FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	FML runtime system reference implementation
Student:	Bc. Martin Taibr
Supervisor:	Ing. Konrad Siek, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

FML is a variant of Feeny, a small, dynamic, object-oriented language with dynamic dispatch designed to teach runtime system concepts through implementation. Specifically, CTU students implement a version of FML during NI-RUN. However, FML lacks a good reference implementation that is complete, well-engineered, legible to students, and with internal design matching good runtime implementation practices. The goal of this thesis is to provide the code of such a reference implementation, and to accessibly document it both in-code and through an external document (the text of the thesis).

The student will:
- investigate existing implementations of FML,
- design a runtime system with possible adjustments to the existing FML specification,
- implement the design, with an emphasis on specific features:
  - improve and document the existing reference FML bytecode compiler and interpreter,
  - implement a modular garbage collector, ideally implementing multiple collection schemes, and
  - implement a just-in-time compiler.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 16 January 2023 in Prague.

Master's thesis

FML RUNTIME SYSTEM REFERENCE IMPLEMENTATION

Bc. Martin Taibr

Faculty of Information Technology
Katedra teoretické informatiky
Supervisor: Ing. Konrad Siek, Ph.D.
May 4, 2023

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Taibr Martin. *FML runtime system reference implementation.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Tables

List of code listings

I'd like to thank my supervisor for all his patience, advice and help and to my family and friends for their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 4, 2023 .....

Abstract

This thesis focuses on the design and implementation of an interpreter for the FML language which is used at CTU for teaching the Runtime systems course. FML is a small, dynamically typed, object oriented language based on Feeny and ML. The reference implementation created this way will contain a garbage collector (GC) and a just-in-time (JIT) compiler.

Keywords FML, teaching language, GC, garbage collector, JIT, just-in-time compiler

Abstrakt

Tato práce se zaměřuje na návrh a implementaci interpreteru pro jazyk FML, který je využívaný na ČVUT k výuce předmětu Runtime systémy. FML je malý, dynamicky typovaný, objektově orientovaný jazyk inspirovaný jazyky Feeny a ML. Takto vzniklá referenční implementace bude obsahovat garbage collector (GC) a just-in-time (JIT) kompilátor.

Klíčová slova FML, výukový jazyk, GC, garbage collector, JIT, just-in-time kompilátor

List of Abbreviations

AOT	Ahead-of-Time
AST	Abstract Syntax Tree
CISC	Complex Instruction Set Computer
ISA	Instruction Set Architecture
GC	Garbage Collector
JIT	Just-in-Time
REX	Register Extension
RISC	Reduced Instruction Set Computer
VM	Virtual Machine

Introduction

As students of computer science progress through their studies, they are tasked with implementing compilers, analyzers, interpreters and other tools for various programming languages.

Usually these are not general purpose languages but are designed for teaching specific topics and must strike a balance between being too simple, where students may not learn much from implementing them, and being too complex, where students cannot implement them in a reasonable timeframe.

One such language is FML, for which students implement a bytecode compiler and interpreter as part of the Runtime Systems course. However, the code of the current reference implementation of FML is not as clean and readable as it could be and lacks a garbage collector (GC) and just-in-time (JIT) compiler.

A garbage collector is a runtime mechanism for managing memory used by a computer program. It automates memory management, freeing developers from the burden of manual memory management, reducing the likelihood of memory leaks and memory-related bugs. The garbage collector is responsible for identifying objects that are no longer used by the program and deallocating them. There are many strategies for garbage collection, but in general, it works by periodically traversing the program's heap to find objects that are no longer reachable and freeing their memory.

A JIT compiler is a type of compiler that dynamically translates bytecode or other intermediate representations into native machine code at runtime, rather than ahead of time. This can result in significant performance improvements over interpreting the program. More advanced JIT compilers can optimize code based on runtime information and architecture-specific details to reach or surpass the performance of ahead-of-time (AOT) compilers. However, JIT compilers can also introduce additional overhead during program execution.

The objective of this thesis is to provide a high-quality reference implementation of FML that includes a garbage collector and a JIT compiler. High performance is explicitly not a goal, the implementation should be reasonably efficient while also ensuring that the code is well-engineered and accessible to students.

Chapter 1

Description of the FML Language

This chapter describes parts of the FML teaching language relevant to this thesis. It provides a short overview of its syntax, some aspects of its semantics and details the structure of its bytecode.

FML is a small, imperative, dynamically typed, object-oriented programming language. Its semantics and bytecode are based on Feeny and its syntax is inspired by ML. FML is designed to be simple enough to implement by a student during one semester, yet provides non-trivial features such as inheritance and dynamic dispatch, which makes it useful as a teaching tool.

It should be noted there are multiple implementations of FML. There is the original reference implementation in Rust by Ing. Konrad Siek, Ph.D [1], the new reference implementation in C by Bc. Michal Vlasák [2], as well as a number of publicly available student implementations.

As the contents of the Runtime Systems course evolve, so does the FML language. The Rust version has not been updated for the current semester (summer 2022/2023) and as a result the newer C version has diverged slightly. [3]

This thesis is based on the original Rust implementation. The sources used in this chapter therefore are:

- the website of the Runtime Systems course from previous years [4]
- the original reference implementation available on GitHub [1]

Whenever this thesis refers to the reference implementation, it means the original Rust implementation.

1.1 Language Elements

Since FML source code compiles to bytecode and the JIT compiler operates on the bytecode, not source code or abstract syntax tree (AST), we will only explain the syntax briefly to aid understanding code snippets. A much more elaborate explanation is available in the sources cited above and a complete explanation would necessarily only duplicate that effort.

1.1.1 Basic Syntax

FML has the same syntax for comments as C. Line comments begin with `//` and end at the end of the line. Block comments begin with `/*` and end with `*/`. For example:

- `// This is a comment`
- `/* This is also a comment */`

Variables are declared using `let` and `=`, for example `let x = 1;`. The initial value is required. All variables are mutable and can be reassigned using '`->`', for example `x -> 2;`.

