`mochi`: A Scheme Implementation for WebAssembly

Kosi Nwabueze[1] Anirudh Rahul[1]

[1]Massachusetts Institute of Technology
{kosinw, arahul}@mit.edu

Abstract

This article discusses the implementation and execution of a Scheme compiler and virtual machine operating on the WebAssembly binary instruction format. In particular, we detail the memory model discipline, bytecode specification, compiler design, and virtual machine execution behind our implementation.

1 Introduction

Scheme is a general-purpose programming language which features a minimal and homoiconic syntax inherited from the Lisp family of programming languages. Scheme is also minimal in its execution structure, embodying a little more than the core of the lambda calculus. However, Scheme has the capability to be more expressive than many modern languages due to features such as continuations and hygienic macros.

We propose a design for a special-purpose compiler and virtual machine for the Scheme programming language, lovingly named, **Mochi**. Specifically, Mochi targets a subset of the R^5RS Scheme standard [**?**]. Mochi is implemented using a custom stack machine architecture similar in structure to the Java Virtual Machine. By implementing a Scheme, we learned many of the powerful concepts fundamental to creating compilers such as creating intermediate representations, large-scale system design, and virtual machine architecture. We also learned why it can be difficult to implement some features that are fundamental to the Scheme programming language such as tail call optimization (partially working, but disabled in final build) and continuations.

Our custom stack machine would operate on top of WebAssembly and would be possible to use in a large variety of applications and platforms such as in the browser or in web applications frameworks such as Node.js. Our custom stack machine was partially inspired by the register machine from Chapter 5 of *Structure and Interpretation of Computer Programs*; however, we decided to pursue a stack machine design similar to the JVM.

We implemented out compiler, **Ricecakes**, using the TypeScript programming language. Our compiler emits a portable bytecode format which we have named **Flour**. The benefits of using TypeScript over something like MIT Scheme is mostly familiarity, as both authors have mostly experience using JavaScript. However, using TypeScript has the added advantage that we can compile both on the web and natively, using a platform like Node.js. Ultimately, if our implementation reaches a stable build we plan to bootstrap [2] our compiler and reimplement the compiler using Scheme.

We implemented our stack machine, **Dango**, in AssemblyScript, which is a highly optimized subset of TypeScript [1] designed to compile down to WebAssembly. The benefits of using AssemblyScript would be that it allows us to write our stack machine at a higher level of abstraction, and not worry about small details related to register

management and low-level control flow but focus on more important issues such as memory management.

WebAssembly is a relatively new, but widely adopted, portable binary instruction format designed to compile and run on browsers at native speeds. Other than speed considerations, some of the benefits of WebAssembly include the interoperability it provides with JavaScript and native Web APIs.

Compilers, some of which tend to be millions of lines of source code (such as Stallman's GCC), must be flexible and additive in order to be maintainable, two core themes present throughout this course. Alongside those themes, we believe writing a compiler will incorporate many concepts taught in 6.905 and coalesce them into one integrated project. Such concepts include but are not limited to generic procedures, pattern matching, register machines, compilation, and domain-specific languages.

2 Design Overview

In creating this Scheme compilation and execution ecosystem we wanted to emphasize the ideas we learned from 6.905 such as additive programming, flexibility, abstraction, creating domain specific languages, modularity, and engineering for extensibility. At the highest level we wanted to make our project modular by decoupling our compiler, byte code specification, and virtual machine into separate projects.

Our compiler **Ricecakes** is a TypeScript program which compiles Scheme into our Flour bytecode specification. We were particularly concerned about the scalability of in terms of functionality, such as adding more language features, supporting macros, and implementing tail calls. To combat the problem of scalability, we integrated concepts taught in 6.905 into the design of our compiler. We integrated domain-specific languages through a parser generator language developed for 6.031 to describe the grammar of our language. We integrated generic procedures/multimethods into TypeScript by using the popular arrows package (permissive license). Of course, we integrated compilation concepts from Chapter 5 (Evaluation) of *Software Design for Flexibility* [3].

