants of PNVI that seem less desirable:

- **PNVI-address-taken**: an earlier variant of PNVI-ae that allowed integer-to-pointer casts to recreate provenance for objects whose address has been taken (irrespective of whether it has been exposed); and
- **PNVI-wildcard**: a variant that gives a "wildcard" provenance to the results of integer-to-pointer casts, delaying checks to access time.

The PVI semantics, originally developed informally in ISO WG14 working papers [Memarian et al. 2018; Memarian and Sewell 2016a], was motivated in part by the GCC documentation [FSF 2018]:

> *"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."*

which presumes there is *an* "original" pointer, and by experimental data for `uintptr_t` analogues of the first test of §2, which suggested that GCC and ICC sometimes track provenance via integers (see `xy` and `yx` variants). However, discussions at the 2018 GNU Tools Cauldron suggest instead that at least some key developers regard the result of casts from integer types as potentially broadly aliasing, at least in their GIMPLE IR, and such test results as long-standing bugs in the RTL backend.

4 REFINING THE BASIC PROVENANCE MODEL: PHENOMENA AND EXAMPLES

Pointer/integer casts The ISO standard (6.3.2.3) leaves conversions between pointer and integer types almost entirely implementation-defined, except for conversion of integer constant `0` and null pointers, and for the optional `intptr_t` and `uintptr_t` types, for which it guarantees that any *"valid pointer to* **`void`***"* can be converted and back, and that *"the result will compare equal to the original pointer"*. As we have seen, in a post-DR260 CR provenance-aware semantics, *"compare equal"* is not enough to guarantee the two are interchangeable, which was clearly the intent of that phrasing. All variants of PNVI-* and PVI support this, by reconstructing or preserving the original provenance respectively.

```
// 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);
}
```

Inter-object integer arithmetic Below is a `uintptr_t` analogue of the §2 example `pointer_offset_from_ptr_subtraction_global_xy.c`, attempting to move between objects with `uintptr_t` arithmetic. In PNVI-*, this has defined behaviour. For PNVI-plain: the integer values are pure integers, and at the **`int`**`*` cast the value of `ux+offset` matches the address of `y` (live and of the right type), so the resulting pointer value takes on the provenance of the `y` storage instance. For PNVI-ae and PNVI-ae-udi, the storage instance for `y` is marked as *exposed* at the cast of `&y` to an integer, and so the above is likewise permitted there.

In PVI, this is UB. First, the integer values of `ux` and `uy` have the provenances of the storage instances of `x` and `y` respectively. Then `offset` is a subtraction of two integer values with non-equal single provenances; we define the result of such to have the empty provenance. Adding that empty-provenance result to `ux` preserves the original `x`-storage instance provenance of the latter, as does the cast to **`int`**`*`. Then the final `*p=11` access is via a pointer value whose address is not consistent with its provenance. Similarly, PNVI-* allows (contrary to current GCC/ICC O2) a `uintptr_t` analogue of the first test of §2, on the left below. PVI forbids this test.

```
// pointer_offset_from_int_subtraction_global_xy.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int x=1, y=2;
int main() {
  uintptr_t ux = (uintptr_t)&x;
  uintptr_t uy = (uintptr_t)&y;
  uintptr_t offset = uy - ux;
  printf("Addresses: &x=%"PRIuPTR" &y=%"PRIuPTR\
         " offset=%"PRIuPTR" \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 UB?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

```
// provenance_basic_using_uintptr_t_global_xy.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int x=1, y=2;
int main() {
  uintptr_t ux = (uintptr_t)&x;
  uintptr_t uy = (uintptr_t)&y;
  uintptr_t offset = 4;
  ux = ux + offset;
  int *p = (int *)ux; // does this have UB?
  int *q = &y;
  printf("Addresses: &x=%p p=%p &y=%"PRIxPTR\
         "\n",(void*)&x,(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);
  }
}
```

```
// 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");
}
```

Both choices are defensible here: PVI will permit more aggressive alias analysis for pointers computed via integers (though those may be relatively uncommon), while PNVI-* will allow not just this test, which as written is probably not idiomatic desirable C, but also the essentially identical XOR doubly linked list idiom, using only one pointer per node by storing the XOR of two, on the right above. Opinions differ as to whether that idiom matters for modern code.

There are other real-world but rare cases of inter-object arithmetic, e.g. in the implementations of Linux and FreeBSD per-CPU variables, in fixing up pointers after a `realloc`, and in dynamic linking (though arguably some of these are not between C abstract-machine objects). These are rare enough that it seems reasonable to require additional source annotation, or some other mechanism, to prevent compilers implicitly assuming that uses of such pointers as undefined.

Pointer provenance for pointer bit manipulations It is a standard idiom in systems code to use otherwise unused bits of pointers: low-order bits for pointers known to be aligned, and/or high-order bits beyond the addressable range. The example on the right (which assumes `_Alignof(int) >= 4`) does this: casting a pointer to `uintptr_t` and back, using bitwise logical operations on the integer value to store some tag bits.

To allow this, we suggest that the set of unused bits for pointer types of each alignment should be made implementation-defined. In PNVI-* the intermediate value of `q` will have empty provenance, but the value of `r` used for the access will re-acquire the correct provenance at cast time. In PVI we make the binary operations used here, combining an integer value that has some provenance ID with a pure integer, preserve that provenance.

(A separate question is the behaviour if the integer value with tag bits set is converted back to pointer type. In ISO the result is implementation-defined, per 6.3.2.3p{5,6} and 7.20.1.4.)

