# Supplementary Material for: Reconciling High-level Optimizations and Low-level Code with Twin Memory Allocation

Anonymous Author(s)

## 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 $\text{alive}_M(l)$, 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 logical 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 $2^{\text{ptrsz}(0)} - 1$) 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 $1, \ldots, \text{memtwins} - 1$) are the base addresses of the *twin blocks*. For every block allocated via **malloc** or **alloc**, 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 $\text{alive}_M(l_1)$ and $\text{alive}_M(l_2)$, then $[M(l_1).P(s)_{i_1}, M(l_1).P(s)_{i_1} + P(l_1).n)$ is disjoint from $[M(l_2).P(s)_{i_2}, M(l_2).P(s)_{i_2} + P(l_2).n)$. 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 $\text{alive}_M(l_1)$ and $\text{alive}_M(l_2)$, then $M(l_1).P(s)_{i_1} \notin [M(l_2).P(s)_{i_2}, M(l_2).P(s)_{i_2} + P(l_2).n)$. This second condition is required to handle 0-sized blocks.

$\text{convert}(s, i, s')$ 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 $\text{convert}(s, i + o, s') = i' + o$ and $\text{convert}(s', i', s) = i + o$.

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

A logical memory address is of the shape $\text{Log}(l, o, s)$, 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 $\text{inbounds}_M(l, o)$, 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 $\text{strict\_inbounds}_M(l, o)$, if

*Conference'17, July 2017, Washington, DC, USA*
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{array}{rcl}
\mathit{stmt} & ::= & \mathit{reg}{=}\mathit{inst} \mid \mathbf{br}\ \mathit{op}, \mathit{label}, \mathit{label} \mid \mathbf{br}\ \mathit{label} \\
 & & \mid\ \mathbf{call}\ \mathit{ty}\ \mathit{funcname}\,(\mathit{ty}\ \mathit{op}[, \mathit{ty}\ \mathit{op}]^*)\ \overline{\mathit{fnattr}} \mid \mathbf{store}\ \mathit{ty}\ \mathit{op}, \mathit{ty}{*}\ \mathit{op} \\
 & & \mid\ \mathbf{ret\ void} \mid \mathbf{ret}\ \mathit{ty}\ \mathit{op} \mid \mathbf{unreachable} \\
 & & \\
\mathit{inst} & ::= & \mathbf{load}\ \mathit{ty}, \mathit{ty}{*}\ \mathit{op}, \mathbf{align}\ \mathit{op} \mid \mathbf{alloca}\ \mathit{ty}, \mathbf{align}\ \mathit{op} \\
 & & \mid\ \mathbf{ptrtoint}\ \mathit{ty}\ \mathit{op}\ \mathbf{to}\ \mathit{ty} \mid \mathbf{inttoptr}\ \mathit{ty}\ \mathit{op}\ \mathbf{to}\ \mathit{ty} \\
 & & \mid\ \mathbf{addrspacecast}\ \mathit{ty}\ \mathit{op}\ \mathbf{to}\ \mathit{ty} \\
 & & \mid\ \mathbf{getelementptr}\ [\mathbf{inbounds}]\ \mathit{ty}\ \mathit{op}\ [,\ [\mathbf{inrange}]\ \mathit{ty}\ \mathit{op}]^* \\
 & & \mid\ \mathbf{select}\ \mathit{ty}\ \mathit{op}, \mathit{ty}\ \mathit{op}, \mathit{ty}\ \mathit{op} \mid \mathbf{freeze}\ \mathit{ty}\ \mathit{op} \\
 & & \mid\ \mathbf{psub}\ \mathit{ty}\ \mathit{op}\ \mathit{op} \mid \mathbf{icmp}\ \mathit{cond}, \mathit{op}, \mathit{op} \\
 & & \mid\ \mathbf{phi}\ \mathit{ty}\ [\mathit{op}, \mathit{label}], \ldots, [\mathit{op}, \mathit{label}] \\
 & & \mid\ \mathbf{call}\ \mathit{ty}\ \mathit{funcname}\,(\mathit{ty}\ \mathit{op}[, \mathit{ty}\ \mathit{op}]^*)\ \overline{\mathit{fnattr}} \\
 & & \\
 & & \mid\ \mathit{aop}\ [\mathbf{nuw}]\ [\mathbf{nsw}]\ \mathit{ty}\ \mathit{op}, \mathit{op}\ \ \texttt{// arithmetic ops} \\
 & & \mid\ \mathit{div}\ [\mathbf{exact}]\ \mathit{ty}\ \mathit{op}, \mathit{op}\ \ \texttt{// divisions} \\
 & & \mid\ \mathit{lop}\ \mathit{ty}\ \mathit{op}, \mathit{op}\ \ \texttt{// logical ops} \\
 & & \mid\ \mathit{castop}\ \mathit{ty}\ \mathit{op}\ \mathbf{to}\ \mathit{ty}\ \ \texttt{// casting ops} \\
 & & \mid\ \ldots \\
 & & \\
\mathit{op} & ::= & \mathit{reg} \mid \mathit{constant} \mid \mathbf{poison} \\
\mathit{bty} & ::= & \mathbf{i}\mathit{sz} \mid \mathit{ty}\ast\ \mathit{addrspace}\ n\ \ \texttt{// base type} \\
\mathit{vty} & ::= & <\mathit{sz} \times \mathit{bty}>\ \ \texttt{// vector type} \\
\mathit{sty} & ::= & \{\ \mathit{ty}\ [, \mathit{ty}]^*\ \}\ \ \texttt{// struct type} \\
\mathit{aty} & ::= & [\ \mathit{sz} \times \mathit{ty}\ ]\ \ \texttt{// array type} \\
\mathit{ty} & ::= & \mathit{bty} \mid \mathit{vty} \mid \mathit{sty} \mid \mathit{aty} \\
 & & \\
\mathit{funcname} & ::= & \mathbf{malloc} \mid \mathbf{free} \mid \ldots \\
\mathit{fnattr} & ::= & \mathbf{alwaysinline} \mid \mathbf{readnone} \mid \mathbf{readonly} \mid \mathbf{writeonly} \ldots \\
\mathit{cond} & ::= & \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{ugt} \mid \mathbf{uge} \mid \mathbf{ult} \mid \mathbf{ule} \mid \mathbf{sgt} \mid \mathbf{sge} \mid \mathbf{slt} \mid \mathbf{sle} \\
\mathit{fcond} & ::= & \mathbf{oeq} \mid \mathbf{ogt} \mid \mathbf{olt} \mid \mathbf{ole} \mid \mathbf{one} \mid \mathbf{ord} \mid \mathbf{ueq} \mid \mathbf{uno} \mid \ldots \\
\mathit{aop} & ::= & \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{shl} \\
\mathit{div} & ::= & \mathbf{udiv} \mid \mathbf{sdiv} \mid \mathbf{urem} \mid \mathbf{srem} \mid \mathbf{lshr} \mid \mathbf{ashr} \\
\mathit{lop} & ::= & \mathbf{and} \mid \mathbf{or} \mid \mathbf{xor} \\
\mathit{castop} & ::= & \mathbf{trunc} \mid \mathbf{zext} \mid \mathbf{sext} \mid \mathbf{fptrunc} \mid \mathbf{fpext} \mid \mathbf{fptoui} \mid \mathbf{fptosi} \mid \ldots
\end{array}
$$

