Feeny Language

Hale

January 7, 2025

Contents

Acknowledgments

This project was first introduced in the University of California, Berkeley graduate course "Virtual Machines and Managed Runtimes" (CS294) taught by Mario Wolczko and Patrick S. Li in 2015. The course provides a hands-on approach to understanding virtual machines by implementing Feeny from scratch, starting with a simple AST interpreter and progressing to an optimized bytecode virtual machine.

The current implementation, while inspired by the original course project, has been independently developed with all components reimplemented from the ground up. Notable modifications include an extended parser grammar for enhanced language features and a focus on robust virtual machine implementation.

1 Language Fundamentals

1.1 Language Overview

Feeny is an object-oriented, imperative, dynamically typed programming language designed for educational purposes. It shares similar core functionality with popular scripting languages like Python, JavaScript, and Ruby, while maintaining a lean implementation that can be completed by a single student in a semester-long course.

1.2 Lexical Structure

1.2.1 Comments

Comments in Feeny begin with a semicolon (;) and continue to the end of the line:

```
var c = 0     ;Initialize c to zero
while c < 9: ;Proceed when c is less than nine
```

1.2.2 Indentation Structure

Feeny uses indentation to indicate code blocks. Lines ending with a colon (:) automatically wrap the following indented block in parentheses:

```
while i < 10:
    do-this()
    do-that()

;equivalent to:
```

```
while i < 10: (do-this() do-that())
```

1.2.3 Comma Handling

Commas in Feeny are treated identically to whitespace and are used solely for readability:

```
f(a, b, c)  ;equivalent to f(a b c)
```

1.2.4 Parenthesis Rules

Identifiers immediately followed by opening parentheses have special handling:

```
f(a(1 + 2))   ;f function call with result of a(1 + 2)
f(a (1 + 2))  ;f function call with two arguments
```

1.3 Basic Types and Values

1.3.1 Integers

Integers in Feeny are 32-bit values supporting basic arithmetic operations:

```
var x = 42         ;integer literal
var y = x + 10     ;arithmetic operation
var z = x.add(10) ;equivalent method call
```

1.3.2 Null

Null is a special object representing an empty environment with no slots:

```
var empty = null
if empty: ;will not execute as null is false
```

1.3.3 Arrays

Arrays are fixed-length sequences with an initial value, accessed using square brackets:

```
var arr = array(10, 0)  ;length 10, all elements 0
arr[0] = 42             ;set first element
arr.set(0, 42)          ;equivalent method call
var val = arr[0]        ;get first element
var val = arr.get(0)    ;equivalent method call
var len = arr.length()  ;get array length
```

set and get are methods for setting and getting array elements, they can be overloaded, like arr[i,j] = 42 equivalent to arr.set(i,j,42) if the arr object has rewrite the set method.

1.3.4 Objects

Objects in Feeny use a slot-based system with variable and method slots:

```
object:
    var x = 10
    var y = 20
    method add():
        this.x + this.y
```

1.4 Control Flow

1.4.1 Conditional Statements

If expressions in Feeny, one important thing is when condition is evaluated to null object, it will be treated as false, otherwise, it will be treated as true(0 is true)

```
if condition:
    true-branch
else:
    false-branch
```

1.4.2 Loops

While loops in Feeny, the condition rule is the same as if expression

```
while condition:
    body
```

1.5 Operator Syntax

All operators in Feeny are syntactic sugar for method calls:

```
x + y    ;expands to x.add(y)
x - y    ;expands to x.sub(y)
x * y    ;expands to x.mul(y)
x / y    ;expands to x.div(y)
x % y    ;expands to x.mod(y)
x < y    ;expands to x.lt(y)
```

```
x > y    ;expands to x.gt(y)
x <= y   ;expands to x.le(y)
x >= y   ;expands to x.ge(y)
x == y   ;expands to x.eq(y)
```

like array's set and get methods, if the object has rewrite the add method, the + operator will invoke the object own add method.

1.6 Scoping Rules

Feeny implements several scoping levels:

- Global scope for top-level definitions
- Local scope for function variables
- Object scope for slots within objects
- The `this` keyword is required for accessing object slots

2 Language Grammar

The grammar of Feeny is defined using Extended Backus-Naur Form (EBNF). Terminal symbols are shown in **bold**, and nonterminals are in plain text.