Our bytecode format **Flour** is specified by a set of TypeScript files that specifies the exact details for serializing/deserializing data structures into bytecode. Such details include the opcodes for different virtual machine operations, the bit patterns and encodings of native values, and the assosciated metadata for programs as a whole. When designing Flour we wanted to make it as flexible as possible, opening up the possibility to introduce as many design changes as possible. Currently, Flour deals with 32-bit words and uses 32-bit addresses; however, we could easily modify our binaries to be 64-bit addressable if needed. Just as easily, we could easily make more changes to our bytecode specification in the future such as adding more opcodes and more native types.

Our virtual machine **Dango** is a WebAssembly program compiled from AssemblyScript that executes Flour bytecode in the browser. By executing our programs on our virtual machine Dango we are inherently making our system more extensible to other platforms. In the future we could accordingly re-implement our virtual machine to target different a platform as it is written in a high-level programming language similar to JavaScript.

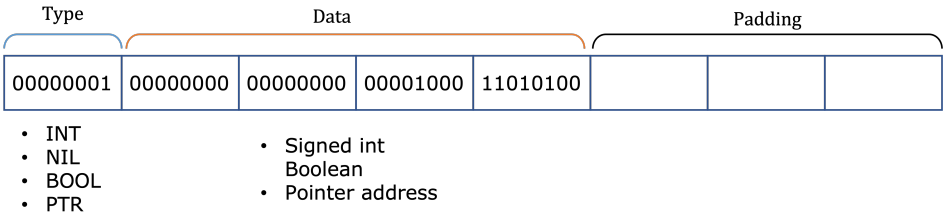

Figure 1: Flour unboxed values are represented in 8 bytes

3 Memory Model

Scheme, like most Lisp languages, is a high-level, garbage collected language which abstracts away menial memory management details away from its users. Due to this property, our implementation must have a way to represent complicated memory details embedded in its memory model. We manage all the data during the execution of a Scheme program with four primary data structures: *environments*, *the heap*, *the stack*, and *constant pools*.

3.1 Representation of Values

Since Scheme has a dynamic, but strong typing discipline, we must associate types with the values at run-time and not the variable bindings at compile-time in our implementation. Mochi's values can be broadly split into two categories: *unboxed* and *boxed* values.

Unboxed values appear directly on the run-time stack and are encoded as an 8 byte bit pattern. Unboxed values can be thought of as primitive values in other languages such as Java or TypeScript. Figure 1 describes how unboxed values are generally encoded in memory. In C, an unboxed value might be implemented by the following struct:

```
typedef struct {
    enum { FIXNUM, NIL, BOOLEAN, BOX, CHARACTER } type;
    union {
        int                 fixnum;
        bool                boolean;
        void*               box;
        char                c;
    } as;
} UnboxedValue;
```

The unboxed types we currently support are: *fixnums*, *nil*, *booleans*, *characters*, and *boxes*. Fixnums are fixed-size, 32-bit signed integers. Nil is a sentinel value which represents the Scheme empty list. Booleans are true or false. Characters are 7-bit ASCII encoded character values. Boxes are memory addresses to boxed values. For simplicity and alignment reasons we decided to store all unboxed types in a consistent 8 bytes of memory (3-6 bytes of padding).

Boxed values appear on the run-time heap and in the data sections of our memory model. Since boxed values are available on the heap, they are subject to garbage collection by the virutal machine. Boxed values represent dynamically sized objects (unlike the fixed-size unboxed values) such as symbols and pairs and are pointed to by unboxed values of type box. Boxed values can be thought of as somewhat equivalent to reference types in Java or objects in TypeScript. In C, a boxed value might be implemented by the following struct:

```
typedef struct {
    enum { PAIR, CLOSURE, SYMBOL, STRING, VECTOR } type;
    union {
        struct { UnboxedValue *car; UnboxedValue *cdr; } pair;
        char            *symbol_or_string;
        UnboxedValue    *vector;
        struct { int arity; Environment *enclosing; Chunk *code; } closure;
    };
} BoxedValue;
```

The boxed types we currently support are: *symbol*[1] and *closure*. Support is planned for pair, *string*, and *vector*. Symbols are immutable, interned sequences of characters which are typically used for equality checking (and are performance since its reference equality). Strings are similar to symbols; however, they are more designed for printing and are therefore not interned. Pairs are a set of pointers, car and cdr (used for historical reasons), which can be used to implement more complicated data structures such as lists and alists. Vectors are fixed-length, contiguous regions of memory. Closures are representations of Scheme lambda objects which contain information such as arity, the environment the closure was constructed in, and the corresponding chunk which represents the body of the lambda.

