# RASM: COMPILING RACKET TO WEBASSEMBLY

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Grant Matejka

June 2022

COMMITTEE MEMBERSHIP

| | |
|---|---|
| TITLE: | Rasm: Compiling Racket to WebAssembly |
| AUTHOR: | Grant Matejka |
| DATE SUBMITTED: | June 2022 |
| COMMITTEE CHAIR: | John Clements, Ph.D.<br>Professor of Computer Science |
| COMMITTEE MEMBER: | Aaron Keen, Ph.D.<br>Professor of Computer Science |
| COMMITTEE MEMBER: | Stephen Beard, Ph.D.<br>Professor of Computer Science |

ABSTRACT

Rasm: Compiling Racket to WebAssembly

Grant Matejka

WebAssembly is an instruction set designed for a stack based virtual machine, with an emphasis on speed, portability and security. As the use cases for WebAssembly grow, so does the desire to target WebAssembly in compilation. In this thesis we present Rasm, a Racket to WebAssembly compiler that compiles a select subset of the top forms of the Racket programming language to WebAssembly. We also present our early findings in our work towards adding a WebAssembly backend to the Chez Scheme compiler that is the backend of Racket. We address initial concerns and roadblocks in adopting a WebAssembly backend and propose potential solutions and patterns to address these concerns. Our work is the first serious effort to compile Racket to WebAssembly, and we believe it will serve as a good aid in future efforts of compiling high level languages to WebAssembly.

# ACKNOWLEDGMENTS

Thanks to:

- God, and my lord Jesus Christ for blessing me in this life and giving me hope for the future.
- My wife for constantly encouraging me and having my back during the long days.
- My family and parents for supporting my academic career.
- My advisor, Dr. John Clements, for helping me throughout this process.

TABLE OF CONTENTS

APPENDICES

# LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# INTRODUCTION

In this thesis I present my initial work and findings in compiling the Racket[27] programming language to WebAssembly[6], or Wasm. I present and evaluate Rasm[37], a Racket to WebAssembly compiler that supports a subset of the Racket language. I also investigate what requirements and roadblocks presented themselves in our early efforts of adding a WebAssembly backend to the currently existing Racket backend[29], Chez Scheme compiler.

WebAssembly is now a W3C recommendation[22] and one of the four native languages of the web[4]. This support and encouragement has driven the need to bring more technologies into the WebAssembly ecosystem. WebAssembly is a lower level language than the existing programming language of the web, JavaScript, and serves as a target language to compile to. There is no other serious effort in attempting to target WebAssembly from Racket and our contributions to this field is a minimum viable product compiler[37] and some initial findings in adding a WebAssembly backend to the Chez Scheme compiler that sits under Racket.

We also include an outline of our investigation into adding a WebAssembly backend into the current Racket Chez Scheme backend, in which we acknowledge roadblocks discovered in our research and propose solutions that will aid future efforts in adding a WebAssembly backend to Racket.

In chapter 2, we discuss other efforts in compiling languages to WebAssembly, and what other efforts have been made in compiling Racket specifically. We discuss early

efforts in compiling Racket to WebAssembly, but also more established efforts of compiling Racket to Python and JavaScript.

In chapter 3, we will take a more in-depth look at WebAssembly. We will discuss interesting aspects of WebAssembly and what sets it apart from more traditional assembly languages.

Chapter 4 discusses the Racket programming language, what unique features the language has to offer and what challenges the language presents when targeting WebAssembly as a compilation target.

Chapter 5 discusses the Chez Scheme Racket backend and what it looks like to compile Racket to a new compilation target.

In chapter 6, we address our initial findings and roadblocks in adding a WebAssembly backend to the Chez Scheme compiler. We go over how arbitrary jumping and the function application structure in the Chez Scheme generated code do not fit well into WebAssembly, and what mitigations we have come up with to address these concerns.

In chapter 7, we discuss the compiler we built and the details of the implementation. We will discuss important areas of study we relied on in compiling Racket, techniques for validating our compiler throughout development and what compromises had to be made.

Due to the continually evolving WebAssembly environment, compiling languages to WebAssembly changes very often and in the future work section we will share some features and efforts that could aid in compiling other high level languages to WebAssembly.

Chapter 2

# RELATED WORK

We will now discuss a project that aims to compile Racket to WebAssembly, and identify, describe and discuss more established projects compiling Racket to Python and JavaScript. We will then discuss some of the most mature projects compiling other programming languages to WebAssembly.

## 2.1 Compiling Racket to WebAssembly

The only existing project in compiling Racket to WebAssembly is wacket[44]. However, it is not meant to be considered a functional compiler[31]. The compiler only operates on arithmetic expressions, and the only supported operations are binary arithmetic with 32 bit integers.

## 2.2 Transpiling Racket

While there has not been much effort in compiling Racket to WebAssembly, there have been other projects that sought to compile Racket to other high level languages.

### 2.2.1 Pycket

The first project we will look at is Pycket[18][43]. It compiles Racket programs to the Python programming language. The work here was highly influential to other projects such as RacketScript and eventually Rasm.

Pycket is built upon the RPython[15] framework, which is a framework for implementing dynamic languages in Python, and thus the generation of machine code does not take place in the existing Racket stack, and the project relies on a separate backend than the current Chez Scheme backend of Racket.

There are two modes in Pycket, OLD and NEW. The OLD mode translates fully expanded Racket programs to a JSON abstract syntax tree, and then proceeds to evaluate this AST in Python, while the NEW mode builds from Racket's linklets.

### 2.2.2 RacketScript

The closest effort to our project is RacketScript[45], which compiles Racket to JavaScript. In its existing implementation, RacketScript compiles fully expanded Racket programs to JavaScript. However, due to the higher level nature of the JavaScript programming language, RacketScript was able to lift higher level features of Racket into JavaScript. Some features and structures of Racket were able to be reimplemented almost directly in JavaScript, while they would require extensive reworking in a WebAssembly environment.

Examples of these structures include lists and list operations. WebAssembly has no form of lists or arrays, so supporting this behavior requires an implementation from scratch, while JavaScript supports lists and a plethora of other operations out of the box. While RacketScript was able to translate nearly directly to these higher level forms, we had to translate them over to much lower level constructs.

However, the success and community's support of this project showed the clear desire for bringing Racket to other environment, and specifically the web.

## 2.3 Compiling Languages to WebAssembly

There are many efforts and languages that desire to take advantage of the WebAssembly ecosystem, and in this section we will discuss some of the largest efforts in compiling to WebAssembly.

### 2.3.1 Emscripten

Emscripten[47] is one of the most commonly used WebAssembly compilers. This compiler is built upon the LLVM ecosystem and compiles C/C++ to WebAssembly, and can even compile LLVM IR to WebAssembly. Support for the C/C++ standard libraries is very good, and most portable programs can be compiled over to WebAssembly with minimal effort. This theoretically means that any language that can compile to the LLVM IR can then be compiled to WebAssembly.

This compiler is used to port legacy code bases to WebAssembly. One such example is AutoCAD, which has been compiled to WebAssembly[21], and is now available as a web app. This application is much too computationally demanding to be built upon JavaScript, but now, with WebAssembly, it has been ported over to the browser. This project is built upon decades-old C/C++, has heavy computational needs, and served as a great use case of both Emscripten and WebAssembly.

### 2.3.2 AssemblyScript

AssemblyScript[12] is a programming language designed to behave very similarly to JavaScript and compile directly to WebAssembly. AssemblyScript has a custom compiler and it relies on Binaryen[11], which is the compiler toolchain for Web-

Assembly that Emscripten relies on. AssemblyScript aims to operate as similarly to Typescript[39], a typed superscript of JavaScript, while still compiling cleanly to WebAssembly. The entire project was built with an emphasis on familiarity of existing programming paradigms, with an emphasis on the compilation to wasm.

However, the AssemblyScript project is still missing some serious features that prohibit some projects from adopting it. One example is AssemblyScript's lack of support for closures. The team says they are waiting for the function reference and garbage collection WebAssembly proposals to be finished, but the lack of closure support is one of AssemblyScript's biggest weaknesses as JavaScript developers have come to expect this functionality.

Chapter 3

# WEBASSEMBLY

WebAssembly is an instruction set architecture, ISA, for a secure, stack-based virtual machine. While WebAssembly was originally intended for the web, the team recognized the importance of other embedding environments and has left the door open for out of browser embeddings. We will now discuss the motivation surrounding the WebAssembly project, the language specification for WebAssembly and the WebAssembly runtime.

## 3.1 Motivation for WebAssembly

WebAssembly is the latest effort in empowering developers on the web to incorporate complex client-side logic into their apps. Throughout the history of the web, there have been many efforts with this same goal (e.g. Java Applets, Adobe Flash, Microsoft's ActiveX, Google's Native Client), but for reasons we will discuss, these projects have failed to satisfy this goal.

### 3.1.1 Early Efforts Toward Client Side Compute

One of the first efforts towards the goal of client side compute was Java Applets. Java Applets were browser plug ins that, as the name suggests, were powered by Java. However, major difficulties arose for the technology as malware was rampant, which led to questionable reliability, and these applets were memory hungry[35]. Java Applets were relevant at a time when browsers and JavaScript were young and Java graph-

ical user interface, GUI, tooling sparse, so as the ecosystem attempted to mature, JavaScript and Flash won out over Java Applets. Throughout this time JavaScript continued to get a lot faster and optimizations for the technology continue to be developed. Flash was the most successful effort before WebAssembly. Adobe Flash support officially ended just recently[30]. Flash was a plug in created and owned by Adobe that allowed multimedia content to be available in the browser. This technology allowed many games and audio/visual streaming applications to finally come to the web. Flash was plagued with frequent bugs, security flaws and malware uses[38] and ultimately it was never allowed on the iPhone[32]. Other efforts include Google's native client[46] (NaCL) and Microsoft's ActiveX. Native Client support ended in 2021[3] and only a subset of ActiveX features are available in a special configuration of Microsoft's Edge browser[2], for legacy web applications. Throughout the course of all these projects, the JavaScript ecosystem has continued to grow and mature and is currently the best option for front end web development.

### 3.1.2 JavaScript

According to the Stack Overflow developer survey, JavaScript is the most used programming language, with 64.8% of developers using the language[5]. While JavaScript has proven itself to be a useful programming language, it has had issues living up to the consistent performance desires and flexibility of the developer community. Now WebAssembly serves as an alternative. There are various performance advantages of WebAssembly, such as smaller download sizes thanks to the bytecode format rather than text[16], more consistent performance and lower level optimization abilities. JavaScript is great at performing simple tasks and DOM manipulation, but for heavy computations WebAssembly performs much better. WebAssembly is not meant to replace JavaScript, but it allows developers to outsource any complex calculations to

a WebAssembly module in their JavaScript applications. There is a very well defined and easily accessible WebAssembly API in the current JavaScript ecosystem, so developers can compile their projects to WebAssembly and easily integrate them into their projects and the web ecosystem in general.

## 3.2 Language Specification of WebAssembly

In 2017, the minimum viable product for the WebAssembly API and binary format was completed and every browser vendor had a WebAssembly runtime implemented. WebAssembly at its core is a language specification. This language is in the form of a bytecode, but also has a specified textual format where developers can hand code programs in an s-expression format. This language was specifically crafted around speed, portability and security and was the fourth W3C recommended language that could execute code in a browser[13]. The other languages are HTML, CSS and JavaScript. There are many aspects of WebAssembly that make it very different from existing assembly languages. Some of these aspects include WebAssembly's stack machine execution, strict static typing, high level control flow and structured branching.

