The Toy Virtual Machine Architecture

Design and Implementation Details

Adrià Martín Martín (2398893)
Mathematics and Computer Science
Technische Universiteit Eindhoven
Eindhoven, The Netherlands
a.martin.martin@student.tue.nl

Pau Jaso Molina (1669335)
Mathematics and Computer Science
Autonomous University of Barcelona
Bellaterra, Barcelona
pau.jaso@autonoma.cat

***Abstract*—In this report, we present the design and implementation of the Toy Virtual Machine (ToyVM), a stack-based bytecode interpreter for the Toy language, developed as an independent reimplementation of the reference Simple Language runtime. The goal of the project was to build a fully functional interpreter capable of executing Toy scripts while maintaining a modular, extensible, and correct architecture. The report describes the main design properties of the interpreter, including its fetch–decode–execute loop, dynamic value model, function call handling, and error management.**

***Index Terms*—Abstract Syntax Tree (AST), bytecode interpreters, dynamic typing, inline caches, hidden classes, virtual machines**

I. Introduction

The Toy Virtual Machine (ToyVM) was developed as part of the “Language Virtual Machine Design & Implementation” course to explore the internal architecture of modern dynamic language runtimes. The project’s objective is not only to execute programs written in the Toy language, but also to provide insight into how high-level code is parsed, compiled into bytecode, and executed by a virtual machine as it is done in other industry standard laguages. The Toy language and its virtual machine serve as a simplified model of real-world interpreters such as those used in JavaScript with its V8, CPython, Truffle-based systems or Java (JVM), enabling a clear view of how Abstract Syntax Trees, bytecode formats, and runtime values interact in practice.
The implementation passes all the tests and only has the Object storage & inline cache.

II. ToyVM Pipeline

The execution pipeline of the ToyVM follows a multi-stage process. First, the source script is parsed into a sequence of tokens, which are then organized into the structured hierarchy of the AST representing the syntax and semantics of the program. Each function in the script corresponds to its own AST, which is traversed in-order to emit bytecode instructions through a shared `CompileContext` instance. This compilation phase encodes the program into a compact BCI (bytecode) format that captures control flow and operations at a low level. At runtime, the virtual machine invokes the `main` function entry point, initializing the interpreter loop. The VM then fetches, decodes, and executes bytecode instructions sequentially, managing intermediate values through a dedicated operand stack and maintaining function state via lightweight call frames. This architecture combines simplicity and extensibility, providing a solid foundation for further performance-oriented enhancements.

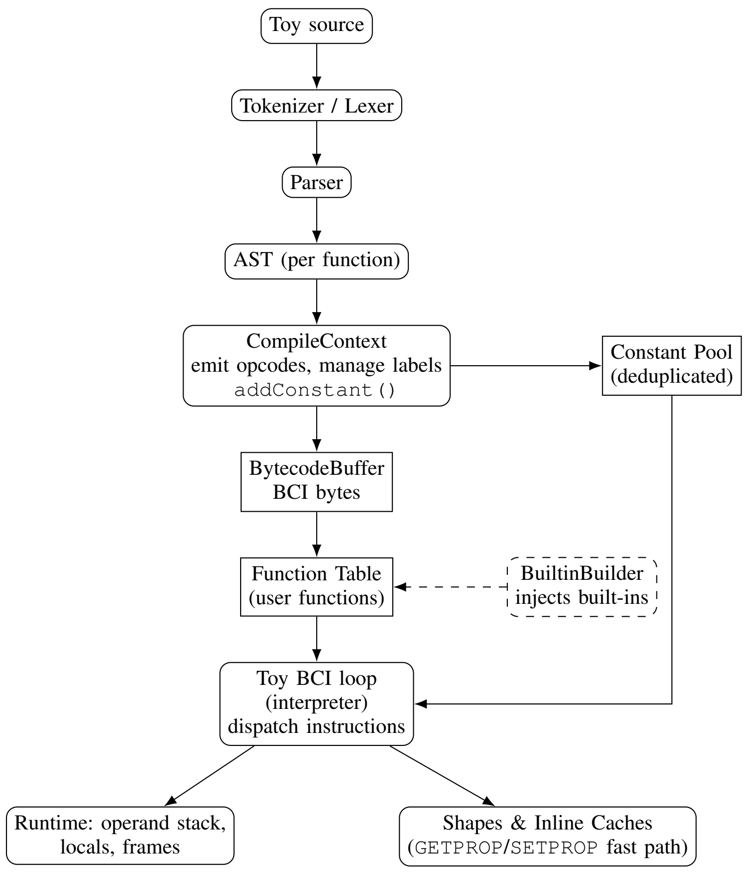

Fig. 1. ToyVM pipeline: from source to bytecode (CompileContext, constant pool, BytecodeBuffer), then interpreter execution with runtime state and IC/shapes.