3.2 Environments

Our stack-based virtual machine at the moment features only a few special-purpose "registers". One of these registers is *IP*, or the instruction pointer, which represents the current location the virtual machine is in the program sequence. Another important register is *ENV*, or the environment register, which points to an environment data structure located on the heap. Environments are data structures which describe what variable names have bindings to what values (encoded as unboxed values) as well as possibly having a pointer to a parent environment. This data structure is analogous to the concept of an environment/frame in the Scheme programming language.

3.3 Calling Convention

As Dango is primarily a stack machine, both procedures and instructions primarily move intermediate values through pushing and popping off our stack data structure. Constant expressions and variable expressions push values onto the stack. Instructions such as conditional jumps, variable definitions, and variable assignments pop values off the stack.

The semantics behind procedure application are unique in our virtual machine. Procedures are typically called with a CALL instruction which pops off the first value on the stack (which must be a boxed closure object or primitive procedure) and executes that procedure. CALL instructions also have an extra *instruction argument* which specifies the number of values from the top of the stack *procedure arguments* which will be removed by the procedure. The virtual machine then creates a new environment and binds the arguments to the formal parameters of the procedure through variable declaration instructions. The return value of the procedure will be the last value on the stack after a RETURN intsruction.

[1] Certain details such as the interning table are currently work in progress.

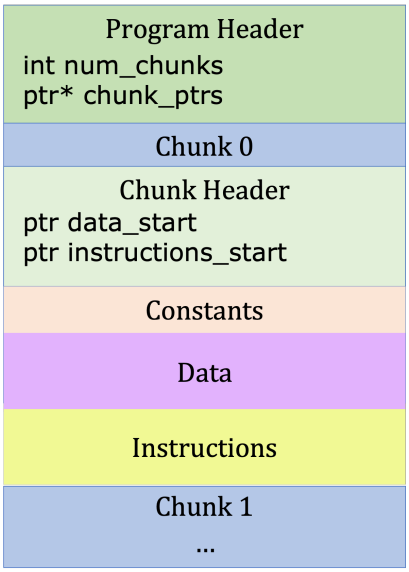

Figure 2: Flour object program format

3.4 Memory Layout

Alongside the stack and environment data structures, there are two addressable pages in our memory model. The first addressable space includes information provided by the binary format emitted by the compiler. Constant pools (both consisting of boxed and unboxed types) are available in this addressable space. Constant pools consist of values laid out in contiguous memory (with alignment) which appear as literals in a Scheme program (such as symbols like 'foo or fixnums such as 42).

The second addressable space includes memory objects allocated dynamically at runtime. This space is a heap data structure subject to stop-and-copy garbage collection. Boxed values which are created during the execution of a program are located here. The roots of the garbage collector include all values on the stack, all values pointed to by the *ENV* and *IP* registers, and values located in constant pools.

4 Bytecode Format

Flour object programs are a portable binary format inspired by the design of class file JVM format and the System V ELF format. Flour object programs have big-endian byte ordering[2]. Object programs are split up into a series of sections called chunks which follow an object header which has a table describing the starting addresses of every chunk in the program. Figure 2 has a graphical depiction of how an object program may look like. Figure 3 shows a hexdump of a factorial Scheme program compiled to the object format. Word sizes are 4 bytes long for this architecture (meaning that addresses and register sizes are 32-bit).

