WASM-PBCHUNK: INCREMENTALLY DEVELOPING A RACKET-TO-WASM COMPILER USING PARTIAL BYTECODE COMPILATION

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

Adam Perlin

June 2023

COMMITTEE MEMBERSHIP

TITLE:	Wasm-PBChunk: Incrementally Developing a Racket-to-Wasm Compiler Using Partial Bytecode Compilation
AUTHOR:	Adam Perlin
DATE SUBMITTED:	June 2023
COMMITTEE CHAIR:	John Clements, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Stephen Beard, Ph.D. Professor of Computer Science

Abstract

Wasm-PBChunk: Incrementally Developing a Racket-to-Wasm Compiler Using Partial Bytecode Compilation

Adam Perlin

Racket is a modern, general-purpose programming language with a language-oriented focus. To date, Racket has found notable uses in research and education, among other applications. To expand the reach of the language, there has been a desire to develop an efficient platform for running Racket in a web-based environment. WebAssembly (Wasm) is a binary executable format for a stack-based virtual machine designed to provide a fast, efficient, and secure execution environment for code on the web. Wasm is primarily intended to be a compiler target for higher-level languages. Providing Wasm support for the Racket project may be a promising way to bring Racket to the browser.

To this end, we present an incremental approach to the development of a Racket-to-Wasm compiler. We make use of an existing backend for Racket that targets a portable bytecode known as PB, along with the associated PB interpreter. We perform an ahead-of-time static translation of sections of PB into native Wasm, linking the chunks back to the interpreter before execution. By replacing portions of PB with native Wasm, we can eliminate some portion of interpretation overhead and move closer to native Wasm support for Chez Scheme (Racket's Backend). Due to the use of an existing backend and interpreter, our approach already supports nearly all features of the Racket language – including delimited continuations, tail-calling behavior, and garbage collection – and excluding threading and FFI support for the time being.

We perform benchmarks against a baseline to validate our approach, to promising results.

ACKNOWLEDGMENTS

Thanks to:

- My family, for their near unending support and encouragement.
- My friends, for helping me stay balanced, and reminding me to go out and have fun occasionally.
- John Clements, for his constant enthusiasm, deep knowledge, and helpful guidance.
- Matthew Flatt, for creating PB and PBChunk and suggesting the research direction

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Since the first browsers were developed, the problem of how to run interactive content served over the web has been ever-present. Fundamentally, web pages are interactive programs served over the internet, and browsers have continually grown in complexity in response to increasing performance demands and security concerns posed by interactive web applications.

Interactive web applications include client side code of some form, and traditionally, client side code on the web has been written in JavaScript. Over the years, numerous other alternative platforms for client side code on the web have been developed, including Java Applets [36], Adobe Flash [2], and Native Client (NaCL) [53]. These client-side platforms all suffered from numerous flaws stemming from security vulnerabilities, lack of portability, efficiency concerns, and corporate priorities hindering standardization.

Fundamentally, WebAssembly (Wasm) [48] has succeeded where previous client-side web approaches have failed [29]. Wasm is the binary executable format for a stack-based virtual machine that can be implemented in a browser environment. It is fully formally specified [23], and has been explicitly designed with speed, security, and ease of verification in mind [23]. It is not meant to replace JavaScript; rather, it is meant to serve as a compiler target for other higher level languages to run on the web in conjunction with JavaScript. It is supported in all major browsers, including Chrome, Firefox, Safari, and Microsoft Edge [48]. Given the breadth of Wasm support, its standardization, and the growing momentum behind it, numerous high-level language implementations are moving to target Wasm. For example, Emscripten [54] is a compiler for LLVM-based languages that targets WebAssembly, allowing languages such as C, C++, and Rust to run on the web. Targeting Wasm does not come without its challenges however, as the development of Emscripten has indicated [54].

1.1 Motivation

Functional languages bring their own challenges to compilation, and these challenges are still relevant when targeting WebAssembly. Thus, research focused on compiling functional languages into Wasm is ongoing. Racket is one such functional language. Racket is based on Scheme, and has found uses in research [42], [19] and education [15]. Racket uses Chez Scheme [12], an existing native Scheme compiler, as its backend. There are numerous features of Scheme-based languages that complicate compilation – in particular, guaranteed tail-call optimization (known in Scheme as "proper tail calls"), first-class continuation operations, and dynamic typing, as well as the presence of necessary runtime components such as a garbage collector.

Existing efforts to compile Racket and Scheme to WebAssembly include Rasm [32], a proof of concept Wasm compiler for expanded Racket, and Hoot [26], an experimental Wasm compiler for Scheme. Rasm unfortunately lacks runtime support, and Hoot has taken a highly involved approach that has led to the development of a complex separate compilation pipeline. As of today, there is no Wasm implementation of Racket that is compatible with the existing code base and which provides sufficient feature breadth to run most programs.

1.2 Key Contributions

We describe an approach we have implemented for running Racket on Wasm that allows for near complete feature support – including delimited continuations, guaranteed tail call optimization, and garbage collection – and uses a much less involved compilation strategy than existing efforts. Our key contribution is to work at the level of a bytecode called PB that already supports all features of Scheme, and by extension, Racket. We then utilize Emscripten to compile the interpreter for PB (which is written in C), along with the Chez Scheme runtime (also written in C) to Wasm. This yields a version of Racket that can run in a Wasm virtual machine. Performance is a concern here, however. To address this, we develop a partial compilation pass which selects "chunks" of PB to translate into Wasm, yielding a speedup when compared with simply interpreting the chunks as bytecode.

1.3 Content Overview

We first describe the distinguishing features of Racket and WebAssembly to provide necessary background. We then describe particular features of Scheme that make compilation difficult; in particular, dynamically typed objects [13], and first-class continuations [25]. We briefly address how Chez Scheme, a mature Scheme compiler that forms the backend for the current Racket implementation [20], solves these problems. This sets the stage for understanding some of the unusual features in Chez Scheme's compilation model, allowing us to explain **why** Chez Scheme's existing code generation interface is particularly difficult to adapt for use with WebAssembly.

We then introduce the PB bytecode, an existing virtual instruction set for which a Chez Scheme backend already exists [33]. PB was primarily developed for bootstrapping builds of Racket on new platforms, but we re-purposed it for use with Wasm. As part of this effort, we wrote a disassembly tool for PB that we use to give examples of what the bytecode looks like and to demonstrate how certain high-level Scheme features are compiled.

We then introduce PBChunk, a pre-existing partial compilation approach developed by Matthew Flatt of the Racket project that allows portions of PB to be compiled into equivalent C code. PBChunk provides the inspiration for our own approach.

