An Abstract Model of Compartmentalization with Sharing

Sean Anderson

February 27, 2024

1 Introduction

This document describes the desired behavior of a compartmentalized C system in terms of a correct-by-construction abstract machine. The model aims to fulfill a few key criteria:

- Compartments are obviously and intuitively isolated from one another by construction
- It is suitable for hardware enforcement without placing intensive constraints on the target
- Inter-compartment interactions via shared memory are possible and conform to the C standard

To this last case: we don't necessarily care that compartments' internal behavior conforms to the C standard. In fact the model explicitly gives compartments a concrete view of memory, giving definition to code that would be undefined behavior in the standard. But when it comes to shared memory, the standard has a clear implication that the memory accessible to a piece of code is determined by the provenance of pointers that code can access. This model embraces that principle.

2 Abstract 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 core memory model is shown in Figure 1. The allocator oracle, ranged over by α, is an abstract state encapsulating information about how allocations are arrayed inside compartments or as shared blocks. From the programmer's perspective, α should be opaque, but for purposes of proving correctness of a given system it is instantiated to match the actual behavior of the compiler-allocator-hardware combination.

The *read* and *write* operations always operate on a single abstract memory, which behaves as a flat address space accessed via integers. Operations that access addresses outside of allocated blocks may or may not failstop, determined by the oracle. If they are successful, they behave consistently (multiple consecutive loads yield the same value, a load after a store yields the stored value, etc.) Memories are kept totally separate, fulfilling our first requirement: compartments' memories are definitely never accessible.

Pointer values carry *indices* from the set $\mathcal{C}^+$ 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 integer address representing their current position.

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

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

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

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

(a) Definitions

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

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

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

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

$$\frac{\mathit{heap\_alloc}\ \alpha\ C\ sz = (p, \alpha')}{C, \alpha, M \mid \texttt{malloc}(\mathit{Vint}\ sz) \longrightarrow C, \alpha', M \mid \mathit{Vptr}\ \mathbf{L}(C)\ p}$$

$$\frac{\mathit{fresh}\ b \qquad \mathit{heap\_alloc}\ \alpha\ b\ sz = (p, \alpha')}{\begin{array}{l}C, \alpha, M \mid \texttt{malloc\_share}(\mathit{Vint}\ sz) \longrightarrow \\ C, \alpha', M \mid \mathit{Vptr}\ \mathbf{S}(b, p)\ p\end{array}}$$

$$\frac{\mathit{heap\_free}\ \alpha\ I\ a = \alpha'}{C, \alpha, M \mid \texttt{free}(\mathit{Vptr}\ I\ a) \longrightarrow C, \alpha', M \mid \mathit{Vundef}}$$

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

(b) Selected Memory Steps

Figure 1: Compartmentalized Memory Model

$$\frac{}{C, \alpha, M, e \mid \odot(\mathit{Vptr}\ I\ a) \longrightarrow C, \alpha, M, e \mid \mathit{Vptr}\ I\ (\langle\odot\rangle a)}$$

$$\frac{}{C, \alpha, M, e \mid (\mathit{Vptr}\ I\ a) \oplus (\mathit{Vint}\ i) \longrightarrow C, \alpha, M, e \mid \mathit{Vptr}\ I\ (a\langle\oplus\rangle i)}$$

$$\frac{}{C, \alpha, M, e \mid (\mathit{Vint}\ i) \oplus (\mathit{Vptr}\ I\ a) \longrightarrow C, \alpha, M, e \mid \mathit{Vptr}\ I\ (i\langle\oplus\rangle a)}$$

$$\frac{}{C, \alpha, M, e \mid (\mathit{Vptr}\ I\ a_1) \oplus (\mathit{Vptr}\ I\ a_2) \longrightarrow C, \alpha, M, e \mid \mathit{Vint}\ (a_1\langle\oplus\rangle a_2)}$$

Figure 2: Arithmetic Operations

Figure **??** shows how loads and stores occur in this system. The pointer's index determines which memory it accesses, and its value is treated as the pointer into that memory. A pointer's index can never be changed via arithmetic—all operations either preserve the index, or demote their result to a plain integer.

Allocation The abstract operations, *heap_alloc* and *stk_alloc*, yield addresses 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's index. It also updates the oracle to reflect that loads and stores to the new block will not failstop (see Section 3). Since the $*$*_alloc* operations are parameterized by the compartment or block that they allocate, they are allowed to make decisions based on that information, such as attempting to keep compartment-local allocations in designated regions. But a programmer should not rely on any such assumptions.