**Figure 1.** Syntax of LLVM IR (unrelated instructions omitted for brevity)

it is inbounds and it does not point beyond the end of the block, i.e., $0 \le o < n$.

Physical addresses are of the shape $\mathsf{Phy}(p, s, I, \mathit{cid})$ where $o$ is the physical address, i.e., it is an offset starting at address `0x0`. $s$ is the address space, and $I$ and $\mathit{cid}$ are additional constraints which should be met when the pointer is dereferenced. $I$ is a set of integer addresses which should be inbounds addresses of the dereferencing memory block when the physical pointer is dereferenced. $\mathit{cid}$ is a CallId enforcing that the physical pointer cannot access memory blocks created inside the function call. $I$ and $\mathit{cid}$ 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 $\mathsf{Phy}(o, s)$ whenever $I$ is an empty set and $\mathit{cid}$ is **None**.

To describe $\mathit{cid}$, 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 CallID $\rightarrow$ Time that is part of the memory. If a physical pointer is passed to a function call and $\mathit{cid}$ of the physical pointer is either **None** or a call that has already returned, $\mathit{cid}$ is updated to the call id of the new function call. Otherwise, $\mathit{cid}$ does not change. Inside the function call, even if a physical pointer points to some memory block $l$, dereferencing the physical pointer is UB if the beginning of the lifetime of $l$ is not earlier than call time of $\mathit{cid}$. Escaping the physical pointer (e.g., storing it into a global variable or returning it at the end of the function) does not change $\mathit{cid}$. After the function call is returned, all physical pointers having the call id act as if their $\mathit{cid}$ are **None**. In other words, even if a pointer with a $\mathit{cid}$ is returned back to its caller, it is no longer restricted in how it can be used.

$$
\begin{array}{ll}
\mathrm{Num}(sz) & ::= \{\, i \mid 0 \le i < 2^{sz} \,\} \\
\mathrm{Time} & ::= \mathbb{N} \\
\mathrm{BlockID} & ::= \mathbb{N} \\
\mathrm{CallID} & ::= \mathbb{N} \\
\mathrm{Mem} & ::= \mathrm{Time} \times (\mathrm{BlockID} \twoheadrightarrow \mathrm{Block}) \times (\mathrm{CallID} \twoheadrightarrow \mathrm{Time} \uplus \mathbf{None}) \\
\mathrm{AddrSpace} & ::= \mathbb{N} \\
\mathrm{Block} & ::= \{\, (t, r, n, a, c, P) \mid t \in \{\, \texttt{stack}, \texttt{heap}, \texttt{global}, \texttt{function} \,\} \\
 & \qquad \wedge\, r \in (\mathrm{Time} \times (\mathrm{Time} \uplus \{\infty\})) \wedge n \in \mathbb{N} \wedge a \in \mathbb{N} \wedge c \in \mathrm{Byte}^{n} \\
 & \qquad \wedge\, P \in ((s \in \mathrm{AddrSpace}) \twoheadrightarrow \mathrm{Num}(\mathrm{ptrsz}(s))^{\mathrm{memtwins}}) \,\} \\
\mathrm{LogAddr}(s) & ::= \{\, \mathsf{Log}(l, o, s) \mid l \in \mathrm{BlockID} \wedge o \in \mathrm{Num}(\mathrm{ptrsz}(s)) \,\} \\
\mathrm{PhyAddr}(s) & ::= \{\, \mathsf{Phy}(o, s, I, cid) \mid o \in \mathrm{Num}(\mathrm{ptrsz}(s)) \wedge I \subset \mathrm{Num}(\mathrm{ptrsz}(s)) \\
 & \qquad \wedge\, cid \in \mathrm{CallID} \uplus \{\mathbf{None}\} \,\} \\
\mathrm{Addr}(s) & ::= \mathrm{LogAddr}(s) \uplus \mathrm{PhyAddr}(s) \\
[\![ \mathbf{i}sz ]\!] & ::= \mathrm{Num}(sz) \uplus \{\, \mathbf{poison} \,\} \\
[\![ \langle sz \times ty \rangle ]\!] & ::= \{0, \ldots, sz - 1\} \to [\![ ty ]\!] \\
[\![ ty * \mathrm{addrspace}(s) ]\!] & ::= \mathrm{Addr}(s) \uplus \{\, \mathbf{poison} \,\} \\
\mathrm{Name} & ::= \{\, \texttt{\%x}, \texttt{\%y}, \ldots \,\} \\
\mathrm{Reg} & ::= \mathrm{Name} \to \{\, (ty, v) \mid v \in [\![ ty ]\!] \,\} \\
\mathrm{Byte} & ::= \mathrm{Bit}^{8} \\
\mathrm{Bit} & ::= [\![ \mathbf{i}1 ]\!] \uplus \mathrm{AddrBit} \\
\mathrm{AddrBit} & ::= \{\, (p, i) \mid \exists s.\ p \in \mathrm{Addr}(s) \wedge (0 \le i < \mathrm{ptrsz}(s)) \,\}
\end{array}
$$