### 3.2.1 Stack Machine

WebAssembly is a stack based runtime. This is a major difference between WebAssembly and other assembly type languages. All instructions of WebAssembly operate on or manipulate the stack. For example, a function like `i32.add` would pop two values off the stack, and would expect both of the values to be of the `i32` type. If the type contract is broken, then an exception would occur when instantiating the WebAssembly module. The developer has very minimal control over the stack and only has access to higher level stack operations in which values are popped from or pushed

onto the stack. Function calling conventions are provided by WebAssembly, and the developer has no access to the call/return behavior of the code. While this is expected in a stack machine environment, this is another aspect in which WebAssembly differs from traditional assembly languages where developers have the freedom to manipulate the calling conventions as they please.

There are no registers in WebAssembly, but rather there are global and local variables, as well as provided functions to get/set these variables. An example of a basic WebAssembly module is shown in figure 3.1.

```
1  (module
2    (func $add
3      (param i32) (param i32)
4      (result i32)
5      local.get 0
6      local.get 1
7      i32.add
8    )
9
10   (func $four
11     (result i32)
12     i32.const 2
13     i32.const 2
14     call $add
15   )
16 )
```

**Figure 3.1: Example of WebAssembly's Stack Machine Operations**

Figure 3.1 is a simple WebAssembly module that defines two functions. The first function, defined on line 2, is referred to as `$add`, and it takes in two parameters of the `i32` type and returns the result of adding them together. The second function, defined on line 10, is referred to as `$four`, and it returns the result of two plus

two. Any function calling behavior must fit into these higher level constructs and we have little control over the structure of the stack itself and we must rely on solely pushing/popping values and calling functions.

### 3.2.2 Strict Static Typing

WebAssembly only has four data types. These types are integers and floats, with 32 and 64 bit variants of each. These are the only supported types and any more complex data must be able to be represented in these forms. WebAssembly memory is an array of bytes that is indexable with a 32 bit integer, because of this, we use `i32` values as pointers to objects in memory. When interacting with the host environment, any value passed to a WebAssembly function must be of these types or converted to them, and any returned value will of these numeric types. Our implementation of Rasm provides helper functions to convert JavaScript values to these WebAssembly types and helper functions to convert returned WebAssembly values to their JavaScript counterpart. We will discuss these helper functions later in section 7.4.3. In WebAssembly every variable needs to have a type identifier, every function needs to have types for every parameter and a return type and every block has to have a type annotation for any values at the top of the stack when the block begins and any final values left on the stack. All of these types must be annotated at compile time, and if any type contract is broken at the module's instantiation, an exception will be thrown.

Figure 3.2 shows an example of a complete WebAssembly function with all proper type annotations in the s-expression format of textual WebAssembly.

```
(module
  (func $max
    (param i32) (param i32)
    (result i32)
    (if (result i32)
      (i32.lt_s
        (local.get 0)
        (local.get 1))
    (then (local.get 1))
    (else (local.get 0)))
  )
)
```

**Figure 3.2: Example of WebAssembly's Types and Control Flow**

In this program we are comparing two parameters and returning the maximum parameter value. As you can see in this program, parameters are regarded and scoped as local variables in a WebAssembly function, and you can dereference variables based on their index or label. WebAssembly provides some basic arithmetic operations for each data type and some operations have signed and unsigned variants where necessary, as observed in our use of `i32.lt_s`, the signed 32 bit integer less than instruction. As seen in the example above, WebAssembly also has specific instructions for handling control flow.

### 3.2.3 High Level Control Flow

WebAssembly relies on a small set of control flow instructions that only allow for structured control flow. The block structures of control flow are conditionals, blocks and loops. Every block must be bracketed with an `end` instruction. We will discuss the specifics of the block and loop forms in the next section. WebAssembly also has

a specific call and call_indirect instructions for function calls. The returned values of the call will be pushed onto the top of the stack. The call instruction takes a label to a function to call, while the call_indirect is used for calling function pointers, and relies on an index into a predefined function table along with the function's type to safely call the function. Another interesting instruction is WebAssembly's `unreachable` instruction, which results in an exception if the instruction is ever reached. Due to the fact WebAssembly has no specific trap instruction, developer can either use `unreachable` to simulate trap in certain situations or use imported functions from the host that throw an exception. In Rasm, we utilize `unreachable` to throw a runtime exception in our provided standard library when a function is called with incorrect parameter types. In total, the control flow instructions in WebAssembly are `call`, `call_indirect`, `return`, `if..else`, `block`, `loop`, `br` and its variants and `unreachable`, where `unreachable` can serve as a trap operation, but there is no native WebAssembly support for interrupts or arbitrary branching.

### 3.2.4 Structured Branching

The only way to jump to specific blocks in WebAssembly is with the branch instruction, `br`. The WebAssembly branch instruction is very different from a traditional branch instruction. A traditional branch is able to jump to arbitrary locations or labels, but this is not the case in WebAssembly. In WebAssembly, developers can only branch to blocks the instruction is nested under. The instruction takes as argument either an integer of how many nested blocks to jump out of, or the label of a nesting block that can then be popped out to.

Figure 3.3 is an example of this behavior. In this example, we label every block with its index relative to the branch instruction at line 19. We also include the label, `$LOOP`, on the `loop` block, which can also be used to branch to the start of the

loop. It is important to note that we can only jump to blocks we are nested under. For example, if we were to attempt to `br $LOOP` at line 4, our module would throw an exception as we are not nested under any block with the label `$LOOP` at that point.

```
1  (module
2    (func (export "ex")
3      (local i32)
4      (local.set 0 (i32.const 1))
5  .   ;; block 2 relative to br instruction
6      (block
7        ;; block 1 or $LOOP
8        (loop $LOOP
9          ;; block 0
10         (if
11           (local.get 0)
12           (then
13             ;; decrement local variable
14             (local.set 0
15               (i32.sub
16                 (local.get 0)
17                 (i32.const 1)))
18             ;; jump to beginning of loop
19             (br $LOOP)
20           )
21           ;; Jump to end of block 2 above
22           ;; This serves as a loop break
23           (else (br 2)))
24       )
25     )
26   )
27 )
```

**Figure 3.3: Example of WebAssembly's Branching Constructs**

Branching to a block/conditional jumps to the end of the block, while branching to a loop jumps to the beginning of the loop. This allows for developers to recreate any more complex control flow[20], in a more structured and secure way.

Some of the benefits of WebAssembly's structured control flow are that modules' validity are preserved, as arbitrary jumps are impossible and irreducible loops cannot occur. An irreducible loop is a loop with multiple entry points to its body, and an irreducible loop can be caused by arbitrary jumps or compiler optimizations. These loops make instruction scheduling very difficult and operating under the possibility of these loops requires some form of transformation[41]. Irreducible loops have been an area of challenge for optimization in JIT compilers[24], and with structured control flow, there is no way for an irreducible loop to occur in WebAssembly.

Due to WebAssembly being unable to jump to arbitrary locations or labels, this structured control is a major difference between WebAssembly and more traditional assembly languages, and this difference proves to be a major hurdle in compiling languages to WebAssembly that we will discuss in more detail later.

### 3.3 WebAssembly Runtime

Due to the runtime nature of WebAssembly, Wasm applications are able to be distributed on any platform that supports a runtime. Every major browser vendor has implemented a WebAssembly runtime, but also many non-browser embedding environments exist. Throughout the development of this project, we relied on the Node.js WebAssembly runtime, which is just one example of a non-browser WebAssembly runtime. The expected runtime behavior is defined in the WebAssembly specification in order to preserve the security promises of WebAssembly[6]. A WebAssembly runtime is a secure sandbox, and any interaction with the host environment must be specif-

ically allowed and provided by the host. A WebAssembly module requests imports from the host environment, and does not run with any special privileges. This allows the developers and users to dictate how much of the system the module should have access to. Along with this, any out of bounds memory access results in a trap. Thus, the portability and security concerns of WebAssembly are addressed and fulfilled.

# Chapter 4

# RACKET

The Racket programming language is a modern Lisp language and a descendant of Scheme[27]. Racket is unique for focusing on language oriented programming, and has found its place commonly in classrooms and college curriculum[36][40].

## 4.1 Interesting Aspects of Racket

Racket aims to be a programming language for language oriented programming[28], which has influenced Racket's hygienic macro system and general extensibility. Some extensibility features of the language include context sensitive and module level control over syntax. This allows for even the most basic aspects of the language (function applications, variable reference, etc.) to be overridden and customized. This extensibility combined with the hygienic macro system has allowed for a plethora of sister languages to be developed. One of the greatest examples of the power of these systems is Typed Racket, which is a statically typed sister language of Racket built utilizing Racket's macro system.

## 4.2 Stages of Racket Compilation

At the highest level, a Racket program has syntax very close to that of a Lisp/Scheme program, with expressions nested in parenthesis to form s-expressions and the overall program structure reflecting that of a tree. However, a Racket program can and will then be transformed in many different formats when compiled. When compiling a

Racket module, the module gets translated into its fully expanded representation, then converted into linklets, before eventually getting fed into the Chez Scheme back-end pipeline. Figure 4.1 outlines this compilation process.

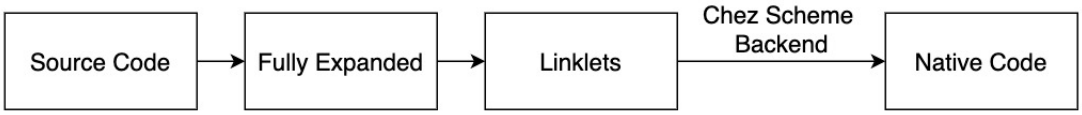



**Figure 4.1: Racket Compilation Pipeline**

For each phase of compilation we describe, we will show the transformation of the simple factorial function shown in figure 4.2.

```
1 (module factorial racket
2     (define (factorial n)
3       (let fac ([n n])
4         (if (equal? 0 n)
5             1
6             (* n (fac (- n 1)))))))
```

**Figure 4.2: Example Racket Factorial Module**

### 4.2.1 Fully Expanded Racket

A fully expanded Racket program is a parsed Racket program that only contains the top forms of the Racket programming language. A top form is the most basic operation/structure of the language. At this form all macros uses are expanded away and identifiers are paired with lexical information. This means that more complex forms such as `cond` are resolved into its most basic forms, as in the case of `cond`, a chain of `if` expressions. The most basic forms of the Racket programming language capture behavior that is unique to each form and is not reproducible by combinations

of the other forms. Rasm compiles from fully expanded Racket programs and we focus on building in support for each of Racket's top forms.

The top forms of Racket that we are most interested in are module definitions, provide statements and general expressions. There are some expressions Rasm currently does not support, such as continuation marks, but Rasm supports the primary expressions of the language such as; `lambda`, `let-values`, `if`, `begin`, etc. A full grammar of the supported forms is outlined in figure 7.1.

