1 Full Semantics

1.1 Syntax

Figure 1 shows syntax of IR with some instructions omitted for brevity.

1.2 Memory

In our semantics, memory Mem is defined as a tuple of a current "time", a partial function from block ids to memory blocks, and a partial function from call ids to call times. We use $M(l)$ to refer to memory block $l$, and $M(cid)$ to refer to the call time of call $cid$. Memory allocation/deallocation (call malloc(), alloca, call free()) increments the time of the memory. The time never decreases during the execution of a program.

The memory has two parameters: First, a partial function $ptrsz$ saying for each address space, if it exists, how large the pointer is. Address space 0 has to exist. Second, the parameter $memtwins$ says how many memory ranges are allocated for each block. The details of this “twin allocation” will be discussed later.

Memory Block. A memory block is defined as a tuple $(t, r, n, a, c, P)$. $t$ is a tag indicating which instruction was used to allocate this block: For memory blocks allocated by alloca, the tag is stack; for malloc it is heap; for global variables, it is global tag; and for functions (i.e., the target of function pointers), it is function.

$r$ is a pair of timestamps defining the lifetime of the block. If $r = (s, e)$, the block is alive in the time range $[s, e)$. When a block is newly allocated, $r$ is assigned $(s, \infty)$ where $s$ is the current time. If the block is freed, $r$ is changed to $(s, e)$ where $e$ is the current time. We say that $l$ is alive, or $alive_M(l)$, if its lifetime has not ended.

$n$ is the size of the block in bytes. $a$ is the alignment of the block in bytes. When a block becomes a concrete block, its integer address ($P(s)$, which we will discuss shortly) must be divisible by $a$.

$c$ is the content of the block, stored as a sequence of $n$ bytes.

$P$ stores, for each address space, the integer addresses of the beginning of the block. These addresses are assigned on allocation. For all address spaces $s$ and twin indices $i$, $P(s)_i + n$ should not exceed the maximal integer address of address space $s$. For example, if address space 1 has size $2^{16}$, then we must have $P(1)_i + n < 2^{16}$. Furthermore, in address space 0, the first and last address (0 and $2^{ptrsz(0)} - 1$) must not be used for any block. For every address space, the first address ($P(s)_0$, or just $P(s)$ for short) is the base address of the block on the physical machine. The remaining addresses (at indices 1, . . . , $memtwins-1$) are the base addresses of the twin blocks. For every block allocated via malloc or alloca, we actually reserve several blocks of the same size in the address space. This lets us prove that it is impossible for others to correctly “guess” the address that the block has been allocated at. The twin blocks’ addresses are not used anywhere in the semantics. However, we demand that all the address ranges covered by the alive blocks of an address space are disjoint: For all address spaces $s$ and all $l_1, l_2, i_1, i_2$, if $(l_1, i_1) \neq (l_2, i_2)$ and $alive_M(l_1)$ and $alive_M(l_2)$, then $M(l_1).P(s_{i_1}, M(l_1).P(s_{i_2}) + P(l_1).n)$ is disjoint from $M(l_2).P(s_{i_1}, M(l_2).P(s_{i_2}) + P(l_2).n)$. Furthermore, the base address of one alive block must not be in the address range covered by another alive block: For all address spaces $s$ and all $l_1, l_2, i_1, i_2$, if $(l_1, i_1) \neq (l_2, i_2)$ and $alive_M(l_1)$ and $alive_M(l_2)$, then $M(l_1).P(s_{i_1}, M(l_2).P(s_{i_2}) + P(l_2).n)$. This second condition is required to handle 0-sized blocks.

convert$(s, i, s')$ is a partial function that maps an integer address $i$ from address space $s$ to $s'$. If there exists $P(s) = i$ and $P(s') = i'$, we have for all offsets $o \leq n$ that convert$(s, i + o, s') = i' + o$ and convert$(s', i', s) = i + o$.

Memory Addresses. There are two kinds of memory addresses (besides poison): logical addresses Log($l, o, s$), and physical addresses Phys($o, s, I, cid$). Both track their address space to be able to detect partial loads of a pointer, and to detect address space crossing on load.

A logical memory address is of the shape Log($l, o, s$), where $l$ is a block id, $o$ is a byte offset from the beginning of the block and $s$ is its address space. An offset $o$ is an inbounds offset of $l$, written inbounds$M(l, o)$, if $o$ is non-negative and not larger than size of the block, i.e., $0 \leq o \leq n$. The offset one-past-the-end is explicitly allowed. Because the last address of address space 0 is never allocated, we know that computing on inbounds addresses can never overflow. The offset is strictly inbounds, written strict_inbounds$M(l, o)$, if
it is in bounds and it does not point beyond the end of the block, i.e., $0 \leq o < n$.

