C provenance semantics: detailed semantics (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

1 INTRODUCTION

This note sketches a mathematical version of the provenance-not-via-integer, address-exposed (PNVI-ae) and provenance-not-via-integer, address-exposed, user-disambiguation (PNVI-ae-udi) models, based on:

- the PNVI semantics in the POPL 2019 paper [Memarian et al. 2019], here called PNVI-plain
- the mathematical version by Martin Uecker, 2019-01-18
- the text summary by Jens Gustedt, 2019-01-22
- email discussion on the C Memory Object Model study group mailing list

Changes for PNVI-ae from PNVI-plain are highlighted. Additional changes for PNVI-ae-udi are highlighted.

This should be read together with the two companion notes, one giving a series of example, and another giving detailed diffs to the C standard text.

This version does not support bytewise copy of a pointer to an unexposed object, which will need quite some additional machinery for PNVI-ae and PNVI-ae-udi beyond the base PNVI (where that automatically works). The design of that machinery should ideally be based on the treatment of representation-byte-accessed pointer values by existing compiler alias analyses and optimisations.

2 THE PNVI-AE-UDI, PNVI-AE, PNVI-PLAIN, AND PVI SEMANTICS

The base PVI and PNVI-plain memory object semantics are manually typeset mathematics simplified from the executable-as-test-oracle Cerberus mechanised Lem [Mulligan et al. 2014] source. We have removed most subobject details, function pointers, and some options. Neither the typeset models or the Lem source consider linking, or pointers constructed via I/O (e.g. via `%p` or representation-byte I/O).

The memory object semantics can be combined with a semantics for the thread-local semantics of the rest of C (expressed in Cerberus as a translation from C source to the *Core* intermediate language, together with an operational semantics for Core) to give a complete semantics for a large fragment of sequential C.

For simplicity, we present the model without the ISO semantics that makes all pointers to an object or region indeterminate at the end of its lifetime, assume two's complement representations, assume NULL pointers have address (and representation) 0, and allow NULL pointers to be constructed from any empty-provenance integer zero, not just integer constant expressions.

2.1 The memory object model interface

In Cerberus, the memory object model is factored out from Core with a clean interface, roughly as in [Memarian et al. 2016, Fig. 2]. This provides functions for memory operations:

- allocate_object (for objects with automatic or static storage duration, i.e. global and local variables),
- allocate_region (for the results of `malloc`, `calloc`, and `realloc`, i.e. heap-allocated regions),
- kill (for lifetime end of both kinds of allocation),
- load, and
- store,

and for pointer/integer operations: arithmetic, casts, comparisons, offseting pointers by struct-member offsets, etc. The interface involves types `pointer_value` (p), `integer_value` (x), `floating_value`, and `mem_value` (v), which are abstract as far as Core is concerned. Distinguishing pointer and integer values gives more precise internal types.

In PNVI-ae, PNVI, and PVI, a provenance π is either $@i$ where i is a storage-instance ID, or the *empty* provenance `@empty`. In PNVI-ae-udi a provenance can also be a symbolic storage instance ID ι (iota), initially associated to two storage instance IDs and later resolved to one or the other.

A pointer value can either be `null` or a pair (π, a) of a provenance π and address a. In PNVI*, an integer value is simply a mathematical integer (within the appropriate range for the relevant C type), while in PVI, an integer value is a pair (π, n) of a provenance π and a mathematical integer n.

2019.

Memory values are the storable entities, either a pointer, integer, floating-point, array, struct, or union value, or unspec for unspecified values, each together with their C type.

3 THE MEMORY OBJECT MODEL STATE

In both PVI and PNVI*, a memory state is a pair (A, M). The A is a partial map from storage-instance IDs to either killed or storage-instance metadata $(n, \tau_{\text{opt}}, a, f, k, t)$:

- size n,
- optional C type τ (or none for allocated regions),
- base address a,
- permission flag $f \in$ {readWrite, readOnly},
- kind $k \in$ {object, region}, and
- for PNVI-ae and PNVI-ae-udi, a taint flag $t \in$ {unexposed, exposed}.