The fully expanded transformation of figure 4.2 is shown in figure 4.3. As described above, it is important to note what higher level structures have been transformed into simpler forms, such as the named let, line 3 of figure 4.2, being represented by a `letrec-values` defining the lambda assigned to the variable `fac`, on line 12 of figure4.3.

```
1  (module factorial racket
2    (#%module-begin
3     (module configure-runtime '#%kernel
4       (#%module-begin
5        (#%require
6         racket/runtime-config)
7        (#%app configure '#f)))
8     (define-values
9      (factorial)
10     (lambda (n)
11       (#%app
12        (letrec-values (((fac)
13                         (lambda (n)
14                           (if (#%app
15                                equal?
16                                '0
17                                n)
18                              '1
19                              (#%app
20                               *
21                               n
22                               (#%app
23                                fac
24                                (#%app
25                                 -
26                                 n
27                                 '1)))))))
28          fac)
29        n)))))
```

**Figure 4.3: Fully Expanded Racket Factorial Example**

### 4.2.2 Linklets

Linklets are the primary element of compilation and evaluation in Racket[14]. A module is usually represented by many linklets, and the combination of linklets and some metadata form a linklet bundle. A linklet is made up of variable definitions and expressions, as well as export and import variable names. The linklet layer is below the macro and syntax object layer and while the general structure of definitions in the linklet layer is similar to that of fully expanded programs, there are some minor

differences between the forms. The primary advantage of compiling from linklets is that it is a lower level than fully expanded Racket programs, there is no need for lexical information and some front end optimizations have taken place (e.g. constant folding).

Figure 4.4 shows the generated linklets from figure4.2. As you can see in figure4.4, identifiers are given unique names and, in this instance, two linklets are generated, which are then fed into the Chez Scheme Backend.

```
1  ;; Linklet 1
2  (linklet
3    ((.get-syntax-literal!) (.set-transformer!) (configure))
4    ()
5    (void) (configure #f) (void))
6
7  ;; Linklet 2
8  (linklet ((.get-syntax-literal!) (.set-transformer!))
9    (factorial)
10   (void)
11   (define-values (factorial)
12     (#%name
13       factorial
14       (lambda (n_1)
15         ((letrec-values
16            (((fac_2)
17              (#%name
18                fac
19                (lambda (n_3)
20                  (if (equal? 0 n_3)
21                      1
22                      (* n_3 (fac_2 (- n_3 1))))))))
23             fac_2)
24           n_1))))
25   (void))
```

**Figure 4.4: Linklets of Racket Factorial Example**

### 4.2.3 Chez Scheme Backend

After linklets, a Racket program gets fed to the Chez Scheme backend to be compiled to native machine code. In the next section, we will discuss the internals of this backend and our initial findings in attempting to add a WebAssembly backend to the Chez Scheme compiler.

# Chapter 5

# COMPILING RACKET

The standard Racket toolchain is now running on top of a Chez Scheme backend that generates native, non-portable machine code. Throughout the course of this project, we spent much of our time attempting to develop a WebAssembly backend for this existing Chez Scheme backend of Racket. We will now reflect on the initial motivation of adopting this backend, what adding a new backend to this compiler entails and where in the compilation process WebAssembly may be the best fit.

## 5.1 From C to Chez Scheme

Racket began adopting the use of a Chez Scheme backend in 2017[29]. Chez Scheme is a superset of the Scheme programming language[25] as described in R6RS[42]. The team sought to transition over to Chez Scheme as they considered maintenance of a Scheme codebase to be better than maintaining a very large C codebase of over 200K lines of code[29]. Due to the fact that Chez Scheme supports all of Racket in a simpler form, and that the foundation of building the compiler had already been laid, we sought to incorporate our own backend to compile Chez Scheme to WebAssembly. This process revealed to us some difficulties that lie in adapting existing compiler architectures and patterns to WebAssembly, and we have developed a few possible solutions to address each concern.

## 5.2 Porting Chez Scheme to a New Platform

When porting Chez Scheme to a new platform, the two primary needs are compiling the C runtime to the platform, and generating machine code for the platform from the compiler.

In the case of WebAssembly, tools like Emscripten[47] are well developed, widely used and reliable at compiling C. So the compilation of the runtime to WebAssembly would not be a major concern. The majority of the difficulty arises when attempting to generate WebAssembly binaries from the Chez Scheme compiler.

The code generation process relies on a machine type that represents the hardware capabilities of the machine the user is compiling to. This machine type represents basic information like name, thread capabilities, hardware platform (x86, x86_64, AArch32, etc) and the host operating system. Due to the nature of WebAssembly running in a virtual machine, not all of these aspects are initially of interest, but could serve as differentiators between the various WebAssembly runtimes.

The WebAssembly code generation itself would be handled by a custom backend, which is a single scheme file that is responsible for defining WebAssembly specific aspects of the compilation process. In this file we would need to create three modules that define the language's registers (which are none for WebAssembly), primitive operations and the assembler.

## 5.3 Nanopass Architecture

The Chez Scheme compiler is built upon a nanopass architecture[33]. The nanopass architecture is a compiler framework developed by Kent Dybvig, and utilizes many

small passes and intermediate languages to process and compile languages. A pass is supposed to be a simple and atomic-like action (generating unique variable names, converting closures, etc) and converts one intermediate language to another. At the end of the process, after all passes have been run, the result is a final language that is expected by the Chez Scheme compiler, which then gets translated nearly directly to machine code.

### 5.3.1 Final Language

The final language generated by the Chez Scheme compiler, before being compiled into the machine specific binary, is much more like a traditional assembly language than a higher level bytecode like WebAssembly. This language is referred to as `L16` in the backend and some of the forms of the language include `jump`, `reg` and `mref`. The structure of the code is in basic blocks with arbitrary jumps between the blocks for control flow and function calls. The combination of blocks and jumps proves to be a challenge to transition to WebAssembly, as WebAssembly has no way to arbitrarily jump between non-nested blocks. We will address these concerns in chapter 6, but with all of this considered, the final generated language may not be the best fit to translate to WebAssembly.

Figure 5.1 shows truncated segments of the final language generated by the Chez Scheme backend for the example shown in figure 4.2. This final language would then be translated into a machine specific, native, binary.

```
1  ...
2
3  fac:
4  dcl.15:
5  0:        mov             %r8, %ac0
6  3:        cmpi            (imm 0), %ac0
7  7:        bne             lf.19(11)
8
9  ...
10
11 134:      mov             %rcx, %ac0
12 137:      jmp             (disp 0 %sfp)
13 Lfail.14:
14 141:      mov             %ac0, %r8
15 144:      movi            (imm 4294967295), %ts
16 154:      jmp             %ts
17 156:      relocation:     (x86_64-jump 65
18                             (library-code #(libspec * 34842)))
19 Lfail.16:
20 156:      mov             %ac0, %r8
21 159:      movi            (imm 8), %rdi
22 166:      mov             %ac0, (disp 8 %sfp)
23 170:      addi            (imm 16), %sfp
24
25 ...
```

**Figure 5.1: Final Language of Racket Factorial Example**

### 5.3.2 Intermediate Language

The final language of the Chez Scheme nanopass architecture would not be the most efficient language to translate to WebAssembly. The differences between WebAssembly and more traditional assembly languages are too great for a direct trans-

lation, but in our research we found an intermediate language that would translate well to WebAssembly.

We hypothesized that the `L7` intermediate language would be a suitable language to compile to WebAssembly because at this stage initial optimizations have taken place, anonymous lambdas have been assigned names and closures have been converted. To add a new backend to the Chez Scheme compiler, we'd have to break out of the nanopass pipeline at this stage and feed this representation to our custom backend where we could then compile the `L7` language to WebAssembly. Branching out of the nanopass architecture is not the best case scenario, but it was a possibility we observed after concluding subsequent passes would create more distance between the generated language and WebAssembly.

Figure 5.2 shows the factorial example of figure 4.2 translated into the `L7` intermediate language of the Chez Scheme backend. We only represent a truncated sample of the generated language, specifically the definition of the `factorial` function itself. It is important to note that at this time, lambdas have been converted to top level case-lambdas, all identifiers have unique names, front end optimizations have taken place and the code snippet in figure 5.2 would be nested inside of a `(labels ...)` expression.

```
...
(#{factorial dfk1dfudxm0lm8mzhkb9sqbi6-0}
 (case-lambda
   [clause
    (#{n dfk1dfudxm0lm8mzhkb9sqbi6-1})
    #f
    1
    (call
      #{dcl dfk1dfudxm0lm8mzhkb9sqbi6-2}
      #f
      #{n dfk1dfudxm0lm8mzhkb9sqbi6-1})]))
(#{fac dfk1dfudxm0lm8mzhkb9sqbi6-3}
  (case-lambda
    [clause
     (#{n dfk1dfudxm0lm8mzhkb9sqbi6-4})
     #f
     1
     (if (call
           #f
           #[#{primref a0xltlrcpeygsahopkplcn-2}
             equal? 591610 (2)]
           '0
           #{n dfk1dfudxm0lm8mzhkb9sqbi6-4})
         '1
         ;; else case
         ...)]))
...
```

**Figure 5.2: L7 Intermediate Language of Racket Factorial Example**

We did not investigate this path further, but we still believe this intermediate language would serve well as a transition point to generate WebAssembly due to similar language constructs.

# Chapter 6

# CONCERNS WHEN COMPILING CHEZ SCHEME TO WEBASSEMBLY

In our research we identified three concerns that prohibited our efforts in adding a WebAssembly backend to the Chez Scheme compiler. These concerns include the existing expectations of the compiler, reliance on arbitrary jumps and unique calling conventions of the generated code.

## 6.1 Existing Expectations

When investigating the Chez Scheme compiler, many small expectations made it difficult to approach and implement WebAssembly.

One expectation of the compiler is targeting a register based machine. This does not translate well to WebAssembly as WebAssembly is a stack machine and has no registers. Mitigating this concern would either require adjusting this expectation or using global/local variables to simulate registers in WebAssembly. An additional concern was the structure in which code is generated in the final stages of the compiler. The compiler currently builds the binary from the bottom up, but in WebAssembly instructions can be bracketed in between guards, which would require some adjustment from the expected behavior.

As explained in the nanopass framework section, if we are to rely on the fully processed code, then we would need to determine the best way to handle some forms instilled by the process, such as register assignments. However, in the end there were other

concerns that restricted our efforts. These include basic block structure with arbitrary jumps between them and the unique calling conventions of the generated scheme code.

## 6.2 Arbitrary Jumps

Traditional assembly languages rely on being able to jump to locations and labels at their discretion. However, as we have discussed, this is not possible in WebAssembly. Therefore, if we are taking a collection of basic blocks and jumps as input we must determine a structure to replicate the expected behavior. One such structure that is common and needed is what we describe as the dispatch pattern. The dispatch pattern is nesting each basic block in one another and surrounding the entire structure in a loop. For each iteration of the loop we can use a flag to determine which basic block to branch to and then at the end of each block we branch back to the beginning of the loop to repeat the process until a final return.

### 6.2.1 The Relooper Algorithm

In our exploration of translating complex control flows to the structured control flow WebAssembly requires, it became clear to us that any efforts currently compiling to WebAssembly would be forced to take an approach similar to what we described above. Just as we suspected this had been the case. Specifically, Emscripten[47] had developed what they call the Relooper algorithm that seems to operate very similarly to our approach. Alon Zakai outlined this algorithm in his original Emscripten paper[47] as well as specifically in a subsequent blog post[48]. The original context of the algorithm was to compile C/C++ code into performant JavaScript code, so this required utilizing JavaScript's conditional and loop constructs to stay adequately

fast. The Relooper algorithm has since been adapted to target WebAssembly's control flow.