```
// provenance_tag_bits_via_uintptr_t_1.c
#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;
  // set low-order bit
  i = i | 1u;
  // cast back to a pointer
  int *q = (int *) i; // does this have UB?
  // cast to integer and mask out low-order 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;           //  does this have UB?
  _Bool b = (r==p);  //  is this true?
  printf("x=%i *r=%i (r==p)=%s\n",x,*r,b?"t":"f");
}
```

Algebraic properties of integer operations The PVI definitions of the provenance results of integer operations, chosen to make `pointer_offset_from_int_subtraction_global_xy.c` forbidden and `provenance_tag_bits_via_uintptr_t_1.c` allowed, has an unfortunate consequence: it makes those operations no longer associative. Compare the examples below:

```
                          // 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;
}
                          // 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;
}
```

The latter is UB in PVI. It is unclear whether this would be acceptable in practice, either for C programmers or for compiler optimisation. One could conceivably switch to a PVI-multiple variant, allowing provenances to be finite sets of storage instance IDs. That would allow the `pointer_offset_from_int_subtraction_global_xy.c` example above, but perhaps too much else besides. The PNVI-* models do not suffer from this problem.

Copying pointer values with `memcpy()` This clearly has to be allowed, and so, to make the results usable for accessing memory without UB, `memcpy()` and similar functions have to preserve the original provenance. The ISO C18 text does not explicitly address this (in a pre-provenance semantics, before DR260, it did not need to). One could do so by special-casing `memcpy()` and similar functions to preserve provenance, but the following questions suggest less ad hoc approaches, for PNVI-plain or PVI. For PNVI-ae and PNVI-ae-udi, the best approach is not yet clear.