Physical addresses are of the shape $\text{Phy}(p, s, I, cid)$ where $o$ is the physical address, i.e., it is an offset starting at address $0x0$. $s$ is the address space, and $I$ and $cid$ are additional constraints which should be met when the pointer is dereferenced. $I$ is a set of integer addresses which should be in bounds addresses of the dereferencing memory block when the physical pointer is dereferenced. $cid$ is a CallId enforcing that the physical pointer cannot access memory blocks created inside the function call. $I$ and $cid$ are both used for supporting more alias analysis rules. They are not used in pointer comparison, pointer subtraction, and pointer to integer casting, but address space casting may update $I$. For simplicity, we write $\text{Phy}(o, s)$ whenever $I$ is an empty set and $cid$ is None.

To describe $cid$, we first introduce the concept of a call id. A call id is a natural number that is uniquely assigned to each function call. For each function call, the time at which it occurs is maintained in the partial map $\text{CallId} \rightarrow \text{Time}$ that is part of the memory. If a physical pointer is passed to a function call and $cid$ of the physical pointer is either None or a call that has already returned, $cid$ is updated to the call id of the new function call. Otherwise, $cid$ does not change. Inside the function call, even if a physical pointer points to some memory block $I$, dereferencing the physical pointer is UB if the beginning of the lifetime of $I$ is not earlier than call time of $cid$. Escaping the physical pointer (e.g., storing it into a global variable or returning it at the end of the function) does not change $cid$. After the function call is returned, all physical pointers having the call id act as if their $cid$ are None. In other words, even if a pointer with a $cid$ is returned back to its caller, it is no longer restricted in how it can be used.
To actually perform a memory access of size $sz > 0$ through a pointer $p$, it must be dereferencable as some block-offset-pair, written $\text{deref}_M(p, sz, l, o)$. If $p$ is a logical pointer $\text{Log}(l, o, 0)$ and $\text{alive}_M(l)$ and $\text{inbounds}_M(l, o + sz)$, then we have $\text{deref}_M(p, sz, l, o)$. If $p$ is a physical pointer $\text{Phy}(p, 0, l, cid)$, there must be a block $l$ and an offset $o$ such that $M(l).P(0) + 0 = p$ and $\text{alive}_M(l)$ and $\text{inbounds}_M(l, o + sz)$ and moreover, for all $p' \in I$, we must have $\text{inbounds}_M(l, p' - M(l).P(0))$, i.e., all these $p'$ are inbounds of the same block) and if $\text{cid} \neq \text{None}$ and $M(\text{cid}) \neq \text{None}$, then $M(l).b < M(\text{cid})$ (i.e., the block was allocated before the function call identified by $\text{cid}$ began, and that function call is still ongoing). If all these requirements are satisfied, we have $\text{deref}_M(p, sz, l, o)$. Notice that $l$ and $o$ are uniquely determined even for physical pointers due to memory blocks being disjoint.

The NULL pointer of address space $s$ is defined as $\text{Phy}(0, s, 0, \text{None})$. Defining NULL as physical pointer allows folding $\text{inttoptr}(0)$ into NULL ($\text{inttoptr}(x)$ is an instruction that casts integer $x$ to pointer), and replacing $p$ by NULL if $p == \text{NULL}$ holds. Also LLVM can optimize NULL + idx into $\text{inttoptr}(idx)$.

Values. $\llbracket ty \rrbracket$ denotes the set of values of type $ty$. An integer value of type $\text{isz}$ is either a concrete number $i$ within range $0 \leq i < 2^{sz}$, or $\text{poison}$. A pointer value of type $\llbracket ty \rrbracket s$ is defined as either a logical address $\text{Log}(l, o, s)$, a physical address $\text{Phy}(o, s, l, cid)$, or $\text{poison}$. There is no distinction between pointer values of different types ($\text{int32}$, $\text{int64}$, ..), but there is a distinction between pointer values of different address spaces. This is needed to make sure that load punning cannot be used to perform an address space cast, and because the size of a pointer may depend on its address space. The register file Reg maps a name to a type and a value of that type.

Byte and Bit denote the set of values that one byte or bit can hold, respectively. A byte can hold 8 bits. A bit can hold either a value of type $\text{int8}$ (i.e., 0, 1, or $\text{poison}$) or the ith bit of a pointer value $p$. Storing to memory involves converting a value to an array of bits. $\llbracket ty \rrbracket (v)$ is a function that converts value $v$ to bits. Similarly, loading a value from memory involves converting bits to a value. $\llbracket ty \rrbracket (b)$ is a function that converts bits $b$ to a value of type $ty$.