The Relooper algorithm[47][48][9] works by converting a control flow graph into a 'loop with a switch statement'. However the original implementation as described in the Emscripten paper[47] is much more elaborate. The algorithm is based on classifying the blocks of a given control flow graph as either a simple block, loop block or multiple block. A simple block is a block with an in edge and an out edge. A loop block represents a loop's control flow with two sub-blocks; an inner block and an out edge to another block. The inner block captures the looping behavior and when execution reaches the end of the block it will return to the beginning. Finally, a multiple block represents any control flow that branches off into multiple blocks before eventually rejoining. The Relooper algorithm then results in blocks arranged under JavaScript's expected high level control flow, rather than a collection of jumps/gotos.

This algorithm translates very well into the realm of WebAssembly, as we are interested in reducing a complex control flow graph into a simple, structured control flow. Every high level control flow operation that the algorithm initially targeted for JavaScript is either directly supported in WebAssembly or it can be supported with the native instructions.

### 6.2.2 Our Relooper Algorithm

We will now describe an algorithm we developed during our research and is similar to the Relooper algorithm, but we are not confident it accurately represents the Relooper algorithm's behavior. As to avoid confusion with the official Relooper algorithm, we will refer to our implementation as the dispatch algorithm.

In figure 6.1 and figure 6.2 we show how to translate an arbitrary control flow to the dispatch pattern. In the WebAssembly code of figure 6.2, the innermost block is where the branch to the next execution block will take place. A jump to a block will take execution to the end of the block, and thus the actual code to be executed for each block is found below the end of each nested block. We have thus commented each block number in the section of where this code would be placed and we also comment what possible values for the `$next_block` local variable could be.

We must rely on a pattern similar to the dispatch algorithm for arbitrary control flow, but for recognizable control flow patterns, WebAssembly offers native solutions. In figure 6.3 and figure 6.4 we show how both conditionals and loops would be supported in native WebAssembly instructions and with the dispatch pattern respectively.

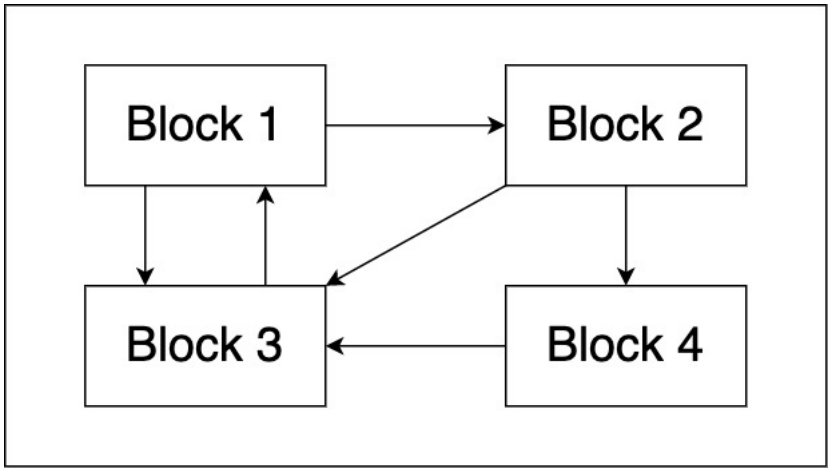
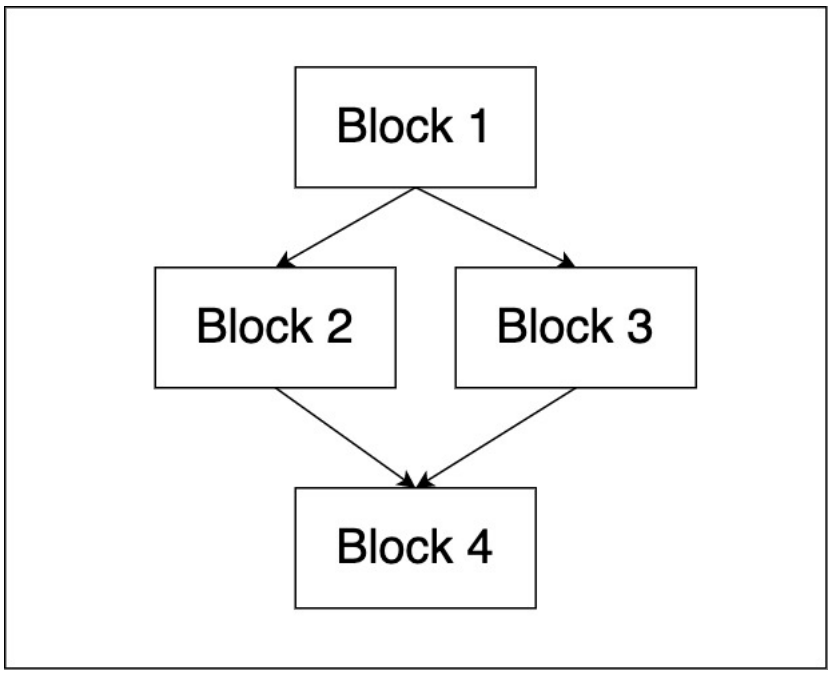



**Figure 6.1: Arbitrary Control Flow Example**

```
(loop $EXEC_LOOP
  (block
    (block
      (block
        (block
          (block
            ;; Dispatch to Next Block Here
            ;; With the br_table command we can
            ;; Jump to whichever block we need to next
            local.get $next_block
            br_table 0 1 2 3
          )
          ;; Block 1
          ;; Set next block on condition
          ;; Can be block 2 or 3
          local.set $next_block
          br $EXEC_LOOP
        )
        ;; Block 2
        ;; Set next block on condition (2 or 3)
        local.set $next_block
        br $EXEC_LOOP
      )
      ;; Block 3
      local.set $next_block ;; Has to be Block 1
      br $EXEC_LOOP
    )
    ;; Block 4
    local.set $next_block ;; Has to be Block 3
    br $EXEC_LOOP
  )
)
```

**Figure 6.2: WebAssembly Code for Arbitrary Control Flow**



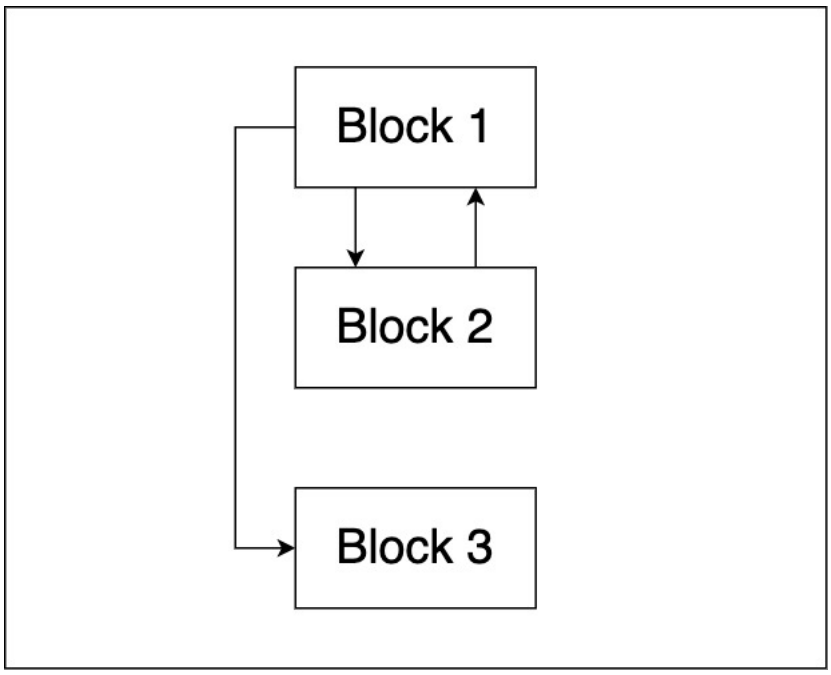

**Listing 6.1: Native WebAssembly Conditional**

```
(if
  ;; Block 1
  (then
    ;; Block 2
  )
  (else
    ;; Block 3
  )
)
;; Block 4
```

**Listing 6.2: Dispatch Pattern Conditional**

```
(loop $EXEC_LOOP
  (block
    (block
      (block
        (block
          br $Next_Block
        )
        ;; Block 1
        ;; Initial Branch
        br $EXEC_LOOP
      )
      ;; Block 2
      br $EXEC_LOOP
    )
    ;; Block 3
    br $EXEC_LOOP
  )
  ;; Block 4
)
```

**Figure 6.3: Translating Conditional Control Flow to WebAssembly**




**Listing 6.3: Native WebAssembly Loop**

```
(block
  (loop
    ;; Block 1
    ;; Break on condition
    br_if 1

    ;; Block 2
  )
)
;; Block 3
```

**Listing 6.4: Dispatch Pattern Loop**

```
(loop $EXEC_LOOP
  (block
    (block
      (block
        br $next_block
      )
      ;; Block 1
      ;; Initial Branch
      br 2
    )
    ;; Block 2
    br 1
  )
  ;; Block 3
)
```

**Figure 6.4: Translating Looping Control Flow to WebAssembly**

An initial look makes one think there may be a hit to performance in using this pattern, so we ran a few tests to determine if there was, and how much of a slowdown developers can expect. We will now outline a small example with some performance measurements measured with JavaScript's native timing commands.

We measured the performance of a simple loop program in WebAssembly. For each iteration of the loop we decrement a counter. If the counter is odd we subtract 3, and if the counter is even we only subtract one. We implemented the program with native constructs as well as with the dispatch pattern described above. We included some simple calls to JavaScript from within the WebAssembly code to both demonstrate importing host functionality, and to include some more complicated operations than WebAssembly arithmetic instructions. These imports get passed to the WebAssembly module as the second argument of the `WebAssembly.instantiate` command found on line 12 of listing 6.6.

We ran the programs with a lightweight node.js host file to measure the execution time of each WebAssembly program. While these measurements may not give us the most accurate execution times, we are interested in their relative performance to each other. For each input, we ran the exported WebAssembly function 10 times and averaged the execution times.

Figure 6.5 shows the general control flow graph of the program we are comparing the performance of.

Figure 6.6 is the host file we ran our measurements with, while figure 6.7 is the program using native WebAssembly control flow instructions and figure 6.8/6.9 is the same program utilizing our dispatch pattern.