2.1 Program Structure

program → scope_stmt

2.2 Statements

scope_stmt → (var_decl | fn_decl | expression)∗

var_decl → **var** IDENTIFIER = expression

fn_decl → **defn** IDENTIFIER **(** parameters? **) :** INDENT scope_stmt DEDENT

slot_stmt → var_slot | method_slot

var_slot → **var** IDENTIFIER = expression

method_slot → **method** IDENTIFIER **(** parameters? **) :** INDENT scope_stmt DEDENT

2.3 Expressions

expression → assign
assign → lvalue = assign | compare
lvalue → chain(. IDENTIFIER |[expression])∗
compare → term (comp_op term)∗
comp_op → < | <= | > | >= | ==
term → factor ((+ | -) factor)∗
factor → unary ((* | / | %) unary)∗
unary → - ∗ chain

2.4 Chain and Primary Expressions

chain → primary chain_suffix∗
chain_suffix → [expression]
| . IDENTIFIER ((args?))?

primary → NUMBER | **null** | IDENTIFIER | (expression)
| if_expr | while_expr | object_expr | array_expr | printf_expr
| IDENTIFIER (args?)

2.5 Complex Expressions

if_expr → **if** expression **:** INDENT scope_stmt DEDENT
(**else :** INDENT scope_stmt DEDENT)?
while_expr → **while** expression **:** INDENT scope_stmt DEDENT
object_expr → **object** (expression | **:**) **:** INDENT slot_stmt ∗ DEDENT
array_expr → **array** (expression , expression)
printf_expr → **printf** (STRING (, expression) ∗)

2.6 Helpers

parameters → IDENTIFIER (, IDENTIFIER)∗
args → expression (, expression)∗

2.7 Lexical Rules

$$\begin{aligned}
\text{NUMBER} &\rightarrow \text{DIGIT}+ \\
\text{IDENTIFIER} &\rightarrow \text{ALPHA } (\text{ALPHA} \mid \text{DIGIT})* \\
\text{STRING} &\rightarrow \texttt{"}\ [\texttt{"}]*\ \texttt{"} \\
\text{ALPHA} &\rightarrow [\text{a-zA-Z_}] \\
\text{DIGIT} &\rightarrow [\text{0-9}] \\
\text{INDENT} &\rightarrow \text{increase in indentation level} \\
\text{DEDENT} &\rightarrow \text{decrease in indentation level}
\end{aligned}$$

2.8 Operator Precedence

Operators are listed in order of decreasing precedence:

1. Unary operators (-)
2. Multiplicative operators (*, /, %)
3. Additive operators (+, -)
4. Comparison operators (<, <=, >, >=, ==)
5. Assignment (=)

Note: All binary operators are transformed into method calls during parsing. For example, `a + b` becomes `a.add(b)`.

3 Object Model

The Feeny language implements a tagged pointer object model for efficient memory usage and runtime performance. The object system combines immediate values for primitives with heap-allocated objects for complex data structures.

3.1 Tagged Pointers

All values in Feeny are represented as tagged pointers using the lower 3 bits:

$$\begin{aligned}
\text{TAG_MASK} &= 7\ (0\text{b}111) \\
\text{TAG_BITS} &= 3 \\
\text{INT_TAG} &= 0\ (0\text{b}000) \\
\text{HEAP_TAG} &= 1\ (0\text{b}001) \\
\text{NULL_TAG} &= 2\ (0\text{b}010)
\end{aligned}$$

This allows:

- Direct integer encoding (shifted left by 3 bits)
- Heap object references (aligned addresses with tag bit set)
- Null value representation (special tag)

3.2 Runtime Objects

All runtime objects share a common header containing the object type:

$$\text{RTObj} = \left\{ \text{type} : \text{ObjType} \quad \text{Object type identifier} \right.$$

The system defines the following object types:

$$\begin{aligned} \text{GLOBAL\_TYPE} &= 0 \\ \text{NULL\_TYPE} &= 1 \\ \text{INT\_TYPE} &= 2 \\ \text{ARRAY\_TYPE} &= 3 \\ \text{OBJECT\_TYPE} &= 4 \end{aligned}$$

3.3 Object Structures

3.3.1 Arrays

Arrays are heap-allocated objects with a length field and variable-sized slot array:

$$\text{RArray} = \begin{cases} \text{type} : \text{ObjType} & \text{Always ARRAY\_TYPE} \\ \text{length} : \text{size\_t} & \text{Number of elements} \\ \text{slots}[] : \text{intptr\_t} & \text{Array elements} \end{cases}$$

3.3.2 Class Instances

Class instances are heap-allocated objects with a parent reference and variable slots:

$$\text{RClass} = \begin{cases} \text{type} : \text{ObjType} & \text{Class-specific type} \\ \text{parent} : \text{intptr\_t} & \text{Parent object reference} \\ \text{var\_slots}[] : \text{intptr\_t} & \text{Instance variables} \end{cases}$$