Finally, we introduce our own partial-compilation system, Wasm-PBChunk. We describe the system at a high level, including some of the relevant algorithms for selecting chunks of PB to partially compile. We provide explanatory samples of the system's output, so the reader can understand what PB translated to Wasm looks like. We benchmark our system using a subset of the Larceny [30] benchmarking suite, which demonstrates the breadth of programs our system is capable of running and allows us to analyze the system's strong and weak points.

We hope that our work will be informative for others who wish to port languages to Wasm – especially other functional languages and languages with a non-traditional runtime model.

Chapter 2

RELATED WORK

2.1 Racket-to-wasm compilation

The most notable previous work on Racket-to-Wasm compilation was a Master's Thesis completed by Matejka [32], in which a standalone compiler for expanded Racket programs was created.

Matejka initially attempted to solve the Racket-to-Wasm compilation problem by creating a backend for Chez Scheme. The work here did not end up proving to be fruitful, for the simple reason that the semantics of Wasm are not easily adapted to the fundamental *execution model* that is used in Chez Scheme generated code. The difficulties in bridging the execution semantics will be described in more detail later.

Matejka's compiler, `rasm`, operates on expanded Racket, re-using the existing Racket expander. The compiler supports a subset of top-level forms and most expression forms – the notable exception being continuation and syntax related primitives. Some core functions, such as `car` and `cdr`, are re-implemented, but `rasm` does not support the core Racket runtime and so notably lacks support for garbage collection, threading, and numerous other features.

2.2 Scheme-to-wasm compilation

2.2.1 Guile to Wasm (Hoot)

The Hoot project [26] is one of the most complete and well documented current efforts towards an Ahead-of-Time Wasm compiler for a Scheme-based language. Hoot is an experimental compiler for GNU Guile [22], an implementation of R6RS Scheme [38]. Hoot is especially unique in that it aims to be as complete as possible, meaning that it features support for a value representation for Scheme objects, variable arguments, tail calls, delimited continuations, and the numeric tower.

Of particular interest is the delimited continuation implementation; to implement this feature, a virtual stack was needed. In order to use a virtual stack instead of the Wasm stack, every call needs to be a tail call. A `tailify` transformation was added to Guile's CPS intermediate language that converts every call into a tail call [51]. Non tail calls are dealt with by splitting the continuation of a non-tail call into a separate continuation that is pushed onto a virtual stack, before performing a tail call. Returns are then executed by popping the most recent continuation from the runtime stack and again performing a tail-call. Continuations themselves are implemented by "slicing" and restoring this virtual stack. Continuations will be described in more detail later.

Note that tail call support relies on the upcoming tail call proposal [50] for Wasm that adds the `return_call` and `return_call_indirect` instructions. Additionally, Hoot's value representation [51] is entirely dependent on Wasm's upcoming GC proposal [49] since it relies on new support for GC-allocated Wasm structs.

2.3 Racket Transpilation

There are a number of projects which do not target Racket specifically, but are aimed at providing alternative implementations of Racket – either through novel interpreter techniques, or transpiling Racket to another higher-level language.

2.3.1 Pycket

Pycket is what is known as a “Tracing JIT Compiler”; in effect, the project made use of the RPython framework in order to produce a Just-in-Time Compiler based on a preexisting interpreter. The key idea behind Pycket was to implement an AST-based Racket interpreter in RPython, and then use the RPython tool-chain to generate a JIT compiler from that interpreter. The project had promising performance results, exceeding the performance of the standard Racket implementation in some cases [7]. Unforunately, RPython does not support the generation of JIT compilers which target Wasm – otherwise, the Pycket approach would be a promising option.

2.3.2 RacketScript

RacketScript is a compiler from Racket to JavaScript. Like many other alternative Racket compilers, RacketScript uses fully-expanded Racket programs as input. It translates expanded Racket programs to ECMAScript 6. The RacketScript project is fairly experimental, and notably does not support tail call optimization, continuations, or the full suite of Racket numeric operations [52]. As Matejka notes in his thesis, the problem of translating Racket to JavaScript is quite different than the Wasm problem, primarily because many high-level features of Racket – such as lists and higher-order functions – have JavaScript analogues, which greatly simplifies the translation process [32].

Chapter 3

INTRODUCTION TO RACKET

This chapter gives a brief overview of Racket, a research language with some unique features that are worthwhile to discuss. Racket provides a very interesting and robust programming environment, and expanding the reach of this programming environment is one of the core motivations behind this thesis.

3.1 Functional Core

The core of Racket is a small, functional language based on Scheme. Scheme itself was originally developed at MIT by Guy Steele and Gerald Sussman as a more functional dialect of Lisp [40], and as a way to explore Carl Hewitt's "Actor Model" of computation [24]. Scheme is heavily based on the Lambda Calculus, as can be seen in its simplistic syntax and its relatively simple evaluation model.

Part of the basic philosophy of Scheme is to provide a minimal set of primitives on which much more complex constructs can be built. Complexity can be built up incrementally through the use of syntax transformers that rewrite higher-level syntax into just the core forms.

Racket also contains a very small set of core forms that implement the core behavior of the language, and are shown in Figure 3.1. Notice how compact the set of core forms actually is; it is quite minimal, given that Racket is a fully-featured modern language. In Racket, much of the complexity that would otherwise be embedded in the core language is defined on top of the core language using syntax transformations.

```
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)
|    (quote-syntax datum)
|    (quote-syntax datum #:local)
|    (with-continuation-mark expr expr expr)
|    (#%plain-app expr ...+)
|    (#%top . id)
|    (#%variable-reference id)
|    (#%variable-reference (#%top . id))
|    (#%variable-reference)
```

Figure 3.1: Racket top-level expression forms

3.2 Language Oriented Programming

Racket is described in the Racket Manifesto [17] as a language for creating other programming languages. What this means, fundamentally, is that Racket is about allowing a user to write code to solve a particular problem in the language that is most appropriate for the task.

Racket features an advanced, hygienic macro system on which numerous other languages can be built.

Racket's macro expansion system is more advanced than Scheme's, and allows for creating new languages that may differ in syntax but are still parsed and expanded into core Racket. Through this facility, alternate languages such as Typed Racket, a gradually typed variant of Racket, and Scribble, a language for writing documentation, have been developed.

```
(module example racket
  (define (map f lst)
    (cond
        [(null? lst) '()]
        [(pair? lst) (cons (f (car lst))
                           (map f (cdr lst)))])))
```

Figure 3.2: Module implementing a simple "map" function

```
(linklet ((.get-syntax-literal!) (.set-transformer!))
((1/map map))
  (void)
  (define-values (1/map)
    (#%name
     map
     (lambda (f_1 lst_2)
       (if (null? lst_2)
           (let-values () '())
           (if (pair? lst_2)
               (let-values ()
                 (cons (f_1 (car lst_2))
                   (1/map f_1 (cdr lst_2))))
               (void)))))))
```

Figure 3.3: Expanded linklet for "map" module

Within Racket itself, there are numerous sub-languages, such as `match` for pattern matching, and the Racket object system, that are defined as syntax transformations on top of the base language of Racket.

