$C \in \mathcal{C} \quad b \in block \quad m \in mem \quad \alpha \in oracle$

$\mathcal{C}^+ ::= \mathbf{L}(C) \mid \mathbf{S}(b, base) \qquad base \in \mathbb{Z}$

$val ::= \ldots \mid ptr\ c\ off \qquad c \in \mathcal{C}^+, off \in \mathbb{Z}$

$$
\begin{aligned}
read &\in oracle \to mem \to \mathbb{Z} \rightharpoonup val \\
write &\in oracle \to mem \to \mathbb{Z} \to val \rightharpoonup mem \\
M &\in \mathcal{M} \subseteq \mathcal{C}^+ \to mem \\
alloc &\in oracle \to \mathcal{C} + block \to \mathbb{Z} \rightharpoonup (\mathbb{Z} \times oracle) \\
free &\in oracle \to \mathcal{C}^+ \to \mathbb{Z} \rightharpoonup oracle
\end{aligned}
$$

(a) Definitions

$$\frac{read\ (M\ C)\ off = v}{C, \alpha, M \mid *(ptr\ \mathbf{L}(C)\ off) \longrightarrow C, \alpha, M \mid v}$$

$$\frac{write\ (M\ C)\ off\ v = m' \qquad M' = M[C \mapsto m']}{C, \alpha, M \mid *(ptr\ \mathbf{L}(C)\ off) := v \longrightarrow C, \alpha, M' \mid v}$$

$$\frac{read\ (M\ b)\ (base + off) = v}{C, \alpha, M \mid *(ptr\ \mathbf{S}(b, base)\ off) \longrightarrow C, \alpha, M \mid v}$$

$$\frac{write\ (M\ b)\ (base + off)\ v = m' \qquad M' = M[b \mapsto m']}{C, \alpha, M \mid *(ptr\ \mathbf{S}(b, base)\ off) := v \longrightarrow C, \alpha, M' \mid v}$$

(b) Reads and writes

$$\frac{alloc\ \alpha\ C\ sz = (p, \alpha')}{C, \alpha, M \mid \texttt{malloc}(int\ sz) \longrightarrow C, \alpha', M \mid ptr\ \mathbf{L}(C)\ p}$$

$$\frac{fresh\ b \qquad alloc\ \alpha\ b\ sz = (p, \alpha')}{C, \alpha, M \mid \texttt{malloc}(int\ sz) \longrightarrow C, \alpha', M \mid ptr\ \mathbf{S}(b, p)\ 0}$$

$$\frac{}{C, \alpha, M \mid (\texttt{int})(ptr\ \mathbf{L}(C)\ off) \longrightarrow C, \alpha, M \mid int\ off}$$

$$\frac{}{C, \alpha, M \mid (\texttt{int})(ptr\ \mathbf{S}(b, base)\ off) \longrightarrow C, \alpha, M \mid int\ (base + off)}$$

$$\frac{}{C, \alpha, M \mid (\texttt{t*})(int\ i) \longrightarrow C, \alpha, M \mid ptr\ \mathbf{L}(C)\ i}$$

(c) Allocations and Casts

Figure 1: Compartmentalized Memory Model

# 1 Abstract Sharing Semantics

We define a C semantics that separates the world into compartments, ranged over by $A$, $B$, $C$, etc., each with its own separate memory. The basics are shown in Figure 1. The core of this memory model is the allocator oracle, ranged over by $\alpha$; this abstract state encapsulates information about how allocations are arrayed inside compartments or as shared blocks.

The *read* and *write* operations always operate on a single abstract memory, which behaves as a single flat address space accessed via integers, except that operations that access addresses outside of previously allocated blocks can failstop depending on the oracle. Memories are kept totally separate, and accesses in one can't interact with the others.

Pointer values carry one of two types of indices that determine which memory they access: $\mathbf{L}(C)$, for local pointers into the compartment $C$, and $\mathbf{S}(b, base)$, where $b$ is an abstract block identifier and *base* is an integer. A "super-memory" $M$ is a map from such indices to memories. Pointers

also always carry an offset. (This means that we can always convert a pointer to either an abstract block-offset model or to an integer.)

Figure **??** shows how loads and stores occur in this system. Shared pointers are converted to integer offsets, while local pointers just use their existing offsets, and then the *read* and *write* operations occur.

**Allocation** The abstract operation, *alloc*, yields an address at which to locate a new block, either within a compartment's memory or in its own isolated block. In the latter case, the address provided becomes the new base of that block. It also produces a new oracle that, among other things, will guarantee that loads and stores to the new block will not failstop. Since the *alloc* operation is parameterized by the compartment or block in question, it is allowed to give one compartment an address that is not currently allocated within it, even if other compartments might have that address allocated.

**Casting** Figure **??** also gives a demonstration of how integer-pointer and pointer-integer casts work. We can always cast pointers to integers; integers cast to pointers always point into the compartment that performed the cast.

## 2 Cross-compartment interfaces

In this system, each function is assigned to a compartment, and we annotate all function arguments and returns that are meant to take values that will be used as shared pointers. Control passes from one compartment to another whenever a call is made to an inactive compartment, and on return from such a call. It is undefined behavior to pass or return a pointer without a sharing annotation, and all pointers passed with annotation must have $\mathbf{S}(\dots)$ as their index, else this is also UB. As a consequence, a compartment can never obtain a $\mathbf{L}(\dots)$-indexed variable from a compartment other than itself. Using static escape analysis, we can approximate the allocations that should be shared, replacing calls to `malloc` with `malloc_shared` or identifying stack allocations that need to be sharable.