Expressions are separated by semicolons (`;`). Blocks are delimited by `begin` and `end`. The last expression in a block is its return value regardless of whether it is followed by a semicolon. A block is therefore an expression.

Variable declarations and assignments, conditionals, loops and function calls are all also expressions. Conditionals use the `if` keyword and loops use `while`.

Functions are introduced using the `function` keyword. They can have multiple parameters and always have one return value. The body can be a simple expression of a block containing multiple other expressions. There is no return keyword, the expression which the body evaluates to is the return value. This also means all functions have exactly one entry and exactly one exit point.

FML has one built-in function, `print`, which takes a format string and a variable number of arguments. The format string can contain tilde (`~`) characters which are replaced by the corresponding arguments. It returns the value `null`.

The format string supports certain escape sequences such as `\n`, `\t`, `\"`, `\\` and `\~`.

Code listing 1.1 Examples of FML functions

```
function add(x, y) -> x + y;
function add1(x) -> begin
    print("before ~\n", x);
    x <- x + 1;
    print("after ~\n", x);
    x
end;

print("~\n", add(1, 2));
print("~\n", add1(3));
```

1.1.2 Types

The FML language has only 5 types:

- Unit with its only value `null`.
- Boolean with values `true` and `false`.
- Integer which is a 32-bit signed integer.
- Array - a fixed size collection indexable by integers which can contain 0 or more values of any type.
- Object - a user-defined structure which can have fields, methods and operators and can inherit from other values (including primitive types and arrays).

FML is dynamically typed which means types are associated with values, not variables. Variables can be assigned values of any type and the type assigned to a variable can change during its lifetime.

Strings and functions are not types. They cannot be assigned to variables, passed to user-defined functions or returned from them. The string literal used in the `print` function is a special case and the only place where a string is used in FML.

Unit, boolean and integer are primitive types. They are allocated directly on the stack and passed by value. Arrays and objects are allocated on the heap and passed by reference.[1]

Code listing 1.2 Example of an array being passed by reference. The modification performed inside the function is visible outside of it.

```
// An array of size 3, all elements initialized to 0
let arr = array(3, 0);
print("~\n", arr[0]); // prints 0

function modify(x) -> begin
    x[0] <- 42;
end;

modify(arr);
print("~\n", arr[0]); // prints 42
```

1.1.2.1 Primitive Types

All primitive types support comparison using the `==` and `!=` operators or equivalently using the `eq` and `ne` methods. Comparisons using `==` between different types return `false`. There are no automatic conversions between types.

Unit supports only equality and inequality comparisons. The `null` value is only equal to itself.

Boolean supports `==` and `!=` and logical operators listed in table 1.1.

Operator	Method
==	eq
!=	ne
&	and
|	or

Table 1.1 Boolean operators and their corresponding methods

Integer supports more comparisons and also mathematical operators listed in table 1.2.

1.1.2.2 Arrays

An array is constructed using the `array` keyword which is syntaxcically similar to a function call that takes two arguments. The first argument is the size of the array and the second is an expression used to initialize the elements.

Arrays have the `set` and `get` methods for indexing. The syntax using square brackets is compiled into calls of these methods as shown in table 1.3 . They can contain elements of any type, including other arrays. An array containing a reference to itself is valid but might cause

[1] Since this paragraph describes language semantics, it doesn't matter where values are allocated. It would be equally valid to allocate primitive types on the heap. However since they have to be passed by value it's more natural and efficient to allocate them on the stack and copy them each time they're used. This is how the reference implementation works.

Operator	Method
`<=`	le
`<`	lt
`>=`	ge
`>`	gt
`==`	eq
`!=`	ne
`+`	add
`-`	sub
`*`	mul
`/`	div
`%`	mod

Table 1.2 Integer operators and their corresponding methods

issues such as infinite recursion when printing it. The specification doesn't address this edge case and the reference implementation doesn't handle it.

Syntax	Corresponding method call
`arr[index] <- new_value`	`arr.set(index, new_value)`
`arr[index]`	`arr.get(index)`

Table 1.3 Array operators and their corresponding methods

Arrays are interesting from a teaching perspective because the initializer can be an arbitrary expression and can have side effects which means the value it returns can be different for each element. The bytecode compiler has to take this into account and despite the syntax looking simple, it might have to generate a loop to initialize the array.

1.1.2.3 Objects and Inheritance

Objects are declared using the `object` keyword and can optionally specify a value they inherit from using the `extends` keyword. The parent can be any of the 5 types, that includes primitives, arrays and other objects. If `extends` is not specified, the parent is `null`.

Methods receive an extra hidden argument called `this` which allows them to access the object they're being called on, it's other methods, operators and fields.

Method dispatch works by looking for the method (or operator) on the current object. If it's not found, it does the same on the parent until either the method is found or there are no more parents.

Field access is not dispatched to parents. If a field is not found on the current object, it's an error.

Each use of the `object` keyword in code results in a separate class definition being created, which is then saved in the program's bytecode representation. A class has statically known fields, methods, and operators; they can't be added or removed at runtime.

1.1.3 Scoping

In FML, variable scoping is lexical, which means the scope of a variable is known statically at compile time based on where it is declared. Variables are not accessible outside the block in which they are declared.

FML supports both global and local variables. Global variables are declared outside of any function, method, or block and are accessible throughout the entire program. Local variables are declared inside a block. This block can be in a function or method but also in the implicit entry function.

1.1.4 Example

The following example shows the syntax of FML and how it can be used to implement the stack data structure.