```
// 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);
}
```

Copying pointer values bytewise, with user-`memcpy` One of the key aspects of C is that it supports manipulation of object representations, e.g. as in the following naive user implementation of a `memcpy`-like function, which constructs a pointer value from copied bytes. This too should be allowed. PNVI-plain makes it legal: the representation bytes have no provenance, but when reading a pointer value from the copied memory, the read will be from multiple representation-byte writes. We use essentially the same semantics for such reads as for integer-to-pointer casts: checking at read-time that the address is within a live object, and giving the result the corresponding provenance. For PNVI-ae and PNVI-ae-udi, the current proposal is to mark storage instances as exposed whenever representation bytes of pointers to them are read, and use the same semantics for reads of pointer values from representation-byte writes as for integer-to-pointer casts. This is attractively simple, but it does means that integer-to-pointer casts become permitted for all storage instances for which a pointer has been copied via `user_memcpy`, which is arguably too liberal. It may be possible to add additional annotations for code like `user_memcpy` to indicate (to alias analysis) that (a) their target memory should have the same provenance as their source memory, and (b) the storage instances of any copied pointers should not be marked as exposed, despite the reads of their representation bytes. This machinery has not yet been designed.

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

One might instead think of recording symbolically in the semantics of integer values (e.g. for representation-byte values) whether they are of the form "byte n of pointer value v", or perhaps "byte n of pointer value of type t", and allow reads of pointer values from representation-byte writes only for such. This is more complex and rather ad hoc, arbitrarily restricting the integer computation that can be done on such bytes. If one wanted to allow (e.g.) bitwise operations on such bytes, as in `provenance_tag_bits_via_repr_byte_1.c`, one would essentially have to adopt a PVI model. However, note that to capture the 6.5p6 preservation of effective types by character-type array copy (*"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."*), we might need something like a very restricted version of PVI: some effective-type information attached to integer values of character type, to say"byte n of pointer value of type t", with all integer operations except character-type stores clearing that info.

As Lee observes [private communication], to make it legal for compilers to replace user-memcpy by the library version, one might want the two to have exactly the same semantics. Though strictly speaking that is a question about the compiler intermediate language semantics, not C source semantics.

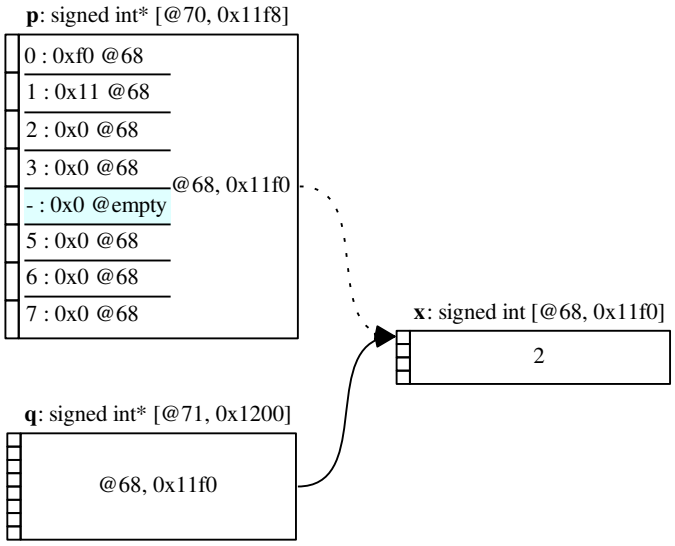
PVI makes user-memcpy legal by regarding each byte (as an integer value) as having the provenance of the original pointer, and the result pointer, being composed of representation bytes of which at least one has that provenance and none have a conflicting provenance, as having the same.

Real `memcpy()` implementations are more complex. The glibc `memcpy()`[glibc 2018] 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 we believe C2x should support it for suitably annotated code. Virtual memory manipulation is outside our scope at present.

Reading pointer values from byte writes In all these provenance semantics, pointer values carry their provenance unchanged, both while manipulated in expressions (e.g. with pointer arithmetic) and when stored or loaded as values of pointer type. In the detailed semantics, memory contains abstract bytes rather than general C language values, and so we record provenance in memory by attaching a provenance to each abstract byte. For pointer values stored by single writes, this will usually be identical in each abstract byte of the value.

However, we also have to define the result of reading a pointer value that has been partially or completely written by (integer) representation-byte writes. In PNVI-*, we use the same semantics as for integer-to-pointer casts, reading the numeric address and reconstructing the associated provenance iff a live storage instance covering that address exists (and, for PNVI-ae and PNVI-ae-udi, if that instance has been exposed). To determine whether a pointer value read is from a single pointer value write (and thus should retain its original provenance when read), or from a combination of representation byte writes and perhaps also a pointer value write (and thus should use the integer-to-pointer cast semantics when read), we also record, in each abstract byte, an optional pointer-byte index (e.g. in 0..7 on an implementation with 8-byte pointer values). Pointer value writes will set these to the consecutive sequence 0, 1, .., 7, while other writes will clear them. For example, the code on the left below sets the fourth byte of `p` to `0`. The memory state on the right, just after the `*q=2`, shows the pointer-byte indices of `p`, one of which has been cleared (shown as -). When the value of `p` is read (e.g. in the `q=p`), the fact that there is not a consecutive sequence 0, 1, .., 7 means that PNVI-* will apply the integer-to-pointer cast semantics, here successfully recovering the provenance `@68` of the storage instance `x`. Then the write of `q` will itself have a consecutive sequence (its pointer-byte indices are therefore suppressed in the diagram). Any non-pointer write overlapping the footprint of `p`, or any pointer write that overlaps that footprint but does not cover it all, would interrupt the consecutive sequence of indices.

```
int x=1;
int main() {
  int *p = &x;
  if (*((unsigned char*)&p+4)==0)
    *((unsigned char*)&p+4)=0;
  int *q = p;
  *q=2;
}
```



In PNVI-plain a representation-byte copy of a pointer value thus is subtly different from a copy done at pointer type: the latter retains the original provenance, while the former, when it is loaded, will take on the provenance of whatever storage instance is live (and covers its address) *at load time*.

The conditional in the example is needed to avoid UB: the semantics does not constrain the allocation address of `x`, so there are executions in which byte 4 is not `0`, in which case the read of `p` would have a wild address and the empty provenance, and the write `*q=2` would flag UB.

Pointer provenance for bytewise pointer representation manipulations To examine the possible semantics for pointer representation bytes more closely, especially for PNVI-ae and PNVI-ae-udi, consider the following. As in `provenance_tag_bits_via_uintptr_t_1.c`, it manipulates the low-order bits of a pointer value, but now it does so by manipulating one of its representation bytes (as in `pointer_copy_user_dataflow_direct_bytewise.c`) instead of by casting to `uintptr_t` and back. In PNVI-plain and PVI this will just work, respectively reconstructing the original provenance and tracking it through the (changed and unchanged) integer bytes.

In PNVI-ae and PNVI-ae-udi, we regard the storage instance of `x` as having been exposed by the read of a pointer value (with non-empty provenance in its abstract bytes in memory) at an integer (really, non-pointer) type. Then the last reads of the value of `p`, from a combination of the original `p=&x` write and later integer byte writes, use the same semantics as integer-to-pointer casts, and thus recreate the original provenance.