In PNVI-ae-udi, A also maps all symbolic storage instance IDs ι, to sets of either one or two (non-symbolic) storage instance IDs. One might also need to record a partial equivalence relation over symbolic storage instance IDs, to cope with the pointer subtraction and relational comparison cases where one learns that two provenances are equal but both remain ambiguous, but that is debatable and not spelt out in this document.

The M is a partial map from addresses to abstract bytes, which are triples of a provenance π, either a byte b or unspec, and an optional integer pointer-byte index j (or none). The last is used in PNVI* to distinguish between loads of pointer values that were written as whole pointer writes vs those that were written byte-wise or in some other way.

3.1 Mappings between abstract values and representation abstract-byte sequences

The M models the memory state in terms of low-level abstract bytes, but store and load take and return the higher-level memory values. We relate the two with functions repr(v), mapping a memory value to a list of abstract bytes, and abst(τ, bs), mapping a list of abstract bytes bs to its interpretation as a memory value with C type τ.

The repr(v) function is defined by induction over the structure of its memory value parameter and returns a list of sizeof(τ) abstract bytes, where τ is the C type of the parameter. The base cases are values with scalar types (integer, floating and pointers) and unspecified values. For an unspecified value of type τ, it returns a list with abstract bytes of the form (@empty, unspec, none). Non-null pointer values are represented with lists of abstract bytes that each have the provenance of the pointer value, the appropriate part of the two's complement encoding of the address, and the $0..\text{sizeof}(\tau) - 1$ index of each byte. Null pointers are represented with lists of abstract bytes of the form (@empty, 0, none). In PVI, integer values are represented similarly to pointer values except that the third component of each abstract byte is none. In PNVI*, integer values are represented by lists of abstract bytes, with each of their first components always the empty provenance, and each of their third components again none. Floating-point values are similar, in all the models, except that the provenance of the abstract bytes is always empty. For array and struct/union values the function is inductively applied to each subvalue and the resulting byte-lists concatenated. The layout of structs and unions follow an implementation-defined ABI, with padding bytes like those of unspecified values.

The abst(τ, bs) function is defined by induction over τ. The base cases are again the scalar types. For these, sizeof(τ) abstract bytes are consumed from bs and a scalar memory value is constructed from their second components: if any abstract byte has an unspec value, an unspecified value is constructed; otherwise, depending on τ, a pointer, integer or floating-point value is constructed using the two's complement or floating-point encoding. For pointers with address 0, the provenance is empty. For non-0 pointer values and integer values, in PVI the provenance is constructed as follows: if at least one abstract byte has non-empty provenance and all others have either the same or empty provenance, that provenance is taken, otherwise the empty provenance is taken. In PNVI*, when constructing a pointer value, if the third components of the bytes all carry the appropriate index, and all have the same provenance (which will be guaranteed if pointer types all have the same size), the provenance of the result is that provenance. Otherwise, the A part of the memory state is examined to find whether a live storage instance exists with a footprint containing the pointer value that is being constructed. If so, in PNVI-plain, its storage instance ID is used for the provenance of the pointer value, otherwise the empty provenance is used. In PNVI-ae and PNVI-ae-udi, when constructing a pointer value, if A has to be examined then, matching the relevant integer-to-pointer cast semantics below, the storage instance must have been exposed, otherwise the result have the empty provenance. In PNVI-ae-udi, if there are two such live storage instances, with IDs i_1 and i_2, the resulting pointer value is given a fresh symbolic storage instance ID ι, and A is updated to map ι to $\{i_1, i_2\}$. This can only happen if the two storage instances are adjacent and the address is one-past the first and at the start of the second. For array/struct types, *abst*() recurses on the progressively shrinking list of abstract bytes.

3.2 Memory operations

The successful semantics of memory operations is expressed as a transition relation between memory states, with transitions labelled by the operation (including its arguments) and return value:

$$(A, M) \xrightarrow{\text{LABEL}} (A', M')$$

For example, the transitions

$$(A, M) \xrightarrow{\text{load}(\tau, p)=v} (A', M')$$

describe the semantics of a load(τ, p) in memory state (A, M), returning value v and with resulting memory state (A', M'). The semantics also defines when each operation flags an out-of-memory (OOM) or undefined behaviour (UB) in a memory state (A, M).