Code listing 1.3 A stack implemented as a linked list.

```
function Stack() -> object begin
    let head = null;

    function push(element) -> this.head <- Node(element, this.head);

    function peek() -> begin
        if null == this.head then
            null
        else
            this.head.element;
    end;

    function pop() -> begin
        let element = this.peek();
        if null != this.head then
            this.head <- this.head.prev;
        element
    end;
end;

function Node(element, prev) -> object begin
    let element = element;
    let prev = prev;
end;

function test_eq(l, r) -> begin
    if l == r then
        print("OK (got/expected ~)\n", l)
    else
        print("Got ~, expected ~\n", l, r)
end;

let s = Stack();
test_eq(s.pop(), null);
test_eq(s.peek(), null);
s.push(1);
test_eq(s.peek(), 1);
s.push(2);
test_eq(s.peek(), 2);
s.push(3);
test_eq(s.peek(), 3);
test_eq(s.pop(), 3);
test_eq(s.peek(), 2);
```

1.2 The Bytecode Virtual Machine

1.2.1 Bytecode Format Overview

A program compiled to bytecode has the following components:

- Constant pool - a vector a program objects.
- Globals - a vector of indices into the constant pool.
- Entry point - an index into the constant pool representing the function that will be called when the program starts.
- Code - a vector of opcodes. All methods are compiled into the same linear sequence of opcodes.

FML has two separate concepts with similar names - program objects, which are data known at compile time, and runtime objects, which will be described in the next section.

1.2.1.1 Program Objects

Program objects are stored in the constant pool and are referenced by indices. There are seven different kinds of program objects:

- Null - the `null` literal.
- Boolean - a boolean literal - either `true` or `false`.
- Integer - a single value of type integer.
- String - a character string in the UTF-8 encoding. They can represent format strings and the names of functions, methods, operators, fields and labels.
- Slot - an index into the constant pool. It always points to a string and is used to represent the name of an object's field or a global variable.[2]
- Method - a method is a structure can represent an object's member function (colloquially called a method in many programming languages) or a global function. It contains its name as an index into the constant pool, the number of arguments it takes, the number of local variables it has and information how to locate its opcodes in the code vector in the form of an address and length.
- Class - a structure describing the fields and methods of an object declared by the `object` keyword in FML source code. It contains a vector of infices into the constant pool, each of which points to either a method or a slot representing a field name.

Since functions and methods are implemented the same way in bytecode, the terms can often be used interchangeably. The reference implementation uses a struct called `Method` for both of them, so this thesis will often use the term `method` to refer to both.

1.2.1.2 Globals

Similarly to classes, the vector of globals contains indices that point to either methods or slots. These represent global functions or the names of global variables respectively.

[2]Slots provide a level of indirection that is not necessary and they are no longer present in the newer C implementation.

1.2.2 The Virtual Machine's State

The previous section described the static structure representing a compiled program. This section will describe structures representing the runtime state of the program and how they relate to the static structures.

FML values inside the virtual machine are called runtime objects. There are five types of them corresponding to the five types of FML values - Null (for the Unit type), Boolean, Integer, Array and Object. According to the specification, runtime objects are referred to by pointers. The specification, however, only specifies observable behaviour of the VM, iplementations are free to use any representation they deem suitable. In particular, it is unnecessary to allocate primitive values on the heap and use pointers to them. In this case the pointer would become a tagged union that either embeds the value of a primitive type in itself or points to a heap-allocated object.

The FML virtual machine has the following components:

- Heap
- Frame stack
- Operand stack
- Instruction pointer

The heap is an area of memory that stores runtime objects. A pointer is an index into the heap and points to the beginning of a runtime object.

Frame stack and an operand stack are separate entities. The frame stack contains frames of all active functions and methods as well as their return addresses. A frame is a structure that contains the return address and a vector of pointers representing local variables.

The operand stack is temporary storage for the values being processed by the VM. Most opcodes work with the operand stack by pushing values onto it or popping them from it. Some opcodes also work with the frame stack or the heap.

The VM also has an instruction pointer which is the index of the current opcode. Executing an opcode updates this instruction pointer, usually by incrementing it but jumps, calls and returns set it to a new, arbitrary value.

1.2.3 Opcodes

Finally, we can describe FML's 17 opcodes, along with their effects on the VM's state.

Each opcode is a tagged union. The opcode's kind is represented by 1 byte and each opcode can have additional data.

1.2.3.1 Literal

Pushes a primitive value which is a compile time constant onto the operand stack.

It contains an index into the constant pool. The index must point to a program object of type Null, Boolean or Integer which is then converted into the corresponding runtime object and pushed onto the operand stack.

Bumps the instruction pointer.

1.2.3.2 Get Local

Reads the value of a local variable and pushes it onto the stack.

It contains an index into the current function's frame that specifies which variable.

Bumps the instruction pointer.

1.2.3.3 Set Local

Reads the value on top of the operand stack and stores it into a local variable.

It contains an index into the current frame that specifies which variable. Does not pop the value from the operand stack.

Bumps the instruction pointer.

1.2.3.4 Get Global

Reads the value of a global variable and pushes it onto the stack.

It contains an index into the constant pool. The pointed to program object must be a string which specifies the name of the global variable. This variable must exist in the globals vector.

Bumps the instruction pointer.

1.2.3.5 Set Global

Reads the value on top of the operand stack and stores it into a global variable.

It contains an index into the constant pool. The pointed to program object must be a string which specifies the name of the global variable. This variable must exist in the globals vector. Does not pop the value from the operand stack.[3]

Bumps the instruction pointer.

1.2.3.6 Call Function

Calls a global function (not an object's method).

Contains two fields. The first one is an index into the constant pool. It must point to a string which specifies the name of the function. The name must exist in the globals vector and must point to a method program object. The second field is the number of arguments this function expects.

Creates a new local frame on the frame stack. Pops the specified number of arguments from the operand stack and stores them into the new frame's locals in reverse order. Therefore the value popped first is the last argument and the value popped last is the first argument.[4] Fills the rest of the frame's locals with `null` values.

First bumps the instruction pointer, then sets the frame's return address to the new value which therefore points to the next opcode after the call, and finally sets the instruction pointer to the address of the called function's first opcode.

1.2.3.7 Return

Returns from the current global function or object's method.

Contains no additional data.

Sets the instruction pointer to the return address of the current frame. Pops the current frame from the frame stack.

1.2.3.8 Label

Does not modify the VM's state except the instruction pointer.

Contains an index into the constant pool which must point to a string which is the label's name. There must be no other label with the same name in the entire program, not even in different methods.