**Figure 2.** Semantic domain

To actually perform a memory access of size $sz > 0$ through a pointer $p$, it must be *dereferencable* as some block-offset-pair, written $\mathsf{deref}_M(p, sz, l, o)$. If $p$ is a logical pointer $\mathsf{Log}(l, o, 0)$ and $\mathsf{alive}_M(l)$ and $\mathsf{inbounds}_M(l, o + sz)$, then we have $\mathsf{deref}_M(p, sz, l, o)$. If $p$ is a physical pointer $\mathsf{Phy}(p, 0, I, cid)$, there must be a block $l$ and an offset $o$ such that $M(l).P(0) + o = p$ and $\mathsf{alive}_M(l)$ and $\mathsf{inbounds}_M(l, o + sz)$ and moreover, for all $p' \in I$, we must have $\mathsf{inbounds}_M(l, p' - M(l).P(0))$ (i.e., all these $p'$ are inbounds of the same block) and if $cid \neq \mathbf{None}$ and $M(cid) \neq \mathbf{None}$, then $M(l).b < M(cid)$ (i.e., the block was allocated before the function call identified by $cid$ began, and that function call is still ongoing). If all these requirements are satisfied, we have $\mathsf{deref}_M(p, sz, l, o)$. 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 $\mathsf{Phy}(0, s, \emptyset, \mathbf{None})$. Defining NULL pointer as physical pointer allows folding **inttoptr**(0) into NULL (**inttoptr**($x$) is an instruction that casts integer $x$ to pointer), and replacing $p$ by NULL if $p$ == NULL holds. Also LLVM can optimize NULL + idx into **inttoptr**($idx$).

***Values.*** $[\![ ty ]\!]$ denotes the set of values of type $ty$. An integer value of type $\mathbf{i}sz$ is either a concrete number $i$ within range $0 \le i < 2^{sz}$, or **poison**. A pointer value of type '$ty * \mathrm{addrspace}(s)$' is defined as either a logical address $\mathsf{Log}(l, o, s)$, a physical address $\mathsf{Phy}(o, s, I, cid)$, or **poison**. There is no distinction between pointer values of different types (`i32*`, `i64*`, ..), but there is a distinction between pointer values of different address spaces.[1] 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 **i**1 (i.e., 0, 1, or **poison**) or the $i$th bit of a pointer value $p$. Storing to memory involves converting a value to an array of bits. $ty\Downarrow(v)$ is a function that converts value $v$ to bits. Similarly, loading a value from memory involves converting bits to a value. $ty\Uparrow(b)$ is a function that converts bits $b$ to a value of type $ty$.

$$
\begin{array}{lll}
ty\Downarrow & \in & [\![ ty ]\!] \to \mathrm{Bit}^{\mathrm{bitwidth}(ty)} \\
ty\Uparrow & \in & \mathrm{Bit}^{\mathrm{bitwidth}(ty)} \to [\![ ty ]\!]
\end{array}
$$

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

[1] 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.

$$\begin{aligned} \text{getbit} &\in [\![bty]\!] \rightarrow \mathbb{N} \rightarrow \text{Bit} \\ \text{getbit}\, n\, i &= \text{platform dependent} \\ & \quad \text{where } n \in [\![\mathbf{i}sz]\!], 0 \le i < sz \\ \text{getbit}\, p\, i &= (p, i) \\ & \quad \text{where } p \in [\![ty * \text{addrspace}(s)]\!], \\ & \quad 0 \le i < \text{ptrsz}(s) \end{aligned}$$

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 $\iota$ updates the register file $R \in \text{Reg}$ and the memory $M \in \text{Mem}$, denoted $R, M \stackrel{\iota}{\hookrightarrow} R', M'$. 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:

$$\begin{aligned} [\![r]\!]_R &= R(r) && \text{// register} \\ [\![C]\!]_R &= C && \text{// constant} \\ [\![\mathbf{poison}]\!]_R &= \mathbf{poison} && \text{// poison} \end{aligned}$$

We propose to add one new instruction to LLVM: **psub**.[2] This instruction takes two pointers $p_1, p_2$ and returns $p_1 - p_2$ 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.[3]

***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 $\text{cast2int}_M(l, o, s)$ and $\text{cast2ptr}(o, s)$:

$$\begin{aligned} \text{cast2int}_M(l, o, s) &= (P(s) + o)\%2^{\text{ptrsz}(s)} \quad \text{where} \\ & \quad M(l) = (t, r, n, a, c, P) \\ \text{cast2ptr}(o, s) &= \text{Phy}(o, s, \emptyset, \mathbf{None}) \end{aligned}$$

Casting from a logical pointer to integer, or $\text{cast2int}_M(l, o, s)$, 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** $\text{Log}(l, o, s)$' computes $\text{cast2int}_M(l, o, s)$ and returns it. '**ptrtoint** $\text{Phy}(o, s, I, cid)$' simply returns $o$. If the size of the destination type $\mathbf{i}sz$ 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 $\text{cast2ptr}(o, s)$, returns a physical pointer with no provenance information. One option here is to add $o$ to $I$ 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))`.[4]

'**inttoptr** $\mathbf{i}sz \ \ o$ **to** $ty * \text{addrspace}(s)$' is equivalent to $\text{cast2ptr}\,(o, s)$ if the size of the source type $\mathbf{i}sz$ is equivalent to the size of the destination type $ty * \text{addrspace}(s)$. 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 casted to a pointer in another address space using **addrspacecast**.