3.3 Racket BC vs. Racket CS

Since Racket is an evolution of Scheme, the Racket project originally evolved from an implementation of a Scheme interpreter along with a set of graphical libraries [20]. The original Scheme interpreter was written in C, so as the project evolved a significant amount of C code was added to improve the interpreter by building a JIT compiler and new runtime components. This left a large portion of the Racket compiler and runtime as a C-implemented system, when experience showed that the Racket-implemented portions of the project were much easier to maintain [20]. Thus, an effort was started to re-implement the Racket compiler itself in a higher-level language. This caused a bifurcation in implementations; the legacy implementation is now known as the "BC" implementation, and the new implementation is known as Racket CS, for Racket on Chez Scheme.

3.4 Compilation Pipeline

The compilation process for Racket CS is comprised of the high-level steps outlined in Figure 3.4. A Racket module is first parsed into S-expression form using the reader. Racket's macro-expander both parses S-expressions into valid Racket (including lexical information), and expands any macros that are present. The expansion process is recursive, since macros may expand to other macros. The expanded source is further split into linklets, where each linklet features only the core forms. Figure 3.2 shows an example Racket module implementing the higher-order function primitive `map`. One of the linklets from the fully expanded form of the module – in particular, the linklet containing the expanded version of `map` – is shown in Figure 3.3. Notice that the `cond` has been expanded into nested `if` forms, the `define` has been expanded into `define-values`, and intermediate `let-values` forms have been introduced.

Since Chez Scheme is a compiler for R6RS [38] Scheme specifically, the expanded Racket linklets must first be translated into Scheme using a "schemify" pass. Though highly similar, expanded Racket is not necessarily valid Scheme and thus some transformations have to take place [20]. Finally, the new Scheme linklets can be fed through

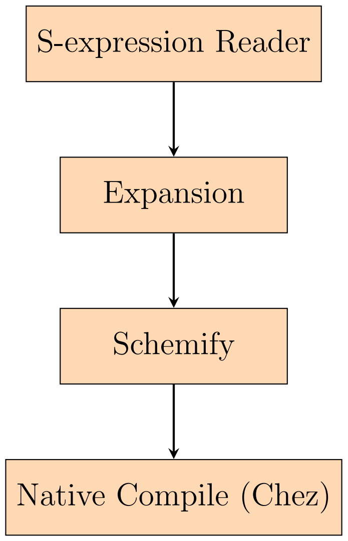


Figure 3.4: Racket CS high-level compilation steps

Chez Scheme to be compiled to native code. The native-compiled Scheme is dynamically linked with a runtime system including the garbage collector and a core runtime that provides IO, threading, and the like.

3.5 Uses (Research, Education)

One of the primary motivations for creating Racket was as an educational tool [17]. Racket's creators wanted a better way to teach Scheme, a language that is featured in the famous textbook *Structure and Interpretation of Computer Programs* [1]. This led to the development of an interactive IDE, DrRacket (originally, DrScheme) [18] and a suite of interactive libraries. Further, a textbook for teaching principles of Computer Science in Racket, *How to Design Programs* (HtDP), was created [15].

Racket is also fundamentally a language for Programming Languages research. In particular, Racket's macro expander served as a test bed for a new approach to hygienic macro expansion using "sets of scopes" [19], and Typed Racket allowed for the invention of "occurrence typing" [42].

Chapter 4

COMPILING RACKET AND SCHEME

This chapter gives a brief overview of some challenges present in compiling a Scheme-based language such as Racket. We give a brief overview of how Chez Scheme, Racket's backend, deals with these challenges. This background is particularly necessary because the approach we will introduce deals directly with code generated by Chez Scheme, which makes heavy use of the concepts that will be discussed in this chapter.

4.1 Chez Scheme and Racket CS

Chez Scheme is the work of Kent Dybvig and others. It is a compiler for an augmented version of R6RS [38] Scheme that has been continually developed since its inception nearly 30 years ago [12].

Chez Scheme was originally proprietary software licensed by Kent Dybvig. The code was acquired by Cisco when Dybvig began working at the company, and in 2017, Cisco decided to open source the project. This allowed the project to be forked and built upon, allowing for experiments with hosting the Racket compiler on Chez Scheme [20].

Chez Scheme is an ideal compiler and runtime environment for Racket to target for a variety of reasons. Since Chez Scheme implements a version of Scheme, the Chez Scheme system already features support for tail calls and efficient first-class continuations [20]. Continuation support in particular is very rare for a programming environment to support out of the box, but it has been a core feature of Chez Scheme's design since the beginning. Chez Scheme also supports a numeric tower similar to Racket's, which includes arbitrary precision arithmetic. Thus, core numeric operations in Racket were more straightforward to transfer over when porting Racket's backend to Chez Scheme [20].

As a final reason, Chez Scheme is a mature project with a fully-featured runtime that includes a garbage collector and threading support. The compiler can output native code for a variety of machine architectures, and thus gives Racket access to native-level performance on multiple platforms.

4.2 Challenges in Scheme Compilation

Two features that complicate Chez Scheme's compilation model are described below. Scheme's support for first-class continuations and its dynamically-typed object model are challenging to implement efficiently and thus it is worth briefly introducing Chez Scheme's approach to implementing these features.

4.2.1 Compiling Continuations

First-class continuations are an unusual feature present in Scheme and require a carefully designed runtime model.

In Scheme, a continuation represents a captured execution context. The execution context can be conceptualized as the "remainder" of a computation. For example, in the expression `(* 10 (+ 1 2))`, when the sub-expression `(+ 1 2)` is being evaluated, the (most) current continuation would be `(* 10 [])`, where `[]` is a *hole* to be filled in with the result of a sub-expression.

A continuation acts like a procedure object, but when invoked it restores the execution context that was captured. The arguments to the continuation procedure *fill* the holes in the execution context. Thus, if `(* 10 [])` was captured as some continuation `k`, then `(k 3)` would restore the execution context as `(* 10 3)`, *in addition* to any outer levels of context.

