# Design Specification

*Memory Management Tools*

---

**Anvil Memory**

*Author: UniquesKernel*

*Date: August 27, 2025*

## Contents

## 1 Introduction to the Problem

Consider a program $P \equiv (I, M)$ where $I$ denotes an ordered sequence of instructions and $M$ denotes an ordered sequence of available memory addresses. Let $A \subset I$ denote the set of allocation instructions and let $B \subset I$ denote the set of deallocation instructions. For $P$ to be temporally memory safe, it must satisfy that for all execution paths $\pi \in \Pi(P)$ there exists a bijective mapping $T_\pi : A_\pi \to B_\pi$. Moreover $P$ exhibits exponential complexity, with a lower bound of $2^{|C|}$ where $C \subset I$ denotes the set of conditional instructions.

Individual tracking of allocation-deallocation pairs admits no systematic method for managing this complexity, as programs with merely $|C| > 10$ conditional instructions yield $2^{|C|} > 1024$ execution paths. This is because such methods operate at the incorrect granularity. Furthermore, traditional approaches such as RAII and reference counting, while offering systematic tracking mechanisms, operate at the same granularity. These methods merely elevate individual allocation management to a higher level of abstraction without reducing the inherent complexity.

We reduce this complexity by observing that instructions in I exhibit temporal dependencies that admit natural groupings, termed **scopes**. Scopes may be nested, creating hierarchical structure between allocation-deallocation pairs. These hierarchical structures cause spatial fragmentation, separating allocation and deallocation pairs in $I$ making verification of the bijective mapping error prone. By managing all allocations within a scope as a unified set, verification complexity reduces from exponential in $C$ to linear verification in the root scope. Since nested scope allocations belong to the root set, destroying the set eliminates all nested allocations, regardless of execution path.

## 2 Program, a Definition From First Principle

**Definition 2.1** Program**.**

Let a program $P$ be defined as a tuple $(\mathcal{M}, I)$, where $I$ is some finite ordered sequence of instructions $(I_n)_{n=0}^{N-1}$ for some $N \in \mathbb{N}$ and $\mathcal{M}$ is a finite ordered sequence of memory locations (addresses) $(M_i)_{i=0}^{M-1}$ for some $M \in \mathbb{N}$.

**Definition 2.2** Program Configuration**.**

We introduce the notion of a configuration of $P$ defined as a pair $\langle \mathcal{V}, \pi \rangle$, where

$$\mathcal{V} : \mathcal{M} \to \mathbb{N}$$

is a function that maps each memory address to its stored value and $\pi \in [0, |I|)$ is a **Program Counter**, denoting the next instruction that should be executed.

Let $\mathcal{S}_P$ be the set of all possible program configurations.

**Definition 2.3** Instruction Semantics**.**

Let $\mathcal{J}$ be the set of all instruction types (e.g., add, jump, load). For each instruction type $j \in \mathcal{J}$, define its semantics as a partial function $[j] : \mathcal{S}_P \rightharpoonup \mathcal{S}_P$ that maps a configuration to the next configuration after executing $j$. (Note: This function may depend on parameters of the instruction, such as memory addresses, which are part of the instruction encoding.)

**Definition 2.4** Transition Relation for a Program**.**

For a program $P = (\mathcal{M}, I)$, the transition relation $\Rightarrow_P \subseteq \mathcal{S}_P \times \mathcal{S}_P$ is defined by:

$$\langle \mathcal{V}, \pi \rangle \Rightarrow_P \langle \mathcal{V}', \pi' \rangle \quad \texttt{iff} \quad [I_\pi](\langle \mathcal{V}, \pi \rangle) = \langle \mathcal{V}', \pi' \rangle,$$

where $[I_\pi]$ is the semantic function of the instruction $I_\pi$.

**Definition 2.5** Atomic Instruction**.**

An instruction $i \in I$ of program $P$ is **atomic** if its execution is represented by a single, indivisible application of the program's transition relation $\Rightarrow_P$.
Formally, instruction $i$ is atomic if for any configuration $S = \langle \mathcal{V}, \pi \rangle$ where $I_\pi = i$, the transition $S \Rightarrow_P S'$ to a next configuration $S'$ represents the complete effect of executing $i$.

**Definition 2.6** Temporality**.**

As per definitions 2.2 and definition 2.4, we have introduced the concept of a configuration on $P$ and the concept of change in a configuration. We introduce the notion of a **tick**, as a discrete temporal event during which a single atomic instruction, see definition 2.5, is executed.

## 3 Memory Safety

## 4 Memory Management

Let a program $P \equiv (i_n)_{n\in\mathbb{N}}$ be an ordered sequence of instructions. Let $A \subset P$ denote the set of allocation instructions and $B \subset P$ the deallocation instructions.

**Definition 4.1** Memory-safe Program**.**

A program $P$ is *memory-safe* if for all execution paths $\pi \in \Pi(P)$:

1. Exists bijection $T_\pi : A_\pi \to B_\pi$ where:
   - $\forall a_i \in A_\pi,\ T_\pi(a_i) = b_i$ is unique.
   - $a_i$ lexically precedes $b_i$ on $\pi$.
2. The deallocation graph $G_\pi = (V_\pi, E_\pi)$ is acyclic.
3. $\text{mem}(a_i) \cap \text{mem}(a_j) = \emptyset$ for $i \neq j$.

### 4.1 Scope-Centralized Memory Management

**Definition 4.2** Memory Arena**.**

A *memory arena* $\mathcal{A} \equiv (\mathcal{M}, F, \Lambda)$ where:

- $\mathcal{M} = \{\mathcal{M}_1, ..., \mathcal{M}_k\}$ - is a finite set of disjoint memory blocks..
- $F \subseteq \bigcup_{i=1}^{k} \mathcal{M}_i$ - is the set of free addresses.
- $\Lambda$ - is an allocator with allocation strategy $\sigma$.