III. Architecture Overview

A. Bytecode Compilation Process

Node-level code generation: Transformation of high-level AST representation into executable bytecode is driven by an explicit compilation phase. Each node is responsible for emitting the appropriate sequence of bytecode instructions that represent its semantics. To support this, the abstract class `ToyStatementNode`, from which all functional Toy nodes inherit, was extended with a new abstract `compile(CompileContext ctx)` method. This method is overridden by every concrete node class to implement its own code generation logic. Nodes that contain child expressions (such as binary operations, conditionals, or loops) call the `compile` method of their sub-nodes first, following an in-order traversal that guaranties the correct evaluation order and stack layout before emitting their own instruction.

Constant pool: to store complex or reusable objects—such as strings, function references, and big numeric literals—outside the main bytecode stream. Instead of embedding large values directly into the instruction sequence, constants are added to a dedicated pool. Each unique object is assigned a 16-bit index, and duplicate entries are automatically avoided using an internal lookup map. At runtime, the interpreter resolves these indices against the shared constant pool loaded with the function's bytecode.

The compiler mainly emits stack ops (**`PUSH_I64`**, **`PUSH_K`**), locals (**`LOAD_L`**, **`STR_K`**), control flow (**`JMP`**, **`JNE`**), calls/returns (**`CALL`**, **`RET`**), arithmetic, and property access (**`GETPROP`**, **`SETPROP`**).

B. Execution Model

This VM implementation executes bytecode using a classical fetch–decode–execute loop implemented as a single `while(true)` over a read-only byte array. At each iteration the interpreter reads one opcode byte at `pc`, increments `pc`, and dispatches via a `switch` statement to the corresponding handler:

```
    byte op = code[pc++]; switch (op) {
    ...}
```

The behaviour of each instruction are implemented in small helper methods (e.g., `opPUSH_I64`, `opJNE`, `opCALL`), which improves readability.

Calls are initiated by `CALL`, which pops an expected `VFunction`, decodes the arity operand, collects arguments from the operand stack, pads missing parameters with `NULL`, and invokes the corresponding `RootCallTarget`. The result (or `NULL`) is boxed and pushed back onto the operand stack. Program termination is signaled by `RET`, which returns the top-of-stack (or `NULL` if empty).

C. Runtime Stack and Locals

The runtime of the VM is stack-based, where all intermediate values, function arguments, and results are passed through a dedicated call stack. Each instruction consumes or produces one or more stack elements, following the classical postfix evaluation discipline. For example, binary arithmetic operators (`ADD`, `SUB`, `MUL`, `DIV`) pop two operands and push a single result, while `CALL` pops a variable number of arguments and the function reference itself.

In addition, every function activation maintains a fixed-size `Locals` array used to store local variables and temporary bindings. The current local environment is allocated upon function entry and deallocated automatically when the `RET` instruction is executed. Variable access is performed using `LOAD_L` and `STR_K`, which load and store values in the local slot specified by a 16-bit index. This design keeps variable lookups constant-time and avoids repeated name resolution during execution.

D. Type (Un)boxing – 'Value' System

To handle the dynamic typing model in a type-safe way, the VM represents all runtime data through a unified interface called `Value`. Each concrete runtime type—such as integers, bigintegers, booleans, strings, objects, and functions—is implemented as a subclass (`VLong`, `VFloat`, `VBool`, `VString`, `VObject`, `VFunction`, etc.). This model allows every value manipulated by the interpreter to share a common interface for operations like addition, comparison, or property access, while still encapsulating the type-specific behavior behind virtual dispatch.

This design effectively "boxes" primitive data into small wrapper objects, giving the runtime full control over their semantics. Such encapsulation ensures that the Toy language behaves consistently and independently from the underlying platform.

A small mapping table (`BOXER`) translates between Java primitives and `Value` objects, allowing constants, arguments, and return values to move seamlessly between the interpreter and host environment. This mechanism guaranties that the runtime only ever operates over a finite, well-defined set of types, simplifying the interpreter loop and avoiding uncontrolled type polymorphism inside the virtual machine.

