Design Specification

Memory Management Tools

Anvil Memory

Author: UniquesKernel

Date: March 24, 2025

Contents

1 Memory Management

Let us define a program P as an ordered sequence of instructions $(i_n)_{n\in\mathbb{N}}$. Let us further define $A \subset P$ as the set of allocation instructions (e.g., malloc, calloc) and $B \subset P$ as the set of deallocation instructions (e.g., free). We shall say that a program P is memory-safe (exhibits neither memory leaks nor double frees) if and only if:

Definition 1.1 Memory-safe Program (no leaks or double frees)**.**

1. For every static control-flow path π (a path through the program, abstracting away loops and recursion), there exists a bijective mapping $T_\pi : A_\pi \to B_\pi$ where:
   - $A_\pi \subset A$ denotes the set of allocations on π.
   - $B_\pi \subset B$ denotes the set of deallocations on π.
   - $\forall a_i \in A_\pi \exists! b_i \in B_\pi : T_\pi(a_i) = b_i$.
2. $\forall (a_i, b_i) \in T_\pi$, a_i lexically precedes b_i on π.
3. Let G denote a graph wherein nodes represent allocation-deallocation pairs (a_i, b_i), and edges represent cleanup dependencies (i.e., b_i cannot be executed unless b_j has already been executed). The graph G must be acyclic for all π.
4. $\forall (a_i, b_i), (a_j, b_j) \in T_\pi$, the memory regions allocated by a_i and a_j must be disjoint.

The aforesaid definition, although mathematically precise, does not account for the dynamic nature of memory allocation through loops and recursion. Nevertheless, it suffices for our present analysis of memory management approaches in languages with explicit memory control, particularly *C*. We shall now examine three successively more centralized approaches to the problem of memory management.

1.1 Decentralized Path-Based Memory Management

Consider a program $P = (i_n)_{n\in\mathbb{N}}$ with a set $C = \{i_c \in P | i_c \text{ is a conditional instruction}\}$ of conditional branch instructions. The number of distinct paths S in the static control-flow graph satisfies the inequality $S \geq 2^{|C|}$.

In the decentralized approach, each path π through the program must independently satisfy the bijective mapping T_π between allocations and deallocations. As $|C|$ increases, the complexity grows exponentially. With merely ten independent conditional statements, one must verify memory correctness across 2^{10} distinct paths. For non-trivial programs where $|C| \gg 10$, this verification becomes combinatorially intractable.

When $|C| = 20$, the programmer must ensure correct memory management across approximately 10^6 paths. Should any single path fail to satisfy Definition 1.1, the program manifests either memory leaks or erroneous double-free operations. Thus, the decentralized approach imposes a verification burden that grows exponentially with program complexity.

1.2 Scope-Centralized Memory Management

We now introduce the concept of memory arenas as a mechanism for centralizing memory management at the scope level, thereby reducing the complexity of the verification problem.

For a program P with control-flow graph G, let us observe that for any well-structured scope, there exist entry and exit nodes G_a and G_b such that all paths $S_{a \to b}$ between these points diverge from G_a and reconverge at G_b.

A memory arena establishes a form of virtual allocation wherein we allocate from a pre-allocated block belonging to the arena rather than directly from the system. If an arena A is constructed at G_a and destroyed at G_b, then the deallocation of A propagates retrospectively through all paths $S_{a \to b}$, releasing all virtual allocations derived from A.

This transformation centralizes memory management from a per-path concern to a per-scope concern. Rather than maintaining T_π for each path π, we need only ensure proper arena lifetime management at scope boundaries. The exponential path complexity is thus reduced to linear scope complexity.

For optimal application of this approach, an arena should remain confined to its originating scope. If extension beyond this scope is necessary, the arena's instantiation should be elevated to an enclosing scope with appropriate lifetime characteristics.

1.3 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.

2 Memory Arena

References