[2]This design decision was due to a mistake when writing the reference implementation.

```
00000000: 46 4c 4f 55 52 00 00 00  00 00 00 03 00 00 00 18 |FLOUR...........|
00000010: 00 00 00 50 00 00 00 8d  2a 6d 6f 64 75 6c 65 2a |...P....*module*|
00000020: 00 00 00 40 00 00 00 40  00 00 00 00 00 00 00 00 |...@...@........|
00000030: 02 00 00 00 01 00 00 00  02 00 00 00 00 00 00 00 |................|
00000040: 0a 00 00 01 01 01 00 00  00 68 02 00 00 00 68 08 |.........h....h.|
00000050: 28 63 68 6b 20 31 29 00  00 00 00 78 00 00 00 78 |(chk 1)....x...x|
00000060: 00 00 00 00 00 00 00 00  02 00 00 00 01 00 00 00 |................|
00000070: 02 00 00 00 00 00 00 00  01 00 00 00 64 0a 00 00 |............d...|
00000080: 02 02 01 00 00 00 67 02  00 00 00 67 08 28 63 68 |......g....g.(ch|
00000090: 6b 20 32 29 00 00 00 00  cd 00 00 00 cd 00 00 00 |k 2)....Í...Í...|
000000A0: 00 00 00 00 00 02 00 00  00 01 00 00 00 02 00 00 |................|
000000B0: 00 00 00 00 00 01 00 00  00 00 00 00 00 01 00 00 |................|
000000C0: 00 01 00 00 00 01 00 00  00 01 00 00 00 01 00 00 |................|
000000D0: 00 65 01 00 00 00 66 02  00 00 00 66 00 00 00 00 |.e....f....f....|
000000E0: 03 02 00 00 00 07 07 00  00 00 02 06 00 00 00 0a |................|
000000F0: 02 00 00 00 65 05 00 00  00 32 02 00 00 00 66 00 |....e....2....f.|
00000100: 00 00 00 04 02 00 00 00  00 07 00 00 00 02 02 00 |................|
00000110: 00 00 65 02 00 00 00 66  02 00 00 00 01 07 00 00 |..e....f........|
00000120: 00 02 02 00 00 00 67 07  00 00 00 02 09 02 00 00 |......g.........|
00000130: 00 64 00 00 00 00 05 02  00 00 00 67 07 00 00 00 |.d.........g....|
00000140: 02 08                                           |..              |
```

Figure 3: Hexdump of a factorial Flour object program

4.1 Object Header

Flour object headers start with an 8-byte signature that identifies the file as a Flour object file. In ASCII, this sequence are the letters *FLOUR* followed by three bytes of padding. The signature is then followed by a 4-byte unsigned integer, N, which tells the virtual machine how many chunks are in the program. After that sequence, is an array of size N which contains 4-byte memory addresses to the beginning of each chunk. In C, this header would be represented as:

```
typedef struct {
    char        signature[8];
    uint32_t    num_chunks;
    ChunkHeader *chunk_start[num_chunks];
} ObjectHeader;
```

4.2 Chunk

Flour *chunks* are the fundamental, manipulable units of code and data in our portable bytecode format. Abstractly, a chunk has a one-to-one correspondence to the body of a lambda expression in the Scheme source code. A chunk is a section of binary data beginning with a header followed by a vector of unboxed values (a pool of unboxed constants) used as literals in the source code of the chunk. Following the unboxed constant pool is a boxed value constant pool which holds all the boxed constants which appear in the source code. Finally, the chunk ends with an instructions section which contains the series of bytecode instructions to be executed by the virtual machine.

A chunk header begins with an 8-byte sequence which is a unique identifier for that chunk. Following that sequence is the address of the boxed constant pool. Following

that value is the address to the first instruction of the chunk. In C, the header of a chunk would be represented as:

```
typedef struct {
    char        signature[8];
    BoxedValue  *boxed_pool;
    Instruction *instructions;
} ChunkHeader;
```

5 Compiler Design

Ricecakes is a multi-pass, ahead-of-time compiler for the Scheme programming language. Ricecakes currently features very few optimization techniques found in many modern compiler; however, since we have introduced flexibility and additive design into our system through generic procedures, adding optimizations such as tail call, constant propagation, and dead code elimination would not be a destructive process. Ricecakes understands source code which abides by the Scheme formal syntax and targets the Flour bytecode object format. In the following sections we will discuss the design behintd each individual pass the compiler applies.