[3] The spec from previous semesters says the value is popped, that is incorrect, it's only peeked.

[4] This might be more intuitive to understand from the perspective of the compiler pushing arguments onto the stack before a function call - the arguments are pushed left to right, therefore they have to be popped right to left.

The label serves as target for jump and branch opcodes.[5]

Bumps the instruction pointer.

1.2.3.9 Jump

Unconditionally jumps to a label.

Contains an index into the constant pool which must point to a string which is the label's name.

Sets the instruction pointer to the address of the label opcode.

1.2.3.10 Branch

Conditionally jumps to a label.

Contains an index into the constant pool which must point to a string which is the label's name.

Pops a value from the operand stack. If the value is `null` or `false`, it is considered falsy. Otherwise it is considered truthy.

If the value is falsy, bumps the instruction pointer. If the value is truthy, sets the instruction pointer to the address of the label opcode.

1.2.3.11 Print

Prints values according to the format string.

Contains two fields. First an index into the constant pool which must point to a string program object which is the format string. Second the number of arguments.

Pops the specified number of arguments from the operand stack, the format string must contain the same number of placeholders. The arguments in reverse order (same as the Call Function opcode) are used to fill placeholders in the format string.

Pushes a `null` onto the operand stack.

Bumps the instruction pointer.

1.2.3.12 Array

Creates a new array.

Contains no additional data.

Pops a value from the operand stack. This becomes the initial value of the array's elements. Pops a second value from the operand stack. This becomes the array's length and must be a non-negative integer. Creates a new runtime object of type Array and pushes a pointer to it onto the operand stack.

This opcode does not initialize the array's elements according to FML's semantics as described in 1.1.2.2. The initial value is typically `null` and the array's elements are initialized later by a loop the bytecode compiler generates after the Array opcode.

Bumps the instruction pointer.

1.2.3.13 Object

Creates a new object.

Contains an index into the constant pool which must point to a class program object. Allocate a new runtime object of type Object. Pops values from the operand stack to initialize its fields

[5] Since this opcode effectively does nothing, one common optimization in student implementations is to remove them from the code vector completely when loading the bytecode and instead only store the addresses in a separate location.

according to the class program object. Pops one more value from the operand stack which becomes the object's parent.[6]

Pushes a pointer to the new object onto the operand stack.

Bumps the instruction pointer.

1.2.3.14 Get Field

Reads the value of an object's field and pushes it onto the stack.

Contains an index into the constant pool which must point to a string program object which is the field's name.

Pops a value from the operand stack which must be a pointer to an object. Reads the value of the specified field from the object and pushes it onto the operand stack.

Bumps the instruction pointer.

1.2.3.15 Set Field

Reads the value on top of the operand stack and stores it into an object's field.

Contains an index into the constant pool which must point to a string program object which is the field's name.

Pops a value from the operand stack, this becomes the new value. Pops a second value from the operand stack which must be a pointer to an object. Stores the new value into the specified field of the object.[7]

Bumps the instruction pointer.

1.2.3.16 Call Method

Calls an object's method.

Contains two fields. The first one is an index into the constant pool. It must point to a string which specifies the name of the method. The second field is the number of arguments this method expects, including the implicit `this` pointer.

Pop the specified number of arguments from the operand stack, the last one becomes the receiver.

If the receiver is `null`, a boolean, an integer or an array, no call is performed. If the method name matches the name of a built-in method or operator, the result is calculated by the interpreter and pushed onto the operand stack. Otherwise a runtime error occurs.

If the receiver is an object, the interpreter looks for the specified method on the object and if not found, its parent recursively. If the method is not found, a runtime error occurs. If the method is found, creates a new local frame on the frame stack. Fills it with the arguments including the receiver and `null` values for the remaining locals. Sets the frame's return address to the address of the next opcode after the Call Method opcode.

If no call was performed, bumps the instruction pointer. Otherwise sets the instruction pointer to the address of the called method's first opcode.

1.2.3.17 Drop

Removes the value on top of the operand stack.

Contains no additional data.

Bumps the instruction pointer.

---

[6] The description of this opcode and some others is greatly simplified because it's not relevant to this implementation of a garbage collector or JIT compiler.

[7] Note that unlike Set Local and Set Global, this opcode does remove the value from the operand stack.

Chapter 2

Description of x86-64 Assembly and Machine Code

This chapter describes elements of the x86-64 instruction set architecture necessary for writing a simple JIT compiler. It provides a short introduction to the basics, then compares 64-bit calling conventions and details how instructions are encoded, including special cases of memory operands.

The x86-64 instruction set architecture (ISA), also known as x64 or AMD64, is a common architecture for desktop and server computers. It, or a simplified variant, is also used at FIT CTU as part of several courses so many students are familiar with it to some extent. Therefore it was chosen as the target architecture for this JIT compiler.

The main source used throughout this chapter and during implementation is the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 1 and 2 [5]. Most importantly in Volume 1 it was Chapter 3.4 Basic Program Execution Registers and in Volume 2 Chapter 2 Instruction Format and parts of Chapters 3-6 detailing the individual instuctions used in this JIT compiler.

Introduction to x64 Assembly [6] served as a more gentle introduction.

x86-64 is a complex instruction set computer (CISC)[1] architecture with variable-length instructions. It has 16 general-purpose 64-bit registers: RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI and R8-15. They can be accessed as 32-bit registers by replacing the leading R with an E.

The RSP register is the stack pointer. The stack is used for storing local variables, return addresses, arguments which don't fit in registers and other things. It grows downwards, i.e. when pushing a value onto the stack, the stack pointer is decremented and the value is stored at the new address. Popping values from the stack increments the stack pointer.

The ISA has a large number of instructions but for implementing a simple JIT compiler only a small subset will be needed:

- Instructions for moving data between registers and memory such as MOV, PUSH and POP.
- Arithmetic instructions such as ADD, SUB, IMUL and IDIV.
- Logical instructions such as AND and OR.
- Comparison instructions such as CMP and TEST.