Storage instance creation When a new storage instance is created, either with allocate_region (for the results of `malloc`, `calloc`, and `realloc`, i.e. heap-allocated regions), or with allocate_object (for objects with automatic or static storage duration, i.e. global and local variables), in non-`const` and `const` variants: a fresh storage-instance ID i is chosen; an address a is chosen from newAlloc(A, al, n), defined to be the set of addresses of blocks of size n aligned by al that do not overlap with 0 or any other allocation in A; and the pointer value $p = (@i, a)$ is returned. In all three cases the storage-instance metadata A is updated with a new record for i, and this is initially marked as `unexposed`. In the allocate_object case the size n of the allocation is the representation size of the C type τ. In the allocate_region$(al, \tau, \texttt{readOnly}(v))$ case, the last of the three rules, the memory M is updated to contain the representation of v at the addresses $a..a + \text{sizeof}(\tau) - 1$.

$$\frac{[\text{LABEL: allocate\_region}(al, n) = p] \quad i \notin \text{dom}(A) \quad a \in \text{newAlloc}(A, al, n) \quad p = (@i, a)}{A, M \to A[i \mapsto (n, \texttt{none}, a, \texttt{readWrite}, \texttt{region}, \texttt{unexposed})], M}$$

$$\frac{[\text{LABEL: allocate\_object}(al, \tau, \texttt{readWrite}) = p] \quad i \notin \text{dom}(A) \quad a \in \text{newAlloc}(A, al, n) \quad n = \text{sizeof}(\tau) \quad p = (@i, a)}{A, M \to A(i \mapsto (n, \tau, a, \texttt{readWrite}, \texttt{object}, \texttt{unexposed})), M}$$

$$\frac{[\text{LABEL: allocate\_object}(al, \tau, \texttt{readOnly}(v)) = p] \quad i \notin \text{dom}(A) \quad a \in \text{newAlloc}(A, al, n) \quad n = \text{sizeof}(\tau) \quad p = (@i, a)}{A, M \to A(i \mapsto (n, \tau, a, \texttt{readOnly}, \texttt{object}, \texttt{unexposed})), M([a..a + n - 1] \mapsto \text{repr}(v))}$$

Storage instance lifetime end When the storage instance of a pointer value $(@i, a)$ is killed, either by a `free()` for a heap-allocated region or at the end of lifetime of an object with automatic storage duration, the storage-instance metadata A of storage instance i is updated to record that i has been killed.

$$\frac{[\text{LABEL: kill}(p, k)] \quad p = (@i, a) \quad k = k' \quad A(i) = (n, \_, a, f, k', \_)}{A, M \to A(i \mapsto \texttt{killed}), M}$$

Load To load a value v of type τ from a pointer value $p = (@i, a)$, there must be a live storage instance for i in A, the footprint of τ at a must be within the footprint of that allocation, and the value v must be the abstract value obtained from the appropriate memory bytes from M.

$$\frac{[\text{LABEL: load}(\tau, p) = v] \quad p = (@i, a) \quad A(i) = (n, \_, a', f, k, \_) \quad [a..a + \text{sizeof}(\tau) - 1] \subseteq [a'..a' + n - 1] \quad v = \text{abst}(\tau, M[a..a + \text{sizeof}(\tau) - 1])}{A, M \to A, M}$$

For PNVI-ae and PNVI-ae-udi, if the recursive-on-τ computation of abst$(\tau, M[a..a + \text{sizeof}(\tau) - 1])$ involves a call of abst at any non-pointer scalar type for a region of M including an abstract byte with non-empty provenance, and the corresponding storage instance is live, it is marked as exposed. This applies e.g. for reads of pointer values via **`char`**$*$ pointers, and for union type punning reads at `uintptr_t` of pointer values.