At the runtime level, at a particular point during execution where a continuation capture occurs, the continuation can be represented by the current "chain" of stack frames. This is because the remainder of program execution is determined by these frames [25]. Continuation capture, then, must mark or store these frames for future use. When a continuation is *reinstated*, the current sequence of stack frames must be overwritten with the previously stored sequence.

One method of implementation for a continuation model is to use a heap-allocated linked-list for all stack frames, allocating a new frame on the heap every time a procedure is called. This is inefficient, though, because it adds overhead for *every* procedure call, and a garbage collector must do more work to collect old stack frames [25].

Chez Scheme uses a hybrid stack-heap approach. A *control stack* is used, which is a heap-allocated linked list of *stack segments*, where a stack segment is comprised of one or more individual frames [25].

Continuation Capture is implemented by splitting the current stack segment in two; the upper portion becomes a new stack frame, and stack frames in the preceding portion are sealed off as part of the continuation. Splitting the segment is a constant time operation up until the current stack segment is exhausted, at which point a new stack segment must be allocated and copying must occur. Thus, continuation capture is inexpensive on average [25].

Continuation Reinstatement requires copying an old portion of a stack segment over top of the current one. Because a captured portion of a stack segment could be quite large, there is an upper bound placed on the size of a stack segment that will be copied [25]. If the stack segment that makes up the continuation is too large, then the stack segment is split in two, and only the top portion (the "closest" frames) are copied over initially. The stack segment that comprises the continuation must be walked over to find a splitting point that occurs at a frame boundary within the segment [25].

4.2.2 Object Representation

Scheme is dynamically typed and handles memory allocation for the user. There are some native sized types – notably the `fixnum` and `flonum` types – that can fit in registers. Most other types – including arbitrary precision numbers, strings, and pairs – are handled as pointers at runtime. Scheme programs must be able to inspect types at runtime in order to behave appropriately.

One option is to store every Scheme value on the heap along with an associated type tag. This is inefficient, however, because in this case any type check will involve a memory access.

Chez Scheme employs a system of pointer tagging in which all values are aligned on an 8-byte boundary, allowing the lower three bits to be used as the type tag [13]. In the case of small data types such as fixnums, the upper bits of a "pointer" simply store the data itself instead of an address. Since there are more than 8 Scheme types, the most common ones (such as pairs and fixnums) are picked to have dedicated type tags, and a separate *escape* tag is used as a catch-all to indicate that a type tag should instead be read from the object that is pointed to.

The pointer bits are divided into segment and offset. Each segment corresponds to a *metatype* – an entry in a segment table. The entries themselves contain information that is used by the garbage collector, including generation information and a dirty vector that indicates assignment. Thus, objects with the same *metatype* occupy the same segment of address space, which is practically useful for Chez Scheme's memory management. This segmenting of objects based on *metatype* properties is a form of the BIBOP (Big Bag of Pages) model [13].

4.3 Impact On Generated Code

A consequence of Chez Scheme's runtime model for continuations and objects is that machine code generated by Chez Scheme will look quite different from what might be generated by a C compiler. In particular, the procedure calling model explicitly separates manipulation of the Scheme call stack from the calls themselves, by using branches for performing a jump to the callee and explicitly performing all frame allocation and return address placement. This is quite different from Wasm's model of function calls, in which allocation of a new stack frame for the callee is an implicit part of the `call` instruction's behavior.

Chapter 5

WEBASSEMBLY (WASM)

This section provides a brief overview of WebAssembly, an emerging platform for running client-side code served over the web.

5.1 History

5.1.1 Client Side Code on the Web

There is a long history of attempts to bring client-side code to the web. Numerous frameworks have been developed over the years, including Adobe Flash, Java Applets, Microsoft ActiveX, and Google's NaCL (Native Client). Ultimately, all have had notable flaws leading to a discontinuation of their use.

It is worth briefly reviewing the aforementioned frameworks, and describing why they are no longer in use. This allows Wasm's existence to be better put into context.

Java Applets were one of the first entries into the space of client-side code on the web, but they faced numerous difficulties. For one, Java was a proprietary technology owned by Sun Microsystems, and thus any efforts to standardize Java and Java Applets would necessarily be tied to Sun's desires. Further, Java Applets were memory intensive and faced numerous security vulnerabilities [29].

Java Applets were largely superseded by Adobe Flash, a web scripting platform from Adobe. Flash allowed for interactive applets with images and video, which were important capabilities to have on the web at the time.

Flash, again, was heavily tied to one company, and security was a large issue. Flash's high resource requirements meant that it was notably not offered as a capability in the IOS web browser. One piece of botnet malware for Flash was so severe that in 2010, almost 10% of all Mac computers were infected [3]. In the mid 2010s, major browser vendors began disabling Flash by default and support for Adobe Flash officially ended in 2020 [3].

Native Client, or NaCL, is a sandboxed environment for running native x86 programs. While security is never guaranteed, the use of an inner and outer sandbox provide assurances that programs cannot misbehave [53]

One of NaCL's largest downsides is its lack of portability; it only supports x86. PNaCL, or Portable Native Client, was an answer to this. PNaCL supports portable LLVM bytecode instead of native code [11]. The primary hindrance to wide adoption of PNaCL was the choice of LLVM as an executable format. LLVM is a very complex bytecode, and its functionality is heavily tied to the Clang project, so its specification is not static. One possible alternative would be to standardize a subset of LLVM for exclusive use on the Web, but with the effort required to do this a new portable executable format for web programming could just as well be created.

5.1.2 JavaScript

JavaScript has so far been the undisputed winner in client-side web programming, with an ECMA standard now in its 13th edition [14]. Originally developed by Brendan Eich over a period of less than two weeks, the first JavaScript interpreter shipped with NetScape Navigator. Since this time, JavaScript has become a "success disaster" of sorts; though the language has numerous warts in its design, it is now one of the most popular programming languages, and every major browser now sports an advanced JavaScript runtime complete with JIT compilation and heavy optimizations.

5.1.3 Emscripten

If we already have JavaScript, why do we need anything else as a compiler target? Emscripten is a project that has explored bringing traditionally native compiled languages – such as C and C++ – to the web [54]. Emscripten has always been a compiler for LLVM, the intermediate representation used by the Clang project. Initially, Emscripten was developed in order to translate LLVM into JavaScript. This was a rather unusual task in that it was effectively the inverse job of a typical compiler; while a typical compiler generally translates a higher-level language into a lower-level one, Emscripten was focused on translating LLVM – an intermediate form of assembly language – into JavaScript, a higher-level language.

