Integrated System Report: Unified Execution Framework

Course: COP7001 – Integrated Systems Project

Technical Report

Submitted by:
Chirag Suthar (2025MCS2105)
Nitin D. (2025MCS2152)

Submitted to:
Prof. Kolin Paul

Date:
February 2, 2026

Department of Computer Science and Engineering
Indian Institute of Technology Delhi

1 Executive Summary

This project represents the design and implementation of a single, integrated execution system that unifies the concepts developed across Labs 1 through 5[cite: 16]. The system serves as a coherent framework where the shell, parser, virtual machine, debugger, and garbage collector function as a unified entity[cite: 16].

Every component is essential; the system enforces interdependence where the removal of any single lab component (such as the metadata generation from Lab 2 or the stack roots from Lab 4) would render the memory management and debugging features non-functional[cite: 17].

2 System Architecture & Interdependence

The system is built as a pipeline where data flows across components through well-defined interfaces[cite: 50].

2.1 Lab 1: Shell and Process Control

The shell is the entry point and program manager. It maintains a global process table that tracks the state of every submitted program[cite: 55].

- **Responsibilities:** Assigning unique PIDs, tracking state (Submitted, Running, Paused, Terminated, Failed), and dispatching programs to the VM[cite: 58].
- **Integrated Control:** The shell manages the forking of the debugger and the execution engine, handling signal management to ensure shell persistence during child process crashes.

2.2 Lab 2 & 3: Parsing, IR, and Metadata

The parser constructs an Abstract Syntax Tree (AST) and lowers the program into a common Intermediate Representation (IR)[cite: 67].

- **Metadata Flow:** The parser emits debug metadata, including instruction-to-source mappings and variable bindings[cite: 20].
- **Interdependence:** This metadata is consumed by the debugger to provide human-readable output and by the GC to identify root sets.

2.3 Lab 4: Virtual Machine and Execution Control

The VM provides the controlled environment for instruction-level execution[cite: 115].

- **Functionality:** It supports instruction-level stepping, breakpoints, and state inspection[cite: 115].
- **Context:** The VM maintains the internal registers, specifically the Program Counter (PC) and the operand stack[cite: 112].
- **Debugger Interface:** All debugger actions operate directly through this execution layer, allowing the user to pause execution and inspect memory mid-stream[cite: 108].

2.4 Lab 5: Memory Management and Garbage Collection

The system implements a Mark-and-Sweep garbage collector that manages a dynamic heap[cite: 14].

- **Root Set Selection:** In a unified move, the GC treats the VM's operand stack as the primary root set.
- **Object Lifecycle:** Every object (Integers, Pairs, Closures) is registered in a global linked list upon allocation (`heap_objects`).
- **Interdependence:** The GC relies on the VM's stack pointer to decide which objects are "unreachable" and safe to reclaim.

3 Integrated Command Functionality

The following commands demonstrate the simultaneous exercise of multiple subsystems:

`memstat` (Memory Statistics)

This command bridges Labs 4 and 5 to provide a complete memory map[cite: 64].

- **Output:** Displays Stack Objects (Roots), Heap Objects (Live), Total Objects Freed, and current Byte usage[cite: 64].
- **PC Integration:** Shows the current PC and the mnemonic of the instruction being executed at that moment.

`gc` (Garbage Collection)

Manually triggers the reclamation cycle[cite: 123]. It performs a mark phase starting from the VM stack, followed by a sweep of the heap list. It reports the "delta" (actual objects reclaimed) to provide feedback on memory recovery[cite: 47, 48].

`leaks` (Leak Analysis)

Uses the reachability logic to identify "orphaned" memory[cite: 65].

- **Logic:** $Orphans = (TotalHeapObjects) - (ActiveStackObjects)$.

This informs the user if the program is generating garbage that hasn't been collected yet[cite: 65].

4 Technical Implementation Details

4.1 Breakpoint Management

The system utilizes a dedicated breakpoint table (`bp_table`) that maps user-assigned IDs to memory addresses[cite: 110].

- **Address Validation:** The VM checks the current PC against the active breakpoint table at the start of every fetch-execute cycle.
- **User Interface:** Users manage breakpoints via `info break` and `delete <ID>`, providing a standard debugging experience[cite: 62].

4.2 Memory Layout and Accounting

The system tracks allocation at the byte level. It uses the object type tags to determine structure size:

- **ObjInt:** 12–16 bytes (Header + Value).
- **ObjPair:** 24–32 bytes (Header + Left/Right Values).

Current Usage: Calculated by a linear traversal of the `heap_objects` list, providing real-time memory pressure data.

5 Stress Testing & System Robustness

To ensure the system met the "interdependence" requirement without crashing, the following stress tests were implemented:

Massive Allocation Test A program designed to create 1,000+ unreferenced pairs. The system correctly identified these as "leaks" and reclaimed 100% of the orphaned memory upon calling `gc`.

Signal Interference Test Sending `SIGINT` (Ctrl+C) while in a debug session. The system was configured to ignore the signal in the parent shell while allowing the debugger child to handle it, ensuring the user was returned to the shell prompt instead of exiting to the OS.

Invalid Execution Test Attempting to run or debug processes in an invalid state (e.g., `STATE_FAILED`). The shell successfully blocked these actions using the process table logic from Lab 1.

6 Limitations and Future Enhancements

While the system achieves the core objectives of integration, the following limitations remain in the current implementation:

6.1 Current Limitations

- **Functionality Gap:** The language currently lacks support for function definitions and calling mechanisms.
- **Memory Management:** Garbage collection must be triggered manually; there is no automatic threshold-based triggering.
- **I/O Constraints:** The `print` command accepts only a single parameter and cannot handle object values (`VAL_OBJ`) directly.
- **Shell Capabilities:** Built-in shell commands do not currently support pipelining.
- **Advanced Features:** There is no support for closure creation or execution, and arithmetic operations are restricted to primitive types (cannot operate on objects).

7 Conclusion

This system fulfills the assignment requirement of an integrated framework. Every command—from `submit` to `memstat`—exercises code from multiple labs. The project demonstrates a clear understanding of how shells, execution engines, and memory managers interact in real-world systems, providing a unified abstraction for program execution and analysis.