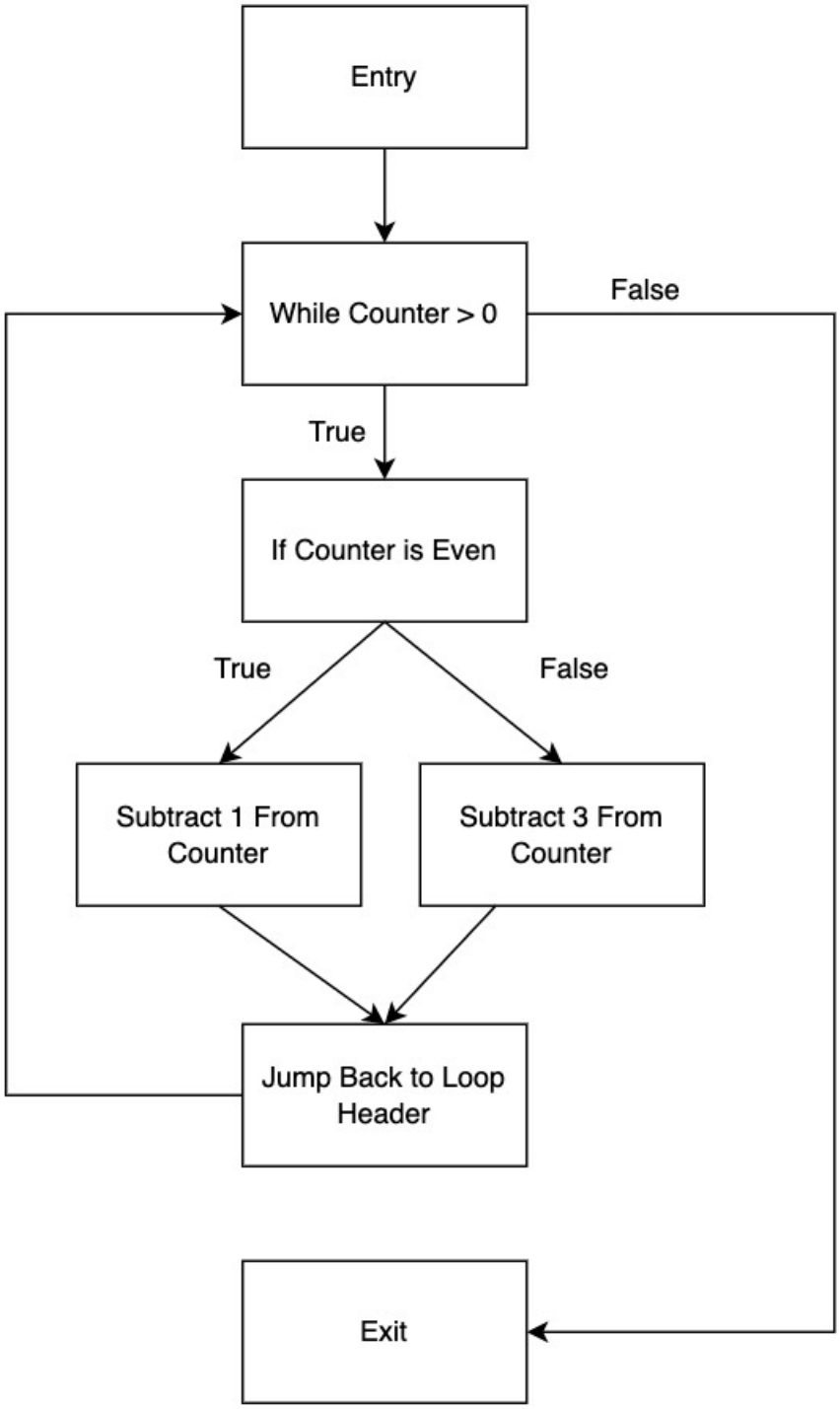



**Figure 6.5: Native vs. Dispatch Performance Test Control Flow Graph**

```
const fs = require("fs");
const filenames = ["native.wasm", "dispatch.wasm"];
const counter_values = [
  10_000_000, 20_000_000, 40_000_000, 60_000_000,
  80_000_000, 100_000_000, 200_000_000,
];
filenames.forEach((fn) => {
  const bytes = fs.readFileSync(fn);
  WebAssembly.instantiate(bytes, {
    env: {
      log: (i) => {},
      subOne: (i) => i - 1,
      subThree: (i) => i - 3,
    },
  }).then((obj) => {
    counter_values.forEach((i) => {
      let run_times = [];
      for (run = 0; run < 10; run++) {
        const start = Date.now();
        obj.instance.exports.test(i);
        const stop = Date.now();
        run_times.push(stop - start);
      }
      const avg_time = (
        run_times.reduce((a, b) => a + b, 0)
          / run_times.length
      ).toFixed(3);
      console.log(
        `${fn} (i = ${i}) avg time : ${avg_time} ms
          [${i}, ${avg_time}]`
      );
    });
  });
});
```

**Figure 6.6: Performance Testing Lightweight Host File**

```
(module
  (import "env" "log"
    (func $log (param i32)))
  (import "env" "subOne"
    (func $subOne (param i32) (result i32)))
  (import "env" "subThree"
    (func $subThree (param i32) (result i32)))

  (func $test (export "test") (param $i i32)
    (block $BREAK
      (loop $LOOP
        (call $log (local.get $i))
        (i32.eqz
          (i32.gt_s (local.get $i) (i32.const 0)))
        br_if $BREAK
        (if
          (i32.and (local.get $i) (i32.const 1))
          ;; odd subtract 3
          (then
            (local.set $i
              (call $subThree (local.get $i))))
          ;; even subtract 1
          (else
            (local.set $i
              (call $subOne (local.get $i)))))
        br $LOOP
      )
    )
  )
)
```

**Figure 6.7: Test A Using Native WebAssembly Control Flow**

```
1  (module
2    (import "env" "log"
3      (func $log (param i32)))
4    (import "env" "subOne"
5      (func $subOne (param i32) (result i32)))
6    (import "env" "subThree"
7      (func $subThree (param i32) (result i32)))
8
9    (func $test (export "test") (param $i i32)
10     (local $next_block i32)
11     (local.set $next_block (i32.const 0))
12
13     (block $BREAK
14       (loop $EXEC_LOOP
15         (block
16           (block
17             (block
18               (block
19                 (block
20                   (local.get $next_block)
21                   br_table 0 ;; 0 -> Loop Test
22                            1 ;; 1 -> Conditional Test
23                            2 ;; 2 -> True
24                            3 ;; 3 -> False
25                            6 ;; 4 -> Break
26                 )
27                 (; Loop Test Block ;)
28                 (call $log (local.get $i))
```

**Figure 6.8: Test B Using Our Dispatch Pattern: Part 1**

```
                (local.set $next_block
                  (if (result i32)
                    (i32.gt_s (local.get $i) (i32.const 0))
                    (then (i32.const 1))
                    (else (i32.const 4))))
                br $EXEC_LOOP
              )
              (; Conditional Test Block ;)
              (local.set $next_block
                (if (result i32)
                  (i32.and (local.get $i) (i32.const 1))
                  (then (i32.const 2))
                  (else (i32.const 3))))
              br $EXEC_LOOP
            )
            (; Odd Block - Subtract 3 ;)
            (local.set $i
              (call $subThree (local.get $i)))
            (local.set $next_block (i32.const 0))
            br $EXEC_LOOP
          )
          (; Even Block - Subtract 1 ;)
          (local.set $i
            (call $subOne (local.get $i)))
          (local.set $next_block (i32.const 0))
          br $EXEC_LOOP
        )
      )
    )
  )
)
```

**Figure 6.9: Test B Using Our Dispatch Pattern: Part 2**

Table 6.1 and figure 6.10 show the results of our performance testing, with the average time over ten runs for each input value for both the native instructions and the dispatch pattern. All execution time results were measured in milliseconds.

**Table 6.1: Native vs. Dispatch Performance Test Results Table**

| Input Size | 10M | 20M | 40M | 60M | 80M | 100M | 200M | 400M |
|---|---|---|---|---|---|---|---|---|
| Native (ms) | 105.9 | 212.3 | 419.2 | 627 | 834.9 | 1048.8 | 2135 | 4271.5 |
| Dispatch (ms) | 112 | 235.9 | 472.8 | 714.2 | 965.9 | 1213.5 | 2417.9 | 4750.1 |

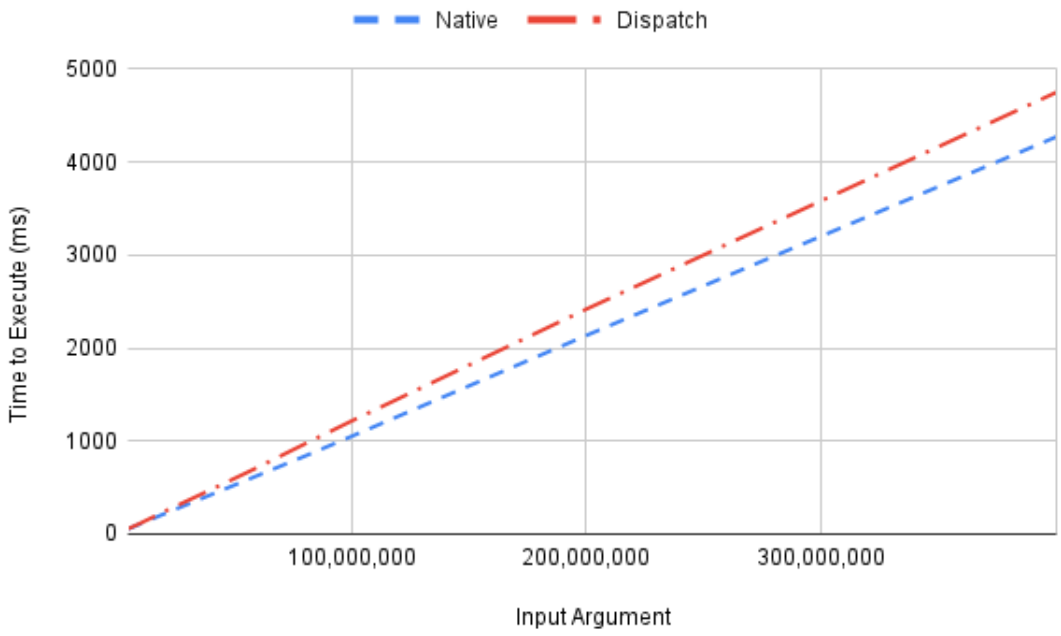



**Figure 6.10: Native vs. Dispatch Performance Test Results Chart**

Based on this simple test, there was a clear performance advantage to relying on the native instructions, rather than using the generic branching dispatch pattern. Our calculations presented in table 6.1 show, on average, the dispatch pattern program took 12.4% more time than the native instruction program to complete.

Another drawback of the dispatch pattern is hurting the readability of the textual WebAssembly, in which hand tuned WebAssembly has a clear advantage over automatically generated wasm.

An alternate approach to handling arbitrary control flow, would be to perform sufficient static analysis of the control flow, in order to pick up on some higher level control flow structures. We would be able to optimize for conditionals and simple loops, but some of the more undefined control flow behavior we would have to rely on the pattern described above. Luckily, Chez Scheme does not have any form of jumps/gotos, so if we are able to integrate at the level before the jumps get injected, we could utilize natively supported WebAssembly control flow structures such as conditionals, blocks and loops.

### 6.3 Calling Conventions

In order to support the continuation operations of Racket, the code generated by the Chez Scheme compiler does not use a traditional stack but rather a separate, heap-allocated, linked list of stack segments[10]. This means that the generated code has no form of call or return, but rather relies on jumps. Handling a function call includes jumping to the function body and then placing the return value in an expected place and jumping back to the caller's location. This pattern does not mesh well into the WebAssembly model, as the WebAssembly ISA has no arbitrary jump instruction, and there is no way to make a call without pushing something onto the stack.

One possible remedy is that if we have access to all the source code at compile time, we can treat all functions as a single large function containing all the basic blocks, and the jumps between caller/callee can be handled with the dispatch pattern described

above. We could simulate register behavior with WebAssembly local/global variable constructs and use variables as pseudo-registers.

Another approach could be to implement a trampolining system[17]. In a trampoline, functions return all necessary information to call subsequent functions, and the code gets executed in a continuous loop with each iteration applying the function information returned by the previous iteration. This can be implemented by relying on a memory based continuation or function pointers, and trampolining would allow us to call functions and recur without overloading the WebAssembly call stack. A proper trampolining implementation would also help us avoid having to rely on tail call optimizations[23], and could be implemented on top of our current system due to how we implement all closures as objects in memory already, with all necessary calling information.

