Selective GC Zoning for Dynamic Compatibility in Static Languages: A Case Study in Mojo

By YoungGwan Jung

Abstract

Modern languages like Mojo strive to combine Python syntax compatibility with static, high-performance execution. However, dynamic features such as `eval()` and `globals()`, which involve runtime object creation, mutable scopes, and potential cyclic references, are naturally at odds with a strictly static memory model. To address this challenge, this paper proposes **Selective GC Zoning**, a strategy that isolates dynamic features within a dedicated garbage-collected subspace (GC Zone), while preserving the static memory model for the rest of the code. This design enables near-native performance for general Mojo code while allowing selective Python compatibility for specific dynamic features. In addition, we discuss object escape constraints between static and GC-managed regions, concurrency scenarios, different GC algorithm choices, and future extensions of this approach.

---

1. Introduction

Mojo is a modern language that inherits Python's intuitive syntax while aiming for system-level performance via static typing and AOT (Ahead-Of-Time) compilation [7]. However, Pythonic features (e.g., `eval()`, `globals()`) introduce dynamic object creation, mutable namespaces, and potential cycles—incompatible with the deterministic memory management typically found in static systems [1][9].

Selective GC Zoning addresses this tension by confining dynamic language constructs to a limited scope. Specifically:

1. **General Mojo code:** Maintains static memory strategies (ownership, Arena/Rc, compile-time checks).

2. **GC Zone:** Activated only when using dynamic constructs (`eval()`, `globals()`), employing a specialized garbage collector within that zone.
3. **Zone lifecycle:** Once the dynamic operation completes, the zone and all its objects are collected, preventing overhead from spreading into the static domain.

This paper outlines the design, implementation considerations, and performance implications of this approach.

---

2. Related Work

2.1 Memory Management in Existing Languages

- **Python (CPython):** Uses reference counting with a cycle-detecting garbage collector [1].
- **Rust:** Avoids GC by leveraging strict ownership and lifetime rules [2].
- **Swift:** Uses ARC (Automatic Reference Counting) for class types [3].
- **JavaScript (V8):** Employs full-heap GC, optimized by inline caching and hidden classes [4].
- **C/C++ (via Boehm GC):** Can use conservative GC [5], though it is not part of the official language standard.

2.2 Partial or Selective GC Concepts

- Some systems or embedded script engines isolate dynamic scripting components in a separate memory region [6].
- The core novelty here lies in applying a “zoned” GC approach to Mojo, so that dynamic features operate in a delimited GC space, preserving the static domain for high-performance AOT-compiled code.

---

3. Selective GC Zoning: Overview

3.1 Core Concept

- **Static Code Realm:** Ordinary Mojo logic with Arena/Rc or ownership-based memory management.
- **GC Zone:** Dedicated region for `eval()`, `globals()`, and other dynamic needs, featuring garbage collection.

3.2 Key Characteristics

1. **Dedicated Heap (GcHeap):** Allocations for dynamic objects are stored in a separate GC-managed heap.
2. **Restricted Scope:** The GC Zone is initiated when calling `eval()` or `globals()`; it remains active only for that scope.
3. **Object Escape Prevention:** A compiler-level mechanism ensures that objects from the GC Zone cannot “leak” into the static domain in ways that violate memory safety.
4. **Automatic Cleanup:** When the zone concludes, a GC pass (Mark-and-Sweep or RC+Cycle) reclaims the memory of the dynamically created objects.

---

4. Architecture & Implementation

```
[Static Code] -> [GC Zone (eval/globals)]
  | Arena/Rc,        | GcHeap, GcObject,
  | Ownership         | cleanup on exit
```

4.1 Data Structures & Example Functions

```
struct GcZone:
    var heap: GcHeap
    var env: Dict[String, GcObject]

    fn eval(code: String) -> GcObject:
        // Internally parses the code and stores objects in GcHeap
        ...
        return result

    fn cleanup():
        // Trigger GC when the zone is no longer needed
        self.heap.collect_all()
```

Basic Example

```
fn static_function() -> Int:
    let code = "3 + 4"  # A Python expression
    let zone = GcZone()
    let result_obj = zone.eval(code)
    let result_int = result_obj.unwrap_int()  # Safely extract an
integer
    zone.cleanup()  # Cleanup once evaluation is done
    return result_int
```

In this snippet, `static_function()` resides in Mojo's static environment but uses `eval()` to interpret a Python expression dynamically. The resulting object (`GcObject`) remains in the GC Zone; only primitive data (like `Int`) is extracted and passed back to static code, preventing unsafe references to GC-managed data.

4.2 Escape Analysis & Compiler Enforcement

- **Escape Analysis:** At compile time, ensure that GC-managed objects do not leak into the static domain in a way that would break static memory guarantees.
- **Primitive Extraction (Unwrapping):** Methods like `unwrap_int()`, `unwrap_string()` are used to retrieve only primitive values from GC objects.
- **Static ↔ GC Zone Interface:** Strings, numbers, and booleans may be copied safely from the GC zone; more complex data structures remain in the zone, or they become invalid once the zone terminates.

These constraints ensure that dynamic objects cannot persist beyond the lifetime of their GC Zone, preserving the safety and predictability of the static code base.

4.3 Selecting a GC Algorithm

Within the GC Zone, there are two primary options:

1. **Mark-and-Sweep**
   - **Pros:** Straightforward handling of cycles, robust for arbitrary object graphs.

- **Cons:** Potential for noticeable pause times if the zone contains many objects.

2. **Reference Counting + Cycle Detection**
   - **Pros:** Reclaims unused objects promptly as reference counts drop.
   - **Cons:** Requires an additional cycle detection mechanism, as reference counting alone cannot handle cycles.