```
// provenance_tag_bits_via_repr_byte_1.c
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
int x=1;
int main() {
  int *p=&x, *q=&x;
  // read low-order (little endian) representation byte of p
  unsigned char i = *(unsigned char*)&p;
  // check the bottom two bits of an int* are not used
  assert(_Alignof(int) >= 4);
  assert((i & 3u) == 0u);
  // set the low-order bit of the byte
  i = i | 1u;
  // write the representation byte back
  *(unsigned char*)&p = i;
  // [p might be passed around or copied here]
  // clear the low-order bits again
  *(unsigned char*)&p = (*(unsigned char*)&p) & ~((unsigned char)3u);
  // are p and q now equivalent?
  *p = 11;          // does this have defined behaviour?
  _Bool b = (p==q); // is this true?
  printf("x=%i *p=%i (p==q)=%s\n",x,*p,b?"true":"false");
}
```

Copying pointer values via encryption To more clearly delimit what idioms our proposals do and do not allow, consider copying pointers via code that encrypts or compresses a block of multiple pointers together, decrypting or uncompressing later.

In PNVI-plain, it would just work, in the same way as `user_memcpy()`. In PNVI-ae and PNVI-ae-udi, it would work but leave storage instances pointed to by those pointers exposed (irrespective of whether the encryption is done via casts to integer types or by reads of representation bytes), similar to `user_memcpy` and `provenance_tag_bits_via_repr_byte_1.c`.

One might argue that pointer construction via `intptr_t` and back via any value-dependent identity function should be required to work. That would admit these, but defining that notion of "value-dependent" is exactly what is hard in the concurrency thin-air problem [Batty et al. 2015], and we do not believe that it is practical to make compilers respect dependencies in general.

In PVI, this case involves exactly the same combination of distinct-provenance values that (to prohibit inter-object arithmetic, and thereby enable alias analysis) we above regard as having empty-provenance results. As copying pointers in this way is a very rare idiom, one can argue that it is reasonable to require such code to have additional annotations.

Copying pointer values via control flow We also have to ask whether a usable pointer can be constructed via non-dataflow control-flow paths, e.g. if testing equality of an unprovenanced integer value against a valid pointer permits the integer to be used as if it had the same provenance as the pointer. We do not believe that this is relied on in practice. For example, consider exotic versions of `memcpy` that make a control-flow choice on the value of each bit or each byte, reconstructing each with constants in each control-flow branch

```
// pointer_copy_user_ctrlflow_bytewise_abbrev.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; // does this have undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

```
// 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);
}
```

In PNVI-plain these would both work. In PNVI-ae and PNVI-ae-udi they would also work, as the first exposes the storage instance of the copied pointer value by representation-byte reads and the second by a pointer-to-integer cast. In PVI they would give empty-provenance pointer values and hence UB.

Integer comparison and provenance If integer values have associated provenance, as in PVI, one has to ask whether the result of an integer comparison should also be allowed to be provenance dependent (provenance_equality_uintptr_t_global_xy.c). GCC did do so at one point, but it was regarded as a bug and fixed (from 4.7.1 to 4.8). We propose that the numeric results of all operations on integers should be unaffected by the provenances of their arguments. For PNVI-*, this question is moot, as there integer values have no provenance.

Pointer provenance and union type punning Pointer values can also be constructed in C by type punning, e.g. writing a pointer-type union member, reading it as a `uintptr_t` union member, and then casting back to a pointer type. (The example assumes that the object representation of the pointer and the object representation of the result of the cast to integer are identical. This property is not guaranteed by the C standard, but holds for many implementations.)

The ISO standard 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. We propose that these reinterpretations be required to be implementation-defined, and, in PNVI-plain, that the usual integer-to-pointer cast semantics be used at such reads.

For PNVI-ae and PNVI-ae-udi, the same semantics as for representation-byte reads also permits this case: the storage instance is deemed to be exposed by the read of the provenanced representation bytes by the non-pointer-type read. The integer-to-pointer cast then recreates the provenance of `x`.

For PVI, we propose that it be implementation-defined whether the result preserves the original provenance (e.g. where they are the identity).

```
// provenance_union_punning_3_global.c
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
int x=1;
typedef union { uintptr_t ui; int *up; } un;
int main() {
  un u;
  int *p = &x;
  u.up = p;
  uintptr_t i = u.ui;
  int *q = (int*)i;
  *q = 11;  // does this have UB?
  printf("x=%d *p=%d *q=%d\n",x,*p,*q);
  }
  return 0;
}
```

Pointer provenance via IO Consider now pointer provenance flowing via IO, e.g. writing the address of an object to a string, pipe or file and reading it back in. We have 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 (provenance_via_io_percentp_global.c, provenance_via_io_bytewise_global.c, and provenance_via_io_uintptr_t_global.c) The first gives a syntactic indication of a potentially escaping pointer value, while the others (after preprocessing) do not. Somewhat exotic though they are, these idioms are used in practice: in graphics code for serialisation/deserialisation (using `%p`), in xlib (using `SCNuPTR`), and in debuggers.