### 6.4 Settling on Fully Expanded Programs

In the end, we did not have suitable time to add a WebAssembly backend to Chez Scheme, but we believe our experience and advice is greatly beneficial for any efforts moving forward, not only for Chez but for all languages seeking to compile to WebAssembly. We acknowledge that the current structure of the Chez Scheme compiler, reliance on arbitrary jumps and the generated calling conventions all prohibit WebAssembly from being efficiently ported into the compiler. With further development efforts and some of the patterns presented above, we believe WebAssembly could someday serve as a good compilation target for Chez Scheme, and thus Racket.

Due to time constraints and the complexity of compiling Racket to WebAssembly, we decided to write our own custom compiler to compile fully expanded racket programs to WebAssembly. We resolved to compile from this stage due to the Chez Scheme

concerns outlined above, lack of familiarity of Racket's linklet forms and general usability of fully expanded Racket.

# Chapter 7

# IMPLEMENTATION

We present our proof of concept Racket to WebAssembly compiler. We rely on fully expanded racket modules that we then compile to the WebAssembly textual representation. We also developed and provide a custom JavaScript host file that instantiates the generated WebAssembly module and wraps the exported functions for easy use in any context.

The full source code of our project can be found at our publicly hosted GitHub repository[37].

## 7.1 General Architecture

The general structure of our compiler is as follows. Our compiler takes a file path as a command line argument and loads in the file's contents. We then rely on Racket's built in expansion procedure to expand the Racket program to its topmost forms. The provided racket file either needs to have a top level module, a `#lang racket` declaration or both. We then handle parsing the other top level forms from the syntax object and generate an initial AST to feed into our compiler passes. We handle the processing of the AST in a few passes, before we transform the intermediate form into our final expected representation. The passes will be explained more in the following section, but some general passes are generating unique identifiers and discovering environmental bindings. After all the passes have resulted in a final form of the program, we proceed to generating the textual WebAssembly. We rely on the s-expression format of WebAssembly and map each of our expected structures to their

corresponding WebAssembly operations. We have also defined a very small standard library of some basic Racket operations, reimplemented in WebAssembly.

### 7.1.1 Command Line Interface

Our compiler is in the form of a command line application that takes as input the path of the Racket program to compile. We have provided a flag to enable `dev` mode, which will also save the intermediate representations of the program in files under a relative `dev` directory.

### 7.1.2 Expansion

We expand the provided file contents with the default Racket syntax reader. This results in a syntax object that represents the program expanded to all of the top-forms. The expanded program is a representation of the Racket program in the simplest structures of the language.

A fully expanded racket program allows us to only need to compile the most basic forms of Racket rather than having to parse every operation that is a combination of the most basic. Most of the processing in the expansion phase is generating our first intermediate form from the provided syntax object, as well as resolving any bound identifiers in this phase, if a binding is known. We also process some initial data types at this time. We resolve any constant data types by wrapping the values in an identifying structure, such as `Int` or `Float`. We rely on code from the RacketScript project[45] for parsing of the provide statements. Once we have fully converted the expanded racket program to our expected format we pass the AST onto our nanopass framework.

### 7.1.3 Supported Top Forms

At this time, we only support a subset of the top level forms. The most impactful forms we do not support are any submodule, or module expanding forms. We also do not support any syntax operations or continuation marks. Figure 7.1 defines the grammar of the subset of Racket Rasm supports.

The module form found nested in the `module-body-form` is for handling `#lang` declarations as modules defined under a `#lang` declaration are wrapped in another module when expanded, and we sought to support this behavior. Racket developers frequently rely upon the `#lang` declaration as well as simultaneously a `(module ...)` definition, so as to improve usability, if a language declaration and a module definition is used, we lift all of the submodule definitions into a top level module. This is the only form of submodule we support, and we only support this behavior in this single instance.

### 7.1.4 Compiler Passes

Our compiler relies on a series of passes, inspired by the nanopass framework. We only use one intermediate language, but a series of passes manipulate the language before the final representation is generated. We first perform some basic static analysis to verify that no top level definitions overshadow each other, as this is not allowed in WebAssembly. Our compiler then performs the following passes: generating unique identifiers, lift lambdas, generate initialization function, closure conversion and type discovery.

```
top-level-form ::= (module id module-path
                     (#%plain-module-begin
                       module-body-form ...))

module-body-form ::= module-top-level-form
                   | (#%provide raw-provide-spec ...)
                   | (module id module-path
                       (#%plain-module-begin
                         expr ...))

module-top-level-form ::= expr
                        | (define-values (id ...) expr)

expr ::= id
       | (#%plain-lambda formals expr ...+)
       | (case-lambda (formals expr ...+) ...)
       | (if expr expr expr)
       | (begin expr ...+)
       | (begin0 expr expr ...)
       | (let-values ([(id ...) expr] ...)
           expr ...+)
       | (letrec-values ([(id ...) expr] ...)
           expr ...+)
       | (set! id expr)
       | (quote datum)
       | (#%plain-app expr ...+)
       | (#%top . id)
       | (#%variable-reference id)
       | (#%variable-reference (#%top . id))

formals ::= (id ...)
          | (id ...+ . id)
          | id
```

**Figure 7.1: Rasm Supported Grammar**

#### 7.1.4.1 Generating unique identifiers

We rely on the first pass of the compiler to generate unique identifiers for each variable in the original Racket program. When we encounter a `let` form the bound variables overwrite any currently bound variable definitions. We capture this behavior by generating unique identifiers for newly bound variables and replacing any occurrence of the former identifier with the unique name. This allows us to not have to worry about scoping rules at later stages of the compilation and the eventual WebAssembly generation. WebAssembly has instructions for local variables so we can rely on this with each of our unique id's and the scoping will be handled automatically.

#### 7.1.4.2 Lift lambdas

In this pass we want to recognize any lambda forms and raise them to top level functions, while also collecting any environmental needs of the function. WebAssembly has no recognition of anonymous functions, so we rely on generating a unique name for the lambdas and replacing the lambda with the identifier. We then proceed to determine which identifiers to expect to be provided in the environment. Since every identifier is unique, we simply step through the module and capture every identifier we have inherited a declaration for thus far. Since when we lift lambdas to top level functions we recognize and attach a list of all defined identifiers in the environment, closure conversion is simply reorganizing the lambda to closer to the expected WebAssembly form.

#### 7.1.4.3 Generate Initialization Function

In WebAssembly any global variable must be declared with a constant value. Due to this, we need to define an initialization function that is called at the instantiation of the WebAssembly module. This function will then be responsible for initializing global variables with their correct value. In this pass, we gather every top level variable declaration and create the initialization function and combine each variable's initialization expression as the body.

WebAssembly requires that the start function to be called takes no parameters and returns no values. The provided function is then automatically invoked after the module's 'tables and memories have been initialized'[7], and thus before the values for the global variables are needed.

A simple example would be the declaration of `x` in the Racket code in figure 7.2.

```
1 (define x (+ 1 2))
```

**Figure 7.2: Defining Racket Variable as Result of Expression**

This is a contrived example and `x` could be set to `3` statically with constant folding, but if we were to rely on a direct compilation of the expression we would have to use the WebAssembly code shown in figure 7.3.

```
(module
    (global $x (mut i32) (i32.const 0))

    (func $init
        (global.set $x
            (i32.add (i32.const 1) (i32.const 2))))

    (start $init)
)
```

**Figure 7.3: Defining WebAssembly Global Variable as Result of Expression**

#### 7.1.4.4 Closure Conversion

The result of this pass is our final representation of the program and every function is represented as a closure with bound environment expectations/identifiers. We process each form of our current representation and for each application we determine if we are applying a function or the result of an expression. We call all functions indirectly, but we wrap applications of expressions with an internal application function that will retrieve the needed data from the applied closure. In this pass we also lift any locals of a function. This means that we identify any local variables a function has and declare them at the beginning of the function, as this is a requirement in WebAssembly. After this pass, every function is represented as a closure and has a name, list of parameters, list of expected environment parameters, list of local variables and a list of expressions as the body of the closure.

#### 7.1.4.5 Discover types

Our final pass is used to discover as many types for identifiers as possible. Most types are unknowable statically, but the primary type we care about is if an identifier

is initialized to a lambda we lifted. If this is the case we state the identifier's type as the lambda's name. We rely on this functionality because when we compile to WebAssembly, if we attempt to apply a lambda that we have lifted, we must retrieve the lambda's index in the function table to correctly call said lambda.

## 7.2 Code Generation

Once all the passes have been completed, the final representation of the program gets converted to WebAssembly. Very little computation takes place at this stage, and most of the work is in representing objects and applying closures. Outside of these two contexts, we simply register memory for our WebAssembly program, initialize the function table and translate each of the expressions to their WebAssembly counterpart. The function table is an indexable table of function pointers that is used to call closures and unknown functions. In this stage, we also append our standard library to the WebAssembly module.

### 7.2.1 Representing Values

Everything in the WebAssembly code we generate is a pointer, including integers and floats. We can represent pointers in WebAssembly as 32 bit integers, as the full memory address space is addressable in this size. When we encounter an integer or float, we allocate space for the value, store the value and return the pointer to the value. We do not support any version of garbage collection, so Rasm programs must be able to stay within the WebAssembly memory bounds. We allocate a maximum of 16MB of memory for a Rasm module, but in some runtimes WebAssembly can have access to up to 4GB of memory. We will discuss garbage collection in WebAssembly in more detail in the future work section, section 8.2.

#### 7.2.1.1 Type Tags

We rely on a type tag system in order to identify what types each object is at runtime. We use the first byte pointed to by a pointer to represent what type an allocated object is at runtime. We have tags for integers, floats, pairs and functions, which are the only data types we support. More complicated data types can be represented as a combination of these. For example, a list is simply a chain of pairs.

Table 7.1 describes each data type we generate and the corresponding runtime data stored for each type.

**Table 7.1: Type Tag Definitions**

| Data Type | Byte 1 | Bytes 2-5 | Bytes 6-9 |
|---|---|---|---|
| Integer | 0 | 64 Bit Integer Value | |
| Float | 1 | 64 Bit Float Value | |
| Function | 2 | Function Table Index | Pointer to Environment Array |
| Pair | 3 | Pointer to First Value | Pointer to Second Value |

For integers and floats, we know the first byte is the type id, and the next 8 bytes is the 64 bit value itself. Functions have the first byte the same as the others, and the next 4 bytes is the address of the function in the function table, and the next 4 bytes are a pointer to a flat array of pointers to the environment variables. We can represent the environment variables as a simple array and index into it, because we know the environmental needs of closures at compile time, and can index in to the array depending on the index of the parameter. For pairs, the first byte is the tag and the next 4 bytes is the pointer to the first element and the next 4 bytes is the pointer to the second element. We use a pointer to address 0 to represent null, and all arrays are zero terminated.

### 7.2.2 Applying Closures

As stated above, we represent closures as objects in memory, so the primary effort in applying closures is retrieving the function table index, determining the signature of the function and retrieving the closure's environment. To make it so we did not need to determine every function's signature at compile time, we have every function take two parameters. The first is a pointer to the flat array of parameters and the second to the captured environment. Since we know the parameter and environment needs at compile time, we can create the array of values at runtime and index into the statically known index to retrieve the runtime value. However, since we do rely on WebAssembly's local variables we need to initialize all the local variables with the parameter values before we execute the function. With all of this together, we can use WebAssembly's `call_indirect` instruction to call our closure. The arguments to this instruction are the type of the function we are applying, the function table index of the function and the parameters needed by the function.