3.3.3 Class Templates

Class templates store the static structure of classes:

$$\text{TClass} = \begin{cases} \text{type} : \text{ObjType} & \text{Class type identifier} \\ \text{poolIndex} : \text{int} & \text{Index in method pool} \\ \text{varNames} : \text{Vector} & \text{Instance variable names} \\ \text{funcNameToPoolIndex} : \text{Map} & \text{Method name to pool index mapping} \end{cases}$$

3.4 Object Creation

The system provides factory functions for creating different types of objects:

- `newIntObj(int value)`: Creates a tagged integer value
- `newNullObj()`: Creates a null value
- `newArrayObj(int length, RTObj* initValue)`: Creates an array with given length and initialization value
- `newClassObj(ObjType type, int slotNum)`: Creates a class instance with given type and number of slots
- `newTemplateClass(ObjType type, int index)`: Creates a class template

3.5 Memory Management

The object system integrates with a garbage collector that:

- Uses forwarding pointers for object relocation (marked by BROKEN_HEART type)
- Handles both immediate values and heap objects
- Maintains object references through collection cycles

The combination of tagged pointers and garbage collection provides an efficient and safe memory management system for the Feeny runtime.

3.6 Arithmetic Details

The arithmetic operations in our tagged primitive system require careful consideration. Let's examine how we can perform these operations efficiently using our tagging scheme.

Let's denote $f(x)$ as the tagged representation of an integer value x, where:

$$f(x) = 8x$$

This representation means we shift every integer left by 3 bits (multiply by 8) to make room for our tag bits. The key insight is that we can perform some arithmetic operations directly on these tagged representations without converting back and forth.

For addition, we want to compute $f(x + y)$ given $f(x)$ and $f(y)$. Through algebraic manipulation:

$$\begin{aligned} f(x + y) &= 8(x + y) \\ &= 8x + 8y \\ &= f(x) + f(y) \end{aligned}$$

For multiplication, we have two possible implementations:

$$\begin{aligned} \text{Option 1 :} & f(x \cdot y) = f(x) \cdot f^{-1}(f(y)) \\ \text{Option 2 :} & f(x \cdot y) = f^{-1}(f(x) \cdot f(y)) \end{aligned}$$

where f^{-1} represents untagging operation (dividing by 8).

For division and modulo operations, we need to untag both operands before performing the operation, then retag the result:

$$\begin{aligned} f(x/y) &= f(\text{UNTAG}(f(x))/\text{UNTAG}(f(y))) \\ f(x \bmod y) &= f(\text{UNTAG}(f(x)) \bmod \text{UNTAG}(f(y))) \end{aligned}$$

Similar to addition, subtraction can be performed directly on tagged values:

$$\begin{aligned} f(x - y) &= 8(x - y) \\ &= 8x - 8y \\ &= f(x) - f(y) \end{aligned}$$

For comparison operations (op $\in \{<, >, =, \leq, \geq\}$), the implementation follows this pattern:

$$f(a \text{ op } b) = ((a \text{ op } b \ ?\ 1 : 0) \oplus 1) \ll 1$$

For example, for "less than" operation:

$$\begin{aligned} f(a < b) &= ((a < b \ ?\ 1 : 0) \oplus 1) \ll 1 \\ &= \begin{cases} 2 & \text{if } a < b \text{ is false} \\ 0 & \text{if } a < b \text{ is true} \end{cases} \end{aligned}$$

This implementation uses bitwise XOR ($\oplus$) with 1 to invert the boolean result, and then shifts left by 1 bit to maintain our tagging scheme. The result is either 2 (representing Null) when the comparison is false, or 0 when the comparison is true.

This representation allows us to perform arithmetic operations efficiently while maintaining the type safety provided by our tagging scheme. The key advantage is that for several operations (notably addition and subtraction), we can operate directly on the tagged values without any conversion overhead.