In the ISO standard, the text for `fprintf` and `scanf` for `%p` says 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"* (again construing the pre-DR260 "compare equal" as implying the result should be usable for access), and the text for `uintptr_t` and the presence of `SCNuPTR` in `inttypes.h` weakly implies the same there.

But then what can compiler alias analyses assume about such a pointer read? In PNVI-plain, this is simple: at `scanf`-time, for the `%p` version, or when a pointer is read from memory written by the other two, we can do a runtime check and potential acquisition of provenance exactly like an integer-to-pointer cast.

In PNVI-ae and PNVI-ae-udi, for the `%p` case we mark the associated storage instance as exposed by the output, and use the same semantics as integer-to-pointer casts on the input. The `uintptr_t` case and representation-byte case also mark the storage instance as exposed, in the normal way for these models.

For PVI, there are several options, none of which seem ideal: we could use a PNVI-like semantics, but that would be stylistically inconsistent with the rest of PVI; or (only for the first) we could restrict that to provenances

that have been output via `%p`), or we could require new programmer annotation, at output and/or input points, to constrain alias analysis.

Pointers from device memory and linking In practice, concrete memory addresses or relationships between them sometimes are determined and relied on by programmers, in implementation-specific ways. Sometimes these are simply concrete absolute addresses which will never alias C stack, heap, or program memory, e.g. those of particular memory-mapped devices in an embedded system. Others are absolute addresses and relative layout of program code and data, usually involving one or more *linking* steps. For example, platforms may lay out certain regions of memory so as to obey particular relationships, e.g. in a commodity operating system where high addresses are used for kernel mappings, initial stack lives immediately below the arguments passed from the operating system, and so on. The details of linking and of platform memory maps are outside the scope of ISO C, but real C code may embody knowledge of them. Such code might be as simple as casting a platform-specified address, represented as an integer literal, to a pointer. It might be more subtle, such as assuming that one object directly follows another in memory—the programmer having established this property at link time (perhaps by a custom linker script). It is necessary to preserve the legitimacy of such C code, so that compilers may not view such memory accesses as undefined behaviour, even with increasing link-time optimisation.

We leave the design of exactly what escape-hatch mechanisms are needed here as an open problem. For memory-mapped devices, one could simply posit implementation-defined ranges of such memory which are guaranteed not to alias C objects. The more general linkage case is more interesting, but well outside current ISO C. The tracking of provenance through embedded assembly is similar.

Pointers from allocator libraries Our semantics special-cases `malloc` and the related functions, by giving their results fresh provenances. This is stylistically consistent with the ISO text, which also special-cases them, but it would be better for C to support a general-purpose annotation, to let both `stdlib` implementations and other libraries return pointers that are treated as having fresh provenance outside (but not inside) their abstraction boundaries.

Compilers already have related annotations, e.g. GCC's `malloc` attribute *"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"* (https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes).

5 IMPLICATIONS OF PROVENANCE SEMANTICS FOR OPTIMISATIONS

In an ideal world, a memory object semantics for C would be consistent with all existing mainstream code usage and compiler behaviour. In practice, we suspect that (absent a precise standard) these have diverged too much for that, making some compromise required. As we have already seen, the PNVI semantics would make some currently observed GCC and ICC behaviour unsound, though at least some key GCC developers already regard that behaviour as a longstanding unfixed bug, due to the lack of integer/pointer type distinctions in RTL. We now consider some other important cases, by example.

Optimisation based on equality tests Both PNVI-* and PVI let `p==q` hold in some cases where `p` and `q` are not interchangeable. As Lee et al. [2018] observe in the LLVM IR context, that may limit optimisations such as GVN (global value numbering) based on pointer equality tests. PVI suffers from the same problem also for integer comparisons, wherever the integers might have been cast from pointers and eventually be cast back. This may be more serious.

