Lab 5 – Mark-Sweep Garbage Collector

1. Introduction

Modern virtual machines rely on automatic memory management to simplify programming and prevent memory-related errors. In this lab, the bytecode virtual machine developed in Lab 4 is extended with a stop-the-world Mark–Sweep Garbage Collector (GC).

The goal of this lab is to design, integrate, and evaluate a basic garbage collector that can correctly reclaim unreachable heap objects while preserving all live data.

2. Motivation for Garbage Collection

Manual memory management using malloc/free is error-prone and can lead to memory leaks, dangling pointers, and double frees.

A garbage collector automates memory reclamation by identifying which objects are still reachable by the program.

The Mark–Sweep algorithm is one of the simplest and most widely studied GC techniques, making it well-suited for educational virtual machines.

3. Overall GC Design

The implemented garbage collector follows a classic stop-the-world Mark–Sweep design. During GC execution, program execution is paused, and the heap is traversed in two phases: mark and sweep.

All heap objects are maintained in a linked list, which allows linear traversal during the sweep phase.

4. Root Identification Strategy

Root identification is a critical step in garbage collection. In this VM, the operand stack serves as the root set. Each value on the stack is examined, and if it represents a heap object, that object is treated as a root.

This strategy ensures that all objects directly accessible by the running program are preserved.

5. Mark Phase

In the mark phase, the collector traverses all objects reachable from the root set. Each object contains a marked flag, which is initially false. When an object is visited, it is marked and its referenced child objects are recursively visited.

The marked flag prevents infinite traversal in the presence of cyclic references.

6. Sweep Phase

After marking is complete, the sweep phase scans the heap list. Any object that remains unmarked is considered unreachable and is safely freed.

Objects that survive the sweep are unmarked to prepare for the next garbage collection cycle. This phase ensures that memory occupied by dead objects is reclaimed.

7. Integration with the Virtual Machine

The garbage collector is tightly integrated with the virtual machine.

Heap allocation is performed through a dedicated allocation function, and the GC can be explicitly invoked using a gc(vm) call. This design allows both test programs and the VM itself to trigger garbage collection when needed.

8. Correctness and Testing

A comprehensive set of test cases was used to validate correctness. These tests cover reachable and unreachable objects, transitive reachability, cyclic references, closure-like object graphs, and heap cleanup with no roots.

All tests passed, demonstrating that the GC correctly distinguishes live objects from garbage in all scenarios.

9. Performance and Stress Evaluation

Performance evaluation was conducted by allocating large numbers of objects and explicitly invoking garbage collection.

The results show that execution time increases linearly with the number of heap objects, which is expected for a Mark–Sweep collector. Despite its simplicity, the GC remains stable under heavy allocation pressure.

10. Limitations

The current implementation is a stop-the-world collector and does not support incremental or generational garbage collection.

Additionally, marking is implemented recursively, which may limit scalability for extremely deep object graphs.

11. Conclusion

This lab successfully demonstrates the integration of a Mark–Sweep garbage collector into a stack-based virtual machine.

The collector correctly reclaims unreachable memory, handles cycles and complex object graphs, and satisfies all functional requirements of the assignment. While simple, this implementation provides a strong foundation for exploring more advanced garbage collection techniques.