Store To store a value v of type τ to a pointer value $p = (@i, a)$, there must be a live storage instance for i in A, which must be writable, and the footprint of τ at a must be within the footprint of that allocation. The memory M is updated with the representation bytes of the value v.

$$\frac{[\text{LABEL: store}(\tau, p, v)] \quad p = (@i, a) \quad A(i) = (n, \_, a', \texttt{readWrite}, k, \_) \quad [a..a + \text{sizeof}(\tau) - 1] \subseteq [a'..a' + n - 1]}{A, M \rightarrow A, M([a..a + \text{sizeof}(\tau) - 1] \mapsto \text{repr}(v))}$$

For PNVI-ae-udi, the kill, load, and store rules above must be adapted. If $p = (\iota, a)$ and $A(\iota) = \{i\}$, the other premises and conclusion of the appropriate above rule apply. If $A(\iota) = \{i_1, i_2\}$ and the premises are satisfied for one of the two, say i_j, the rest of the rule applies except that in the final state A is additionally updated to map ι to $\{i_j\}$.

The memory operations flag out-of-memory (OOM) and undefined behaviour (UB) as follows:

allocate_region(al, n) / allocate_object(al, τ, `readwrite`) / allocate_object(al, τ, `readOnly`(v)):	
OOM out of memory	if newAlloc(A, al, n) = {} or newAlloc(A, al, sizeof(τ)) = {}
load(τ, p) / store(τ, p, v) / kill(p):	
UB null pointer	if p = `null`
UB empty provenance	if $p = (@\texttt{empty}, a)$
UB killed provenance	if $p = (@i, a)$ and $A(i) = \texttt{killed}$
load(τ, p) / store(τ, p, v):	
UB out of bounds	if $p = (@i, a)$, $A(i) = (n, \_, a', f, k, \_)$, and $[a..a + \text{sizeof}(\tau) - 1] \not\subseteq [a'..a' + n - 1]$
store(τ, p, v):	
UB read-only	if $p = (@i, a)$ and $A(i) = (n, \_, a', \texttt{readOnly}, k, \_)$
kill(p):	
UB non-alloc-address	if $p = (@i, a)$, $A(i) = (n, \_, a', f, k, \_)$, and $a \neq a'$

For PNVI-ae-udi, the rules above must be adapted. In the case where $p = (\iota, a)$ and $A(\iota) = \{i\}$, the semantics is exactly as for $p = (i, a)$, while if $A(\iota) = \{i_1, i_2\}$, one has UB only if the conditions above apply to both i_1 and i_2.

3.3 Pointer / Integer operations

Pointer subtraction Pointers $p = (@i, a)$ and $p' = (@i', a')$ can be subtracted if they have the same provenance ($i = i'$), there is a live storage instance for i in A, and both a and a' are within or one-past the footprint of that allocation (in ISO C the last will always hold, otherwise UB would have been flagged in earlier pointer arithmetic). The result is the numerical difference $a - a'$ divided by sizeof(dearray(τ))), where dearray(τ) returns τ if it is not an array type, and otherwise returns its element type. This rule is stated for PNVI and PNVI-ae, returning pure integer. For PVI, diff_ptrval constructs the same integer but with `@empty` provenance. For PNVI-ae-udi, because subtraction of pointers with different provenance should be UB, if one of the two pointers has a symbolic storage instance ID ι, mapped by A to $\{i_1, i_2\}$, while the other either has a provenance $@i'$ or an ι' mapped to a singleton $\{i'\}$, then i' must be either i_1 or i_2, and ι is resolved to that in the A of the final state. Otherwise UB. If both pointers are ambiguous, say mapped to $\{i_1, i_2\}$ and $\{i'_1, i'_2\}$, then if those two sets share a single element which satisfies the other rule preconditions, both symbolic storage instance IDs are resolved to that. Otherwise UB. If the sets have two pairs of elements that satisfy the other conditions (which we believe can only happen if the addresses are equal), then subtraction is permitted but the symbolic storage instance IDs are left unresolved. Otherwise UB. For example, suppose `p` and `q` have been produced by separate casts from an integer which is ambiguously one-past one allocation and at the start of another. Then after `p-q` or `p<q` we know they must have been the same provenance, but we still don't know which. (Alternatively, we could change the semantics to record an identity relation over symbolic storage instance IDs, and additional modifications to the rules below beyond what is in this draft, but that seems to be unwarranted complexity).