Can a function argument alias local variables of the function? In general one would like this to be forbidden, to let optimisation assume its absence. Consider first the example below, where `main()` guesses the address of `f()`'s local variable, passing it in as a pointer, and `f()` checks it before using it for an access. Here we see, for example, GCC -O0 optimising away the **`if`** and the write `*p=7`, even in executions where the `ADDRESS_PFI_1PG` constant is the same as the `printf`'d address of `j`. We believe that compiler behaviour should be permitted, and hence that this program should be deemed to have UB — or, in other words, that code should not normally be allowed to rely on implementation facts about the allocation addresses of C variables.

```
// pointer_from_integer_1pg.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(int *p) {
  int j=5;
  if (p==&j)
    *p=7;
  printf("j=%d &j=%p\n",j,(void*)&j);
}
int main() {
  uintptr_t i = ADDRESS_PFI_1PG;
  int *p = (int*)i;
  f(p);
}
```

The PNVI-* semantics deems this to be UB, because at the point of the `(int*)i` cast the `j` storage instance does not yet exist (let alone, for PNVI-ae and PNVI-ae-udi, having been exposed by having one of its addresses taken and cast to integer), so the cast

gives a pointer with empty provenance; any execution that goes into the **if** would thus flag UB. The PVI semantics flags UB for the simple reason that `j` is created with the empty provenance, and hence `p` inherits that.

Varying to do the cast to **int*** in `f()` instead of `main()`, passing in an integer `i` instead of a pointer, this becomes defined in PNVI-plain, as `j` exists at the point when the abstract machine does the `(int*)i` cast. But in PNVI-ae and PNVI-ae-udi, the storage instance of `j` is not exposed, so the cast to **int*** gives a pointer with empty provenance and the access via it is UB. This example is also UB in PVI.

At present we do not see any strong reason why making this defined would not be acceptable — it amounts to requiring compilers to be conservative for the results of integer-to-pointer casts where they cannot see the source of the integer, which we imagine to be a rare case — but this does not match current O2 or O3 compilation for GCC, Clang, or ICC.

```
// pointer_from_integer_1ig.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  int *p = (int*)i;
  if (p==&j)
    *p=7;
  printf("j=%d &j=%p\n",j,(void*)&j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1IG;
  f(j);
}
```

Allocation-address nondeterminism Note that both of the previous examples take the address of `j` to guard their `*p=7` accesses. Removing the conditional guards gives the left and middle tests below, that one would surely like to forbid:

```
// pointer_from_integer_1p.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(int *p) {
  int j=5;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t i = ADDRESS_PFI_1P;
  int *p = (int*)i;
  f(p);
}
```

```
// pointer_from_integer_1i.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1I;
  f(j);
}
```

```
// pointer_from_integer_1ie.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  uintptr_t k = (uintptr_t)&j;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1I;
  f(j);
}
```

Both are forbidden in PVI for the same reason as before, and the first is forbidden in PNVI-*, again because `j` does not exist at the cast point.

But the second forces us to think about how much allocation-address nondeterminism should be quantified over in the basic definition of undefined behaviour. For evaluation-order and concurrency nondeterminism, one would normally say that if there exists any execution that flags UB, then the program as a whole has UB (for the moment ignoring UB that occurs only on some paths following I/O input, which is another important question that the current ISO text does not address).

This view of UB seems to be unfortunate but inescapable. If one looks just at a single execution, then (at least between input points) we cannot temporally bound the effects of an UB, because compilers can and do re-order code w.r.t. the C abstract machine's sequencing of computation. In other words, UB may be flagged at some specific point in an abstract-machine trace, but its consequences on the observed implementation behaviour might happen much earlier (in practice, perhaps not very much earlier, but we do not have any good way of bounding how much). But then if one execution might have UB, and hence exhibit (in an implementation) arbitrary observable behaviour, then anything the standard might say about any other execution is irrelevant, because it can always be masked by that arbitrary observable behaviour.

Accordingly, our semantics nondeterministically chooses an arbitrary address for each storage instance, subject only to alignment and no-overlap constraints (ultimately one would also need to build in constraints from programmer linking commands). This is equivalent to noting that the ISO standard does not constrain how implementations choose storage instance addresses in any way (subject to alignment and no-overlap), and hence that programmers of standard-conforming code cannot assume anything about those choices. Then in PNVI-plain, the `..._1i.c` example is UB because, even though there is one execution in which the guess is correct, there is

another (in fact many others) in which it is not. In those, the cast to `int*` gives a pointer with empty provenance, so the access flags UB — hence the whole program is UB, as desired. In PNVI-ae and PNVI-ae-udi, the `..._1i.c` example is UB for a different reason: the storage instance of `j` is not exposed before the cast `(int*)i`, and so the result of that cast has empty provenance and the access `*p=7` flags UB, in every execution. However, if `j` is exposed, as in the example on the right, these models still make it UB, now for the same reason as PNVI-plain.