Figure 4 shows formal semantics of **addrspacecast**. If the given pointer is a physical pointer $\text{Phy}(o, s, I cid)$, the instruction translates both the offset $o$ and the inbounds offsets $I$ using $\text{convert}(s, \_, s')$. 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 $\text{Log}(l, o, s)$, its block id is maintained, and the new offset is calculated as follows. First, the pointer is casted to integer using $\text{cast2int}_M$. 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 $\text{convert}(1, x, 2) = x$ (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 $p = \text{Log}(l, 15, 1)$ will have integer address $(8 + 15)\%16 = 7$, but **addrspacecast** $p$ *to* $2 = \text{Log}(l, 15, 2)$ will return $(8 + 15)\%32 = 23$. This breaks our property that any pointer $p$ can be replaced with **inttoptr**(**ptrtoint**($p$)).

[2] Currently **psub** is implemented as an intrinsic function, @**llvm.psub**. For simplicity, it is represented as an instruction in this document.
[3] SPEC CPU2017 has up to 2% speedup.

[4] 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.

$$\frac{(\iota = \text{“}r = \textbf{ptrtoint}\ ty * \text{addrspace}(s)\ op\ \textbf{to}\ \textbf{i}sz\text{”})\quad \mathsf{Log}(l,\ o,\ s) = [\![op]\!]_R \quad \mathsf{cast2int}_M(l,\ o,\ s) = j}{R,\ M \xhookrightarrow{\iota} R[r \mapsto j\%2^{sz}],\ M}$$

$$\frac{(\iota = \text{“}r = \textbf{ptrtoint}\ ty * \text{addrspace}(s)\ op\ \textbf{to}\ \textbf{i}sz\text{”})\quad \mathsf{Phy}(o,\ s,\ I,\ cid) = [\![op]\!]_R}{R,\ M \xhookrightarrow{\iota} R[r \mapsto o],\ M}$$

$$\frac{(\iota = \text{“}r = \textbf{ptrtoint}\ ty * \text{addrspace}(s)\ op\ \textbf{to}\ \textbf{i}sz\text{”})\quad \textbf{poison} = [\![op]\!]_R}{R,\ M \xhookrightarrow{\iota} R[r \mapsto \textbf{poison}],\ M}$$

$$\frac{(\iota = \text{“}r = \textbf{inttoptr}\ \textbf{i}sz\ op\ \textbf{to}\ ty * \text{addrspace}(s)\text{”})\quad \textbf{poison} \neq [\![op]\!]_R \quad \mathsf{cast2ptr}([\![op]\!]_R,\ s) = p}{R,\ M \xhookrightarrow{\iota} R[r \mapsto p],\ M}$$

$$\frac{(\iota = \text{“}r = \textbf{inttoptr}\ \textbf{i}sz\ op\ \textbf{to}\ ty * \text{addrspace}(s)\text{”})\quad \textbf{poison} = [\![op]\!]_R}{R,\ M \xhookrightarrow{\iota} R[r \mapsto \textbf{poison}],\ M}$$

**Figure 3.** Semantics of **ptrtoint**, **inttoptr**

$$\frac{(\iota = \text{“}r = \textbf{addrspacecast}\ ty_1 * \text{addrspace}(s_1) op\ \textbf{to}\ ty_2 * \text{addrspace}(s_2)\text{”})\quad \mathsf{Phy}(o, s_1, I, cid) = [\![op]\!]_R \quad o' = \mathsf{convert}(s_1, o, s_2) \quad I' = \{\, \mathsf{convert}(s_1, i, s_2) \mid i \in I \,\}}{R, M \xhookrightarrow{\iota} R[r \mapsto \mathsf{Phy}(o', s_2, I', cid)], M}$$

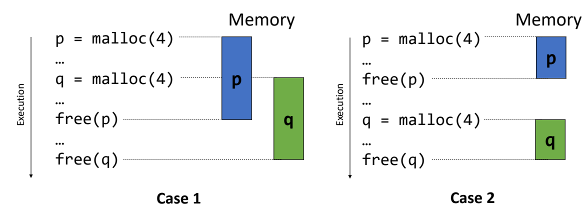
$$\frac{(\iota = \text{“}r = \textbf{addrspacecast}\ ty_1 * \text{addrspace}(s_1) op\ \textbf{to}\ ty_2 * \text{addrspace}(s_2)\text{”})\quad \mathsf{Log}(l, o, s_1) = [\![op]\!]_R \quad o' = (\mathsf{convert}(s_1, \mathsf{cast2int}_M(l, o, s_1), s_2) - \mathsf{cast2int}_M(l, 0, s_2))\%2^{\mathsf{ptrsz}(s_2)}}{R, M \xhookrightarrow{\iota} R[r \mapsto \mathsf{Log}(l, o', s_2)], M}$$

**Figure 4.** Semantics of **addrspacecast**

***Pointer Comparison.*** Now we define the semantics of pointer comparison. **icmp** compares two pointers in the same address space, and returns a value of type **i**1. 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 ecsape $p$ if $p$ is inbounds address, because integer address of $p$ is always positive if it is inbounds.[5]

The definition of **icmp eq** $p1$ $p2$ is as follows.

1. If $p1, p2$ are both logical addresses $\mathsf{Log}(l_1, o_1, s)$ and $\mathsf{Log}(l_2, o_2, s)$, 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., $\neg(0 \leq o_1 < n_1) \vee \neg(0 \leq o_2 < n_2)$, 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.[6] 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 $\mathsf{Phy}(o_1, s, I_1, cid_1)$, $\mathsf{Phy}(o_2, s, I_2, cid_2)$: the result is equivalent to $o_1 = o_2$.
3. If $p1 = \mathsf{Phy}(o_1, s, I_1, cid_1)$ and $p2 = \mathsf{Log}(l_2, o_2, s)$ or vice versa, the result is equivalent to $o_1 = \mathsf{cast2int}_M(l_2, o_2, s)$.