4 Virtual Machine Design

4.1 Machine Architecture

The Feeny VM adopts a stack-based design with four key components:

4.1.1 Core Components

1. **Global Variable Map**
   - Maintains name-to-value mapping for all global variables
   - Supports lookup and update operations by name
   - Implemented as a hash table for efficient access
2. **Current Local Frame**
   - Contains function arguments and local variables
   - Stores return address and parent frame link
   - Fixed-size slots (*num_args + num_locals*)
   - Direct indexed access for performance
3. **Operand Stack**
   - Holds temporary results during expression evaluation
   - Basic operations: push, pop, peek
   - Used for passing arguments and intermediate values
4. **Instruction Pointer**
   - Points to next instruction to execute
   - Updated by control flow instructions
   - Used for function returns and branches

4.2 Instruction Selection

The instruction set is designed around common operations while keeping the implementation simple:

4.2.1 Basic Instructions

- `Lit(i)`: Load constants from pool
- `GetLocal(i)/SetLocal(i)`: Local variable access
- `GetGlobal(i)/SetGlobal(i)`: Global variable access
- `Drop()`: Stack manipulation
- `Print(format, n)`: Output operations
- `Array()`: Array creation

4.2.2 Object Instructions

- `Object(c)`: Object instantiation
- `GetSlot(i)/SetSlot(i)`: Slot access
- `CallSlot(i,n)`: Method invocation
- Special handling for primitive receivers (int/array)

4.2.3 Control Flow Instructions

- `Label(i)`: Define jump targets
- `Branch(i)`: Conditional jumps
- `Goto(i)`: Unconditional jumps
- `Return()`: Function returns
- `Call(i,n)`: Function calls

4.3 Optimizations

4.3.1 Instruction Level

- Combined instructions for common patterns
- Specialized paths for primitive operations
- Stack caching for repeated access

4.3.2 Method Dispatch

- Inline caching of method lookups
- Fast path for primitive receivers
- Method table indexing

4.3.3 Frame Access

- Direct slot indexing
- Frame reuse when possible
- Stack-relative addressing

The VM design prioritizes:

- Simple and clear instruction semantics
- Efficient execution of common operations
- Predictable memory usage patterns
- Straightforward debugging and maintenance

This architecture provides a good balance between implementation complexity and runtime efficiency, while supporting all features of the Feeny language.

5 Garbage Collection

Feeny garbage collector implements a semi-space copying collector with a Cheney-style algorithm. The heap is divided into two equal spaces: from-space and to-space, with objects being copied between them during collection.

5.1 Memory Layout

The initial heap configuration consists of:

$$\begin{aligned}\text{heap\_size} &= 1\text{MB} = 1024 \times 1024 \text{ bytes}\\ \text{heap\_start} &= \text{base address of from-space}\\ \text{to\_space} &= \text{base address of to-space}\end{aligned}$$

5.2 Object Structure

Each object in the heap has a header containing its type information. For forwarding pointers during collection:

$$\text{BROKEN\_HEART} = \text{special type indicating forwarded object}$$

Object sizes are calculated based on their type:

$$\text{size}(obj) = \begin{cases} 0 & \text{for INT\_TYPE and NULL\_TYPE} \\ \text{sizeof(RArray)} + \text{length} \times \text{sizeof(intptr\_t)} & \text{for ARRAY\_TYPE} \\ \text{sizeof(RClass)} + n_{vars} \times \text{sizeof(intptr\_t)} & \text{for class instances} \end{cases}$$

5.3 Collection Algorithm

The collection process follows these steps:

1. **Initialization**:

$$\text{to\_ptr} = \text{to\_space}$$

2. **Root Set Scanning**: Traverse and copy all objects reachable from:

- Global variables
- Stack frames
- Operand stack

3. **Forwarding Objects**: When copying an object:

$$\text{forward}(obj) = \begin{cases} obj & \text{if not heap pointer or already forwarded} \\ \text{copy\_to\_new\_space}(obj) & \text{otherwise} \end{cases}$$

4. **Scanning Phase**: For each copied object:

- Update all internal pointers using forwarding addresses
- Handle different object types (arrays, class instances) appropriately

5.4 Heap Management

The collector triggers under two conditions:

$$\text{trigger\_gc if: } \text{heap\_ptr} + \text{nbytes} > \text{heap\_start} + \text{heap\_size}$$
$$\text{or: } \frac{\text{heap\_ptr} - \text{heap\_start}}{\text{heap\_size}} \times 100 > 90\%$$

Heap expansion occurs when:

$$\text{usage\_after\_gc} > 70\% \text{ or insufficient space}$$

The expansion doubles the heap size:

$$\text{new\_size} = \text{heap\_size} \ll 1$$

5.5 Memory Allocation

The allocation function `halloc` implements a bump pointer allocator:

1. Align request to 8 bytes:

$$\text{aligned\_size} = (n + 7)\& \sim 7$$

2. Check space and trigger collection if needed:

$$\text{usage} = \frac{\text{heap\_ptr} - \text{heap\_start}}{\text{heap\_size}} \times 100$$

3. Allocate by incrementing bump pointer:

$$\text{result} = \text{heap\_ptr}$$
$$\text{heap\_ptr} = \text{heap\_ptr} + \text{aligned\_size}$$

This implementation provides efficient memory management while maintaining the simplicity of a copying collector. The automatic heap expansion ensures the system can handle growing memory requirements, while the triggering conditions help maintain optimal performance.