$\text{inttoptr}$

To convert an individual bit, we define a partial function getbit $v i$ that returns ith bit of a value $v$ with base type $bty$. If $v$ is an integer, getbit $v i$ returns ith bit of the integer $v$. If the integer $v$ is $\text{poison}$, all its bits are $\text{poison}$; otherwise all bits are either 0 or 1. If $v$ is a pointer, getbit $v i$ returns either $\text{poison}$ if $v$ is $\text{poison}$ or a pair $(p, i)$ which is an element of AddrBit denoting the ith bit of a non-poison pointer $p$. We

\footnote{This matches LLVM’s plans to move to a ptr type: https://lists.llvm.org/pipermail/llvm-dev/2015-February/081822.html.}
define all bits of poison to be poison, regardless of its type.

getbit ∈ [bty] → N → Bit

getbit n i = platform dependent

where n ∈ [siz], 0 ≤ i < sz

getbit p i = (p, i)

where p ∈ [ty + addrspace(s)],

0 ≤ i < ptrsz(s)

Further details of these definitions will be given together with the load/store instructions.

1.3 Instructions

In this subsection, we introduce the semantics of the IR instructions. Instruction i updates the register file R ∈ Reg and the memory M ∈ Mem, denoted R, M ← R′, M′. Note that program text and program counter are omitted in state because every operation explained in this section increments the program counter by one and does not change the program text. The aforementioned global map from call id to call time is omitted as well. For semantics of branch instructions, we follow earlier work [? ].

The value [op]R of an operand op in register file R is given by:

[p]R = R(r) // register
[C]R = C // constant

[poison]R = poison // poison

We propose to add one new instruction to LLVM: psub. This instruction takes two pointers p1, p2 and returns p1 − p2 as an integer. Currently LLVM uses ’ sub (ptrtoint p1), (ptrtoint p2) ’ to subtract two pointers. This is already correct in this semantics, but using psub can improve compiler’s optimization power.

Integer→Pointer Casting. We formally define the semantics of ptrtoint and inttoptr instructions. Figure 3 shows semantics of ptrtoint and inttoptr. There are two auxiliary functions cast2intM(l, o, s) and cast2ptr(o, s):

cast2intM(l, o, s) = (P(s) + o)%ptrsz(s)

where

M(l) = (t, r, n, a, c, p)

cast2ptr(o, s) = Phy(o, s, 0, None)

Casting from a logical pointer to integer, or cast2intM(l, o, s), yields an integer P(s) + o based on block l. If P(s) + o overflows the size of the address space, it wraps around to 2’s complement. (This can only happen if the pointer is not inbounds.) The semantics of ptrtoint is easily represented by cast2int. ’ptrtoint Log(l, o, s)’ computes cast2intM(l, o, s) and returns it. ’ptrtoint Phy(o, s, l, cid)’ simply returns o. If the size of the destination type siz is larger than the size of the source type, it is zero-extended. If it is smaller than it, most significant bits are truncated.

Casting from an integer to a pointer, or cast2ptr(o, s), returns a physical pointer with no provenance information. One option here is to add o to l upon casting, making sure that if this pointer is ever dereferenced, it is still in the block that it started out in. However, that would invalidate replacing p by inttoptr(ptrtoint(p)).

inttoptr is o to ty * addrspace(s)’ is equivalent to cast2ptr(o, s) if the size of the source type siz is equivalent to the size of the destination type ty * addrspace(s). If the size of the source type is larger, high bits of o are truncated. If the size of the source type is smaller, o is zero-extended.

In this semantics, inttoptr and ptrtoint instructions are allowed to freely move around, be removed, or be introduced.

Address-Space Casting. LLVM IR is a general-purpose intermediate language, and it can be used to compile programs for GPUs as well. The address space of a GPU typically disjoint from the one of the CPU, and moreover, many have multiple address spaces themselves. A programmer can choose which memory to use for allocation. To handle that, LLVM IR tracks address space of a pointer it its type. A pointer in one address space can be cast to a pointer in another address space using addrspacecast.

Figure 4 shows formal semantics of addrspacecast. If the given pointer is a physical pointer Phy(o, s, Icid), the instruction translates both the offset o and the inbounds offsets I using convert(s, _, s’). If the result of convert is not defined (the function is partial, after all), the result is poison. If converting any offset in I fails, it is poison as well. addrspacecast is a capturing operation, as is ptrtoint.

If the given pointer is a logical pointer Log(l, o, s), its block id is maintained, and the new offset is calculated as follows. First, the pointer is cast to integer using cast2intM. Next, the integer is converted into the corresponding integer address in s’ using convert. Finally, offset is calculated by getting relative offset from l in s’.