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

[5] Note that LLVM is rather conservative, so it assumes that p == NULL does not capture only if p is some system memory allocating function.

[6] 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 $p1 = \mathsf{Log}(l, o_1, s)$ and $p2 = \mathsf{Log}(l, o_2, s)$, we check whether the offsets $o_1$ and $o_2$ are inbounds. This condition is needed because $l + o_1$ or $l + o_2$ can overflow at runtime. If this is the case, the result is $o_1 \leq_u o_2$. Otherwise, we allow non-deterministic choice.
2. If $p1 = \mathsf{Log}(l_1, o_1, s)$, $p2 = \mathsf{Log}(l_2, o_2, s)$ and $l_1 \neq l_2$, the result is nondeterministic choice.
3. If $p1 = \mathsf{Phy}(o_1, s, I_1, cid_1)$ and $p2 = \mathsf{Phy}(o_2, s, I_2, cid_2)$, the result is $o_1 \leq_u o_2$.
4. If $p1 = \mathsf{Phy}(o_1, s, I_1, cid_1)$ and $p2 = \mathsf{Log}(l_2, o_2, s)$, it is $o_1 \leq_u \mathsf{cast2int}_M(l_2, o_2, s)$.

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**.

$$(\iota = \text{“}r = \textbf{icmp}\ op\ ty * \text{addrspace}(s)\ op_1\ op_2\text{”})$$
$$\frac{\dfrac{}{\textbf{poison} = [\![op_1]\!]_R \vee \textbf{poison} = [\![op_2]\!]_R}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \textbf{poison}], M}$$

***Memory Allocation / Deallocation.*** Memory allocating operations create a new memory block and pick integer base addresses $P(s)$ 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 $P(0)_i$ may not be 0 and the last strictly inbounds address $P(0)_i + n - 1$ may not be $2^{\mathsf{ptrsz}(0)} - 1$, i.e., the first and last byte 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. Furthermore, $P(s)_i + n$ must not exceed the size of $s$. Finally, for all $s$, all the $[P(s)_i, P(s)_i + n)$ 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 allocated and the block contains all members of the aggregated type. **malloc**(0) returns NULL.

The pointers returned by **alloca** and **malloc** all have address 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 $\mathsf{deref}_m(p, 1, l, 0)$ 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 $\tau_{cur}$.

***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 $p = \mathsf{Log}(l, o, s)$, **getelementptr** $p$, $i$ returns $\mathsf{Log}(l, (o + i')\%2^{\mathsf{ptrsz}(s)}, s)$ 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 $p = \mathsf{Phy}(o, s, I, cid)$, then **getelementptr** $p$, $i$ simply returns $\mathsf{Phy}((o + i')\%2^{\mathsf{ptrsz}(s)}, s, I, cid)$ where $i'$ is $i$ multiplied by size of its element type. This operation does not affect $I$ and $cid$, which enables optimizing **getelementptr** $p$, 0 to $p$. In the **inbounds** variant, the returned pointer has an updated inbounds set $I' = I \cup \{o, ((o + i')\%2^{\mathsf{ptrsz}(s)})\}$. This allows for further alias analysis even on physical pointers. Also, tracking inbounds addresses and checking them later instead of returning **poison** instantly allows reordering 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 Figure 9.