If each call to `eval()` yields only a small set of objects, Mark-and-Sweep may be sufficient. In scenarios with numerous or long-lived objects within the GC Zone, RC+Cycle might offer more responsive collection. The choice can be context-dependent and may evolve as Mojo's dynamic ecosystem matures.

---

5. Performance & Concurrency

5.1 Theoretical Performance Analysis

Let:

- `n` = number of dynamic zone invocations (e.g., `eval()` calls),
- `o` = average number of objects per zone,
- `t_gc(o)` = time to collect `o` objects.

Then total GC overhead is:

Tgc*total*≈*n*×*tgc*(*o*)*T*{\mathrm{gc_total}} \approx n \times t_{\mathrm{gc}}(o)

In most usage patterns, `eval()` would be invoked relatively infrequently, and the number of objects per invocation (`o`) would remain small. Thus, the overall GC overhead becomes negligible relative to the rest of the static code's execution. Selective GC Zoning localizes garbage collection to specific dynamic tasks, safeguarding overall performance.

5.2 Simple Benchmark / Simulation Outline

- **Scenario 1:** 90% static code, 10% dynamic calls (short `eval` scripts).
- **Scenario 2:** 50% static code, 50% dynamic calls (larger scripts).

Metrics:

- GC time as a percentage of total runtime.
- Frequency of GC triggers and memory usage patterns.
- Performance degradation when `n` or `o` grow significantly.

Even rudimentary tests can demonstrate that minimal `eval()` usage leads to negligible GC costs and that excessive or unplanned dynamic calls can indeed accumulate overhead.

5.3 Concurrency Considerations

If Mojo supports multithreading, handling concurrent accesses to a GC Zone poses additional challenges:

- **Stop-the-world GC:** Simplifies implementation but halts all threads while collection occurs.
- **Parallel GC:** Multiple threads may collect the GC Zone in parallel, requiring synchronization mechanisms or advanced concurrent GC strategies.
- **Restricted concurrent eval:** The language could forbid multiple threads from entering the same GC Zone at once, thereby reducing complexity at the cost of reduced parallelism.

This paper primarily focuses on a single-threaded environment, leaving parallel and concurrent GC as a potential area for future exploration.

---

6. Zone Lifecycle & Reusability

6.1 Zone Lifecycle

1. **Initialization:** `eval()` or `globals()` call triggers the creation of a new GcZone.
2. **Active Use:** Objects are allocated, modified, or returned within the GC Zone.
3. **Cleanup:** At scope exit, the system calls `cleanup()`, performing a final GC pass.
4. **Deactivation:** The zone is fully released, returning control to static code.

6.2 Reusability & Nested Zones

- **Immediate Destruction:** A new GC Zone is created and destroyed for each `eval()`, which is straightforward but may become costly if `eval()` is called very frequently.
- **Sticky/Long-lived Zone:** The zone is kept active across multiple invocations, potentially retaining objects between calls. This saves overhead on repeated creation but can grow memory usage over time.
- **Nesting:** One might call `eval()` within another `eval()`, creating nested zones. Managing multiple concurrent zones complicates lifetime tracking and garbage collection, so an initial implementation might disallow or tightly restrict nesting.

---

7. Limitations & Future Work

1. **DSL Restrictions:** The dynamic subset for `eval()` might omit certain Python features (like async, closures) to reduce complexity.
2. **GC Object Escape Analysis:** Enforcing non-escape at compile time is critical for safety but adds complexity to the language and runtime.
3. **Parallel/Concurrent GC:** This paper assumes single-threaded execution; parallel or concurrent strategies will require additional mechanisms.
4. **Multiple GC Zones:** Handling inter-zone references or object exchange is a non-trivial challenge for advanced usage scenarios.
5. **Zone-Aware JIT:** Future work could integrate JIT compilation strategies that optimize code within the GC Zone.

---

8. Conclusion

Selective GC Zoning offers a practical way to integrate Python-like dynamic functionality into the otherwise static, high-performance Mojo environment. By confining runtime-evaluated code to a specialized GC subspace, the rest of Mojo retains static safety and predictable performance. The incremental overhead applies only where dynamic behavior is truly needed—aligning with the principle of “pay only for what you use.”

Our design illustrates how a low-level, static-oriented language can selectively accommodate dynamic features without universally adopting a garbage-collected model. Future research will explore concurrency support, nested or parallel GC Zones, and zone-aware JIT optimizations to extend the approach to a broader set of use cases.

---

9. References

[1] Guido van Rossum et al. The Python Language Reference Manual. Python Software Foundation.

[2] Matsakis, N. D., & Klock, F. S. (2014). The Rust Language. https://www.rust-lang.org

[3] Apple Inc. Swift Programming Language Guide. https://developer.apple.com/documentation/swift

[4] Google V8 Team. V8 JavaScript Engine Internals. https://v8.dev/

[5] Boehm, H. J. (2004). A garbage collector for C and C++. http://www.hboehm.info/gc/

[6] Serrano, M., Gallesio, E., & Loitsch, F. (2006). Heap management and garbage collection in dynamic languages. Journal of Functional Programming.

[7] Modular Inc. Mojo Language Docs. https://docs.modular.com/mojo/

[8] Chisnall, D. (2020). C is not a low-level language: Your computer is not a fast PDP-11. Communications of the ACM, 63(11), 78–85.

[9] Jones, R., Hosking, A., & Moss, E. (2011). The Garbage Collection Handbook: The Art of Automatic Memory Management. CRC Press.