

Lab 5 Report: Mark-Sweep Garbage Collector

January 16, 2026

1 Introduction

This report presents the design and implementation of a stop-the-world mark-sweep garbage collector integrated into the bytecode virtual machine from Lab 4. The collector manages heap-allocated objects based on reachability from VM roots. The goal is a correct, deterministic GC that works with the VM instruction set and object model used in the project.

2 Objective

The collector must:

- Allocate and track heap objects.
- Discover roots from VM runtime state.
- Mark all objects reachable from the roots.
- Sweep unreachable objects and reclaim memory.
- Remain safe under stress and deep object graphs.

3 Design Methodology

The GC design follows a minimal and auditable approach:

- A single global heap list records all live allocations.
- All runtime values are boxed so references are explicit.
- Marking is iterative (no recursion) to avoid call-stack overflow.
- Sweeping is linear in the number of heap objects.

This provides a clear and deterministic execution model that is easy to test.

4 System Architecture

4.1 VM Components

The VM is stack-based and executes bytecode with a fixed instruction set. Execution state includes:

- Program counter (PC).
- Operand stack (values used by instructions).
- Call stack (return addresses for `CALL/RET`).
- Global memory (STORE/LOAD slots).

4.2 Object Model

The object model is minimal but sufficient for Lab 5 requirements:

- **OBJ_INT**: boxed integer.
- **OBJ_PAIR**: pair with left and right references.
- **OBJ_FUNCTION**: placeholder for closure tests.
- **OBJ_CLOSURE**: closure referencing a function and environment.

Each object carries a mark bit and a link in the global heap list.

4.3 Value Representation

Values are boxed to make object references explicit:

- **VAL_NIL**: empty value (no reference).
- **VAL_OBJ**: pointer to a heap object.

This makes root discovery and traversal uniform for both stack and memory slots.

5 Root Discovery

Root discovery scans the VM state for references:

- Operand stack entries currently in use.
- Global memory slots.

The VM exposes a single `vm_visit_roots` callback that iterates these locations and invokes a visitor for each object reference.

6 Mark Phase

The mark phase uses an explicit stack to avoid recursion. The algorithm is:

```
mark_from_roots():
  for each root:
    push(root)
    while stack not empty:
      obj = pop()
      if obj is already marked: continue
      mark obj
      for each child in obj:
        push(child)
```

Child enumeration is implemented by `obj_visit_children` based on object kind. For `OBJ_PAIR`, both left and right are visited; for closures, the function and environment are visited.

7 Sweep Phase

Sweeping walks the heap list:

- If an object is unmarked, it is removed from the list and freed.
- If an object is marked, its mark bit is cleared for the next cycle.

This guarantees that any object not reachable from roots is reclaimed.

8 Instruction Extensions

To build object graphs from bytecode, the VM adds:

- **PAIR**: pop two values and allocate a pair.
- **LEFT**: pop a pair and push its left value.
- **RIGHT**: pop a pair and push its right value.

These opcodes are implemented in the assembler, loader, and VM executor, and are used by the test suite to validate object traversal.

9 Correctness and Safety

9.1 Runtime Checks

The VM validates:

- Stack overflow/underflow.
- Division by zero.
- Invalid jump addresses.

- Invalid memory indices.
- Invalid opcodes in bytecode.

These checks prevent undefined behavior and make failures explicit.

9.2 GC Safety

The GC is stop-the-world, so no object mutation occurs during marking. All allocations are linked into the heap list. Root discovery is performed through a single VM entry point, which keeps GC logic isolated from VM execution.

10 Testing Methodology

Testing is structured to match Lab 5 requirements.

10.1 VM Opcode Suite

The opcode suite validates VM execution behavior (arithmetic, jumps, calls, memory, and error conditions). It also includes object opcodes: `PAIR`, `LEFT`, and `RIGHT`.

10.2 GC Interactive Suite

An interactive suite `VM/test.c` covers:

- Basic reachability and unreachable collection.
- Transitive reachability.
- Cycles.
- Deep object graphs.
- Closure capture.
- Stress allocation.

10.3 GC Automated Suite

The automated test `test.c` exercises:

- Reachability and collection after root removal.
- Cycles and safe reclamation.
- Rooting via VM memory slots.
- Deep chains of pairs.
- Stress allocation and full sweep.

11 Test Results

Test	Result
VM opcode suite	PASS
GC interactive suite (VM/test.c)	PASS
GC automated suite (test.c)	PASS

12 Performance Evaluation

The VM reports execution time per program and instruction counts. GC tests were run on the development machine using the provided suites. The results indicate correctness under stress and no observable errors in memory safety checks.

13 Limitations and Enhancements

13.1 Current Limitations

- Fixed-size VM stacks and memory arrays.
- Fixed-size mark stack for traversal.
- Manual GC triggering rather than threshold-based triggering.

13.2 Future Enhancements

- Dynamic mark stack growth for very wide graphs.
- Allocation thresholds to trigger GC automatically.
- Generational or incremental GC to reduce pause times.

14 Conclusion

The system integrates a correct stop-the-world mark-sweep collector into the bytecode VM. The design uses boxed values, explicit root discovery, and iterative marking to ensure correctness and robustness. The solution meets Lab 5 requirements and is verified by both interactive and automated tests.