An Abstract Model of Compartmentalization with Sharing

Sean Anderson

March 1, 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. 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 machine integers (*int*). 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' local memories are definitely never accessible to other compartments.

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

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

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

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

$$M \in \mathcal{M} \subseteq \mathcal{C} + block \to mem$$

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

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

$$\begin{aligned}
read &\in oracle \to \mathcal{C} + block \to mem \to int \rightharpoonup val \\
write &\in oracle \to \mathcal{C} + block \to mem \to int \to val \rightharpoonup mem \\
heap_alloc &\in oracle \to \mathcal{C} + block \to int \rightharpoonup (int \times oracle) \\
heap_free &\in oracle \to \mathcal{C} + block \to int \rightharpoonup oracle \\
stk_alloc &\in oracle \to \mathcal{C} + block \to int \rightharpoonup (int \times oracle) \\
stk_free &\in oracle \to \mathcal{C} + block \rightharpoonup oracle \\
perturb &\in oracle \rightharpoonup oracle
\end{aligned}$$

Figure 1: Definitions

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

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

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

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

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

$$\frac{fresh\ b \qquad heap\_alloc\ \alpha\ b\ sz = (p, \alpha')}{C, \alpha, M, e \mid \texttt{malloc\_share}(Vint\ sz) \longrightarrow C, \alpha', M, e \mid Vptr\ \mathbf{S}(b, p)\ p}\textsc{MallocLocal}$$

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

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

Figure 2: Selected Memory Steps

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

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

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

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

Figure 3: Arithmetic Operations Involving Pointers

A pointer value consists of a pair: a "base" 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.

Figure **??** shows how loads and stores occur in this system. The pointer's base determines which memory it accesses, and its value is treated as the pointer into that memory. A pointer's base can never be changed via arithmetic—all operations either preserve the base, 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 base. 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 to protect using a page-table-based enforcement mechanism. When proving program properties in this setting, one should quantify over all possible oracles, but when proving correctness of an implementation we restrict ourselves to oracles that reflect its actually behavior.

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 3, 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 2.

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

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

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

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

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

Figure 4: Call and Return Semantics (Continuations ommitted)

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 $(id, \mathbf{L} \mid \mathbf{S}, sz)$. (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 4. 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 Axioms of the Oracle

The allocation oracle hides the behavior of the compiler behind an abstract interface that produces addresses on command. Our semantics behaves nondeterministically in a general sense, but all nondeterministic behaviors are captured by the oracle. Given a particular oracle, the semantics is deterministic.

$$\frac{write\ \alpha\ B\ m\ a\ v = m'}{read\ \alpha\ B\ m'\ a\ v}\textsc{ReadWrite}$$

$$\frac{a \neq a' \qquad write\ \alpha\ m\ a'\ v = m'}{read\ \alpha\ B\ m\ a = read\ \alpha\ B\ m'\ a}\textsc{WriteStable}$$

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad a_1 \leq a < a_2}{\exists v. read\ \alpha\ B\ a = v}\textsc{ReadLiveH}$$

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad a_1 \leq a < a_2}{\exists m'. write\ \alpha\ m\ a\ v = m'}\textsc{WriteLiveH}$$

$$\frac{(B, a_1, a_2) \in LiveS(\alpha) \qquad a_1 \leq a < a_2}{\exists v. read\ \alpha\ B\ a = v}\textsc{ReadLiveS}$$

$$\frac{(B, a_1, a_2) \in LiveS(\alpha) \qquad a_1 \leq a < a_2}{\exists m'. write\ \alpha\ m\ a\ v = m'}\textsc{WriteLiveS}$$

(a) Facts about Reads and Writes

$$\frac{heap\_alloc\ \alpha\ B\ s = (i, \alpha')}{LiveH\ \alpha'\ (B, i, i + s)}\textsc{HeapAlloc}$$

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad heap\_alloc\ \alpha\ B\ s = (i, \alpha')}{i + s < a_1 \text{ or } a_2 \leq i}\textsc{HeapAllocDisjH}$$

$$\frac{(B, a_1, a_2) \in LiveS(\alpha) \qquad heap\_alloc\ \alpha\ B'\ s = (i, \alpha')}{i + s < a_1 \text{ or } a_2 \leq i}\textsc{HeapAllocDisjS}$$

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad heap\_alloc\ \alpha\ B'\ s = (i, \alpha')}{LiveH\ \alpha'\ (B, a_1, a_2)}\textsc{HeapAllocStabH}$$

$$\frac{heap\_alloc\ \alpha\ B\ s = (i, \alpha')}{LiveS(\alpha') = LiveS(\alpha)}\textsc{HeapAllocStabS}$$