[1] CISC means the ISA tends to have complex instructions that can do a lot of work in a single instruction, often supporting complex addressing modes, as opposed to reduced instruction set computers (RISC) which have simpler instructions.

- Control flow instructions such as CALL, RET, JMP and Jcc.[2]
- Various other instructions for debugging and other purposes such as NOP, UD2, INT3, CQO.

Many of these instructions take one or more operands which can be registers, memory locations or immediate values (constants).

There are two commonly used syntaxes for x86-64 assembly. In Intel syntax registers are specified simply by their name, memory operands are specified by using square brackets and the destination operand is on the left. The other syntax is called AT&T syntax. It requires a leading `%` for registers, memory operands are specified by using parentheses and the destination operand is on the right. There are other differences as well. This thesis uses Intel syntax.

A few example instructions:

- `mov rax, 42`: stores the value 42 in RAX.
- `add rax, 42`: adds the value 42 to the value in RAX and stores the result in RAX.
- `add rax, rbx`: adds the values in registers RAX and RBX and stores the result in RAX.
- `add rax, [rbx]`: reads a value from memory at the address in RBX, adds it to the value in RAX and stores the result in RAX.

x86-64 is a deep and complex topic that is impossible to adequately cover here. This chapter aims to provide only a high-level overview. Topics covered in slightly more depth are calling conventions, memory addressing and instruction encoding.

2.1 Calling Conventions

When writing assembly code that interacts with other functions, it is necessary to follow a set of rules called the calling convention. It specifies which registers or parts of the stack are used for passing arguments and returning values, which registers must be preserved by whom and other important details. We will explore two calling conventions here.

When talking about arguments and return values, we will mainly consider 32 and 64-bit integers and 64-bit pointers since FML doesn't support floating-point numbers.

In both x86-64 calling conventions, the stack must be aligned to a 16-byte boundary before the call instruction is executed. A lack of alignment typically manifests itself as a segmentation fault.

Each register is considered either volatile or non-volatile in a given calling convention. Volatile (caller-saved) means that if the caller wants its value to be preserved after a function call, it must save it on the stack before the call and restore it afterwards. Non-volatile (callee-saved) means that if the callee wants to use it, it must save the value on the stack before modifying it and restore it before returning.

2.1.1 Microsoft x64 Calling Convention

The Microsoft x64 calling convention is used by Microsoft Windows.

The first four arguments are passed in registers RCX, RDX, R8 and R9 [7]. Additional arguments are passed on the stack. The return value is saved in RAX if it fits, otherwise the caller allocates space for it on the stack and sends the callee a pointer to it as the first argument in RCX. Additionally, the caller must allocate 32 bytes of shadow space on the stack for the callee to use.

Caller-saved registers are RAX, RCX, RDX and R8-11.

Callee-saved registers are RBX, RBP, RSP, RDI, RSI and R12-R15.

[2] Jcc is a family of instructions for conditional jumps such as JE, JNE, JG, JGE, JL, JLE.

2.1.2 System V AMD64 Calling Convention

The System V AMD64 calling convention is used by Linux, macOS and many other Unix-like operating systems.

The first six arguments are passed in registers RDI, RSI, RDX, RCX, R8 and R9 [8]. Additional arguments are passed on the stack. The return value is saved in RAX and RDX if it fits, otherwise the caller allocates space for it on the stack and sends the callee a pointer to it as the first argument in RDI. This means this calling convention can return two 64-bit values using just registers. There is no shadow space.

Caller-saved registers are RAX, RCX, RDX, RDI, RSI and R8-R11.

Callee-saved registers are RBX, RBP, RSP and R12-R15.

2.2 Memory Addressing

Many x86-64 instructions can take a memory operand. It is specified in square brackets and has three components: a base register, an index register with its scale and a displacement (offset). It can look like this: `mov rax, [r15 + rsi * 8 + 0x45]`. All components are optional. The scale can be 1, 2, 4 or 8. If it's 1, it can also be omitted. For example in `mov rax, [rcx + rdx]`, RCX is the base, RDX is the index scaled by 1 and there is no displacement.

2.3 Instruction Encoding

For educational purposes, it was also decided to write a custom assembler instead of using an existing one.[3] This section describes how x86-64 instructions are encoded into machine code.

The encoding of an instruction consists of the following components:

- Prefixes - optional, each has 1 byte.
- Opcode - 1-3 bytes.
- ModR/M byte - optional, 1 byte. Specifies the operands, either registers or part of the addressing mode. Can also contain an extension of the opcode.
- SIB byte - optional, 1 byte. Specifies part of the addressing mode.
- Displacement - optional, 1, 2 or 4 bytes. Represents an offset added to the memory address.
- Immediate value - optional, 1, 2, 4 or 8 bytes. Represents an immediate value used as an operand.

The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 [5], Chapter 2 is an indispensable resource for fully understanding this topic. It explains the information covered here in much greater detail and contains tables with the meanings of each field of the ModR/M and SIB bytes. Since the encoding depends on the given instruction and its form, Chapters 3-6 are also required to determine how a particular instruction is encoded. Furthermore, the AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions [9]

[3] As a result, a considerable amount of time was spent understanding and implementing the encoding of x86-64 instructions, especially the various addressing modes, with the expectation that they would be important when implementing the JIT compiler. This assumption proved to be incorrect. At the time of writing, the JIT compiler contains a total of three instructions using a memory operand, and none of them use the more complex form with an index register.

contains largely the same information presented differently and might be helpful for a better understanding.[4]

The most important prefix here is the REX[5] prefix. Its high 4 bits always contain the value `0b0100`. The low 4 bits are called W, R, X and B. REX.W specifies that the instruction is using 64-bit operands.

Each register has a number from 0 to 15. The original 8 32-bit registers have numbers 0-7 which requires 3 bits. Access to the other 8 64-bit registers is requires a 4th bit which can be specified in the REX prefix. REX.R/X/B provide an extra bit for registers specified in the ModR/M.reg, SIB.index and ModR/M.rm fields respectively. This means they are used when selecting registers R8-15.