### 7.2.3 Standard Library

We have reimplemented some basic functions from the Racket standard library. It is not a complete effort. However, reimplementing the standard library is not pivotal to prove the validity of our approach. Any IO operations will have to be provided by the host environment, and in WebAssembly we could either call them directly or the standard library could wrap them. Currently, we import IO from our JavaScript host environment for debug logging.

## 7.3 JavaScript Host

Along with our compiler, we provide a JavaScript host file that instantiates the WebAssembly module and wraps the module's exports. All exports, including constant values, are wrapped by JavaScript functions for retrieval and use. We wrap all WebAssembly exports, so the exports can be handled as the developer would expect in a JavaScript setting while also providing enough flexibility. In particular we offer four variations of every export. These variations are; `jsTojs`, `jsTowasm`, `wasmTojs` and `wasmTowasm`. Every variation represents the function's parameter types and return type. This means a `jsTojs` function will take JavaScript values as parameters and return a JavaScript value as a result. Following, a `wasmTojs` function will take WebAssembly values as parameters and return a JavaScript value as a result. Each export is accessible under the property name the user wants to use. For example, if there is an exported function `add` and the user wants to use the `jsTowasm` variant, they would access it with `wasm_module.rasm.jsTowasm.add(...)`.

A wrapper is necessary as we represent all WebAssembly values as pointers into memory, and we will discuss converting between WebAssembly pointer and JavaScript values in section 7.4.3. For any exported function that we can statically calculate its number of parameters, we initialize the function object's length property as the number of arguments to the function, as would be expected in a normal WebAssembly exported function.

## 7.4 Using Rasm

With the generated WebAssembly and the provided host file, a user can easily get started with WebAssembly. All that is necessary for a developer to get started with

Rasm is using `wat2wasm` to convert the generated WebAssembly text to a binary, passing the binary data into the provided `rasm.instantiate` function and then utilizing the WebAssembly module however they please, with helper functions provided to translate between WebAssembly and JavaScript values.

### 7.4.1 wat2wasm

Our compiler generates a textual form of WebAssembly. The translation of the textual form to binary is one to one as each instruction is represented by a byte, rather than a string of characters. To ease development, we rely on a third party tool called `wat2wasm` from the `wabt` toolkit. This is a well established community toolkit and very commonly used in the WebAssembly community. The `wat2wasm` tool is used to generate a WebAssembly binary from a WebAssembly text file. The generated binary can then be loaded into the JavaScript native `instantiate` function.

### 7.4.2 Instantiating a Module

In a JavaScript project using Rasm, all a user would need to do is require the Rasm module and then instantiate the module with the `rasm.instantiate` function. The instantiate method takes as argument the bytes of the wasm module to be instantiated. The bytes can be retrieved using `fs`'s read file function. The Rasm instantiate method returns a promise, due to the asynchronous nature of this operation, and in the callback parameter the user can use the WebAssembly object however they please.

An example use case below, figure 7.4, shows a user calling the exported function `add` using the Rasm module.

```
1 const rasm = require("./rasm");
2
3 rasm.instantiate(module_bytes)
4     .then((obj) => {
5         obj.rasm.jsTojs.add(1, 2);
6     });
```

**Figure 7.4: Example of Instantiating a Rasm Module**

### 7.4.3 WebAssembly Memory Pointers and JavaScript Values

When operating between WebAssembly and JavaScript, developers have to juggle between WebAssembly memory pointers and JavaScript values. This proves to be clunky and a hindrance, so as well as wrapping exported functions in a variety of configurations, we have defined a few helper functions that translate between WebAssembly pointers and JavaScript values.

We have provided interfaces to translate a WebAssembly pointer into its corresponding JavaScript counterpart. Supported return types at this point are BigInts, numbers and closures. A Closure that gets returned from WebAssembly is wrapped in a JavaScript function that calls the WebAssembly application function when evaluated, and thus the returned closure behaves just as a JavaScript function would, but interacts with WebAssembly as expected.

As well as provided functionality to translate WebAssembly pointers to JavaScript, we provide operations for developers to translate JavaScript values to WebAssembly. This is useful for calling WebAssembly functions from JavaScript when you need to provide specific JavaScript arguments as parameters. We currently support translating numbers, BigInts and booleans to WebAssembly from JavaScript. We don't currently support translating JavaScript functions to WebAssembly supported clo-

sures. We feel this is not necessary at the moment, as it is still possible to pass a WebAssembly closure pointer to a WebAssembly function, thus this captures the passing functions as parameters use case.

## 7.5 Validation

We will now discuss our approach to validating the correctness of our compiler and the behavior of the generated WebAssembly code.

Throughout the development of our compiler, we relied on test driven development. With this we developed a custom testing framework that would host our WebAssembly module and assert the behavior of specified test cases throughout a series of validation programs.

The structure of these tests were a series of Racket programs that we used to verify each form was being compiled appropriately and the exported values/functions from this program were correct.

### 7.5.1 Testing Framework

Due to the need to operate WebAssembly modules from within a host, we defined a custom testing host file that would instantiate a Rasm module and execute a test case based on JavaScript object specification.

WebAssembly functions can return results that are too complicated for a basic expected value comparison, so for each test case a user specifies a specific callback that is used to retrieve the value we are testing against. This is shown above in listing 7.5 under the `callback` property of the test case.

As well as a callback, a user specifies a basename, description and expected value for each test case. The basename represents the basename of the Racket validation program we are testing, the description represents a description of what we are testing and the expected field is the final expected JavaScript value produced by applying the callback.

Figure 7.5 shows an example of how a user would specify a test case.

```
{
    basename: 'return_func',
    description:
        'Test calling a closure that returns a closure',
    callback:
        (obj) => obj.rasm.toJS(obj.rasm.jsTojs.func()(3)),
    expected: 4,
}
```

**Figure 7.5: Rasm Test Case Example with Callback**

If a particular test case is not complex enough to need a callback, a user can rather specify an export name and an array of parameters to pass to this export. If no parameters are necessary, the property need not be defined. An example of a test case specification with no callback is found in figure 7.6.

```
{
    basename: 'arithmetic_funcs',
    description: 'Test Basic Arithmetic',
    export: 'add1',
    params: [45742],
    expected: 45743,
}
```

**Figure 7.6: Rasm Test Case Example**

The source code for our testing framework is found in appendix A.

### 7.5.2 Validation Programs

In total we rely on over 40 Racket programs to ensure the general validity of our compiler. Examples of these programs include basic nested arithmetic and closure conversion, and were used to verify correct behavior and expected values, rather than to verify performance of the generated code. A subset of these validation programs are available in appendix B for reference, and all such programs are available publicly on the GitHub repository[37].

# Chapter 8

# FUTURE WORK

There is still plenty of work remaining for Racket to fully compile to WebAssembly. Some of the work is simply a matter of time, such as the implementation of the standard library, but some of the work will require interesting solutions.

## 8.1 Lack of Control Over the Stack

One such area of concern in compiling to WebAssembly is having to manage the call stack. WebAssembly is a stack machine, and the developer has very limited control over the structure of the stack. This proves to be a hindrance when compiling functional languages to WebAssembly, like Racket and Scheme.

There are a few ways to mitigate this concern. One mitigation would be to recognize any control flows between functions where a function call is unnecessary and optimize the operation into a loop. This is called tail calling[23], and is very important to allow comparable performance to native Racket. There is currently a tail call proposal in the works that would allow developers to optimize away unnecessary information and only keep necessary call information. The only runtime that has support for tail calling is Chrome, but it is under an experimental flag. Sadly, most other engines do not see tail call support as a priority at this time. Eventually, with proper garbage collection, we could even replicate the heap allocated continuation that Racket is built with.

Another mitigation would be to implement a trampolining based system we described in section 6.3. In a trampolining system, the generated code can simulate the stack however needed, and the WebAssembly stack would never need to exceed a single frame.

## 8.2 Garbage Collection

Another concern in compilation is having to manage memory. Memory in WebAssembly is a linear array of bytes, with no garbage collection. This means any compiled code is responsible for making sure there is enough memory to execute the desired program and that allocated objects are managed well. A garbage collection proposal[8] is in the works and under active development, but until standardization, developers need to build in some runtime around their generated code. This is especially important to compiling Racket, as Racket has a runtime that it relies upon in normal execution. Potential solutions include either compiling the existing Racket runtime, or building a custom WebAssembly runtime for Racket when the needed proposals are at a stable state.

## 8.3 Debugging

There is currently very little support for debugging WebAssembly. Most developers' current approach is to debug the source code of their program and hope the generated WebAssembly is correct. This works well for simple projects, but as complexity grows, this approach does not scale. This effect is exacerbated as the WebAssembly module gets run in a host environment that adds another layer of abstraction and developers are thus responsible for determining where problems arise in their projects. With very little debugging support, developers are faced with a challenge of determining

the correctness of their source code, generated WebAssembly, WebAssembly host environment interface and the eventual execution of the code. Better debugging tools are in the works, but I believe the poor support around debugging WebAssembly is currently holding back adoption and utilization of WebAssembly.

### 8.4 Control Flow and Registers

I believe one of the biggest hindrances in porting an existing language to WebAssembly is how different the WebAssembly control flow instructions are than current architectures and patterns. The control flow of WebAssembly demands the use of our described dispatch/relooper patterns for any complicated control flows or arbitrary jumps. With these patterns performance is of concern, and time will tell if it can truly scale to the needs of the users.

Along with this, WebAssembly has no form of registers. Some projects use WebAssembly's local and global structures to simulate register use, and similar to the dispatch pattern, this works for now, but, both of these workarounds are clearly not taking advantage of the uniqueness of WebAssembly, and could ultimately hurt optimization of the generated code. I believe more work needs to be done in translating these expectations over to WebAssembly in a reliable and efficient way.

WebAssembly is a lofty project, with a lot of aspirations and a lot of work left. We believe our work shows some of the potential of WebAssembly, while still acknowledging that roadblocks exist for developers carrying over existing languages to the platform. Our experience can serve as an example and we hope our advice can be of value for future efforts.

# Chapter 9

# CONCLUSION

In this thesis we have presented our Racket to WebAssembly compiler. WebAssembly has a standardized implementation in all major browsers, with runtimes existing in many non-browser environments as well. We have done our part in bringing Racket to this ecosystem. There are many aspects of WebAssembly that differentiate it from a traditional assembly language, such as the host environment relationship, static typing and structured control flow. Throughout the development of this compiler we have discovered and addressed many concerns in compiling a modern functional language to WebAssembly. We have also presented our initial findings in adding a WebAssembly backend to the current Racket Chez Scheme backend. We acknowledge the difficulty developers may encounter when porting an existing architecture to WebAssembly, and present solutions that seek to bridge the gap between WebAssembly and other assembly language needs. We believe the work we have outlined here and the solutions presented will be of value to future teams porting any programming language to WebAssembly and hope to see full Racket compilation in the future.

# BIBLIOGRAPHY


[1] Cal Poly Github. http://www.github.com/CalPoly.

