An Abstract Model of Compartmentalization with Sharing

Sean Anderson

March 11, 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
- Compartments can only access shared memory if they have first obtained a valid pointer to it, consistent with the C standard and "capability reasoning"

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. A concrete memory, m, partially maps machine integer (int) addresses to values with a basic axiomatization given in Figure 2. Axioms **WR1** and **WR2** are standard memory axioms.

Axiom **RW** deals with the partial nature of memory: reads and writes that are not defined correspond to failstop behavior. Below, we will discuss how reads and writes to allocated memory are guaranteed to succeed. Other reads and writes might succeed or failstop depending on how the compiler has organized memory. **WR1** implicitly tells us that if a write succeeds, a read from that location will succeed as well. **RW** tells us conversely that a location that is legal to read is also legal to write, regardless of its allocation status.

One memory of this kind is assigned to each compartment, and additional memories will be allocated for shared blocks. Memories are kept totally separate, fulfilling our first requirement: compartments' local memories are definitely never accessible to other compartments. A pointer

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

$$C \in \mathcal{C} \quad b \in block$$

$$\begin{aligned}
\mathcal{C}^+ &::= \mathbf{L}(C) \mid \mathbf{S}(b, base) && base \in int \\
v \in val &::= \ldots \mid Vptr\ c\ a && c \in \mathcal{C}^+, a \in int \\
e &\in ident \rightharpoonup (\mathcal{C}^+ \times int)
\end{aligned}$$

$$\begin{aligned}
state ::= {} & C, M, e, \ldots \mid \texttt{expr} \\
\mid {} & C, M, e, \ldots \mid \texttt{stmt} \\
\mid {} & CALL(f, M, \ldots) \\
\mid {} & RET(M, v, \ldots)
\end{aligned}$$

$$(\longrightarrow) \in state \times state$$

Figure 1: Definitions

$$\begin{aligned}
\mathbf{WR1} &\triangleq write\ m\ a\ v = m' \to \\
&\qquad read\ m\ a = v \\
\mathbf{WR2} &\triangleq read\ m\ a = v \to \\
&\qquad write\ m\ a'\ v' = m' \to \\
&\qquad a \neq a' \to read\ m'\ a = v \\
\mathbf{RW} &\triangleq read\ m\ a = v \to \\
&\qquad \exists m'. write\ m\ a\ v' = m'
\end{aligned}$$

$$\frac{M\ C = m \qquad read\ m\ a = v}{C, M, e \mid *(Vptr\ \mathbf{L}(C)\ a) \longrightarrow C, M, e \mid v}\textsc{ValOfLocal}$$

$$\frac{M\ C = m \quad write\ m\ a\ v = m' \quad M' = M[C \mapsto m']}{C, M, e \mid *(Vptr\ \mathbf{L}(C)\ a) := v \longrightarrow C, M', e \mid v}\textsc{AssignLocal}$$

$$\frac{M\ b = m \qquad read\ m\ a = v}{C, M, e \mid *(Vptr\ \mathbf{S}(b, base)\ a) \longrightarrow C, M, e \mid v}\textsc{ValOfShare}$$

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

Figure 2: Reads and Writes

$$\textbf{HA} \triangleq \mathit{heap\_alloc}\ M\ B\ sz = (a, M') \rightarrow \mathit{LiveH}\ M\ B\ (a, a + sz)$$

$$\textbf{HAM} \triangleq \mathit{Live}\ M\ B\ (a_1, a_2) \rightarrow \exists m. M\ B = m$$

$$\textbf{HAR} \triangleq \mathit{LiveH}\ M\ B\ (a_1, a_2) \rightarrow a_1 \leq a < a_2 \rightarrow \exists v. \mathit{read}\ (M\ B)\ a = v$$

$$\textbf{HAW} \triangleq \mathit{LiveH}\ M\ B\ (a_1, a_2) \rightarrow a_1 \leq a < a_2 \rightarrow \exists m'. \mathit{write}\ (M\ B)\ a\ v = m'$$