5.1 A Simple Program

In the following sections we will consider a simple Scheme program to demonstrate the intermediate outputs generated by a correct implementation of Ricecakes. Below is an iterative program which compiles the factorial of an integer.

```
(define (factorial x)
  (define (fact-iter a product)
    (if (= a 0)
      product
      (fact-iter (- a 1) (* product a))))
  (fact-iter x 1))
```

5.2 Lexical & Syntax Analysis

The first step to compiling any non-regular formal language is parsing. Parsing is a topic which can get very involved especially on the theoretical side. However, due to the Polish notation property of LISP, parsing and lexing becomes a relatively uninteresting topic to discuss. At this point, the Ricecakes compiler parses *lazily*, meaning that at any stage during code generation, the parser will only generate as much of an abstract syntax tree needed.

The tokenizer will generate tokens on demand. In our reference implementation, we implement the tokenization rules through a generic procedure which uses a two-character lookahead to dispatch on what token to output next. In TypeScript, here is what our token type definition looks like:

```
type Token =
    | { variant: TokenVariant.DATUM, value: Datum }
    | { variant: TokenVariant.READER_MACRO, value: ReaderMacro }
    | { variant: TokenVariant.LEFT_PAREN }
    | { variant: TokenVariant.RIGHT_PAREN }
    | { variant: TokenVariant.DOT }
    | { variant: TokenVariant.EOF };
```

The parser will then consume only the tokens it needs from the lexing stream and generate tagged syntax trees which contains type information so that pattern matching is possible during the code generation step. In TypeScript, here is what our abstract syntax tree definition looks like:

```
type SyntaxTree =
    | { variant: SyntaxTreeVariant.ATOM, value: Datum }
    | { variant: SyntaxTreeVariant.LIST, value: SyntaxTree[] }
    | { variant: SyntaxTreeVariant.EOF };
```

5.3 Code Generation

This stage of the compiler works via the same mechanism as the compilers/evaluators described in Software Design for Flexibility. The compiler describes generic procedure handlers based on a predicate which checks the contents of the abstract syntax tree. For example, the compiler has a isLetExpression predicate which then emits bytecode for let expressions and recursively emits bytecode for all subexpressions. Accounting for all cases and special forms is the most time-consuming part of writing the compiler and is mostly not novel. Figure 4 shows an example of the disassembled bytecode of the factorial program.

5.4 Serialization and Linking

This stage of the compiler was also time-consuming to design; however, it packs all of the compiled code (which are represented by JavaScript objects after code generation) into a binary format as seen in Figure 3. This step was also implemented using generic procedures dispatching over types ranging from boxed values (such as fixnums or booleans) to entire chunk headers. The binary format scheme was described in more detail in chapter 4.