The reason why the calculation of offset is complex is due to a possible overflow. Let’s assume that the size of address space 1 is 4, i.e., there are 16 bytes, and the size of address space 2 is 5, i.e., there are 32 bytes. Also, let’s assume that convert(1, x, 2) = x (identity function). Finally, let’s assume that the beginning of a block l is 8 in both address spaces (it must be the same because convert is the identity function). Then, pointer p = Log(l, 15, 1) will have integer address (8 + 15)%16 = 7, but addrspacecast p to 2 = Log(l, 15, 2) will return (8 + 15)%32 = 23. This breaks our property that any pointer p can be replaced with inttoptr(ptrtoint(p)).

This enables replacing p with NULL if p == NULL is given. Also we can make optimizers like GVN insert this if needed, although we didn’t utilize this replacement in our prototypes.
Case 2
malloc(4)

Figure 3. Semantics of `ptroptint`, `inttoptr`

\[(i = \"r = \text{inttoptr} \ ty \ast \text{addrspace}(s) \ op \ to \ ty_2 + \text{addrspace}(s_2)\")\]

\[\text{Log}(l, o, s) = \{ op \}_R \text{ cast2int}_M(l, o, s) = j\]

\[R, M \rightarrow R[r \rightarrow j \% 2^{sz}]. M\]

\[\text{Phy}(o, s, I, cid) = \{ op \}_R \text{ poisons } \{ op \}_R\]

\[R, M \rightarrow R[r \rightarrow \text{poison}]. M\]

\[(i = \"r = \text{addrspacecast} \ ty_1 \ast \text{addrspace}(s_1) \ op \ to \ ty_2 + \text{addrspace}(s_2)\")\]

\[\text{Log}(l, o, s) = \{ op \}_R \quad o' = \text{convert}(s_1, o, s_2) \quad I' = \{ \text{convert}(s_1, i, s_2) \mid i \in I \}\]

\[R, M \rightarrow R[r \rightarrow \text{Phy}(o', s_2, I', cid)]. M\]

\[(i = \"r = \text{addrspacecast} \ ty_1 \ast \text{addrspace}(s_1) \ op \ to \ ty_2 + \text{addrspace}(s_2)\")\]

\[\text{Log}(l, o, s) = \{ op \}_R \quad o' = \left(\text{convert}(s_1, \text{cast2int}_M(l, o, s_1), s_2) - \text{cast2int}_M(l, 0, s_2)\right) \% 2^{ptrsz(s_2)}\]

\[R, M \rightarrow R[r \rightarrow \text{Log}(l, o', s_2)]. M\]

Figure 4. Semantics of `addrspacecast`

Pointers Comparison. Now we define the semantics of pointer comparison. `icmp` compares two pointers in the same address space, and returns a value of type `i1`. In this semantics, `icmp` can freely move across any other operations like `call malloc()`, `free()`. Also, `icmp` on two logical pointers does not capture their integer addresses. In address space `0`, `icmp` `NULL, p` also does not escape `p` if `p` is inbound address, because integer address of `p` is always positive if it is inbound.

The definition of `icmp eq p1 p2` is as follows.

1. If `p1, p2` are both logical addresses, `Log(l_1, o_1, s)` and `Log(l_2, o_2, s)`, we first check whether their block ids are same, e.g., `l_1 = l_2`. If they are same, the comparison is equivalent to `o_1 = o_2`. If `l_1 \neq l_2`, the comparison can evaluate to `false`. However, there are also some cases where comparison is non-deterministic, i.e., it can evaluate to either `false` or `true`. This is the case if either one of the offsets is not strictly inbounds, i.e., `\neg (0 \leq o_1 < n_1) \lor \neg (0 \leq o_2 < n_2)`, or if the lifetimes of the two blocks do not overlap. In other words, the result is only guaranteed to be `false` if both offsets are strictly inbounds and the lifetimes overlap. These are sufficient conditions to ensure that the bit representations of the two pointers on the hardware differ.

This figure visualizes two cases where (1) lifetimes overlap, and (2) lifetimes do not overlap. If lifetimes overlap, the two blocks never have overlapping memory addresses. Therefore comparison on pointers from each of these blocks yields `false` if the offsets are strictly inbounds. Notably, the result of the comparison does not depend on whether `p` or `q` has already been freed. However, if their lifetimes are disjoint, `p` and `q` may overlap their addresses, and comparison on two pointers is nondeterministic value. Note that whether the two blocks overlap or not is determined when the second malloc is called. The result of the comparison does not depend on whether a block is still allocated, so `icmp eq` is allowed to freely move across `free`

2. If `p1, p2` are both physical addresses, `Phy(o_1, s, l_1, cid_1)`, `Phy(o_2, s, l_2, cid_2)`: the result is equivalent to `o_1 = o_2`.