Can a function access local variables of its parent? This too should be forbidden in general. The example on the left below is forbidden by PVI, again for the simple reason that `p` has the empty provenance, and by

```
// pointer_from_integer_2.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f() {
  uintptr_t i=ADDRESS_PFI_2;
  int *p = (int*)i;
  *p=7;
}
int main() {
  int j=5;
  f();
  printf("j=%d\n",j);
}
```

```
// pointer_from_integer_2g.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f() {
  uintptr_t i=ADDRESS_PFI_2G;
  int *p = (int*)i;
  *p=7;
}
int main() {
  int j=5;
  if ((uintptr_t)&j == ADDRESS_PFI_2G)
    f();
  printf("j=%d &j=%p\n",j,(void*)&j);
}
```

PNVI-plain by allocation-address nondeterminism, as there exist abstract-machine executions in which the guessed address is wrong. One cannot guard the access within `f()`, as the address of `j` is not available there. Guarding the call to `f()` with `if ((uintptr_t)&j == ADDRESS_PFI_2)` (`pointer_from_integer_2g.c` on the right above) again makes the example well-defined in PNVI-plain, as the address is correct and `j` exists at the `int*` cast point, but notice again that the guard necessarily involves `&j`. This does not match current Clang at O2 or O3, which print `j=5`.

In PNVI-ae and PNVI-ae-udi, `pointer_from_integer_2.c` is forbidden simply because `j` is never exposed (and if it were, it would be forbidden for the same reason as in PNVI-plain). PNVI-ae and PNVI-ae-udi allow `pointer_from_integer_2g.c`, because the `j` storage instance is exposed by the `(uinptr_t)&j` cast.

The PNVI-address-taken and PNVI-wildcard alternatives A different obvious refinement to PNVI would be to restrict integer-to-pointer casts to recover the provenance only of objects that have had their address taken, recording that in the memory state. PNVI-address-exposed is based on PNVI-address-taken but with the tighter condition that the address must also have been cast to integer.

A rather different model is to make the results of integer-to-pointer casts have a "wildcard" provenance, deferring the check that the address matches a live object from cast-time to access-time. This would make `pointer_from_integer_1pg.c` defined, which is surely not desirable.

Perhaps surprisingly, the PNVI-ae and PNVI-ae-udi variants seem not to make much difference to the allowed tests, because the tests one might write tend to already be UB due to allocation-address nondeterminism, or to already take the address of an object to use it in a guard. These variants do have the conceptual advantage of identifying these UBs without requiring examination of multiple executions, but the disadvantage that whether an address has been taken is a fragile syntactic property, e.g. not preserved by dead code elimination.

The problem with lost address-takens and escapes Our PVI proposal allows computations that erase the numeric value (and hence a concrete view of the "semantic dependencies") of a pointer, but retain provenance. This makes examples like that below [Richard Smith, personal communication], in which the code correctly guesses a storage instance address (which has the empty provenance) and adds that to a zero-valued quantity (with the correct provenance), allowed in PVI. We emphasise that we do not think it especially desirable to allow such examples; this is just a consequence of choosing a straightforward provenance-via-integer semantics that allows the bytewise copying and the bitwise manipulation of pointers above. In other words, it is not clear how it could be forbidden simply in PVI.

However, in implementations some algebraic optimisations may be done before alias analysis, and those optimisations might erase the `&x`, replacing it and all the calculation of `i3` by `0x0` (a similar example would have `i3 = i1-i1`). But then alias analysis would be unable to see that `*q` could access `x`, and so report that it could not, and hence enable subsequent optimisations that are unsound w.r.t. PVI for this case. The basic point is that whether a variable has its address taken or escaped in the source language is not preserved by optimisation. A possible solution, which would need some implementation work for implementations that do track provenance through integers, but perhaps acceptably so, would be to require those initial optimisation passes to record the address-takens involved in computations they erase, so that that could be passed in explicitly to alias analysis. In contrast to the difficulties of preserving dependencies to avoid thin-air concurrency, this does not forbid optimisations that remove dependencies; it merely requires them to describe what they do.

```
// provenance_lost_escape_1.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "charon_address_guesses.h"
int x=1; // assume allocation ID @1, at ADDR_PLE_1
int main() {
  int *p = &x;
  uintptr_t i1 = (intptr_t)p;            // (@1,ADDR_PLE_1)
  uintptr_t i2 = i1 & 0x00000000FFFFFFFF;//
  uintptr_t i3 = i2 & 0xFFFFFFFF00000000;// (@1,0x0)
  uintptr_t i4 = i3 + ADDR_PLE_1;        // (@1,ADDR_PLE_1)
  int *q = (int *)i4;
  printf("Addresses: p=%p\n",(void*)p);
  if (memcmp(&i1, &i4, sizeof(i1)) == 0) {
    *q = 11;  // does this have defined behaviour?
    printf("x=%d *p=%d *q=%d\n",x,*p,*q);
  }
}
```

In PNVI-plain, the example is also allowed, but for a simpler reason that is not affected by such integer optimisation: the object exists at the `int*` cast. Implementations that take a conservative view of all pointers formed from integers would automatically be sound w.r.t. this. At present ICC is not, at O2 or O3.