IV. Implementation Details & Optimitzations

A. Object Representation & Inline Caching

Shapes (hidden classes). Representation of dynamic objects with a split layout: a *shape* (metadata) plus a *fields* array (storage). The shape encodes the structure of an object—i.e., which properties exist and at which slots—while the fields array holds the actual `Values` at those slots. This removes hash lookups from the steady state and turns repeated property access into constant-time slot reads/writes.

Shape graph. The `Shape` class maintains:

- a persistent map `propToSlot : String` $\rightarrow$ `int` assigning each property name to a stable slot index, and
- a transition map `transitions : (op,key)` $\rightarrow$ `Shape` keyed by `TransitionKey{ADD|DEL, key}`.

Objects start at `EMPTY_SHAPE`. Adding a new property `p` follows (or creates) the transition `(ADD,p)` to a successor shape whose `propToSlot` is the parent's map plus `p` $\mapsto$ `nextSlot`. Similarly, deletion uses the `(DEL,p)` transition to a shape whose map omits `p`. Each new shape is immutable (maps are copied-on-write and sealed with `Map.copyOf`), so shapes are safely shareable across objects and ideal for inline caches.

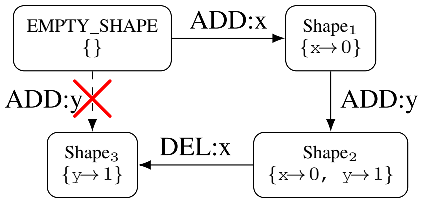

Fig. 2. Shape graph: property additions/deletions create immutable shape transitions; fields are stored by slot.

VObject layout and operations. Each `VObject` carries:

- a `shape` reference (initially `EMPTY_SHAPE`),
- a growable `Value[] fields` array used as slot storage.

A property read resolves to a slot (`shape.slotOf(key)`); if present, `fields[slot]` is returned, otherwise a language-level error (`Undefined property`) is thrown.

Why shapes? Shapes provide:

- *Predictable constant-time access*: property lookup becomes `slotOf` $\rightarrow$ array index.
- *Sharing and caching*: identical object "classes" reuse the same shape, enabling cheap equality checks (`obj.shape()==cachedShape`) in inline caches.
- *Cheap evolution*: property additions/deletions are just transitions in the shape graph; no rehashing on the hot path.

Interpreter integration. The bytecodes `GETPROP`/`SETPROP` use `shape.slotOf` for the generic path and `getFast`/`setFast` for the IC path. A monomorphic cache stores {shape, key, slot} at the call site; on a hit, access is a single array read/write. On a miss (shape change or different key), the interpreter falls back to the generic lookup, updates the cache, and continues.

B. *Bytecode Tooling and Debug Flags*

Toy includes two lightweight debugging utilities that expose internal bytecode and runtime behavior for testing and validation.

1) The `-dump-bci` flag prints a disassembled view of the generated bytecode, showing instruction offsets, opcodes, and their operands. This tool is mainly used to verify the correctness of the compilation phase and to inspect control-flow layout.
2) The `-trace-bci` flag enables step-by-step execution tracing inside the interpreter. When active, a `BciTracer` callback records the current program counter, executed opcode, and a snapshot of the operand stack at each instruction.

C. *Built-in Functions*

Toy also offers a set of *built-in functions* that extend the language with essential operations such as object creation, reflection, and I/O. These are not user-defined Toy functions, but rather Java procedures that are artificially injected into the global `functionTable` at startup through the `BuiltinBuilder`. This ensures that every Toy program has access to a consistent standard library without requiring explicit imports.

V. Conclusion

This report presented the architecture and implementation of ToyVM, a compact stack-based bytecode interpreter for the Toy language. We detailed the end-to-end pipeline (parsing $\rightarrow$ AST $\rightarrow$ bytecode), the compilation infrastructure (`CompileContext`, constant pool), and the interpreter's execution model built around a tight `switch`-dispatch loop. A uniform `Value` system enables a finite, well-defined runtime type domain, while the object representation uses shapes to provide predictable slot-based access and enable monomorphic inline caches. Tooling (`-dump-bci`, `-trace-bci`) improves observability and helped validate correctness. Although optimizations are conservative, the design emphasizes clarity and extensibility, providing a solid baseline for future work such as array strategies, string ropes, and a minimal JIT tier.