3. If `p1 = Phy(o_1, s, l_1, cid_1)` and `p2 = Log(l_2, o_2, s)` or vice versa, the result is equivalent to `o_1 = cast2int_M(l_2, o_2, s)`.

The rules for comparing logical pointers allow `p+n == q` to be folded into `'false'`, which is an optimization currently

\[\text{This is the case of http://lists.llvm.org/pipermail/llvm-dev/2017-April/112009.html.}\]
performed by gcc/llvm. Figure 5 shows the formal rules for
‘icmp eq’.

* icmp ne p1, p2 * is simply defined as a negation of
* icmp eq p1, p2 *. This enables free conversion between p ==
* q and ! (p != q).

* icmp ule p1, p2 * (or p <= q) is defined as follows.
1. If p1 = Log(l, o1, s) and p2 = Log(l, o2, s), we check
whether the offsets o1 and o2 are inbounds. This con-
dition is needed because l + o1 or l + o2 can overlap
at runtime. If this is the case, the result is o1 <= o2.
2. Otherwise, we allow non-deterministic choice.

* icmp ult p1, p2 * (or p < q) is defined as follows.
1. If p1 = Log(l1, o1, s), p2 = Log(l2, o2, s) and l1 != l2,
the result is nondeterministic choice.
2. If p1 = Phy(o1, s, l1, cid1) and p2 = Phy(o2, s, l2, cid2),
the result is o1 <= o2.
3. If p1 = Phy(o1, s, l1, cid1) and p2 = Log(l2, o2, s), it is
o1 <= o2.
4. If p1 = Phy(o1, s, l1, cid1) and p2 = Phy(o2, s, l2, cid2),
the result is o1 <= o2.

Figure 6 shows the rules for ‘icmp ule’. The semantics of
‘icmp ult’ is defined in a similar way to ‘icmp ule’.

For all equality/inequality comparisons, the result is
poison if one or more operands are poison.

\[( i = "r = icmp op ty * addrspace(s) op1 op2") \\
\text{poison} = [op1]_{R_k} \lor \text{poison} = [op2]_{R_k} \\
R, M \rightarrow R[r \rightarrow \text{poison}], M \]

Memory Allocation / Deallocation. Memory allocating op-
erations create a new memory block and pick integer base ad-
dresses P(s) for each address space they operate in, including
some number of twin blocks. Our semantics is parameterized
in how many twin blocks are allocated. In address space 0,
the base addresses P(0), may not be 0 and the last strictly
inbounds address P(0) + n - 1 may not be 2^ptrsz(0) - 1, i.e.,
the first and last base of address space 0 are not allocatable.
In particular, this means that P(0) + n cannot overflow. The
standard operations *alloca, call malloc* only work in address
space 0. In order to maintain the memory invariants, all the
base addresses must be divisible by the alignment. Further-
more, P(s), n must not exceed the size of s. Finally, for all
s, all the |P(s), P(s), n| must be mutually disjoint and
disjoint from all existing alive blocks’ ranges. Figure 7 shows
semantics of *alloca* and *call malloc*.

*alloca* creates a new logical block of the size of ty. Every
bit of value of a new block is initialized with *poison*.
The tag of a new block is stack meaning that the block cannot be
freed by *free*. It is freed when a function returns.

*malloc* creates a new logical block of len bytes, or returns
NULL nondeterministically. If len is *poison*, it is UB. Similar
to *alloca*, every bit is initialized with *poison*. The alignment
of blocks created by *malloc* is determined by the ABI (hence
platform dependent), and it corresponds to the maximum
alignment required for any type. Note that for aggregate

Types like struct type / array type only a single block is allo-
cated and the block contains all members of the aggregated
type. *malloc(0) returns NULL.*

The pointers returned by *alloca* and *malloc* all have ad-
dress space 0.

Figure 8 shows semantics of *free*. *free* invalidates the
memory block that ptr refers to by updating its time range.
Calling *free* on NULL pointer is a NOP. Otherwise, calling
*free* on a pointer p requires deref_p (p, 1, l, 0) for some block l;
otherwise, it is UB. Notice that the offset must be 0. Moreover,
deref only ever holds for pointers of address space 0.

These three operations *alloca, malloc, free* all increment
the time of the memory \( t_{cur} \).

