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

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

$\mathit{val} ::= \ldots \mid \mathit{ptr}\ c\ a \qquad c \in \mathcal{C}^+, a \in \mathit{int}$

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

(a) Definitions

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

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

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

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

(b) Reads and writes

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

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

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

(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 α; 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 $\mathbf{S}(b, \mathit{base})$, 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.

$$\begin{array}{rl} \mathbf{A} ::= & !\alpha\ m\ a\ v \\ & |?\alpha\ m\ (a_1, a_2) \end{array}$$

$$\frac{\mathit{write}\ \alpha\ m\ a\ v = m'}{!\alpha\ m'\ a\ v} \qquad \frac{!\alpha\ m\ a\ v}{\mathit{read}\ \alpha\ m\ a = v}$$

$$\frac{?\alpha\ m\ (a_1, a_2) \quad a_1 \le a < a_2}{\exists v.\mathit{read}\ \alpha\ m\ a = v} \qquad \frac{?\alpha\ m\ (a_1, a_2) \quad a_1 \le a < a_2}{\exists m'.\mathit{write}\ \alpha\ m\ a\ v = m'} \qquad \frac{\mathit{read}\ \alpha\ m\ a = v}{!\alpha\ m\ a\ v}$$

$$\frac{\begin{array}{c} !\alpha\ m\ a\ v \qquad a \neq a' \\ \mathit{write}\ \alpha\ m\ a'\ v' = m' \end{array}}{!\alpha\ m'\ a\ v} \qquad \frac{\begin{array}{c} ?\alpha\ m\ (a_1, a_2) \\ \mathit{write}\ \alpha\ m\ a\ v = m' \end{array}}{?\alpha\ m'\ (a_1, a_2)} \qquad \frac{\mathit{alloc}\ \alpha\ I\ s = (i, \alpha')}{?\alpha'\ m\ (i, i+s)}$$

$$\frac{\begin{array}{c} !\alpha\ m\ a\ v \quad a < i \text{ or } a \le i+s \\ \mathit{alloc}\ \alpha\ I\ s = (i, \alpha') \end{array}}{!\alpha'\ m\ a\ v} \qquad \frac{\begin{array}{c} ?\alpha\ m\ (a_1, a_2) \\ \mathit{alloc}\ \alpha\ I\ s = (i, \alpha') \end{array}}{i+s < a_1 \text{ or } a_2 \le i} \qquad \frac{\begin{array}{c} ?\alpha\ m\ (a_1, a_2) \\ \mathit{alloc}\ \alpha\ I\ s\ m = (i, \alpha') \end{array}}{?\alpha'\ m\ (a_1, a_2)}$$

$$\frac{\begin{array}{c} ?\alpha\ m\ (a_1, a_2) \quad a < a_1 \\ \mathit{free}\ \alpha\ I\ a = \alpha' \end{array}}{?\alpha'\ m\ (a_1, a_2)} \qquad \frac{\begin{array}{c} ?\alpha\ m\ (a_1, a_2) \quad a_2 \le a \\ \mathit{free}\ \alpha\ I\ a = \alpha' \end{array}}{?\alpha'\ m\ (a_1, a_2)} \qquad \frac{\begin{array}{c} ?\alpha\ m\ (a_1, a_2) \\ \mathit{perturb}\ \alpha\ m = (\alpha', m') \end{array}}{?\alpha'\ m'\ (a_1, a_2)}$$

$$\frac{\begin{array}{c} ?\alpha\ m\ (a_1, a_2) \quad !\alpha\ m\ a\ v \quad a_1 \le a < a_2 \\ \mathit{perturb}\ \alpha\ m = (\alpha', m') \end{array}}{!\alpha'\ m'\ a\ v}$$

Figure 2: Assertion Rules

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 When a pointer is cast to an integer type, its value doesn't change—instead, certain integer operations applied to pointer values will demote the result to an integer value. If such a value is cast back to a pointer type, it retains its value. If, however, an integer value is cast to a pointer, the result is always a local pointer to the active compartment, as shown in Figure **??**.

2 Oracle Axioms

To simplify the expression of these axioms, we define a small assertion language within which we can express facts about the values of addresses in memory. We write $!\alpha\ m\ a\ v$ to indicate that, in memory m, address a is expected to contain value v. We write $?\alpha\ m\ (a_1, a_2)$ to indicate that the range of addresses from a_1 to a_2 are allocated as a block, so operations will not failstop (i.e., they will produce some value, but which value may be unknown). Finally, we write $\mathbf{A}[\alpha, m, a]$ to represent the set of assertions binding α, m, and a, and $\mathbf{a}[x/y]$ to indicate the assertion $\mathbf{a}$ substituting y for x.

Local Behavior Such assertions are naturally related to the behavior of the system:

$$\frac{\textit{write}\ \alpha\ m\ a\ v = m'}{!\alpha\ m'\ a\ v} \qquad \frac{!\alpha\ m\ a\ v}{\textit{read}\ \alpha\ m\ a = v}$$

The full set of rules is found in 2. These rules encode facts like: a store followed by a load yields the value that was stored, two consecutive loads yield the same value, and so on. Allocations are guaranteed to be disjoint from any prior allocations. In fact, because all compartments share an oracle, this is the case even for allocations from other compartments—convenient, because we assume that we are targeting a system with a single address space, and this guarantees that our semantics only describes operations that can fit in that address space.

Finally, we introduce the *perturb* operation. The only assertions that are maintained over a call to perturb are those involving addresses in allocated regions. Perturb happens during every function call and return, because the compiler needs to be free to reallocate memory during those operations.

3 Cross-compartment interfaces

In this system, each function is assigned to a compartment. A compartment interface is a subset of the functions in the compartment that are publicly accessible. At any given time, the compartment that contains the currently active function is considered the active compartment. It is illegal to call a private function in an inactive compartment.

Public functions are only permitted to receive pointer arguments if they are shared pointers (that is, they ultimately derive from a call to `malloc_share`). Private functions may take either kind of pointer as argument.

Violations of a compartment interface exhibit failstop behavior.

As a consequence of these rules, a compartment can never obtain a $\mathbf{L}(\dots)$-indexed pointer from a compartment other than itself. It can recieve such a pointer if it is cast to an integer type. If the resulting integer is cast back into a pointer, it will have the same behavior as if it were cast from any other integer: accessing it may cause a failstop or else access the appropriate address in the active compartment's block.