There are of course numerous challenges here, since LLVM features unstructured control flow and lower-level manipulation of the stack, while JavaScript notably does not. Another major challenge was mapping LLVM's strictly-typed semantics onto JavaScript, a dynamically-typed language. As a result of the typing challenges in particular, Emscripten ended up targeting a subset of JavaScript known as asm.js [5] that restricts JavaScript operations to those that have more predictable behavior upon JIT compilation in modern JS engines.

It became increasingly clear, however, that asm.js was far from an ideal compiler target, and thus asm.js arguably provided the inspiration for Wasm.

Emscripten now targets Wasm, but does still make use of some of the same compilation techniques that were developed for targeting JavaScript. Though Wasm is lower-level and is designed to be a better compiler target comparatively (it has static typing guarantees, for example), it still uses structured control flow where LLVM notably does not [23]. The presence of structured control flow in Wasm and the challenges this presents for compilation will be explored further later in this thesis.

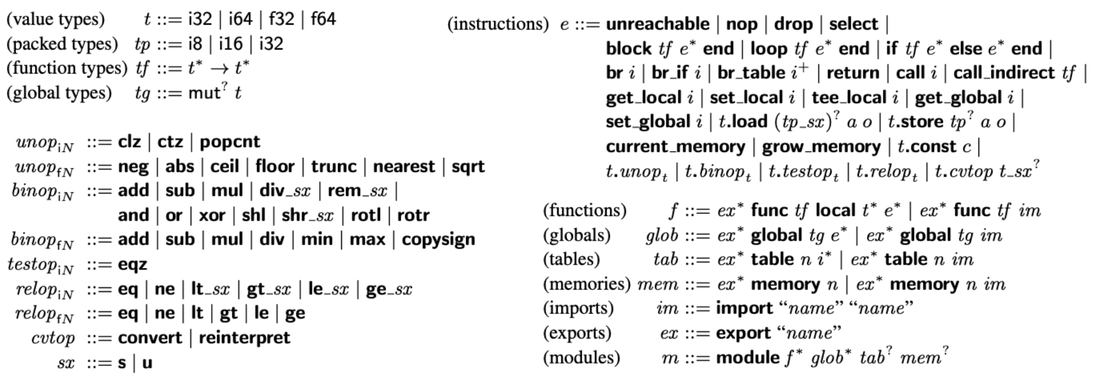
One reason Emscripten has made an impact is because it can be made part of the LLVM toolchain. This then allows any language with a Clang frontend – notably, C and C++ – to be compiled directly to Wasm. To host C and C++ code on the web, Emscripten provides support for its own version of libc [44].

5.2 Enter Wasm

Wasm is not meant to replace JavaScript; instead, it is meant to be a better compiler target to complement JavaScript with high-performance compiled code from other higher-level languages. Wasm code exists in self-contained modules that can interoperate with JavaScript. Thus, performance critical sections of code can be written in a language that is compiled to Wasm, and exported such that JavaScript programs can call into them.

As mentioned previously, Wasm is primarily intended to be a web-compatible compiler target for higher-level languages. Wasm was carefully designed from the beginning with a few specific goals in mind, outlined in [23]; Wasm was designed to be:

- **Safe**. Wasm features memory safety and process isolation. Wasm modules are isolated to the point of near inconvenience; Wasm memory accesses are defined exclusively with respect to memory objects *in a given module.*
- **Fast**. Wasm can be compiled ahead of time to a variety of machine architectures. It is statically typed to allow for a level of safety and greater ahead of



(value types) $t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$

(packed types) $tp ::= \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32}$

(function types) $tf ::= t^* \to t^*$

(global types) $tg ::= \mathsf{mut}^? \; t$

$unop_{\mathrm{i}N} ::= \mathbf{clz} \mid \mathbf{ctz} \mid \mathbf{popcnt}$

$unop_{\mathrm{f}N} ::= \mathbf{neg} \mid \mathbf{abs} \mid \mathbf{ceil} \mid \mathbf{floor} \mid \mathbf{trunc} \mid \mathbf{nearest} \mid \mathbf{sqrt}$

$binop_{\mathrm{i}N} ::= \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div}\_sx \mid \mathbf{rem}\_sx \mid$
$\mathbf{and} \mid \mathbf{or} \mid \mathbf{xor} \mid \mathbf{shl} \mid \mathbf{shr}\_sx \mid \mathbf{rotl} \mid \mathbf{rotr}$

$binop_{\mathrm{f}N} ::= \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div} \mid \mathbf{min} \mid \mathbf{max} \mid \mathbf{copysign}$

$testop_{\mathrm{i}N} ::= \mathbf{eqz}$

$relop_{\mathrm{i}N} ::= \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{lt}\_sx \mid \mathbf{gt}\_sx \mid \mathbf{le}\_sx \mid \mathbf{ge}\_sx$

$relop_{\mathrm{f}N} ::= \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{lt} \mid \mathbf{gt} \mid \mathbf{le} \mid \mathbf{ge}$

$cvtop ::= \mathbf{convert} \mid \mathbf{reinterpret}$

$sx ::= \mathbf{s} \mid \mathbf{u}$

(instructions) $e ::= \mathbf{unreachable} \mid \mathbf{nop} \mid \mathbf{drop} \mid \mathbf{select} \mid$
$\mathbf{block}\; tf\; e^*\; \mathbf{end} \mid \mathbf{loop}\; tf\; e^*\; \mathbf{end} \mid \mathbf{if}\; tf\; e^*\; \mathbf{else}\; e^*\; \mathbf{end} \mid$
$\mathbf{br}\; i \mid \mathbf{br\_if}\; i \mid \mathbf{br\_table}\; i^+ \mid \mathbf{return} \mid \mathbf{call}\; i \mid \mathbf{call\_indirect}\; tf \mid$
$\mathbf{get\_local}\; i \mid \mathbf{set\_local}\; i \mid \mathbf{tee\_local}\; i \mid \mathbf{get\_global}\; i \mid$
$\mathbf{set\_global}\; i \mid t.\mathbf{load}\; (tp\_sx)^?\; a\; o \mid t.\mathbf{store}\; tp^?\; a\; o \mid$
$\mathbf{current\_memory} \mid \mathbf{grow\_memory} \mid t.\mathbf{const}\; c \mid$
$t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop\; t\_sx^?$

(functions) $f ::= ex^*\; \mathbf{func}\; tf\; \mathbf{local}\; t^*\; e^* \mid ex^*\; \mathbf{func}\; tf\; im$

(globals) $glob ::= ex^*\; \mathbf{global}\; tg\; e^* \mid ex^*\; \mathbf{global}\; tg\; im$

(tables) $tab ::= ex^*\; \mathbf{table}\; n\; i^* \mid ex^*\; \mathbf{table}\; n\; im$

(memories) $mem ::= ex^*\; \mathbf{memory}\; n \mid ex^*\; \mathbf{memory}\; n\; im$

(imports) $im ::= \mathbf{import}\; \text{“}name\text{”}\; \text{“}name\text{”}$

(exports) $ex ::= \mathbf{export}\; \text{“}name\text{”}$

(modules) $m ::= \mathbf{module}\; f^*\; glob^*\; tab^?\; mem^?$

Figure 5.1: Full Wasm grammar [23]

time optimization, and its low-level stack operations translate fairly easily to native operations on modern processors.