Address Calculation. The getelementptr instruction is
used to get the address of a subelement of an aggregate
data structure. **getelementptr** does not check whether the
block is alive or not. This enables **getelementptr** to freely
move across *free* calls.

**getelementptr** on a logical pointer yields a logical pointer
with shifted offset. If the operand is p = Log(l, o, s),
getelementptr p, i returns Log(l, (o + i’)%2^ptrsz(s), s) where
i’ is i multiplied by the size of its element type. The
**getelementptr** instruction may have the inbounds tag,
which imposes further requirements on the operands and
helps LLVM do further alias analysis. Concretely, it demands
that both the base pointer and the returned pointer are
inbounds of the block. **getelementptr inbounds** p, i returns
poison if that is not the case.

**getelementptr** on a physical pointer yields a physical
pointer with shifted offset. If the base pointer is p = Phy(o, s, l, cid), then **getelementptr** p, i simply returns
Phy(o + i’%2^ptrsz(s), s, l, cid) where i’ is i multiplied by
size of its element type. This operation does not affect
l and cid, which enables optimizing **getelementptr** p, 0
to p. In the **inbounds** variant, the returned pointer has
an updated inbounds set i’ = i \cup \{(o + i’)%2^ptrsz(s)\}.
This allows for further alias analysis even on physical
pointers. Also, tracking inbounds addresses and checking
them later instead of returning poison instantly allows
ordering of **getelementptr inbounds** and memory allocating/deallocating operations. **getelementptr inbounds** on
a physical pointer is poison if the added offset overflows.

The formal semantics of *getelementptr* is given in Fig-
ure 9.

If the base pointer points to a nested aggregate value,
the getelementptr instruction may have multiple in-
dexes as its operands. In this case, it is allowed for
getelementptr inbounds to point past the range of a sub-
type. For example, ‘int a[5][5]; int* t=&a[0][7];’ is
translated into

\[ \%a = alloca [5 x [5 x i32]], align 16 \]
\[ \%t = getelementptr inbounds [5 x [5 x i32]]* \%a, i64 0, i64 0, i64 7 \]
In this case, %t is not poison.

**Pointer Subtraction.** We define a new instruction psub(ptr1, ptr2) that calculates ptr1 − ptr2. This operation
\[(t = \text{"call void free(i8* op)"})
\]
\[
\log(0, 0, 0) \equiv \|\text{op}\|_R
\]
\[
R, M \hookrightarrow R, M
\]

\[(t = \text{"call void free(i8* op)"})
\]
\[
\log(0, 0, 0) \equiv \|\text{op}\|_R
\]
\[
M(l) = (\text{heap}(b, \infty), n, a, c, P)
\]
\[
R, (r_{\text{cur}}, f, C) \hookrightarrow R, (r_{\text{cur}} + 1, f[l \mapsto (\text{heap}(b, r_{\text{cur}}), n, a, c, P)], C)
\]

\[(t = \text{"call void free(i8* op)"})
\]
\[
\text{Phy}(0, 0, 1, \text{cid}) = \|\text{op}\|_R
\]
\[
M(l) = (\text{heap}(b, \infty), n, a, c, P)
\]
\[
\text{call2int}(l, 0) = o \quad b < \text{calltime}(\text{cid})
\]
\[
R, (r_{\text{cur}}, f, C) \hookrightarrow R, (r_{\text{cur}} + 1, f[l \mapsto (\text{heap}(b, r_{\text{cur}}), n, a, c, P)], C)
\]

Figure 8. Semantics of free() (cases not mentioned here all raise UB)

\[(i = r = \text{getelementptr ty in addr space(s) op}_1 \text{ isz op}_2)
\]
\[
\text{getelementptr-logical}
\]
\[
\log(l, o, s) = \|\text{op}_1\|_R
\]
\[
i = \|\text{op}_2\|_R
\]
\[
R, M \hookrightarrow R[r \mapsto \log(l, (o + \text{bitwidth}(ty) + i) \% \text{ptrsz}(s), s)], M
\]

\[(i = r = \text{getelementptr ty in addr space(s) op}_1 \text{ isz op}_2)
\]
\[
\text{getelementptr-physical}
\]
\[
\text{Phy}(o, s, l, \text{cid}) = \|\text{op}_1\|_R
\]
\[
i = \|\text{op}_2\|_R
\]
\[
R, M \hookrightarrow R[r \mapsto \text{Phy}(o + \text{bitwidth}(ty) + i) \% \text{ptrsz}(s), s, l, \text{cid}], M
\]

\[(i = r = \text{getelementptr ty inbounds ty in addr space(s) op}_1 \text{ isz op}_2)
\]
\[
\text{getelementptr-inbounds-logical}
\]
\[
\log(l, o, s) = \|\text{op}_1\|_R
\]
\[
\text{inbounds}_M(l, o)
\]
\[
i = \|\text{op}_2\|_R
\]
\[
o' = (o + \text{bitwidth}(ty) + i) \% \text{ptrsz}(s)
\]
\[
R, M \hookrightarrow R[r \mapsto l \mapsto \text{Phy}(o', s, l \cup \{o, o'\}, \text{cid})], M
\]