$$\frac{\begin{array}{c} \mathit{expr} = \texttt{malloc(}\ \mathit{Vint}\ sz) \\ \mathit{heap\_alloc}\ M\ C\ sz = (a, M') \end{array}}{C, M, e \mid \mathit{expr} \longrightarrow C, M', e \mid \mathit{Vptr}\ \mathbf{L}(C)\ a}\ \textsc{MallocLoc}$$

$$\frac{\begin{array}{c} M\ b = \bot \qquad \mathit{heap\_alloc}\ M\ b\ sz = (a, M') \\ \mathit{expr} = \texttt{malloc\_share(}\ \mathit{Vint}\ sz) \end{array}}{C, M, e \mid \mathit{expr} \longrightarrow C, M', e \mid \mathit{Vptr}\ \mathbf{S}(b, a)\ a}\ \textsc{MallocShr}$$

$$\frac{v = \mathit{Vptr}\ \mathbf{L}(C)\ a \quad \mathit{heap\_free}\ M\ C\ a = M'}{C, M, e \mid \texttt{free}(v) \longrightarrow C, M', e \mid \mathit{Vundef}}\ \textsc{FreeLoc}$$

$$\frac{\begin{array}{c} v = \mathit{Vptr}\ (\mathbf{S}(b, \mathit{base}))\ \mathit{base} \\ \mathit{heap\_free}\ M\ b\ a = M' \end{array}}{C, M, e \mid \texttt{free}(v) \longrightarrow C, M', e \mid \mathit{Vundef}}\ \textsc{FreeShr}$$

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

Figure 3: Heap Allocation and Integer-Pointer Cast

value consists of a pair: a "base" (ranged over by B) drawn from the set $\mathcal{C}^+$ that determines which memory it accesses, and a machine integer address representing their concrete position. A base is either $\mathbf{L}(C)$, for local pointers into the compartment C, or $\mathbf{S}(b, \mathit{base})$, where b is an abstract block identifier and *base* is a machine integer. A "super-memory" M is a map from bases to memories.

The allocation and free operations for both stack and heap act on the super-memory. The axiomatization for heap allocation is given in Figure 3. Once a region is allocated within a memory, the predicate *LiveH* holds on the allocated region, and reads and writes are guaranteed to succeed within that region of the appropriate base. Further axioms related to how multiple allocations interact with one another will appear later.

This axiomatization serves to abstract away concrete details about memory layout that may be specific to a given compiler-allocator-hardware combination. We can understand any particular instance of $\mathcal{M}$ as an oracle that divines where the target system will place each allocation and, with knowledge of the full layout of memory, determines what happens in the event of an out-of-bounds read or write. This oracular is constrained by the full set of axioms in Section 3.

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 base. Since the $*\_alloc$ operations are parameterized by the full super-memory M and the identity of 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 to protect using a page-table-based enforcement mechanism.

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. As shown in 4, if they involve

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

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

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

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

Figure 4: Arithmetic Operations Involving Pointers

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 **??**.

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 $\mathbf{L}$ or $\mathbf{S}$, so that a function signature is a list of tuples $(id, \mathbf{L} \mid \mathbf{S}, sz)$. (Aside: there is a case to be made we should just allocate all stack objects locally barring some critical use case for share them. Doing so would simplify the model here.)

The allocation and deallocation of stack memory is shown in the step rules in Section 5. In the full semantics, calls and returns step through intermediate states, written CALL and RET. During the step from the intermediate callstate into the function code proper, the semantics looks up the function being called and allocates its local variables before beginning to execute its statement. And during the step from the `return` statement into the intermediate returnstate, the semantics likewise deallocates every variable it had previously allocated.

3 Memory Axioms

The oracular axiomatization of memory hides the behavior of the compiler behind an abstract interface that produces addresses on command. On the surface this creates a nondeterministic semantics, but given any particular instantiation of *mem*, the semantics becomes deterministic.