- **Portable**. Wasm has a precise and very simple formal semantics [23] that makes no reference to any underlying platform. Wasm can be executed in any compliant virtual machine. Wasm was designed so that support for the format could be added with only small extensions to existing JavaScript VMs, to allow for increased adoption.

- **Compact**. Wasm's creators recognized that any format for storing executable code to be served over the web must be compact to allow for easy transmission over the wire. Wasm is defined as a binary format, and this decision alone allows for a more compressed representation. Further, it takes only a **single pass** to verify that a Wasm module is well-formed [23].

5.3 Syntax

Though Wasm code is executed in binary form, Wasm syntax is defined in terms of a formal grammar [23]. The grammar is quite compact, and is shown in Figure 5.1.

Wasm further has a text-based format known as WAT (WebAssembly Text) that uses an S-expression based representation for meta-elements – such as modules, imports, and definitions – as well as instructions themselves.

An example program is shown in Figure 5.2.

5.4 Semantics

5.4.1 Stack-based Operations

Wasm is a stack-machine [23], meaning that all operations are defined with respect to an abstract expression stack. It is established that stack-based bytecode formats generally have a smaller encoding size [37], which is a major reason for this choice in Wasm's design.

Nearly every instruction supported in Wasm will either consume one or more values by popping them off the evaluation stack, or produce one or more values by pushing them onto the evaluation stack. Many instructions do both. For example, the `i32.add` instruction (seen in Figure 5.2) expects two operands of type `i32` to be placed at the top of the evaluation stack, and produces a single result: their sum, with a defined overflow behavior.

5.4.2 Structured Control Flow

A notable feature of Wasm is its *structured* control flow, which is unusual for a low-level bytecode. What this means is that control in Wasm is achieved through the use of the constructs `loop`, `block`, `if/else`, `br`, `br_if`, `call`, `call_indirect`, `return`, `br_table`, and `unreachable` – several of which are similar to features found in high-

```
(module
   (memory 100)
   (export "fact" (func $fact))
   (func $fact (param $n i32) (result i32)
       (local $i i32)
       (local $p i32)

       (i32.const 1) ;; push i32 constant 1 onto stack
       (local.set $p) ;; pop stack value and store in local

       (i32.const 1)
       (local.set $i)

       (block $outer ;; a named BLOCK, $outer.
           (loop $loop ;; a named LOOP, $loop
               (local.get $i)
               (local.get $n)
               (i32.gt_s)
               (br_if $outer) ;; a branch to the END of
                               ;; $outer

               ;; instructions can be nested
               ;; in an expression based form;
               (local.set $p (i32.mul
                               (local.get $p)
                               (local.get $i)))
               (local.set $i (i32.add
                               (local.get $i)
                               (i32.const 1)))

               ;; a branch to the BEGINNING of $loop.
               (br $loop)))

       ;; implicit return
       (local.get $p)))
```

Figure 5.2: An simple factorial routine in Wasm

level languages. Wasm does **not** allow unstructured control flow constructs such as `goto` or arbitrary branching. The use of structured control flow prevents irreducible control flow graphs [23]. A CFG is *reducible* if and only if every sub-graph identifying a loop is dominated by a loop header. This means that loop headers can be identified using back edges in the CFG, leading to a linear time algorithm for loop identification [35]. The structured control flow restriction makes it possible to validate Wasm code in a single pass.

In Figure 5.2, we can observe Wasm's structured control flow. A block is best understood as the introduction of a new "control scope" on a control stack, and a branch behaves more like a "return" operation than a jump. Because of this abstract model, control can only be transferred to blocks in scope, and the point of transfer in the enclosing block is determined by the type of block. In Figure 5.2, we see a `loop` block that is used to implement a local loop for computing a factorial product. Thus, the `br $loop` instruction unwinds the control stack up until the `$loop`, and so control re-enters `$loop`.

We also see a `block`, `$outer`, that wraps the loop. Since `$outer` is a `block`, when the instruction `br_if $outer` is executed all nested blocks *including* `$outer` are popped from the control stack so control resumes *after* the `$outer` block.

5.5 Wasm Implementations

Wasm is now supported in nearly every major browser including Chrome, Firefox, Safari, and Microsoft Edge. It is supported in Chrome's V8 [48], Firefox's Spider-Monkey [39], and Apple's JavaScriptCore [27] JavaScript engines. Additionally, there are multiple standalone Wasm VMs, including WasmTime [47] and Wasmer [46].

The primary role of a Wasm runtime is to execute Wasm programs – according to the Wasm specification – in some fashion. Wasm can be fully interpreted, but it is often AOT or JIT compiled for performance reasons. JIT compilation in particular is popular for in-browser Wasm implementations, because minimizing page startup time is highly important in a browser context.

The Wasm runtime is a standard user space program, but it does need to provide an interface for the Wasm programs it executes to access operating system capabilities. Emscripten's original solution to this problem was to port libc to the browser, providing implementations of routines such as `malloc`, `printf`, and the like that were specific to running in a browser environment [44]. However, this is non-standardized and web-specific, and Wasm is intended to be a truly platform independent standard. As a result, a group of interested parties started the WASI (WebAssembly System Interface) Working Group [45]. The general idea behind WASI is to develop a standard for a platform-independent interface that Wasm programs can call into. As a result, toolchain implementations for various languages including C/C++, Rust, and others can add WASI support to their standard libraries to allow them to target a platform-independent Wasm implementation effectively [44].

Chapter 6

CHALLENGES IN USING A CHEZ SCHEME BACKEND

Chez Scheme uses a nanopass architecture [28] in order to compile Scheme from source form all the way down to machine code. Using the nanopass framework, Chez Scheme defines around 35 intermediate languages and around 50 micro-passes to translate between them [28], where multiple passes may be applied to the same intermediate language. Each intermediate language may extend a previous language by removing high-level constructs in the language being extended, and/or adding new low-level constructs. For example, the `L4` language extends the `L3` language, but removes the `set!` form for variables that have indefinite extent. The pass that translates L3 to L4 is called `convert-assignments`, and its only job is to remove the `set!` form for variables with an indefinite lifetime [6].