The opcode determines which instruction is executed. Many instructions have several variants which accept different operands, often two registers, a register and a memory location or a register and an immediate value. The variants have the same mnemonic in assembly code but different opcodes when compiled to machine code.

The ModR/M byte has three fields:

- mod - 2 bits. Specifies the addressing mode - whether a register is used directly or if it's part of the addressing mode and whether a displacement is used and how many bytes it has.
- reg - 3 bits. Usually specifies a register, which operand depends on the instruction opcode. For some instructions it's used as an extension of the opcode.
- rm - 3 bits. Usually used to specify a register.

The SIB byte is used when the memory operand contains an index register. It also has three fields:

- scale - 2 bits. Specifies the scale of the index register. The 2 bits offer 4 different values, each corresponding to a different scale.
- index - 3 bits. Specifies the index register.
- base - 3 bits. Specifies the base register.

The displacement is an offset added to the memory address. Its length is determined by the mod field of the ModR/M byte.

The immediate value is used if the instruction takes an operand which is a constant. Its length is determined by the instruction opcode.

2.3.1 Special Cases

A closer examination of the tables related to the ModR/M and SIB bytes in the Intel manual reveals that there are some special cases which are not covered by the general description above.[6]

In particular, if the ModR/M.rm field is `0b100`, then the SIB byte is used. The value `0b100` corresponds to the ESP, RSP or R12 registers. When one of them is used as the base register, the SIB byte must be used. The SIB.base field is then set to `0b100` and the SIB.index field is set to its special value, which is also `0b100` and means no index register is used.

This special meaning of SIB.index applies regardless of the value of ModR/M.rm. This means ESP and RSP can't be used as the index register.

[4] Finally, to properly grasp how addressing modes are encoded, the author recommends finding a few worked examples such as https://www.systutorials.com/beginners-guide-x86-64-instruction-encoding/ and https://stackoverflow.com/questions/28664856/how-to-interpret-x86-opcode-map and going through them.

[5] REX stands for register extension

[6] These rules are rather hard to understand. The Intel manual offers Table 2-5. Special Cases of REX Encodings, which documents them in one place. The author found other descriptions helpful, such as https://stackoverflow.com/questions/52522544/rbp-not-allowed-as-sib-base.

Their register number is the same as R12. However that register requires REX.X to be 1 which means it can be differentiated from ESP and RSP. Therefore R12 can be used as the index register.

One more special case is EBP or RBP as base with ModR/M.mod set to `0b00`. This combination in 32-bit mode means the register is not used but a 32-bit displacement is used instead. However in 64-bit mode, this has been changed to mean RIP-relative addressing.

Chapter 3

Design of the Garbage Collector and JIT Compiler

3.1 Garbage Collector

There are many different approaches to garbage collection, each with its own strengths and weaknesses. In this project, the author chose to implement one of the simpler methods - the mark and sweep garbage collector, with support for compaction.

The mark and sweep algorithm works by starting in roots and walking through the heap while marking all reachable objects. [10] [11] Roots are runtime objects that are reachable directly - global variables, local variables and values on the stack. This is the mark phase. It then scans through the heap again and frees all unmarked objects. This is the sweep phase.

This would, however, leave holes in the heap. This is called fragmentation. The heap as implemented in FML is a simple vector of runtime objects and relies on the allocator of the host language. Normally the heap would be a vector of bytes and the objects would take up each a different number of bytes. If the holes became small, there might be enough free space for a new object in total but it would be impossible to allocate it since none of the holes would be big enough.

To solve this problem, the heap is compacted as part of the sweep phase. This is done in three passes. First the garbage collector calculates a new position for each live object. Then it updates all references between objects to these new positions. Finally, it moves the objects to their new positions.

3.2 JIT Compiler

3.2.1 Assembler

The JIT compiler uses a custom assembler that supports a small but sufficient subset of x86-64 instructions.

This assembler first encodes the instructions into an intermediate representation called Encoding which closely mirrors the components described in 2.3. It then serializes the Encoding into a sequence of bytes which is the machine code and machine code can be quite easily deserialized back into Encoding. This two-step approach proved to be very useful for multiple reasons:

- For testing because comparing two Encodings is much easier for a human than comparing two sequences of bytes. If a test of the assembler fails, it prints both Encodings and the

progammer can not only see which bytes are different but also which component of the encoding they correspond to.

- For debugging and exploration. It's possible to use an external compiler to generate machine code for a given instruction, especially with a complex memory addressing mode, then disassemble it to inspect the contents of the ModR/M and SIB bytes.
- Implementing both serialization and deserialization, then making sure their behavior matches over the entire test suite is a good way to decrease the chances of bugs.

3.2.1.1 Alternative Approaches

The author considered two alternative approaches instead of implementing a custom assembler.

The first was to use an existing library. For Rust, the dynasm [12] crate appeared most promising. It's a library for generating machine code at runtime. Dynasm is accompanied by the dynasmrt crate which provides a runtime that handles other tasks such as allocating executable memory.

The second was determining which exact instructions are needed by the JIT and serializing those using an external compiler and hardcoding them into the JIT compiler. The JIT compiler would then only need to patch certain parts of the machine code such as offsets in jumps and calls and immediate values. Originally the author believed even registers could be patched this way but over time he realized he would have to avoid the special cases described in 2.3.1.

Both approaches were rejected. Using an external library would take most of the complexity away so students exploring the reference implementation would learn less about the inner workings of a JIT compiler. Hardcoding the instructions and patching them later was deemed too risky without understanding the whole complexity of their format.

3.2.2 Calling Convention

Rust, the language used in the reference implementation has its own internal calling convention which is unstable. However, it needs to interact with functions generated by the JIT compiler. Therefore it was necessary to choose a calling convention. After evaluating both options presented in 2.1, the author chose System V AMD64 mainly because it allows passing more arguments and returning more values in registers, therefore reducing complexity by avoiding the stack in more cases.