Figure 9. Semantics of getelementptr (cases not mentioned here are all poison)

does not read or write memory. ptr1 and ptr2 should have same address space by type checking. Alias analysis can treat psub specially so it does not consider the addresses of its operands escaped in some cases. Originally clang used ptrtoint and integer arithmetic to emit pointer subtraction; with psub we enable a more precise alias analysis.

1. If ptr1, ptr2 are both logical addresses, they must point to the same logical block; otherwise the result is poison.
2. If ptr1, ptr2 are both physical addresses, say ptr1 = Phy(o1, s, l1, cid1), ptr2 = Phy(o2, s, l2, cid2), the result is (o1 − o2) \% \text{ptrsz}(s).
3. If ptr1 is a logical address and ptr2 is a physical address or vice versa, say ptr1 = Log(l1, o1, s), ptr2 = Phy(o2, s, l2, cid2), the result is equivalent to (\text{cast2int}_M(l_1, o_1, s) − o_2) \% \text{ptrsz}(s).

Figure 10 shows formal semantics of psub. The transformation (p1 = p2) \iff 0 \mapsto p1 \mapsto p2 is valid, but its inverse is not. Similarly, (p1 \mapsto p2) \iff \theta \mapsto p1 \mapsto p2 is valid, but its inverse is not. This instruction is implemented as @llvm.psub intrinsic function in our prototype.

Load and Store. As mentioned in the beginning of this section, we define two meta operations to support conversion between values of types and low-level bit representation.

\[
\text{ty}|_l \quad \in \quad \text{Bit}^{\text{bitwidth}(ty)}
\]

\[
\text{ty}|_l \quad \in \quad \text{Bit}^{\text{bitwidth}(ty)} \rightarrow \text{ty}
\]

For base types, ty|_l transforms poison into the bitvector of all poison bits, and defined values into their standard
\[(i = \text{"r = psub ty * addrspace(s) op1, op2"}) \quad (i = \text{"r = psub ty * addrspace(s) op1, op2"})\]

\[
\begin{align*}
\text{Log}(l_1, o_1, s) &= \text{\text{\texttt{op1}}}_{\text{\texttt{R}}} \\
\text{Log}(l_2, o_2, s) &= \text{\text{\texttt{op2}}}_{\text{\texttt{R}}} \\
R, M &\leftarrow R[r \mapsto (a_1 - o_2)\%2^{\text{\texttt{ptrsz(s)}}}], M
\end{align*}
\]

\[
\begin{align*}
\text{Log}(l_1, o_1, s) &= \text{\text{\texttt{op1}}}_{\text{\texttt{R}}} \\
\text{Log}(l_2, o_2, s) &= \text{\text{\texttt{op2}}}_{\text{\texttt{R}}} \\
R, M &\leftarrow R[r \mapsto \text{\texttt{poison}}], M
\end{align*}
\]

\[(i = \text{"r = psub ty * addrspace(s) op1, op2"})\]

\[
\begin{align*}
\text{Log}(l_1, o_1, s) &= \text{\text{\texttt{op1}}}_{\text{\texttt{R}}} \\
\text{Log}(l_2, o_2, s) &= \text{\text{\texttt{op2}}}_{\text{\texttt{R}}} \\
R, M &\leftarrow R[r \mapsto (a_1 - o_2)\%2^{\text{\texttt{ptrsz(s)}}}], M
\end{align*}
\]

\[
\begin{align*}
\text{Log}(l_1, o_1, s) &= \text{\text{\texttt{op1}}}_{\text{\texttt{R}}} \\
\text{Log}(l_2, o_2, s) &= \text{\text{\texttt{op2}}}_{\text{\texttt{R}}} \\
R, M &\leftarrow R[r \mapsto \text{\texttt{poison}}], M
\end{align*}
\]