Arithmetic and Integer-Pointer Casts Most arithmetic operations are typical of C. The interesting operations are those involving integers that have been cast from pointers. We give concrete definitions to all such operations based on their address. If they involve only a single former pointer, the result will also be a pointer into the same memory; otherwise the result is a plain integer. If the former pointer is cast back to a pointer type, it retains its value and is once again a valid pointer. Otherwise, if an integer value is cast to a pointer, the result is always a local pointer to the active compartment, as shown in Figure 1b.

Calls and Returns There are two interesting details of the call and return semantics: they allocate and deallocate memory, and they can cross compartment boundaries. In the first case, we need to pay attention to which stack-allocated objects are to be shared. This can again be done using escape analysis: objects whose references never escape, can be allocated locally. Objects whose references escape to another compartment must be allocated as shared. Those that escape to another function in the same compartment can be treated in either way; if they are allocated locally but are later passed outside the compartment, the system will failstop at that later point (see Section 4).

We assume that each local variable comes pre-annotated with how it should be allocated, with a simple flag **L** or **S**, so that a function signature is a list of tuples $(id, \mathbf{L} \mid \mathbf{S}, sz)$.

$$e \in \mathit{ident} \rightharpoonup (\mathcal{C}^+ \times \mathit{int})$$

$$\mathit{alloc\_locals}\ C\ \alpha\ \mathit{ls} = \begin{cases} (\alpha, \lambda \mathit{id}.\bot) & \text{if } \mathit{ls} = [\,] \\ (\alpha'', e[\mathit{id} \mapsto (\mathbf{L}(C), i)]) & \text{if } \mathit{ls} = (\mathit{id}, \mathbf{L}, \mathit{sz}) :: \mathit{ls}', \\ & \text{where } \mathit{alloc\_locals}\ C\ \alpha\ \mathit{ls}' = (\alpha', e) \\ & \text{and } \mathit{stk\_alloc}\ \alpha'\ C\ \mathit{sz} = (i, \alpha'') \\ (\alpha'', e[\mathit{id} \mapsto (\mathbf{S}(b, i), i)]) & \text{if } \mathit{ls} = (\mathit{id}, \mathbf{S}, \mathit{sz}) :: \mathit{ls}', \\ & \text{where } \mathit{alloc\_locals}\ C\ \alpha\ \mathit{ls}' = (\alpha', e) \\ & \text{and for fresh } b, \mathit{stk\_alloc}\ \alpha'\ b\ \mathit{sz} = (i, \alpha'') \end{cases}$$

$$\mathit{dealloc\_locals}\ \alpha\ e = \begin{cases} \mathit{stk\_free}\ \alpha'\ C\ i & \text{if for some } \mathit{id}, e\ \mathit{id} = (\mathbf{L}(C), i), \\ & \text{where } \mathit{dealloc\_locals}\ \alpha\ e[\mathit{id} \mapsto \bot] = \alpha' \\ \mathit{stk\_free}\ \alpha'\ b\ i & \text{if for some } \mathit{id}, e\ \mathit{id} = (\mathbf{L}(b, i), i), \\ & \text{where } \mathit{dealloc\_locals}\ \alpha\ e[\mathit{id} \mapsto \bot] = \alpha' \\ \alpha & \text{otherwise} \end{cases}$$

$$\mathit{perturb} \in \mathit{oracle} \to \mathit{oracle}$$

$$\frac{f = (C, \mathit{locals}, s) \quad \mathit{alloc\_locals}\ C\ \alpha\ \mathit{locals} = (\alpha', e)}{\mathit{CALL}(f, \alpha, M) \longrightarrow C, \mathit{perturb}\ \alpha', M, e \mid s}$$

$$\frac{\mathit{dealloc\_locals}\ \alpha\ e = \alpha'}{C, \alpha, M, e \mid \texttt{return}\ v \longrightarrow \mathit{RET}(\alpha, M, v)}$$

Figure 3: Call and Return Semantics (Continuations ommitted)

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

$$\frac{?\alpha\ I\ m\ (a_1, a_2) \quad a_1 \leq a < a_2}{\exists m'.\mathit{write}\ \alpha\ m\ a\ v = m'} \qquad \frac{\mathit{read}\ \alpha\ m\ a = v}{!\alpha\ I\ m\ a\ v} \qquad \frac{\begin{array}{c} !\alpha\ I\ m\ a\ v \quad a \neq a' \\ \mathit{write}\ \alpha\ m\ a'\ v' = m' \end{array}}{!\alpha\ I\ m'\ a\ v}$$

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

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

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

Figure 4: Assertion Rules

3 Axioms of the Oracle

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 $!\alpha\ I\ m\ a\ v$ to indicate that, with oracle state α, the memory m whose index is I is known to map a to v. We write $?\alpha\ I\ m\ (a_1, a_2)$ 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 undefined).