Relevant axioms have been shown above, and now we can fill in the rest, with a special focus on how behavior is or is not preserved over allocations. We define two kinds of statements that we can make about the locations of allocated regions. The proposition $\mathit{LiveH}\ M\ (B, a_1, a_2)$ means

$$\mathit{alloc\_locals}\ M\ C\ [\,] = (M, \lambda id.\bot)$$

$$\begin{aligned}\mathit{alloc\_locals}\ M\ C\ (id, \mathbf{L}, sz) :: ls &= \\ (M'', e[id \mapsto (\mathbf{L}(C), i)])& \\ \text{where } \mathit{stk\_alloc}\ M\ C\ sz = (i, M')& \\ \text{and } \mathit{alloc\_locals}\ M'\ C\ ls = (M'', e)&\end{aligned}$$

$$\begin{aligned}\mathit{alloc\_locals}\ M\ C\ (id, \mathbf{S}, sz) :: ls &= \\ (M'', e[id \mapsto (\mathbf{S}(b, i), i)])& \\ \text{where } \mathit{alloc\_locals}\ M\ C\ ls = (M', e)& \\ \text{and } \mathit{stk\_alloc}\ M'\ C\ sz = (i, M'')&\end{aligned}$$

$$\begin{aligned}\mathbf{SA} &\triangleq \mathit{stk\_alloc}\ M\ B\ sz = (a, M') \rightarrow \\ &\mathit{LiveS}(M') = (B, a, a + sz) :: \mathit{LiveS}(M)\end{aligned}$$

$$\begin{aligned}\mathbf{SAM} &\triangleq (B, a_1, a_2) \in \mathit{LiveS}(M) \rightarrow \\ &\exists m.(M\ B) = m\end{aligned}$$

$$\begin{aligned}\mathbf{SAR} &\triangleq (B, a_1, a_2) \in \mathit{LiveS}(M) \rightarrow \\ &a_1 \leq a < a_2 \rightarrow \exists v.\mathit{read}\ (M\ B)\ a = v\end{aligned}$$

$$\begin{aligned}\mathbf{SAW} &\triangleq (B, a_1, a_2) \in \mathit{LiveS}(M) \rightarrow \\ &a_1 \leq a < a_2 \rightarrow \exists m.\mathit{write}\ (M\ B)\ a\ v = m\end{aligned}$$

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

Figure 5: Call Semantics and Local Variables

that the region from a_1 to a_2 is allocated in the heap of the base B. $\mathit{LiveS}(M)$ is a list of triples (B, a_1, a_2) similarly representing the stack. *LiveS* is map from an oracle to a stack because calls to *stk_free* should only ever deallocate the most-recently-allocated object, so we must maintain their order.

The full set of axioms not already introduced is found in Figure 6. Importantly, allocations are guaranteed to be disjoint from any prior allocations in the same base. (In fact, when targeting a system with a single address space, we further restrict them to be disjoint across all bases.) Addresses in allocated regions are guaranteed successful loads and stores, and once an unallocated address has been successfully accessed it is guaranteed to behave consistently until new memory is allocated anywhere in the system, at which point all unallocated memory again becomes unpredictable.

The *perturb* operation similarly represents the possibility of compiler-generated code using unallocated memory and therefore changing its value or rendering it inaccessible. The only facts 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, but it may happen at other points in the semantics as well.

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 $L(\dots)$-based 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.

$$\textbf{HSTAB1} \triangleq \mathit{LiveH}\ M\ B\ (a_1, a_2) \rightarrow (\mathit{heap\_alloc}\ M\ B\ sz = (\_, M') \vee \mathit{stk\_alloc}\ M\ B\ sz = (\_, M') \vee \mathit{stk\_free}\ M\ B = m') \rightarrow \mathit{LiveH}\ M'\ B\ (a_1, a_2)$$

$$\textbf{HSTAB2} \triangleq \mathit{LiveH}\ M\ B\ (a_1, a_2) \rightarrow \mathit{heap\_free}\ M\ B\ a = M' \rightarrow a \neq a_1 \rightarrow \mathit{LiveH}\ M'\ B\ (a_1, a_2)$$

$$\textbf{SSTAB} \triangleq (\mathit{heap\_alloc}\ M\ B\ sz = (\_, M') \vee \mathit{heap\_free}\ M\ B\ a = M') \rightarrow \mathit{LiveS}(M') = \mathit{LiveS}(M)$$