$$\frac{[\text{LABEL: diff\_ptrval}(\tau, p, p') = x] \quad p = (@i, a) \quad p' = (@i', a') \quad i = i' \quad A(i) = (n, \_, \hat{a}, f, k, \_) \quad x = (a - a')/\text{sizeof}(\text{dearray}(\tau)) \quad a \in [\hat{a}..\hat{a} + n] \quad a' \in [\hat{a}..\hat{a} + n]}{A, M \rightarrow A, M}$$

Pointer relational comparison Pointers $p = (@i, a)$ and $p' = (@i', a')$ can be compared with a relational operator (<, <=, etc.) if they have the same provenance ($i = i'$). The result is the boolean result of the mathematical comparison of a and a'. For PNVI-ae-udi, this has to be adapted in much the same way as the pointer subtraction rule above.

$$\frac{[\text{LABEL: rel\_op\_ptrval}(p, p', op) = b] \quad p = (@i, a) \quad p' = (@i', a') \quad i = i' \quad b = op(a, a') \quad op \in \{\leq, <, >, \geq\}}{A, M \rightarrow A, M}$$

Pointer equality comparison Pointers p and p' can always be compared with an equality operator (`=`, `!=`). The result is true if they are either both null or both non-null and have the same provenance and address; nondeterministically either $a = a'$ or `false` if they are both non-null and have different provenances; and false otherwise. For PNVI-ae-udi, because equality comparison is permitted (without UB) irrespective of the provenances of the pointers, if the two pointers both have determined single provenances after looking up any symbolic IDs in A, this should give `true`, otherwise the middle (nondeterministic) clause should apply. The final A should not resolve any symbolic IDs.

$$\frac{[\textsc{label}\colon \text{eq\_op\_ptrval}(p, p') = b] \quad \begin{cases} b = \texttt{true} & \text{if } p = p' \\ b \in \{(a = a'), \texttt{false}\} & \text{if } p = (\pi, a),\ p' = (\pi', a'), \text{ and } \pi \neq \pi' \\ b = \texttt{false} & \text{otherwise} \end{cases}}{A, M \to A, M}$$

Note that the above nondeterminism appears to be necessary to admit the observable behaviour of current compilers, but a simpler provenance-oblivious semantics is arguably desirable:

$$\frac{[\textsc{label}\colon \text{eq\_op\_ptrval}(p, p') = b] \quad \begin{cases} b = \texttt{true} & \text{if } p = p' = \texttt{null} \\ b = \texttt{true} & \text{if } p = (\pi, a),\ p' = (\pi', a'), \text{ and } a = a' \\ b = \texttt{false} & \text{otherwise} \end{cases}}{A, M \to A, M}$$

Pointer array offset Given a pointer p at C type τ, the result of offsetting p by integer x (either by array indexing or explicit pointer/integer addition) is as follows, where $x = n$ in PNVI*, or $x = (\pi', n)$ in PVI. For the operation to succeed, p must be some non-null $(@i, a)$. Then there must be a live storage instance for i, and the numeric result of the addition of $a + n * \text{sizeof}(\tau)$ must be within or one-past the footprint of that storage instance. Otherwise the operation flags UB. For PNVI-ae-udi, if p is ambiguous (i.e., $p = (\iota, a)$ and $A(\iota) = \{i_1, i_2\}$ then if x is non-zero this should only be defined behaviour for (at most) one of the two, and then ι should be resolved to that one in the final state. If $x = 0$ it does not resolve the ambiguity.

$$\text{iso\_array\_offset\_ptrval}(A, p, \tau, x) = \begin{cases} (@i, a') & \text{if } p = (@i, a) \text{ and } a' = a + n * \text{sizeof}(\tau) \text{ and } A(i) = (n'', \_, a'', \_, \_, \_) \text{ and } a' \in [a''..a'' + n''] \\ \text{UB: out of bounds} & \text{if all except the last conjunct above hold} \\ \text{UB: empty prov} & \text{if } p = (@\texttt{empty}, a) \\ \text{UB: killed prov} & \text{if } p = (@i, a) \text{ and } A(i) = \texttt{killed} \\ \text{UB: null pointer} & \text{if } p = \texttt{null} \end{cases}$$

Pointer member offset Given a pointer p at C type τ, which is a struct or union type with a member m, if p is some non-null (π, a), the result of offsetting the pointer to member m has the same provenance π and the suitably offset a. If p is null, the result is a pointer with empty provenance and the integer offset of m within τ's representation (this is de facto C behaviour, in the sense that the GCC torture tests rely on it; it does not exactly match ISO C).