Assertions are naturally related to the behavior of the system: $! \ldots a\ v$ will hold after a store of v to a, and from that assertion it follows that a load from a yields v. The full set of rules is found in Figure 4.

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 any valid allocator must be that can fit in that address space.

Finally, Figure 3 introduced the *perturb* operation, which mutates the allocator oracle. 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.

4 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 may not receive $L(\dots)$-indexed pointer arguments. Private functions may take either kind of pointer as argument. $L(\dots)$ pointers may also not be stored to shared memory. This guarantees that there can be no confusion between shared pointers and a compartment's own local pointer that escaped its control. 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.

5 Machine Constraints

Now we consider the constraints that this system places on potential implementations. In particular, in a tag-based enforcemen mechanism with a limited quantity of tags, is this system realistic? In general it requires a unique tag per compartment, as well as one for each shared allocation. In the extreme, consider a system along the lines of ARM's MTE, which has four-bit tags. That could only enforce this semantics for a very small program, or one with very little shared memory (fewer than sixteen tags, so perhaps two-four compartments and around a dozen shared objects.)

On the other hand, this semantics is a reasonable goal under an enforcement mechanism with even eight-bit tags (512 compartments and shared objects.) If we go up to sixteen bits, we can support programs with thousands of shared objects.

That said, it only takes a minor adjustment for this model to be enforceable in even the smallest of tag-spaces. Instead of separate dynamic blocks for each shared object, we parameterize each instance of `malloc` with a list of compartments that are allowed to use the pointers that it allocates. We write the identifiers for these `malloc` invocations $\texttt{malloc}_{\overline{C}}$, where $\overline{C}$ is a set of compartments. Then the relevant step rules become those in Figure 5, with calls and returns changed similarly.

This version can be enforced with a number of tags equal to the number of different sharing combinations present in the system—in the worst case this would be exponential in the number of compartments, but in practice it can be tuned to be arbitrarily small. (In extremis, all shared objects can be grouped together to run on a machine with only two tags.) Sadly it strays from the C standard in its temporal memory safety: under some circumstances a shared object can be accessed by a compartment that it has not (yet) been shared with.

$$\mathit{val} ::= \ldots \mid \mathit{Vptr}\ c\ a \quad c \in 2^{\mathcal{C}}$$

$$\begin{aligned}
M \in\ & \mathcal{M} \subseteq 2^{\mathcal{C}} \to \mathit{mem} \\
\mathit{heap\_alloc} \in\ & \mathit{oracle} \to 2^{\mathcal{C}} \to \mathit{int} \rightharpoonup (\mathit{int} \times \mathit{oracle}) \\
\mathit{heap\_free} \in\ & \mathit{oracle} \to 2^{\mathcal{C}} \to \mathit{int} \rightharpoonup \mathit{oracle} \\
\mathit{stk\_alloc} \in\ & \mathit{oracle} \to 2^{\mathcal{C}} \to \mathit{int} \rightharpoonup (\mathit{int} \times \mathit{oracle}) \\
\mathit{stk\_free} \in\ & \mathit{oracle} \to 2^{\mathcal{C}} \to \mathit{int} \rightharpoonup \mathit{oracle}
\end{aligned}$$

$$\frac{\mathit{read}\ (M\ \overline{C})\ a = v}{C, \alpha, M \mid *(\mathit{Vptr}\ \overline{C}\ a) \longrightarrow C, \alpha, M \mid v}$$

$$\frac{\mathit{write}\ (M\ \overline{C})\ a\ v = m' \quad M' = M[\overline{C} \mapsto m']}{C, \alpha, M \mid *(\mathit{Vptr}\ \overline{C}\ a) := v \longrightarrow C, \alpha, M' \mid v}$$

$$\frac{\mathit{heap\_alloc}\ \alpha\ C\ \mathit{sz} = (p, \alpha')}{C, \alpha, M \mid \mathtt{malloc}_{\overline{C}}(\mathit{Vint}\ \mathit{sz}) \longrightarrow C, \alpha', M \mid \mathit{Vptr}\ \mathbf{L}(C)\ p}$$

$$\frac{}{C, \alpha, M \mid (\mathtt{t}*)(\mathit{Vint}\ i) \longrightarrow C, \alpha, M \mid \mathit{Vptr}\ \{C\}\ i}$$

Figure 5: Memory With Explicit Sharing