\section*{Figure 10. Semantics of psub}

\[
p = \text{Log}(l, o, s) \quad \text{deref}_{M}(p, sz, l, o) \quad M(l) = (t, r, n, a, c, P) \quad b = \text{sub}^{\text{\texttt{SI2}}}_{\text{\texttt{SZ}}}(c, o) \quad o\%a = 0
\]

\[
\text{Load}(M, p, sz, a) = b
\]

\[
p = \text{Log}(l, o, s) \quad M(l) = (t, r, n, a, c, P) \quad \text{deref}_{M}(p, sz, l, o) \quad c' = \text{overwrite}^{\text{\texttt{SI2}}}_{\text{\texttt{SZ}}}(c, b, o)
\]

\[
\text{Store}(M, p, b, a) = M'
\]

\section*{Figure 11. Semantics of two auxiliary functions, Load and Store (when \(p\) is logical pointer only). If \(p\) is a logical pointer and it does not match these two cases, it fails.}

\[
isz(b) = \begin{cases} 
n & \text{such that } \forall_{0 \leq i < n} b[i] = n, i \forall b[i] = (\text{ Phy}(n, 0, 0, \text{None}), i) \\
\text{poison} & \text{if there's no such } n
\end{cases}
\]

\section*{Figure 13. Converting a bit vector to an integer}

low-level representation, getbit \(v\ i\) is a function that returns \(i\)th bit of a value \(v\). For vector types, \(ty\|\) transforms values element-wise, where + denotes the bitwise concatenation.

\[
isz(b) = \begin{cases} 
n & \text{such that } \forall_{0 \leq i < sz} b[i] = n, i \forall b[i] = (\text{ Phy}(n, 0, 0, \text{None}), i) \\
\text{poison} & \text{if there's no such } n
\end{cases}
\]

\section*{Figure 12. Converting a value to a bit vector}

\(\text{isz}(b)\) transforms bitwise value \(b\) to an integer of type \(\text{isz}\). It creates integer \(n\) from bits. Notation \(n.i\) is used to represent \(i\)th bit of non-\texttt{poison} integer \(n\). Type punning from pointer to integer yields \texttt{poison} and this explains redundant load-store pair elimination.\footnote{Redundant load-store pair eliminations means removing ‘\(v = \text{load 164 ptr; store v, ptr’}. If reading a logical pointer as integer implicitly casts the pointer, removing this load-store pair is not allowed.} One exception is when the pointer is a physical pointer of address space 0. In this case, type punning yields the integer address of the physical pointer. If any bit of \(b\) is \texttt{poison}, the result of \(\text{isz}(b)\) is \texttt{poison}. Figure 13 shows the definition of \(\text{isz}(b)\).

\(ty\ *\ \text{addrspace(s)}\|\(b\)\) transforms a bitvector \(b\) into a pointer of type \(ty\ *\ \text{addrspace(s)}\). If \(b\) is exactly all the bits of pointer \(p\) in the right order, it returns \(p\). If all bits contain a non-\texttt{poison} integer, it reconstructs a physical pointer of address space 0 with the corresponding integer address. Otherwise, it returns \texttt{poison}. Combined with the definition of \(\text{isz}(b)\), this allows vectorization of loading heterogeneous aggregates containing both aligned pointers and integer.\footnote{For example, vectorizing load and store of struct T(float* a; uintptr_t b) type as <2 x i8> type is allowed.} Figure 13 shows the definition of \(\text{isz}(b)\).

For vector types, \(ty\|\) transforms bitwise representations element-wise.

\[
\langle sz \times ty \rangle | \langle v \rangle = \langle ty | (b_0), \ldots, ty | (b_{sz-1}) \rangle
\]
Figure 14. Converting a bit vector to a pointer.

Figure 15. Semantics of load, store.

Function Call. ‘call ty funcname’ calls a function with arguments which are given as operands of the instruction. call creates a fresh call id cid, and adds (cid, rcur) where rcur is current time of memory M to the global call id map. If an argument x is given to a call, register x inside the call has value updatecid(x). updatecid(x) is a function that updates every bit of x to have current cid if possible. ty is type of the argument.

updatecid(x) = ty[(map(ty)(x), updatebit)]
updatebit(b) = (Phy(o, s, l, cid), i) if b = (Phy(o, s, l, None), i) otherwise
updatebit(b) = b otherwise

By recording call id in physical pointers, alias analysis can assume that physical pointer which is given as argument never aliases with memory blocks allocated inside the function. Any actual access violating this rule would be UB because of the cid checks performed by load and store.

Function Return. When a function call with call id cid returns, its entry in the global call ID map gets changed to None, indicating that the call has ended.

Non-memory Operations. For the remaining operations we follow earlier work [?]. All operations on poison unconditionally return poison except phi and select. The instruction freeze(isz op) non-deterministically chooses an arbitrary non-poison value of the given type if op is poison. Otherwise, it is a NOP. Branching on poison is immediate UB. Select yields poison iff the condition is poison or the selected value is poison.