For this to be sensible, p should point to the start of an object of type τ, with UB otherwise (ISO C suggests this, writing *"The value is that of the named member of the object to which the first expression points"*). However, without a subobject-aware effective-type semantics, we cannot check that here. We could check the storage instance of p's provenance has a footprint large enough to accommodate an object of type τ starting at a, which would guarantee that the result of the offset is within the storage instance.

$$\text{member\_offset\_ptrval}(p, \tau, m) = \begin{cases} (\pi, a + \text{offsetof\_ival}(\tau, m)), & \text{if } p = (\pi, a); \\ \text{offsetof\_ival}(\tau, m), & \text{if } p = \texttt{null}. \end{cases}$$

Casts (PNVI-plain) In PNVI-plain, a cast of a pointer value p to an integer value (at type τ) just converts null pointers to zero and non-null pointer values to the address a of the pointer, if that is representable in τ, otherwise

flagging UB. The provenance of the pointer is discarded.

$$\text{cast\_ptrval\_to\_ival}(\tau, p) = \begin{cases} 0, & \text{if } p = \texttt{null}; \\ a, & \text{if } p = (\pi, a) \text{ and } a \in \texttt{value\_range}(\tau) \\ \text{UB}, & \text{otherwise} \end{cases}$$

In PNVI-plain, an integer-to-pointer cast of 0 returns the null pointer. For a non-0 integer x, casting to a pointer to τ, if there is a storage instance i in the current memory model state (A, M) for which the address of the pointer would be properly within the footprint of the storage instance, it returns a pointer $(@i, x)$ with the provenance of that storage instance. (The "properly within" prevents the one-past ambiguous case.) If there is no such storage instance, it returns a pointer with empty provenance.

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \_) \text{ and } x \in [a..a + n - 1] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

Casts (PNVI-ae) In PNVI-ae, the result of a cast of a pointer value p to an integer value is exactly as in PNVI-plain. In addition, for a cast of pointer value $p = (@i, a)$ with provenance $@i$, where $A(i) = (n, \tau_{\text{opt}}, a, f, k, t)$ is the storage instance metadata for i, the memory state (A, M) is updated to $(A(i \mapsto (n, \tau_{\text{opt}}, a, f, k, \texttt{exposed})), M)$ to mark the that storage instance as exposed.

In PNVI-ae, an integer-to-pointer cast of 0 returns the null pointer. For a non-0 integer x, casting to a pointer to τ, if there is a storage instance i in the current memory model state (A, M) for which the address of the pointer would be properly within the footprint of the storage instance, and storage instance i is exposed, it returns a pointer $(@i, x)$ with the provenance of that storage instance. If there is no such storage instance, it returns a pointer with empty provenance.

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \texttt{exposed}) \text{ and } x \in [a..a + n - 1] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

Casts (PNVI-ae-udi) In PNVI-ae-udi, a cast of a pointer value p to an integer is just like PNVI-ae.

Unlike PNVI-ae, PNVI-ae-udi permits a cast of a one-past pointer to integer and back to recover the original provenance, replacing the integer-to-pointer check that x is properly within the footprint of the storage instance by a check that it is properly within or one-past:

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \texttt{exposed}) \text{ and } x \in [a..a + n] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

But then a PNVI-ae-udi cast of an integer value to a pointer can create a pointer with ambiguous provenance (as in the definition of repr) : if it could be within or one-past two live storage instances, with IDs i_1 and i_2, and both storage instances have been marked as exposed, the resulting pointer value is given a fresh symbolic storage instance ID ι, and A is updated to map ι to $\{i_1, i_2\}$. This can only happen if the two storage instances are adjacent and the address is one-past the first and at the start of the second.