If the base pointer points to a nested aggregate value, the **getelementptr** instruction may have multiple indexes as its operands. In this case, it is allowed for **getelementptr inbounds** to point past the range of a subtype. 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
```

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp eq } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \textsc{icmp-ptr-logical-same-block} \\ \mathsf{Log}(l, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l, o_2, s) = [\![op_2]\!]_R\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 = o_2)], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp eq } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \textsc{icmp-ptr-logical-different-block} \\ \mathsf{Log}(l_1, o_1, s) = [\![op_1]\!]_R \\ \mathsf{Log}(l_2, o_2, s) = [\![op_2]\!]_R \qquad l_1 \neq l_2\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \textbf{false}], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp eq } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \textsc{icmp-ptr-logical-nondet-true} \\ \mathsf{Log}(l_1, o_1, s) = [\![op_1]\!]_R \qquad M(l_1) = (t_1, (b_1, e_1), n_1, a_1, c_1, P_1) \\ \mathsf{Log}(l_2, o_2, s) = [\![op_2]\!]_R \qquad M(l_2) = (t_2, (b_2, e_2), n_2, a_2, c_2, P_2) \\ l_1 \neq l_2 \qquad \neg(0 \leq o_1 < n_1) \vee \neg(0 \leq o_2 < n_2) \vee [b_1, e_1) \cap [b_2, e_2) = \emptyset\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \textbf{true}], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp eq } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \textsc{icmp-ptr-physical} \\ \mathsf{Phy}(o_1, s, I_1, cid_1) = [\![op_1]\!]_R \\ \mathsf{Phy}(o_2, s, I_2, cid_2) = [\![op_2]\!]_R\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 = o_2)], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp eq } ty * \text{ addrspace}(s)\ op_1\ op_2{}''\text{)} \\ \textsc{icmp-ptr-physical-logical} \\ (\mathsf{Phy}(p, s, I, cid_1) = [\![op_1]\!]_R \wedge \mathsf{Log}(l, o, s) = [\![op_2]\!]_R) \vee \\ (\mathsf{Phy}(p, s, I, cid) = [\![op_2]\!]_R \wedge \mathsf{Log}(l, o, s) = [\![op_1]\!]_R)\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (p = \mathsf{cast2int}_M(l, o, s))], M}$$

**Figure 5.** Semantics of **icmp eq**

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp ule } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \mathsf{Log}(l, o_1, s) = [\![op_1]\!]_R \qquad \mathsf{inbounds}_M(l, o_1) \\ \mathsf{Log}(l, o_2, s) = [\![op_2]\!]_R \qquad \mathsf{inbounds}_M(l, o_2)\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 \leq_u o_2)], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp ule } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \mathsf{Log}(l_1, o_1, s) = [\![op_1]\!]_R \qquad \mathsf{Log}(l_1, o_2, s) = [\![op_2]\!]_R \qquad l_1 \neq l_2 \qquad b \in \{\textbf{true}, \textbf{false}\}\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto b], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp ule } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \mathsf{Phy}(o_1, s, I_1, cid_1) = [\![op_1]\!]_R \\ \mathsf{Phy}(o_2, s, I_2, cid_2) = [\![op_2]\!]_R\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 \leq_u o_2)], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp ule } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \mathsf{Phy}(p, s, I, cid_1) = [\![op_1]\!]_R \\ \mathsf{Log}(l, o, s) = [\![op_2]\!]_R\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (p \leq_u \mathsf{cast2int}_M(l, o, s))], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{icmp ule } ty * \text{ addrspace}(s)\ op_1\ op_2\text{”}) \\ \mathsf{Log}(l, o, s) = [\![op_1]\!]_R \\ \mathsf{Phy}(p, s, I, cid_2) = [\![op_2]\!]_R\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (\mathsf{cast2int}_M(l, o, s) \leq_u p)], M}$$

**Figure 6.** Semantics of **icmp ult**

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{alloca } ty,\ \textbf{align } a\text{”}) \\ n = bytewidth(ty) \quad u = \mathbf{i}(8 \times n)\Downarrow(\textbf{poison}) \quad l \text{ fresh} \quad P \text{ unallocated physical addresses}\end{array}}{R, (\tau_{cur}, f, C) \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Log}(l, 0, 0)], (\tau_{cur} + 1, f[l \mapsto (\mathtt{stack}, (\tau_{cur}, \infty), n, a, u, P)], C)}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{call i}8{*}\ \textbf{malloc}(\text{iptrsz}(0)\ len)\text{)”} \\ n = [\![len]\!]_R \quad u = \mathbf{i}(8 \times n)\Downarrow(\textbf{poison}) \quad l \text{ fresh} \quad P \text{ unallocated physical addresss} \quad n > 0\end{array}}{R, (\tau_{cur}, f, C) \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Log}(l, 0, 0)], (\tau_{cur} + 1, f[l \mapsto (\mathsf{heap}, (\tau_{cur}, \infty), n, a, u, P)], C)}$$

$$\frac{\begin{array}{c}(\iota = \text{“}r = \textbf{call i}8{*}\ \textbf{malloc}(\text{iptrsz}(0)\ len)\text{)”} \\ -\end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{NULL}], M}$$

**Figure 7.** Semantics of **alloca**, **call malloc**()

In this case, %t is not **poison**.

***Pointer Subtraction.*** We define a new instruction **psub**$(ptr1, ptr2)$ that calculates $ptr1 - ptr2$. This operation

$$(\iota = \text{“}\textbf{call void free}(\text{i8}*\ op)\text{”})$$
$$\frac{\mathsf{Log}(0,0,0) = [\![op]\!]_R}{R, M \xhookrightarrow{\iota} R, M}$$