Chez Scheme's final intermediate languages are numbered `L15a...L15c, L16`, with `L16` being the last intermediate language before machine code. The `L15*` languages are responsible for various transformations that are just above the machine code level. `L15c` consists of low-level assignments (to temporaries or memory locations) and simple value producing operations, with control flow being accomplished via predicates and jumps. The instruction selection pass consumes `L15c` and outputs `L15d`, where `L15d` augments `L15c` with actual machine instructions. The instruction selection pass relies on Chez Scheme backend modules, one of which exists for each ISA that is supported by Chez Scheme.

Each backend module must implement a specific backend interface, and the backend interface assumes capabilities that would be difficult to implement for Wasm. In his Master's thesis, Grant Matejka outlined some of these difficulties [32]. These observations are worth re-stating and elaborating on, to make the reasons for our own approach clearer.

6.1 Registers

First and foremost, the Chez Scheme backend interface assumes the use of registers. Wasm has no registers; instead, Wasm has locals that are essentially named locations on the virtual stack. A fixed set of locals could be used to emulate registers, however, so this is not necessarily a critical issue. It is unclear how applying Chez Scheme's register allocation pass to a collection of Wasm locals would impact the quality of generated Wasm, given that a Wasm JIT will ultimately be performing its own register allocation.

6.2 Arbitrary Jumps

Jump translation would be one of the more difficult pieces to adapt Chez Scheme's backend to. Any Chez Scheme backend must expose a procedure, `md-handle-jump`, that the Chez Scheme instruction selection pass uses directly when performing the translation between `jump` operations in `L15c` and `L15d`. `md-handle-jump` must translate intermediate jump instructions in `L15c` into a form that is compatible with a given architecture. It is assumed that `md-handle-jump` can handle several jump forms that occur in `L15c`, including: jumping to a memory address, jumping to a label, jumping to a literal (relocated value), or jumping to an immediate.

Jumping to a memory address is simply impossible in Wasm; branches are defined with respect to labels, and calls are defined in a type-safe way with respect to functions (or typed function pointers) [23]. The same issues hold for jumping to a relocated address, or jumping to an immediate value.

Jumping to a predefined *label* would be possible, provided block nesting allows for it. However, a CFG transformation such as Emscripten's Re-Looper [54] would need to be performed to ensure that a block is nested such that it can branch to its successors. This is an involved transformation and would likely involve adding a new pass.

6.3 Calling Conventions

R6RS Scheme guarantees proper tail calling behavior. At an implementation level, this means that if a function's body is reduced to a call with no continuation, the call must re-use the same stack frame. In his paper outlining the formal semantics of tail calls [8], William Clinger notes that under a proper tail-call model, a call is best understood as a *jump* that changes an environment register. The stack is used for storing the current continuation, or the work that remains to be done [10]. As a result, when a call occurs, a new stack frame is only needed to avoid overwriting the current continuation if it exists, *not* simply as a feature of the call.

This insight is crucial to understanding Chez Scheme's calling convention; it does not use stack frames unless it is forced to. By default, it treats every call as a branch, and uses the stack on a by-need basis. Structured returns are never used within Scheme code. Any return is an explicit branch to a return address, where the return address is stored on the stack only if there is a non-empty continuation for the current call.

Wasm is a notable departure from this model, since it *only* uses structured calls. In Wasm's formal semantics [23], calls reduce to the creation of a new locals block on the execution stack with certain locals initialized to the argument values. This simply does not allow us to emulate proper tail-calling behavior at the function level.

Wasm's upcoming tail call proposals may allow a `return_call` instruction with proper tail calling semantics [50], but the current state of Wasm does not support this semantics. This is a major hurdle to cover in using Chez Scheme's backend.

6.3.1 Separate Compilation Pipeline?

In his thesis on Rasm, Grant Matejka theorized that a higher-level intermediate language from a much earlier pass of Chez Scheme might be used as a branching off point for compilation to Wasm, as it is the subsequent passes that transform the intermediate language into a form that is increasingly less amenable for translation to Wasm's control constructs [32]. The hindrance here is that an entirely separate compilation pipeline would need to be created, specifically for Wasm. There would be significant complexity involved, and the pipeline would be highly specific to Wasm and thus more difficult to justify maintaining.

6.4 Final Rationale

We decided not to pursue the path of backend creation. While it may ultimately be possible to adapt Chez Scheme's backend for use with Wasm, it is unfortunately not a straightforward solution. Most likely, new intermediate languages would need to be designed and introduced, with new passes to go along with them. These would have to be somehow slotted in with the existing set of passes, which may be tricky if they are Wasm-specific.

Fundamentally, any Wasm-specific passes would break the abstraction imposed by the backend, which assumes an identical set of passes for every supported ISA (with implementation-level details abstracted away by the backend interface). The approach we will introduce shortly alleviates the need for any backend modification, and thus greatly simplifies the development process.

Chapter 7

PORTABLE BYTECODE (PB)

Our approach relies on a bytecode known as PB, or Portable Bytecode. PB is already a valid output target for Chez Scheme; i.e., a PB backend module has already been created [33]. An interpreter for the bytecode already exists and can be compiled along with Chez Scheme's runtime, thus allowing programs compiled to PB to be executed.

7.1 Porting The Compiler

The primary reason for PB's existence is portability. Chez Scheme is a self-hosted compiler, meaning that the compiler itself is written in Chez Scheme. Chez Scheme itself is a superset of the R6RS standard. If Chez Scheme is to be ported to a new platform, there are several pieces that must be in place. First and foremost, a backend must be created for the new platform, using Chez Scheme's existing backend framework. Many different ISAs are already supported, including x86, ARM, and PowerPC.

With a backend in place, the problem of compiling the *compiler source itself* is a roadblock.

Suppose we wish to compile Chez Scheme for some new platform x. One option would be to use an existing (different) Scheme implementation that can run on platform x to create a new build. Unfortunately, since Chez Scheme is a unique superset of R6RS, we are unlikely to find another Scheme implementation that can compile Chez Scheme on platform x.

Because of this, we have to resort to building Chez Scheme (with the new added backend) on a separate machine with say, architecture y, on which Chez Scheme *is already* supported. Then, the new compiler (which runs on platform y but now has the capability to target platform x) could be used to cross-compile itself, and the resulting binary could be moved to a machine of type x.