Casts (PVI)

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = (@\texttt{empty}, 0) \\ (\pi, n), & \text{otherwise, where } x = (\pi, n) \end{cases}$$

$$\text{cast\_ptrval\_to\_ival}(\tau, p) = \begin{cases} (@\texttt{empty}, 0), & \text{if } p = \texttt{null}; \\ (\pi, a), & \text{if } p = (\pi, a) \text{ and } a \in \texttt{value\_range}(\tau) \\ \text{UB}, & \text{otherwise} \end{cases}$$

Integer operations (PVI) In PVI one also has to define the provenance results of all the other operations returning integer values. Below we do so for the basic operations, though this would also be needed for all the

integer-returning library functions. Most would give integers with empty provenance.

$$\begin{array}{rcl}
\pi \oplus \pi' & = & \begin{cases} \pi, & \text{if } \pi = \pi' \text{ or } \pi' = \texttt{@empty}; \\ \pi', & \text{if } \pi = \texttt{@empty}; \\ \texttt{@empty}, & \text{otherwise.} \end{cases} \\
\text{op_ival}(op, (\pi, n), (\pi', m)) & = & (\pi \oplus \pi', op(n, m)), \text{ where } op \in \{+, *, /, \%, \&, |, \wedge\} \\
\text{op_ival}(-, (\pi, n), (\pi', m)) & = & \begin{cases} (\texttt{@empty}, n - m), & \text{if } \pi = @i \text{ and } \pi' = @i', \text{ whether } i = i' \text{ or not}; \\ (@i, n - m), & \text{if } \pi = @i \text{ and } \pi' = \texttt{@empty}; \\ (\texttt{@empty}, n - m), & \text{if } \pi = \texttt{@empty}. \end{cases} \\
\text{eq_ival}((\pi, n), (\pi', m)) & = & (n = m) \\
\text{lt_ival}((\pi, n), (\pi', m)) & = & (n < m) \\
\text{le_ival}((\pi, n), (\pi', m)) & = & (n \leq m)
\end{array}$$

3.4 No-expose annotation

For PNVI-ae and PNVI-ae-udi, to permit implementations, e.g. of `memcpy`-like functions, to operate on representation bytes but without needlessly leaving all the storage instances that were pointed to in those bytes exposed, we envisaged some "`no-expose`" annotation that users could apply to such code. But now it's not so clear how that could work. We can turn off exposure during execution of annotated code easily enough (though Jens points out that this might not be the right thing for code which is passed a function pointer). But if the user-memcpy code copies bytes via a **`char`** `*` pointer, then the resulting abstract types in memory still have empty provenance (because we're not tracking provenance via the intervening integer values), so when a pointer value is read (after the user-memcpy) from the copy, it will still get empty provenance.

3.5 Provenance of other operations

In addition to the operations defined above, some operations are desugared/elaborated to simpler expressions by the Cerberus pipeline. Their PVI results have provenance as follows; their PNVI* results are the same except that there integers have no provenance:

- the result of address-of (`&`) has the provenance of the object associated with the lvalue, for non-function-pointers, or empty for function pointers.
- prefix increment and decrement operators follow the corresponding pointer or integer arithmetic rules.
- the conditional operator has the provenance of the second or third operand as appropriate; simple assignment has the provenance of the expression; compound assignment follows the pointer or integer arithmetic rules; the comma operator has the provenance of the second operand.
- integer unary `+`, unary `-`, and `~` operators preserve the original provenance; logical negation `!` has a value with empty provenance.
- **`sizeof`** and `_Alignof` operators give values with empty provenance.
- bitwise shifts has the provenance of their first operand.
- Jens Gustedt highlights that atomic operations have their own specific provenance properties, not yet discussed here.

3.6 Provenance for other library functions

TBD

REFERENCES

Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C Semantics and Pointer Provenance. In ***POPL 2019**: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3290380 Proc. ACM Program. Lang. 3, POPL, Article 67. Also available as ISO WG14 N2311, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2311.pdf.

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.

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*. 175–188. https://doi.org/10.1145/2628136.2628143