PNVI-ae and PNVI-ae-udi are more like PVI here: they allow the example, but only because the address of `p` is both taken and cast to an integer type. If these semantics were used for alias analysis in an intermediate language after such optimisation, this would likewise require the optimsation passes to record which addresses have been taken and cast to integer (or otherwise exposed) in eliminated code, to be explicitly passed in to alias analysis.

Should PNVI allow one-past integer-to-pointer casts? For PNVI*, one has to choose whether an integer that is one-past a live object (and not strictly within another) can be cast to a pointer with valid provenance, or whether this should give an empty-provenance pointer value. Lee observes that the latter may be necessary to make some optimisation sound [personal communication], and we imagine that this is not a common idiom in practice, so for PNVI-plain and PNVI-ae we follow the stricter semantics.

PNVI-ae-udi, however, is designed to permit a cast of a one-past pointer to integer and back to recover the original provenance, replacing the integer-to-pointer semantic check that x is properly within the footprint of the storage instance by a check that it is properly within or one-past. That makes the following example allowed in PNVI-ae-udi, while it is forbidden in PNVI-ae and PNVI-plain.

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

The downside of this is that one has to handle pointer-to-integer casts for integer values that are ambiguously both one-past one storage instance and at the start of the next. The PNVI-ae-udi approach to that is to leave the provenance of pointer values resulting from such casts unknown until the first operation (e.g. an access, pointer arithmetic, or pointer relational comparison) that disambiguates them. This makes the following two, each of which uses the result of the cast in one consistent way, well defined:

```
// pointer_from_int_disambiguation_1.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    uintptr_t i = (uintptr_t)p;
    int *r = (int *)i;
    *r=11;  // is this free of UB?
    printf("x=%d y=%d *p=%d *q=%d *r=%d\n",x,y,*p,*q,*r);
  }
}
```

```
// pointer_from_int_disambiguation_2.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    uintptr_t i = (uintptr_t)p;
    int *r = (int *)i;
    r=r-1;  // is this free of UB?
    *r=11;  // and this?
    printf("x=%d y=%d *p=%d *q=%d *r=%d\n",x,y,*p,*q,*r);
  }
}
```

while making the following, which tries to use the result of the cast to access both objects, UB.

```c
// pointer_from_int_disambiguation_3.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    uintptr_t i = (uintptr_t)p;
    int *r = (int *)i;
    *r=11;
    r=r-1;  // is this free of UB?
    *r=12;  // and this?
    printf("x=%d y=%d *p=%d *q=%d *r=%d\n",x,y,*p,*q,*r);
  }
}
```

In this, the `*r=11` will resolve the provenance of the value in one way, making the `r-1` UB.

REFERENCES

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015*. 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

David Chisnall, Justus Matthiesen, Kayvan Memarian, Kyndylan Nienhuis, Peter Sewell, and Robert N. M. Watson. 2016. C memory object and value semantics: the space of de facto and ISO standards. http://www.cl.cam.ac.uk/~pes20/cerberus/notes30.pdf (a revison of ISO SC22 WG14 N2013).

Clive D. W. Feather. 2004. *Indeterminate values and identical representations (DR260)*. Technical Report. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

FSF. 2018. Using the GNU Compiler Collection (GCC) / 4.7 Arrays and pointers. https://gcc.gnu.org/onlinedocs/gcc/Arrays-and-pointers-implementation.html. Accessed 2018-10-22.

Krebbers and Wiedijk. 2012. N1637: Subtleties of the ANSI/ISO C standard. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1637.pdf.

Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph.D. Dissertation. Radboud University Nijmegen.

Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-level Optimizations and Low-level Code with Twin Memory Allocation. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2018, part of SPLASH 2018, Boston, MA, USA, November 4-9, 2018*. ACM.

Kayvan Memarian, Victor Gomes, and Peter Sewell. 2018. n2263: Clarifying Pointer Provenance v4. ISO WG14 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2263.htm.

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *PLDI 2016: 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation (Santa Barbara)*. http://www.cl.cam.ac.uk/users/pes20/cerberus/pldi16.pdf PLDI 2016 Distinguished Paper award.

Kayvan Memarian and Peter Sewell. 2016a. N2090: Clarifying Pointer Provenance (Draft Defect Report or Proposal for C2x). ISO WG14 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2090.htm.

Kayvan Memarian and Peter Sewell. 2016b. What is C in practice? (Cerberus survey v2): Analysis of Responses – with Comments. ISO SC22 WG14 N2015, http://www.cl.cam.ac.uk/~pes20/cerberus/analysis-2016-02-05-anon.txt.

`glibc`. 2018. `memcpy`. https://sourceware.org/git/?p=glibc.git;a=blob;f=string/memcpy.c;hb=HEAD.

WG14 (Ed.). 2018. *Programming languages – C* (ISO/IEC 9899:2018 ed.).