3.2.3 Replacing the main interpreter loop

The core of the bytecode interpreter is a loop that reads the next opcode and branches to the corresponding handler. The simplest way a JIT compiler could speed up the program is by eliminating this overhead. The JIT compiler iterates through the bytecode and generates a block of native code for each opcode. Each block calls the handler of the original opcode. Additionally blocks which represent jumps (opcodes Jump, Branch, CallFunction, CallMethod and Returns) also have to perform a jump in the native code to the correct location. That way the block corresponding to the correct opcode is executed next. The interpreter still does most of the work but there's no overhead from branching and looping.

A key insight here is that control flow is now fully dictated by native code. The instruction pointer inside the VM is still getting updated but it no longer affects anyting. The second speedup comes from slightly changing the inteprter so it no longer updates its instruction pointer when executing jitted code.

This is a relatively simple way to improve performance using a JIT compiler but it only gives a small benefit.

3.2.4 Optimizing integer-only functions

A more complex but also more effective way to speed up the program is to convert some functions to native code entirely so they do all computation in native code without calling back into the interpreter. Depending on how advanced the JIT compiler is, this can be done for all functions or only those that meed certain criteria. The JIT compiler implemented in this thesis is very simple so the criteria for this optimization are quite strict. The only operations permitted are:

- integer arithmetic - method calls that resolve to the built-in methods
- boolean consitions where the result is used to branch - again method calls that resolve to the built-ins
- simple control flow - label, jump, branch and return opcodes
- local variables - set and get opcodes
- integer literals and the drop opcode

These conditions ensure that the native code doesn't accidentally interact with runtime objects it doesn't support and interpret then as integers because that would lead to undefined behavior. Functions that meet these criteria, however, can be sped up significantly.

With a bit more work the conditions could be relaxed. For example, allowing function calls to other functions that meet the same criteria would allow optimizing several more functions from the FML test suite.

Chapter 4

Implementation

The current implementation is based on the reference implementation written in Rust, which was used in previous semesters. The author of this thesis tried to keep most of the structure of the code. Noentheless, some things were simplified and reorganized. For example a fairly shallow change that affected most of the code is that it was formatted and items were grouped so that structures and enums are together at the top and `impl` blocks are together below them. This makes it easier to see at a glance the greater structure of the data and how individual objects relate to each other.

Similarly a lot of small values in many places which were passed by reference are now passed by value. Incidentally this was necessary to avoid invoking undefined behavior due to Rust's aliasing rules when these values were passed into native code generated by the JIT compiler. However, it is also more idiomatic in Rust.

The original implementation consumed a lot of memory because each object instance contained the `Method` structs for its methods. This was changed so that instances only contain indices into the constant pool. This also the correct behavior according to the specification.

4.1 Garbage Collection

The garbage collector is a fairly straightforward mark and sweep implementation. It's not optimized in any way but should be simple to understand.

It's all contained within a single function which takes the frame stack and the operand stack as arguments. They contain the roots necessary to begin tracing which runtime objects are still reachable.

The garbage collector is triggered when the heap is full and the program attempts to allocate. Because the JIT compiler optimizes into native code only functions that don't allocate, the garbage collector doesn't interact with the JIT in any way and doesn't need to traverse the native stack.

4.2 x86-64 Assembler

The assembler supports a number of arithmetic, locical, control flow and other instructions. They are represented as a Rust enum. Because many of them have several variants, accepting different types of operands, the enum can contain several variants per instruction.

Operands are represented as `Reg` and `Mem` objects. `Reg` allows specifying 32 and 64-bit reg-

isters. It currently doesn't support 8-bit registers.[1] `Mem` allows specifying all addresing modes using the 64-bit registers and 32-bit offsets. The assembler also supports 32-bit immediates for many instructions and 64-bit immediates for MOV.

4.3 JIT Memory

There are implementations of executable memory for Windows and for Unix-like systems. The Unix implementation uses `posix_memalign` and `mprotect` and properly makes sure the memory is never writable and executable at the same time. The Windows implementation uses `VirtualProtect`. Both implementations are tested on CI.

4.4 JIT Compiler

Due to time constraints, the JIT compiler is mostly implemented as a single function. It iterates through all functions and attempts to generate an optimized version for each one. If it encounters an unsupported construct, it gives up. The JIT compiler generated unoptimized fallback implementations for all functions.

The unoptimized fallback simply has a block of native code for each opcode which calls into the interpreter. Some opcodes such as `Label` can avoid using the interpreter entirely.

4.5 Testing

The x86-64 assembler contains a large number of tests. All instructions are tested with various combinations of operands, likewise all supported memory addressing modes are tested.

The file `src/tests/asm_compiler.rs` contains whole functions implemented using these instructions which are then executed using `JitMemory` and the result is checked to be correct.

These programs were often written to better understand how x86-64 assembly works. They showcase things such as accessing function arguments which didn't fit into registers and had to be put onto the stack. They're commented with explanations with the hope they'll be useful to students.

Tests like these also provide a quick and easy way to put together a few assembly instructions, calling them with arbitrary arguments and checking the result which is helpful when learning assembly.

[1] This might make implementing boolean operations difficult.

Chapter 5

Performance

High-performance was explicitly not a goal[1], instead the focus was on readability. Nonetheless, the implementation should be reasonably fast and should showcase the perf difference between interpreting bytecode and executing machine code directly.