$$(\iota = \text{“}\textbf{call void free}(\text{i8}*\ op)\text{”})$$
$$\frac{\mathsf{Log}(l,0,0) = [\![op]\!]_R \quad M(l) = (\mathsf{heap}, (b, \infty), n, a, c, P)}{R, (\tau_{cur}, f, C) \xhookrightarrow{\iota} R, (\tau_{cur}+1, f[l \mapsto (\mathsf{heap}, (b, \tau_{cur}), n, a, c, P)], C)}$$

$$(\iota = \text{“}\textbf{call void free}(\text{i8}*\ op)\text{”})$$
$$\frac{\mathsf{Phy}(o,0,I,cid) = [\![op]\!]_R \qquad M(l) = (\mathsf{heap}, (b, \infty), n, a, c, P) \qquad \mathsf{cast2int}(l,0) = o \qquad b < \mathsf{calltime}(cid)}{R, (\tau_{cur}, f, C) \xhookrightarrow{\iota} R, (\tau_{cur}+1, f[l \mapsto (\mathsf{heap}, (b, \tau_{cur}), n, a, c, P)], C)}$$

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

$$(\iota = \text{“}r = \textbf{getelementptr}\ ty*\ \text{addrspace}(s)\ op_1\ \textbf{i}sz\ op_2\text{”})$$
GETELEMENTPTR-LOGICAL
$$\frac{\mathsf{Log}(l,o,s) = [\![op_1]\!]_R \quad i = [\![op_2]\!]_R}{R, M \xhookrightarrow{\iota} R[r \mapsto \mathsf{Log}(l, (o + bytewidth(ty) * i)\%2^{\mathsf{ptrsz}(s)}, s)], M}$$

$$(\iota = \text{“}r = \textbf{getelementptr}\ ty*\ \text{addrspace}(s)\ op_1\ \textbf{i}sz\ op_2\text{”})$$
GETELEMENTPTR-PHYSICAL
$$\frac{\mathsf{Phy}(o,s,I,cid) = [\![op_1]\!]_R \quad i = [\![op_2]\!]_R}{R, M \xhookrightarrow{\iota} R[r \mapsto \mathsf{Phy}((o + bytewidth(ty) * i)\%2^{\mathsf{ptrsz}(s)}, s, I, cid)], M}$$

$$(\iota = \text{“}r = \textbf{getelementptr inbounds}\ ty*\ \text{addrspace}(s)\ op_1\ \textbf{i}sz\ op_2\text{”})$$
GETELEMENTPTR-INBOUNDS-LOGICAL
$$\frac{\begin{array}{c}\mathsf{Log}(l,o,s) = [\![op_1]\!]_R \qquad M(l) = (t, r, n, a, c, P)\\ \mathsf{inbounds}_M(l,o) \qquad \mathsf{inbounds}_M(l, o + bytewidth(ty) * i)\end{array}}{R, M \xhookrightarrow{\iota} R[r \mapsto (l, o + bytewidth(ty) * i, s)], M}$$

$$(\iota = \text{“}r = \textbf{getelementptr inbounds}\ ty*\ \text{addrspace}(s)\ op_1\ \textbf{i}sz\ op_2\text{”})$$
GETELEMENTPTR-INBOUNDS-PHYSICAL
$$\frac{\mathsf{Phy}(o,s,I,cid) = [\![op_1]\!]_R \qquad i = [\![op_2]\!]_R \qquad o' = (o + bytewidth(ty) * i) \qquad o + bytewidth(ty) * i < 2^{\mathsf{ptrsz}(s)}}{R, M \xhookrightarrow{\iota} R[r \mapsto \mathsf{Phy}(o', s, I \cup \{o, o'\}, cid)], M}$$

**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 $ptr1 = \mathsf{Phy}(o_1, s, I_1, cid_1), ptr2 = \mathsf{Phy}(o_2, s, I_2, cid_2)$, the result is $(o_1 - o_2)\%2^{\mathsf{ptrsz}(s)}$.
3. If $ptr1$ is a logical address and $ptr2$ is a physical address or vice versa, say $ptr1 = \mathsf{Log}(l_1, o_1, s), ptr2 = \mathsf{Phy}(o_2, s, I_2, cid_2)$, the result is equivalent to $(\mathsf{cast2int}_M(l_1, o_1, s) - o_2)\%2^{\mathsf{ptrsz}(s)}$.

Figure 10 shows formal semantics of **psub**. The transformation `(p1 - p2) == 0` $\longrightarrow$ `p1 == p2` is valid, but its inverse is not. Similarly, `(p1 - p2) > 0` $\longrightarrow$ `p1 > 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.

$$\begin{array}{rcl} ty\Downarrow & \in & [\![ty]\!] \longrightarrow \mathsf{Bit}^{\mathsf{bitwidth}(ty)} \\ ty\Uparrow & \in & \mathsf{Bit}^{\mathsf{bitwidth}(ty)} \longrightarrow [\![ty]\!] \end{array}$$

For base types, $ty\Downarrow$ transforms **poison** into the bitvector of all **poison** bits, and defined values into their standard

$$(\iota = \text{“}r = \textbf{psub}\ ty*\ \text{addrspace}(s)\ op1,\ op2\text{”})\ \frac{\mathsf{Log}(l, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l, o_2, s) = [\![op_2]\!]_R}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 - o_2)\%2^{\mathsf{ptrsz}(s)}], M}$$

$$(\iota = \text{“}r = \textbf{psub}\ ty*\ \text{addrspace}(s)\ op1,\ op2\text{”})\ \frac{\mathsf{Log}(l_1, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Log}(l_2, o_2, s) = [\![op_2]\!]_R \quad l_1 \neq l_2}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \textbf{poison}], M}$$

$$(\iota = \text{“}r = \textbf{psub}\ ty*\ \text{addrspace}(s)\ op1,\ op2\text{”})\ \frac{\mathsf{Phy}(o_1, s, I_1, cid_1) = [\![op_1]\!]_R \quad \mathsf{Phy}(o_2, s, I_2, cid_2) = [\![op_2]\!]_R}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 - o_2)\%2^{\mathsf{ptrsz}(s)}], M}$$

$$(\iota = \text{“}r = \textbf{psub}\ ty*\ \text{addrspace}(s)\ op1,\ op2\text{”})\ \frac{\mathsf{Log}(l, o_1, s) = [\![op_1]\!]_R \quad \mathsf{Phy}(o_2, s, I_2, cid_2) = [\![op_2]\!]_R \quad o' = \mathsf{cast2int}_M(l, o_1, s)}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o' - o_2)\%2^{\mathsf{ptrsz}(s)}], M}$$

**Figure 10.** Semantics of **psub**

$$\frac{p = \mathsf{Log}(l, o, s) \quad M(l) = (t, r, n, a, c, P) \quad \mathsf{deref}_M(p, sz, l, o) \quad b = \mathsf{substr}^{8\times n}_{sz}(c, o) \quad o\%a = 0}{\mathsf{Load}(M, p, sz, a) = b}$$

$$\frac{p = \mathsf{Log}(l, o, s) \quad M(l) = (t, r, n, a, c, P) \quad M = (\tau_{cur}, f, C) \quad \mathsf{deref}_M(p, sz, l, o) \quad o\%a = 0 \quad c' = \mathsf{overwrite}^{8\times n}_{sz}(c, b, o) \quad M' = (\tau_{cur}, f[l \mapsto (t, r, n, a, c', P)], C)}{\mathsf{Store}(M, p, b, a) = M'}$$

**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.

$$\mathbf{i}sz\Uparrow(b) = \begin{cases} n & \text{such that } \forall_{0 \le i < sz}\, b[i] = n.i \vee \\ & b[i] = (\mathsf{Phy}(n, 0, \emptyset, \textbf{None}), i) \\ \textbf{poison} & \text{if there's no such } n \end{cases}$$