5.5 Additional Passes

At the moment, the reference implementation contains source code for a broken version of tail call optimization (has been disabled). However, the compiler is planned to support an additional optimization passes which rewrites programs in continuation-passing style (which gives automatic tail calls). In addition, a hygienic macro expansion system (based on syntax-case) is also planned.

```
=== *module* ===
00000000    1 CLOSURE          00000101
00000005    | DEFINE_VARIABLE  00000068 'factorial'
0000000a    | GET_VARIABLE     00000068 'factorial'
0000000f    | RETURN
=== (chk 1) ===
00000000    1 DEFINE_VARIABLE  00000064 'x'
00000005    2 CLOSURE          00000202
0000000a    | DEFINE_VARIABLE  00000067 'fact-iter'
0000000f    | GET_VARIABLE     00000067 'fact-iter'
00000014    | POP
00000015    6 GET_VARIABLE     00000064 'x'
0000001a    | CONSTANT         00000003 '1'
0000001f    | GET_VARIABLE     00000067 'fact-iter'
00000024    | CALL             00000002
00000029    | RETURN
=== (chk 2) ===
00000000    2 DEFINE_VARIABLE  00000065 'product'
00000005    | DEFINE_VARIABLE  00000066 'a'
0000000a    3 GET_VARIABLE     00000066 'a'
0000000f    | CONSTANT         00000003 '0'
00000014    | GET_VARIABLE     00000007 '='
00000019    | CALL             00000002
0000001e    | JUMP_IF_FALSE    0000000a <offset 0000002d>
00000023    4 GET_VARIABLE     00000065 'product'
00000028    | JUMP             00000032 <offset 0000005f>
0000002d    5 GET_VARIABLE     00000066 'a'
00000032    | CONSTANT         00000004 '1'
00000037    | GET_VARIABLE     00000000 '-'
0000003c    | CALL             00000002
00000041    | GET_VARIABLE     00000065 'product'
00000046    | GET_VARIABLE     00000066 'a'
0000004b    | GET_VARIABLE     00000001 '*'
00000050    | CALL             00000002
00000055    | GET_VARIABLE     00000067 'fact-iter'
0000005a    | CALL             00000002
0000005f    | RETURN
```

Figure 4: Disassembly of a factorial Flour object program

6 Virtual Machine Design

The Dango virtual machine is a stack-based virtual machine which supports unstructured control flow and automatic memory management. This stack machine is a necessary abstraction over the WebAssembly execution model which supports neither arbitrary control flow nor automatic memory management which are requirements to create efficient Scheme implementations. Arbitrary control flow is necessary to implement optimizations such as tail call which is required by the formal specification of the language. Automatic memory management is required as the semantics of Scheme do not describe how to allocate or deallocate memory.

Below is a complete set of opcodes which the virtual machine understands how to execute.

OP	ARGS	RES
`PRINT`		`print(stack.pop())`
`CONSTANT`	x	`stack` ← `Constant(x)`
`DEFINE_VARIABLE`	x	`ENV.define(x, stack.pop())`
`SET_VARIABLE`	x	`ENV.set(x, stack.pop())`
`GET_VARIABLE`	x	`stack` ← `Environment.get(x)`
`JUMP`	x	jump ahead x bytes
`JUMP_IF_FALSE`	x	if not `stack.pop()` jump ahead x bytes
`CALL`	x	call closure at `stack.pop()` with x args
`RETURN`	a	return a
`CLOSURE`	x	`stack` ←* `Closure`(x_1, x_2)

Note 1: a is the 2nd to last stack element and b is the last stack element. And x represents a 4 byte argument accompanying the instruction, typically in the form of an unsigned integer.
Note 2: x_1 is the chunk number containing the instructions the closure should run encoded in the first 3 bytes of x, and x_2 is the number of arguments the closure accepts encoded in the last byte of x.

7 Conclusion

Over the span of this project, we have found that designing and writing a rudimentary compiler, custom bytecode, and virtual machine was a lot of work! We have little experience in compiler and programming langauge design so we are impressed and satisfied with the work we have completed so far. Although we may not have been able to implement more complicated features such as a continuation-passing style rewriter or hygienic macro expansion, we believe the amount of work we put into this project was a worthwhile experience.

The complete source code for the Mochi project is licensed under a GNU LGPL v3 license. Please try our reference implementation at the official website: `https://mochi-scheme.github.io/`.

A copy of this paper can also be found at `https://mochi-scheme.github.io/paper.pdf`.

Source code is available at the project repository: `https://github.com/kosinw/mochi`.

References

[1] URL https://www.typescriptlang.org/.

[2] Apr 2022. URL https://en.wikipedia.org/w/index.php?title=Bootstrapping_(compilers)&oldid=1085162524. Page Version ID: 1085162524.

[3] Christopher Hanson, Gerald Jay Sussman, and Guy L. Steele. *Software design for flexibility: how to avoid programming yourself into a corner*. The MIT Press, Cambridge, Massachusetts London, England, 2021. isbn:9780262362474.