6 Performance Analysis

The implementation underwent several optimization phases, with significant performance improvements at each step:

6.1 Execution Time Comparison

Version	Real Time (s)	Improvement
Ast Interpreter	33.897	/
Bytecode Interpreter	33.205	baseline
With Garbage Collector	23.110	30.4%
Naive Tagged Primitives	17.005	48.8%
Magic Tagged Primitives	14.806	55.4%

The most significant improvements came from:

1. **Garbage Collection**: Reduced execution time by 30.4% through efficient memory management and reduced system calls (sys time from 6.832s to 0.030s)

2. **Tagged Primitives**: Further 35.9% improvement by eliminating heap allocations for primitive types through:

$$\text{tagged\_value} = \begin{cases} n \ll 1 & \text{for integers} \\ 2 & \text{for null} \end{cases}$$

The final implementation achieves a total performance improvement of 55.4% compared to the initial version, with negligible system time overhead.

7 Thoughts on Memory Management and Garbage Collection

7.1 Bytecode vs AST Interpreter GC

Writing a garbage collector for the AST interpreter would be more challenging compared to the bytecode interpreter, for several key reasons:

- **Control**: The bytecode interpreter provides better control over memory layout and execution state. The memory representation is more explicit and structured.
- **Clarity**: There is a clearer separation between interpreter structures and program data in the bytecode implementation. This makes it easier to identify what needs to be collected.
- **Precision**: Implementing precise garbage collection is more straightforward in the bytecode interpreter due to explicit type information and memory layout.
- **State Management**: The explicit stack management in bytecode makes root set identification simpler, as all live references must be either on the stack or in global variables.

7.2 Array Implementation Analysis

The following pseudocode for the ARRAY bytecode instruction contains a critical error:

```
case ARRAY_INS:
    val init = pop from operand stack
    val length = pop from operand stack
    val array = halloc(8 + 8 + 8 * length.value)
    // Tag + Len + Slots..
    array[0] = ARRAY_TAG
    array[1] = length.value
    array[2 to 2 + length.value] = init
    push array to operand stack
```

The error lies in the array initialization step. The code attempts to initialize all array slots with a single assignment operation (`array[2 to 2 + length.value] = init`), which is incorrect. Instead, it should use a loop to explicitly set each slot to the initialization value:

```
for i = 0 to length.value - 1:
    array[2 + i] = init
```

7.3 GC Performance Considerations

7.3.1 Long-Lived Objects

The current garbage collection algorithm has inefficiencies when dealing with long-lived objects that contain large amounts of memory:

- These objects are repeatedly scanned during each collection cycle
- They are unnecessarily moved between generations
- This repeated processing wastes computational resources

7.3.2 Linked List Performance

For programs making heavy use of linked lists, particularly long lists of integers, our current garbage collector exhibits several performance issues:

- **Memory Layout**: The collector will compact separate linked lists into a more contiguous memory layout

- **Cache Effects**: While this might seem beneficial for cache performance, it can actually be detrimental
- **Access Patterns**: Linked list traversal patterns may not benefit from the compacted layout
- **Overhead**: The cost of repeatedly moving and adjusting these structures may outweigh any potential benefits

7.4 Potential Improvements

To address these issues, several improvements could be considered:

- Implement generational collection with better promotion policies
- Add large object space for handling long-lived, large objects
- Optimize collection strategies for linked data structures
- Consider memory layout patterns that better match access patterns

References

1. **Virtual Machine Design and Implementation**
   - Smith, J.E. and Nair, R., "Virtual Machines", Morgan Kaufman, 2005.
   - Goldberg, A. and Robson, D., "Smalltalk-80: The Language and its Implementation", Addison-Wesley, 1983.
2. **Bytecode Interpretation**
   - Diehl, S., Hartel, P., and Sestoft, P., "Abstract machines for programming language implementation", Future Generation Computer Systems 16, 2000.
   - Bell, J.R., "Threaded code", Communications of the ACM 16(6), 1973.
3. **Memory Management**
   - Jones, R., Hosking, A., and Moss, E., "The Garbage Collection Handbook", CRC Press, 2012.
   - Ungar, D., "Generation Scavenging: A non-disruptive high performance storage reclamation algorithm", 1984.

The implementation draws inspiration from various virtual machine designs and techniques documented in these references, particularly in areas of bytecode interpretation, garbage collection, and dynamic compilation.