The *allocation function* $H : \mathcal{A} \rightharpoonup \mathcal{A} \times \texttt{addr}(\mathcal{M})$ is:

$$H((\mathcal{M}, F, \Lambda)) = \begin{cases} ((\mathcal{M}, F \setminus \{a\}, \Lambda), a) & \text{if } a \in F \text{ via } \sigma \\ \text{undefined} & \text{otherwise} \end{cases}$$

Subject to invariants:

1. Disjointness: $\mathcal{M}_i \cap \mathcal{M}_j = \emptyset \ \forall i \neq j$.
2. No individual deallocations: $\nexists b_i \in B$ for $a_i \in \mathcal{A}$.
3. Mass reclamation: $\bigcup \mathcal{M}_\pi \subseteq F_\pi^{\text{final}}$ at scope exit.

For control-flow graph $G$ with scope entry/exit nodes $G_a, G_b$:

- Arena instantiation at $G_a$ creates $\mathcal{A} = (\mathcal{M}, F, \Lambda)$.
- All paths $S_{a\to b}$ allocate via $H((\mathcal{M}, F, \Lambda))$.
- Destruction at $G_b$ enforces $F^{\text{final}} = \bigcup \mathcal{M}$, satisfying Definition 4.1.

This formalism reduces verification complexity from $O(2^{|C|})$ paths to $O(1)$ scope-level invariants. The allocator $\Lambda$ implements $\sigma$ through $H$ while maintaining:

$$\begin{aligned}\forall \pi \in \Pi(P), &\ 1.\ T_\pi \text{ bijection} \Rightarrow F^{\text{final}} = \bigcup \mathcal{M} \\ &\ 2.\ \text{Lexical precedence} \Rightarrow \text{Scope nesting} \\ &\ 3.\ \text{Disjointness} \Rightarrow \mathcal{M}_i \cap \mathcal{M}_j = \emptyset\end{aligned}$$

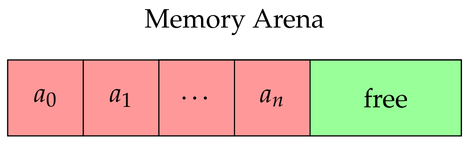

Figure 1: Memory arena structure showing sequential allocations within contiguous blocks.

## 4.2 Initialization-Centralized Memory Management

We now present a further centralization of memory management through what we shall term the "Defer Pattern," which couples allocation and deallocation at the point of initialization.

Utilizing compiler facilities such as the cleanup attribute in GCC, one may specify:

```
MemoryArena *arena DEFER(memory_arena_destroy) = memory_arena_create(...);
```

This construction guarantees deallocation at scope exit, irrespective of the control flow path that leads to said exit.

The Defer Pattern exhibits the following properties:

1. **Spatial Coupling**: The allocation and its corresponding deallocation are textually adjacent in the source code, providing immediate verification of the appropriate $T_\pi(a_i) = b_i$ relationship.

2. **Compiler-Enforced Temporal Sequencing**: The compiler ensures that deallocation occurs at scope exit, enforcing the lexical ordering requirement that $a_i$ precedes $b_i$.

3. **Single-Point Specification**: Memory management decisions are made once, at the point of initialization, rather than distributed throughout the control flow graph.

This approach effectively centralizes memory management responsibilities at initialization points, delegating enforcement to the compiler. The programmer need only ensure proper declaration of cleanup behaviors at allocation sites, substantially reducing the cognitive burden of tracking allocation-deallocation pairs across the program's control flow structure.

We thus observe a progression from decentralized path-based verification, to scope-centralized arena management, to initialization-centralized specification—each stage reducing the cognitive distance between allocation and deallocation logic, and consequently reducing the opportunity for memory management errors.

For compiler-enforced cleanup:

$$\underbrace{\text{MemoryArena}^{*}\mathcal{A}}_{\text{Allocation}} \xrightarrow[\text{Deallocation}]{\text{DEFER}} \underbrace{\mathcal{A} \mapsto (\mathcal{M}, F^{\text{final}}, \Lambda)}_{\text{Scope exit}}$$

where the DEFER macro enforces:

$$\mathcal{A} \mapsto (\emptyset, \bigcup \mathcal{M}, \Lambda)\ \forall \pi \in \Pi(P).$$

## 5 Allocation Strategy

**Definition 5.1** contiguous Memory**.**

Let $\mathcal{M}$ be the set of memory addresses. Let $F$ be the set of free addresses. A subset $C$ of $\mathcal{M}$ is said to be contiguous if $\exists x, y \in \mathcal{M}, C = \{a | x \leq a \leq y, x \leq y\} \wedge C \subseteq F$.

**Definition 5.2** Allocation Strategy**.**

Let $\mathcal{M}$ be the set of memory addresses. Let $F \subseteq \mathcal{M}$ be the set of free memory addresses. Let $A \subseteq \mathcal{M}$ be the set of allocated Memory addresses. Let $n \in \mathbb{N}$ be some natural number greater than zero. Then an allocation strategy $\sigma(F, n)$ is defined as:

$$\sigma(F, n) \rightarrow \phi\left((F \backslash C, A \cup C, C)\right),$$

Where $C \subseteq F$ is some contiguous piece of memory (see 5.1) such that $|C| = n$ and that satisfy some selection policy $\phi$[a] (e.g., first-fit, best-fit, worst-fit) .

[a]We shall intentionally leave $\phi$ as abstractly defined as it is better discussed as an implementation detail.

## References