Several FML programs available in the repository were benchmarked using the following commands:

- `hyperfine --warmup 2 "./fml run $PATH"`
- `hyperfine --warmup 2 "./fml run --jit --debug opt-disable $PATH"`
- `hyperfine --warmup 2 "./fml run --jit $PATH"`

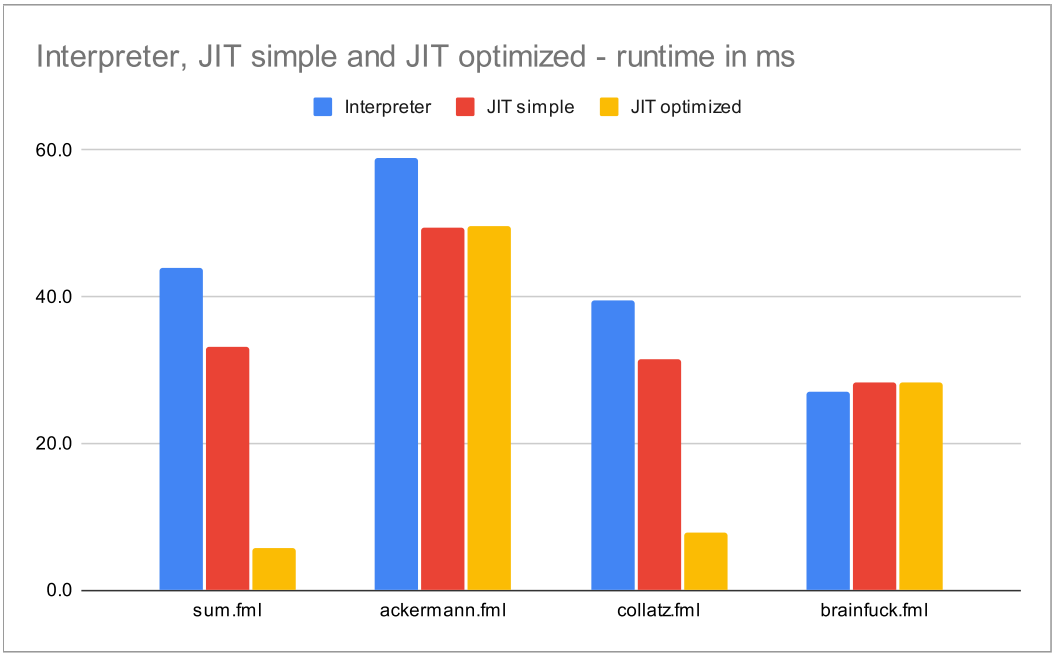
The following graphs compare the results.

Sum.fml and collatz.fml contain functions which satisfy the requirements needed to be compiled into pure native code without calls back into the interpreter. Such a program is then 5-8 times faster.

Most programs still benefitted from the JIT compiler, even if only the main interpreter loop was replaced and saw a roughtly 20% improvement.

Brainfuck.fml is an outlier and shows slightly worse performance with JIT.

[1] The author was told, and has to concur, that students find it motivating when their implementation is faster than the reference one.


Interpreter, JIT simple and JIT optimized - runtime in ms
Interpreter
JIT simple
JIT optimized
60.0
40.0
20.0
0.0
sum.fml
ackermann.fml
collatz.fml
brainfuck.fml

Chapter 6
Conclusions

In this thesis, we aimed to provide a high-quality reference implementation of the FML programming language that includes a garbage collector and a JIT compiler. We started by introducing the FML language and x86-64 assembly language. Then we described the design and implementation of the garbage collector and the JIT compiler. For the garbage collector, we chose to implement a mark-and-sweep algorithm with support for compaction. We also discussed the challenges of fragmentation and explained how our approach addressed this problem. For the JIT compiler, we developed a custom assembler and described how it translated FML bytecode into x86-64 machine code.

Finally, then evaluated how the JIT compiler affects the performance of our implementation.

Most of the time spent working on this thesis was dedicated to the custom assembler. It is unfortunate that we were not able to implement a more sophisticated JIT compiler which would properly use all the implemented instructions.

Overall, our implementation represents an large improvement over the original. We have addressed some of the shortcomings of the previous reference implementation, and provided a cleaner and more accessible codebase with additional features.

Bibliography

1. ING. KONRAD SIEK, PH.D. *FML language and runtime* [online]. 2023. [visited on 2023-05-03]. Available from: `https://github.com/kondziu/FML`.
2. BC. MICHAL VLASÁK. *cfml* [online]. 2023. [visited on 2023-05-03]. Available from: `https://gitlab.fit.cvut.cz/vlasami6/cfml`.
3. ING. KONRAD SIEK, PH.D. *Specs • NI-RUN • FIT CTU Course Pages* [online]. 2023. [visited on 2023-05-03]. Available from: `https://courses.fit.cvut.cz/NI-RUN/specs/index.html`.
4. ING. KONRAD SIEK, PH.D. *Specs • NI-RUN • FIT CTU Course Pages* [online]. 2022. [visited on 2023-05-03]. Available from: `https://courses.fit.cvut.cz/NI-RUN/@B212/specs/index.html`.
5. INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual* [online]. 2022. [visited on 2023-05-04]. Available from: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.
6. INTEL CORPORATION. *Introduction to x64 Assembly* [online]. 2022. [visited on 2023-05-04]. Available from: `https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf`.
7. MICROSOFT CORPORATION. *x64 calling convention* [online]. 2022. [visited on 2023-05-04]. Available from: `https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170&viewFallbackFrom=vs-2017`.
8. H.J. LU, MICHAEL MATZ, MILIND GIRKAR, JAN HUBIČKA, ANDREAS JAEGER, MARK MITCHELL. *System V Application Binary Interface* [online]. 2018. [visited on 2023-05-04]. Available from: `https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf`.
9. ADVANCED MICRO DEVICES. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions* [online]. 2022. [visited on 2023-05-04]. Available from: `https://www.amd.com/system/files/TechDocs/24594.pdf`.
10. ING. KONRAD SIEK, PH.D. *NI-RUN lecture, Memory Management: Garbage collection* [online]. 2022. [visited on 2023-05-04]. Available from: `https://courses.fit.cvut.cz/NI-RUN/lectures/6/index.html`.
11. ROBERT NYSTROM. *Crafting Interpreters* [online]. 2021. [visited on 2023-05-04]. Available from: `https://craftinginterpreters.com/garbage-collection.html`.
12. CENSOREDUSERNAME. *dynasm* [online]. 2023. [visited on 2023-05-04]. Available from: `https://crates.io/crates/dynasm`.

Contents of the Enclosed Medium

- `readme.txt` ..... short description of the contents of the medium
- `exe` ..... directory with the executable program
- `src`
  - `impl` ..... source code of the implementation
  - `thesis` ..... source code of the thesis in LaTeX format
- `text`
  - `thesis.pdf` ..... text of the thesis in PDF format