$$\textbf{HADISJ} \triangleq \mathit{LiveH}\ M\ B\ (a_1, a_2) \vee ((B, a_1, a_2) \in \mathit{LiveS}(M)) \rightarrow \mathit{heap\_alloc}\ M\ B\ sz = (a', M') \rightarrow a' + sz < a_1 \wedge a_2 \leq a'$$

$$\textbf{SADISJ} \triangleq \mathit{LiveH}\ M\ B\ (a_1, a_2) \vee ((B, a_1, a_2) \in \mathit{LiveS}(M)) \rightarrow \mathit{stk\_alloc}\ M\ B\ sz = (a', M') \rightarrow a' + sz < a_1 \wedge a_2 \leq a'$$

$$\textbf{HF} \triangleq \mathit{LiveH}\ M\ (B, a_1, a_2) \rightarrow \mathit{heap\_free}\ M\ B\ a_1 = M'$$

$$\textbf{SF1} \triangleq \mathit{LiveS}(M) = (B, a_1, a_2) :: \_\_ \rightarrow \mathit{stk\_free}\ M\ B\ a_1 = M'$$

$$\textbf{SF2} \triangleq \mathit{stk\_free}\ M\ B\ a_1 = M' \rightarrow \_ :: \mathit{LiveS}(M') = \mathit{LiveS}(M)$$

$$\textbf{PERT1} \triangleq (\mathit{LiveH}\ M\ (B, a_1, a_2) \vee (B, a_1, a_2) \in \mathit{LiveS}(M)) \rightarrow \mathit{perturb}\ M = M' \rightarrow a_1 \leq a < a_2 \rightarrow \mathit{read}\ (M\ B)\ a = v \rightarrow \mathit{read}\ (M'\ B)\ a = v$$

$$\textbf{PERT2} \triangleq \mathit{LiveH}\ M\ (B, a_1, a_2) \rightarrow \mathit{perturb}\ M = M' \rightarrow \mathit{LiveH}\ M\ (B, a_1, a_2)$$

$$\textbf{PERT3} \triangleq \mathit{perturb}\ M = M' \rightarrow \mathit{LiveS}(M) = \mathit{LiveS}(M')$$

Figure 6: Remaining Axioms

$$\begin{aligned} val &::= \ldots \mid Vptr\ B\ a \qquad B \in 2^{\mathcal{C}} \\ e &\in ident \rightharpoonup (2^{\mathcal{C}} \times int) \\ M &\in \mathcal{M} \subseteq 2^{\mathcal{C}} \to mem \\ heap\_alloc &\in \mathcal{M} \to 2^{\mathcal{C}} \to int \rightharpoonup (int \times \mathcal{M}) \\ heap\_free &\in \mathcal{M} \to 2^{\mathcal{C}} \to int \rightharpoonup \mathcal{M} \\ stk\_alloc &\in \mathcal{M} \to 2^{\mathcal{C}} \to int \rightharpoonup (int \times \mathcal{M}) \\ stk\_free &\in \mathcal{M} \to 2^{\mathcal{C}} \rightharpoonup \mathcal{M} \end{aligned}$$

$$\frac{M\ B = m \qquad read\ m\ a = v}{C, M, e \mid *(Vptr\ B\ a) \longrightarrow C, M, e \mid v}$$

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

$$\frac{heap\_alloc\ M\ B\ sz = (p, M')}{C, M, e \mid \texttt{malloc}_B(Vint\ sz) \longrightarrow C, M', e \mid Vptr\ B\ p}$$

Figure 7: Selected Rules for Explicit Sharing

As a consequence of these rules, a compartment can never obtain a $\mathbf{L}(\ldots)$-based 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 let the base of each memory region consist of the set of compartments that have permission to access it. We parameterize each instance of `malloc` with such a set, which will be the base of each pointer that it allocates. We write the identifiers for these `malloc` invocations $\texttt{malloc}_B$. Then we replace the relevant definitions and step rules with those in Figure 7, with call and return steps 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.