[2] Frequently asked questions (faq) about edge in the enterprise. https://docs.microsoft.com/en-us/deployedge/faqs-edge-in-the-enterprise. Accessed on 05.13.2022.

[3] Native client. https://developer.chrome.com/docs/native-client/. Accessed on 05.13.2022.

[4] Roadmap. https://webassembly.org/roadmap/. Accessed on 05.24.2022.

[5] Stack overflow developer survey 2021. https://insights.stackoverflow.com/survey/2021. Accessed on 05.01.2022.

[6] WebAssembly Core Specification. Accessed on 04.20.2022.

[7] WebAssembly Core Specification, Release 2.0. Accessed on 04.25.2022.

[8] WebAssembly Garbage Collection Specification. https://github.com/WebAssembly/gc/. Accessed on 05.07.2022.

[9] Webassembly troubles part 2: Why do we need the relooper algorithm, again? http://troubles.md/posts/why-do-we-need-the-relooper-algorithm-again/. Accessed on 05.20.2022.

[10] Racket. https://github.com/racket/racket, 1997.

[11] Binaryen. https://github.com/WebAssembly/binaryen, 2015.

[12] Assemblyscript. https://github.com/AssemblyScript/assemblyscript, 2017.

[13] World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation, 2019.

[14] Linklets and the core compiler. https://docs.racket-lang.org/reference/linklets.html, 2021.

[15] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: A step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, page 53–64, New York, NY, USA, 2007. Association for Computing Machinery.

[16] S. Atapattu. Bringing you up to speed on how compiling webassembly is faster. https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/wasm/. Accessed on 05.09.2022.

[17] H. G. Baker. Cons should not cons its arguments, part ii: Cheney on the m.t.a. *SIGPLAN Not.*, 30(9):17–20, sep 1995.

[18] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A tracing jit for a functional language. *SIGPLAN Not.*, 50(9):22–34, aug 2015.

[19] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A tracing jit for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 22–34, New York, NY, USA, 2015. Association for Computing Machinery.

[20] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, may 1966.

[21] K. Cheung. Autocad & webassembly: Moving a 30 year code base to the web. https://www.infoq.com/presentations/autocad-webassembly/, Sep 2018. Accessed on 05.15.2022.

[22] B. Couriol. Webassembly 1.0 becomes a w3c recommendation and the fourth language to run natively in browsers. https://www.infoq.com/news/2019/12/webassembly-w3c-recommendation/, Dec 2019. Accessed on 03.17.2022.

[23] S. K. Debray and T. A. Proebsting. Interprocedural control flow analysis of first-order programs with tail-call optimization. *ACM Trans. Program. Lang. Syst.*, 19(4):568–585, jul 1997.

[24] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, page 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[25] R. K. Dybvig. The development of chez scheme. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, page 1–12, New York, NY, USA, 2006. Association for Computing Machinery.

[26] R. K. Dybvig. The development of chez scheme. *SIGPLAN Not.*, 41(9):1–12, sep 2006.

[27] M. Felleisen, R. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. Mccarthy, and S. Tobin-Hochstadt. The racket manifesto. In T. Ball, R. Bodik, B. Lerner, G. Morrisett, and S. Krishnamurthi, editors, *1st Summit on*

*Advances in Programming Languages, SNAPL 2015*, Leibniz International Proceedings in Informatics, LIPIcs, pages 113–128. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, May 2015.

[28] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. A programmable programming language. *Commun. ACM*, 61(3):62–71, feb 2018.

[29] M. Flatt, C. Derici, R. K. Dybvig, A. W. Keep, G. E. Massaccesi, S. Spall, S. Tobin-Hochstadt, and J. Zeppieri. Rebuilding racket on chez scheme (experience report). *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[30] T. Fosmark and H. Arya. Update: Adobe flash player end of support on december 31, 2020 - microsoft lifecycle. https://docs.microsoft.com/en-us/lifecycle/announcements/update-adobe-flash-support#:~:text=Microsoftwillendsupportfor,andtheirotherindustrypartners., Sep 2020.

[31] jared83. Racket and webassembly #2015. https://github.com/racket/racket/issues/2015, 2018. Accessed on 05.27.2022.

[32] S. Jobs. Thoughts on flash. *Apple, Inc*, 2010.

[33] A. W. Keep and R. K. Dybvig. A nanopass framework for commercial compiler development. *SIGPLAN Not.*, 48(9):343–350, sep 2013.

[34] A. W. Keep and R. K. Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 343–350, New York, NY, USA, 2013. Association for Computing Machinery.

[35] s. klabnik. Is webassembly the return of java applets & flash? https://steveklabnik.com/writing/is-webassembly-the-return-of-java-applets-flash.

[36] D. Levy. Racket fun-ctional programming to elementary mathematic teachers. 05 2013.

[37] G. Matejka. Rasm. https://github.com/GrantMatejka/rasm, 2022.

[38] K. McElhearn. The history of adobe flash player: From multimedia to malware, Dec 2020.

[39] Microsoft. Typescript. https://github.com/microsoft/TypeScript, 2014.

[40] T. Partanen, L. Mannila, and T. Poranen. Learning programming online: A racket-course for elementary school teachers in finland. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, page 178–179, New York, NY, USA, 2016. Association for Computing Machinery.

[41] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2):175–188, mar 1999.

[42] M. Sperber, K. Dybvig, M. Flatt, A. V. Straaten, R. Findler, and J. Matthews. Revised6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

[43] S. Tobin-Hochstadt. Pycket. https://github.com/pycket/pycket, 2013.

[44] tsoding. wacket. https://github.com/tsoding/wacket, 2018.

[45] V. Yadav. Racketscript. https://github.com/racketscript/racketscript, 2016.

[46] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. pages 79–93, 2009.

[47] A. Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, page 301–312, New York, NY, USA, 2011. Association for Computing Machinery.

[48] A. Zakai. Reloop all the blocks. http://mozakai.blogspot.com/2012/05/reloop-all-blocks.html, May 2012. Accessed on 05.20.2022.

# APPENDICES

## Appendix A

## TESTING FRAMEWORK

```
const fs = require("fs");
const exec = require("child_process").exec;
const assert = require("assert");
const { test_cases } = require("./test_cases");
var wabt = require("wabt")();
const rasm = require("./rasm");

const my_exec = (cmd, callback) => {
  exec(cmd, (error, stdout, stderr) => {
    if (error) {
      console.error(`error: ${error.message}`);
      return;
    }
    if (stderr) {
      console.error(`stderr: ${stderr}`);
      return;
    }
    if (stdout.trim()) {
      console.log(`stdout: ${stdout}`);
    }
    callback();
  });
};

const my_assert = (want, got, msg) => {
  const epsilon = 0.005;

  assert.ok(want - epsilon < got &&
```

```
29              want + epsilon > got, msg);
30  };
31
32  const copy_rasm = `cp ../rasm.js .`;
33  const copy_index = `cp ../example_index.js .`;
34  const make_compiler = `raco make ../compiler.rkt`;
35
36  my_exec(`${copy_rasm} && ${copy_index} && ${make_compiler}`,
37    () =>
38    test_cases.forEach((test) => {
39      const basename = test.basename;
40      const rkt_path = `../examples/${basename}.rkt`;
41
42      const compile_file =
43        `racket ../compiler.rkt ${rkt_path}`;
44
45      my_exec(`${compile_file}`, () => {
46        console.log(`Test Case: ${basename}`);
47
48        const inputWat = `out/${basename}.wat`;
49        wabt.then((wabt) => {
50          var wasmModule = wabt.parseWat(
51            inputWat,
52            fs.readFileSync(inputWat, "utf8")
53          );
54
55          rasm.instantiate(wasmModule.toBinary({}).buffer)
56              .then((obj) => {
57                if (test.callback) {
58                  const val = test.callback(obj);
59                  my_assert(test.expected,
60                            val,
61                            test.description);
62                } else if (test.export) {
63                  const params =
64                    test.params ? test.params : [];
```

```
65                    const val =
66                      obj.rasm.jsTojs[test.export](...params);
67                    my_assert(
68                      test.expected,
69                      val,
70                      `${test.export}: ${test.description}`
71                    );
72                  }
73              });
74          });
75      });
76    })
77 );
```

**Listing A.1: Rasm Testing Framework**

Appendix B

VALIDATION PROGRAMS

```
1 (module arithmetic racket
2   (provide a1 a2 a3 a4 a5)
3
4   (define a1 (+ (/ 10 (* 1.5 3)) (* 23 10)))
5   (define (a2) (* (+ 12 (/ 12 4)) (- 25 10)))
6   (define (a3 x) (* (+ x (/ 12 4)) (- 25 10)))
7   (define (a4 x y) (* (+ 12 (/ y 4)) (- x 10)))
8   (define (a5 x y z) (* (+ z (/ x 4)) (- y 10))))
```

**Listing B.1: Test Program for Nested Arithmetic with Mixed Argument Types**

```
1 (module condtest racket
2   (provide mycond)
3
4   (define mycond (lambda (x)
5                    (cond
6                      [(< x 0) 0]
7                      [(< x 5) 1]
8                      [(< x 10) 2]
9                      [else 3]))))
```

**Listing B.2: Simple Conditional to Test cond Top Form**

```
1  #lang racket
2
3  (provide add-fact)
4
5  (define (add-fact x y)
6    (letrec ([fact (lambda (n)
7                     (if (equal? 0 n) 1 (* n (fact (- n 1)))))])
8      (+ (fact x) (fact y))))
```

**Listing B.3: Verifying Recursion with letrec Top Form**

```
1  #lang racket
2
3  (provide fact fact2)
4
5  (define Y (lambda (b)
6              ((lambda (f) (b (lambda (x) ((f f) x))))
7               (lambda (f) (b (lambda (x) ((f f) x)))))))
8
9  (define (fact x)
10   ((Y (lambda (fact)
11         (lambda (n)
12           (if (equal? 0 n) 1 (* n (fact (- n 1))))))) x))
13
14
15 (define fact2
16   (Y (lambda (fact)
17        (lambda (n)
18          (if (equal? 0 n) 1 (* n (fact (- n 1))))))))
```

**Listing B.4: Simple Y-Combinator to Verify Closure Handling**

```
1  #lang racket
2
3  (provide ycomb)
4
5  (define (ycomb dc)
6    ((lambda {empty}
7       ((lambda {cons}
8          ((lambda {empty?}
9             ((lambda {first}
10               ((lambda {rest}
11                  ((lambda {Y}
12                     ((lambda {length}
13                        ((lambda {addup}
14                           (addup (cons 3 (cons 17 empty))))
15                         (Y
16                          (lambda
17                              {addup}
18                            (lambda {l}
19                              (if (empty? l)
20                                  0
21                                  (+ (first l)
22                                     (addup (rest l))))))))))
23                      (Y
24                       (lambda
25                           {length}
26                         (lambda {l}
27                             (if (empty? l)
28                                 0
29                                 (+ 1 (length (rest l)))))))
30                      ))
31                   ((lambda {x}
32                        (lambda {y}
33                            (y (lambda {z} (((x x) y) z)))))
34                    (lambda {x}
35                        (lambda {y}
```

```
36                                  (y (lambda {z} (((x x) y) z))))))))
37                          (lambda {l} (l false))))
38                      (lambda {l} (l true))))
39                  (lambda {l} (equal? l empty))))
40              (lambda {a b} (lambda {select} (if select a b)))))
41     13)
42   )
```

**Listing B.5: Complex Lambda Calculus**