**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, $ty\Downarrow$ transforms values element-wise, where $+\!\!+$ denotes the bitvector concatenation.

$$\begin{aligned} \mathbf{i}sz\Downarrow(v) \text{ or } ty*\ \text{addrspace}(s)\Downarrow(v) &= \lambda i.\ \text{getbit}\ v\ i \\ \langle sz\times ty\rangle\Downarrow(v) &= ty\Downarrow(v[0]) +\!\!+ \ldots \\ &\quad +\!\!+\ ty\Downarrow(v[sz-1]) \\ &\quad \text{where } b = b_0 +\!\!+ \ldots +\!\!+ b_{sz-1} \end{aligned}$$

**Figure 12.** Converting a value to a bit vector

$\mathbf{i}sz\Uparrow(b)$ transforms bitwise value $b$ to an integer of type $\mathbf{i}sz$. It creates integer $n$ from bits. Notation $n.i$ is used to represent $i$th bit of non-**poison** integer $n$. Type punning from pointer to integer yields **poison** and this explains redundant load-store pair elimination.[7] 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 **poison**, the result of $\mathbf{i}sz\Uparrow(b)$ is **poison**. Figure 13 shows the definition of $\mathbf{i}sz\Uparrow(b)$.

$ty*\ \text{addrspace}(s)\Uparrow(b)$ 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-**poison** integer, it reconstructs a physical pointer of address space 0 with the corresponding integer address. Otherwise, it returns **poison**. Combined with the definition of $\mathbf{i}sz\Uparrow(b)$, this allows vectorization of loading heterogeneous aggregates containing both aligned pointers and integer.[8] Figure 13 shows the definition of $\mathbf{i}sz\Uparrow(b)$.

For vector types, $ty\Uparrow$ transforms bitwise representations element-wise.

$$\langle sz\times ty\rangle\Uparrow(b) = \langle ty\Uparrow(b_0), \ldots, ty\Uparrow(b_{sz-1})\rangle$$

[7] Redundant load-store pair eliminations means removing ‘`v = load i64 ptr; store v, ptr`’. If reading a logical pointer as integer implicitly casts the pointer, removing this load-store pair is not allowed.

[8] For example, vectorizing load and store of `struct T{float* a; uintptr_t b}` type as `<2 x i8*>` type is allowed.

$$ty * \text{addrspace}(s)\Uparrow(b) = \begin{cases} p & \text{such that } addrspace(p) = s \wedge \forall_{0 \le i < 2^{\text{ptrsz}(s)}}\, b[i] = (p, i) \\ \text{Phy}(n, 0, \emptyset, \textbf{None}) & \text{such that } s = 0 \wedge \text{i}sz\Uparrow(b) = n \\ \textbf{poison} & \text{if there's no such } p \text{ or } n \end{cases}$$

**Figure 14.** Converting a bit vector to a pointer

Figure 11 is the semantics of two auxiliary functions Load and Store. $\text{substr}^n_{sz}(c, o)$ is a partial function $(\text{Bit}^n \times \mathbb{N}) \rightarrow \text{Bit}^{sz}$ which returns $(c[o\times 8], c[o\times 8+1], \ldots, c[o\times 8+sz-1])$. $\text{overwrite}^n_{sz}(c, b, o)$ is a partial function $(\text{Bit}^n \times \text{Bit}^{sz} \times \mathbb{N}) \rightarrow \text{Bit}^n$ that overwrites bits $b$ over $c$ at byte offset $o$. If a dereferenceable physical pointer $p = \text{Phy}(o, s, I, cid)$ is given, $\text{Load}(M, p, sz, a)$ behaves exactly same as $\text{Load}(M, p', sz, a)$ where $p'$ is a logical pointer $\text{Log}(l, o', s)$, which is uniquely determined for $p$ as described in the early part of this section. If there's no such $p'$, $\text{Load}(M, p, sz, a)$ fails. $\text{Store}(M, p, b, a)$ on a dereferenceable physical pointer $p$ behaves exactly same as $\text{Store}(M, p', b, a)$, where $p'$ is a logical pointer which is uniquely determined for the physical pointer $p$. If there's no such $p'$, $\text{Store}(M, p, b, a)$ fails. $\text{Load}(M, p, sz, a)$ and $\text{Store}(M, p, b, a)$ fail if $p$ is not dereferenceable, regardless of $p$'s type.

Now we define semantics of **load**/**store** operations. $\text{Load}(M, p, sz, a)$ returns the bits $p$ points to if it successfully dereferences the pointer with given size $sz$ and alignment $a$. **load** yields $v$ if $\text{Load}(M, p, sz, a)$ successfully returns a value, or UB if $\text{Load}(M, p, sz, a)$ fails. The store operation $\text{Store}(M, p, b, a)$ successfully stores the bit representation $b$ into the memory $M$ and returns the updated memory if $p$ is dereferenceable with the given alignment $a$. **store** is UB if $\text{Store}(M, p, b, a)$ fails, or updates memory to $M'$ otherwise.

$$(\iota = \text{“}r = \textbf{load}\ ty,\ ty{*}\ op,\ \textbf{align}\ a\text{”})$$

$$\frac{\text{Load}(M, [\![op]\!]_R, \text{bitwidth}(ty), a) \text{ fails}}{R, M \xhookrightarrow{\iota} \text{UB}}$$

$$\frac{\text{Load}(M, [\![op]\!]_R, \text{bitwidth}(ty)) = v}{R, M \xhookrightarrow{\iota} R[r \mapsto ty\Uparrow(v)], M}$$

$$(\iota = \text{“}\textbf{store}\ ty\ op_1,\ ty{*}\ op,\ \textbf{align}\ a\text{”})$$

$$\frac{\text{Store}(M, [\![op]\!]_R, ty\Downarrow([\![op_1]\!]_R), a) \text{ fails}}{R, M \xhookrightarrow{\iota} \text{UB}}$$

$$\frac{\text{Store}(M, [\![op]\!]_R, ty\Downarrow([\![op_1]\!]_R)) = M'}{R, M \xhookrightarrow{\iota} R, M'}$$

**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 $(cid, \tau_{cur})$ where $\tau_{cur}$ 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.

$$\text{updatecid}(x) = ty\Uparrow(map(ty\Downarrow(x), \text{updatebit}))$$

$$\text{updatebit}(b) = (\text{Phy}(o, s, I, cid), i) \text{ if } b = (\text{Phy}(o, s, I, \textbf{None}), i)$$

$$\text{updatebit}(b) = b \text{ otherwise}$$

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**(i$sz$ $op$) 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**.