(b) Facts about Heap Alloc

$$\frac{stk\_alloc\ \alpha\ B\ s = (i, \alpha')}{LiveS(\alpha') = (B, i, i + s) :: LiveS(\alpha)}\textsc{StkAlloc}$$

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad stk\_alloc\ \alpha\ B'\ s = (i, \alpha')}{i + s < a_1 \text{ or } a_2 \leq i}\textsc{StkAllocDisjH}$$

$$\frac{(B, a_1, a_2) \in LiveS(\alpha) \qquad stack\_alloc\ \alpha\ B'\ s = (i, \alpha')}{i + s < a_1 \text{ or } a_2 \leq i}\textsc{StkAllocDisjS}$$

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad stk\_alloc\ \alpha\ B\ s = (i, \alpha')}{LiveH\ \alpha'\ (B, a_1, a_2)}\textsc{StkAllocStabH}$$

(c) Facts about Stack Alloc

$$\frac{heap\_free\ \alpha\ B\ a = \alpha'}{LiveS(\alpha') = LiveS(\alpha)}\textsc{HeapFreeStabS}$$

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad a < a_1 \text{ or } a_2 \leq a \qquad heap\_free\ \alpha\ B'\ a = \alpha'}{LiveH\ \alpha'\ (B, a_1, a_2)}\textsc{HeapFreeStabH}$$

$$\frac{stk\_free\ \alpha\ B' = \alpha'}{\_ :: LiveS(\alpha') = LiveS(\alpha)}\textsc{StkFreeStabS}$$

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad stk\_free\ \alpha\ B' = \alpha'}{LiveH\ \alpha'\ (B, a_1, a_2)}\textsc{StkFreeStabH}$$

(d) Facts about Free

$$\frac{LiveH\ \alpha\ (B, a_1, a_2) \qquad perturb\ \alpha = \alpha'}{LiveH\ \alpha'\ (B, a_1, a_2)}\textsc{PerturbStabH}$$

$$\frac{perturb\ \alpha = \alpha'}{LiveS(\alpha') = LiveS(\alpha)}\textsc{PerturbStabS}$$

$$\frac{read\ \alpha\ B\ m\ a = v \qquad perturb\ \alpha = \alpha' \qquad LiveH\ \alpha\ (B, a_1, a_2) \qquad a_1 \leq a < a_2}{read\ \alpha'\ B\ m\ a = v}\textsc{PerturbStabValH}$$

$$\frac{read\ \alpha\ B\ m\ a = v \qquad perturb\ \alpha = \alpha' \qquad (B, a_1, a_2) \in LiveS(\alpha) \qquad a_1 \not\leq a < a_2}{read\ \alpha'\ B\ m\ a = v}\textsc{PerturbStabValS}$$

(e) Facts about Perturb

Figure 5: Assertion Rules

Here we axiomitize the oracle. We define two kinds of statements that we can make about the locations of allocated regions. The proposition *LiveH* α (B, a_1, a_2) means that, according to α, the region from a_1 to a_2 is allocated in the heap and belongs to the base B. $LiveS(\alpha)$ is a list of triples (B, a_1, a_2) representing the stack, such that any $(B, a_1, a_2) \in LiveS(\alpha)$ designates a stack allocated block in α. *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 the order.

The full set of axioms is found in Figure 5. Importantly, 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. Addresses in allocated regions are guaranteed successful loads and stores, and loads following successful stores are always successful (and return the stored value) even if outside of allocated regions.

The *perturb* operation mutates the allocator oracle to represent 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.

As a consequence of these rules, a compartment can never obtain a $\mathbf{L}(\dots)$-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.)

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

$$e \in ident \rightharpoonup (2^{\mathcal{C}} \times int)$$

$$M \in \mathcal{M} \subseteq 2^{\mathcal{C}} \to mem$$

$$heap\_alloc \in oracle \to 2^{\mathcal{C}} \to int \rightharpoonup (int \times oracle)$$

$$heap\_free \in oracle \to 2^{\mathcal{C}} \to int \rightharpoonup oracle$$

$$stk\_alloc \in oracle \to 2^{\mathcal{C}} \to int \rightharpoonup (int \times oracle)$$

$$stk\_free \in oracle \to 2^{\mathcal{C}} \rightharpoonup oracle$$

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

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

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

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

Figure 6: Memory With Explicit Sharing

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 6, 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.