Ideally, we would wish for Chez Scheme to be bootstrapped on a single machine with no dependencies other than the compiler source itself (and standard build tools). This is where the utility of PB comes into play. The Chez Scheme compiler and kernel (including any Scheme-implemented runtime facilities) can be compiled to platform-independent PB bytecode and distributed along with the source. Then, on a new machine of type x, a user can run a PB-build, which will first compile the C portion of the Chez Scheme kernel – importantly, the PB interpreter – for machine type x. The assumption is that x will always have a compatible C compiler. Then, the provided boot files (which include the PB-compiled version of Chez Scheme itself) can be run with the PB interpreter, effectively acting like a temporary Chez Scheme installation on platform x. This new version of Chez Scheme can cross compile the existing source for platform x, thus yielding a native executable.

7.2 PB Instructions

The PB architecture resembles a RISC architecture such as ARM. It has instructions that can generally be divided into the following categories:

- **Numeric**; integer/floating point arithmetic and bitwise operations
- **Control**; several different branching variants

- **Memory and Indirection**; including variants of load and store, and notably a `literal` instruction that allows constant values to be interspersed in the instruction stream and loaded
- **Mov**; numerous variants of `mov`, including those that allow for casting between native floating point and integer values
- **Synchronization**; CAS, lock, fence, and atomic increment instructions
- **FFI**; instructions for reading from and writing to a separate "arena" used for calling functions that use a C-style calling convention, and performing C-style calls

There are a handful of other notable instructions that would not fall under the above categories:

- **Interp** instructions allow for re-starting the PB interpreter loop with a separate execution context. The instructions allow for recursive nesting of interpreter loop calls.
- **Call** instructions support a fixed set of C prototypes that are specific to Chez Scheme runtime functions. The interpreter will execute a `call` instruction as a standard C-call to an address provided as part of the instruction. Arguments are loaded from a set of predefined registers, and the return value is placed in a predefined register.
- **Return** instructions specifically return from an *instance* of the PB interpreter loop. Thus, they may be used for premature exit from an interpreter instance.
- **PBChunk** instructions contain an index into a function pointer table. When a pbchunk instruction is encountered, the interpreter, that is written in C, loads

the indicated function address – that refers to a separate function written in C – from the table and calls the referenced function. Thus, a PBChunk instruction results in a standard C-style call. The result of the call becomes the value of the instruction pointer for the next iteration of the interpreter loop. PBChunk instructions allow for generated fragments of C code to be used in place of the bytecode that follows the `pbchunk` instruction. These generated fragments may improve performance, as they are effectively an "unrolling" of the interpreter loop. Notably, the table of chunk functions must be statically known when the interpreter is compiled, as they are statically linked with the interpreter code. Thus, only a limited set of chunk functions that are specific to a particular program may be used in `pbchunk` instructions.

7.3 Disassembler

Racket contains a `raco` sub-command that is host to many utilities. Among them is `disassemble`, a command that allows for disassembling individual functions that are compiled and stored in-memory. The disassembler has support for x86, Aarch64, and i386. We augmented the disassembler with support for PB.

At a high-level, `raco disassemble` takes a Racket procedure and dispatches the compiled function to a handler for disassembly. `raco` disassemble supports both Racket BC and Racket CS.

In the case of Racket CS, the underlying Chez Scheme debugging facilities are used to extract the procedure and its metadata. The `inspect/object` primitive provided by Chez Scheme is used to extract a list of relocations that are attached to a procedure. The raw bytes consisting of the procedure instructions, along with a list of relocations, are passed to a routine for disassembly.

7.3.1 Relevant Concepts for Disassembly

Code Objects. In Chez Scheme, compiled code is stored in-memory as a code-object. Generally, a code object corresponds to the code for an individual function or closure. Code objects may represent built-in library functions, foreign procedures, and user-defined procedures and closures.

Relocations are used as placeholders for data that may not be known at compile time. The data can be a Scheme object such as a list, symbol or string. A relocation may also be the address of a library or primitive function.

For an example of a primitive function, consider the procedure `+`, taking one or more numbers and returning their sum. The implementation for `+` is a library function within Chez Scheme's boot files, which are loaded into memory when Chez Scheme is started.

Scheme code is compiled to `.so` shared object files that notably are **not** linked to the contents of boot files.

As a result, the code object for `+` will not be present in a compiled Scheme program. Thus, when a particular `.so` file is loaded by Chez Scheme, it must be linked with primitive library functions that are in Chez Scheme's address space. Because of this dynamic linking, all code objects have an attached list of relocations.

Each relocation contains:

- An offset into the code object
- A piece of data to be linked (a Scheme object or a primitive/library function)

- A format which is machine-specific and indicates how the relocated value will actually be loaded in machine instructions.

The linker will fill in each specified relocation entry within a code object at link-time with the appropriate data to be linked.

In PB, relocations generally appear as placeholder bytes immediately following `literal` instructions. At runtime, the `literal` instruction will be used to load the relocated data, which is either 4 or 8 bytes in length, into a register.

RP-Headers (return-point headers) are used for the purposes of garbage collection. They are necessary because the Scheme stack is heap allocated and garbage collected. In the case of a non-tail call, there will be a return back to the previous stack frame, meaning that there *may* be data which needs to be retained in said stack frame. The GC needs to know whether the stack frame for a caller could still contain live data.

For the purposes of disassembly, rp-headers must be differentiated from instructions and treated as raw binary data.

Labels. The address of an instruction within the instruction range for a procedure is considered a label if any other instruction within the procedure branches to it. The disassembler should identify and reconstruct labels in order to provide a more faithful output.

7.3.2 Design of the Disassembler

The disassembler works in two passes.

The first pass is needed to identify addresses that are labels. Since back-branching is allowed, the entire instruction stream must be scanned in order to collect all local branch targets (branch targets within the bounds of the function being disassembled). A non-local branch target is a branch target that is loaded from memory or a `literal` instruction. The disassembler does not give any special treatment to non-local branches.

As an additional portion of the first pass, the disassembler collects rp-headers. Differentiating rp-headers from normal instructions is tricky, because there is no metadata which describes rp-header placement within the instruction stream. Fortunately, for PB a convention is followed: the `adr` instruction, which loads an address at an offset from the instruction pointer, will only load an address immediately following an rp-header. This is because rp-headers occur at "return-points," and the `adr` instruction is nearly always used for loading return addresses. As a result, `adr` instructions can be used as markers for identifying rp-headers.

The second pass of the disassembler is a scan over instructions whilst performing book-keeping using the list of headers, labels, and relocations. Labels are printed appropriately, and the contents of headers and relocation entries are printed as raw, non-instruction data. Care must be taken to skip over an appropriate number of bytes before interpreting the next instruction in the case of non-instruction data. `literal` instructions are an interesting case, as they are technically variable-width. The length of data following a literal instruction is equal to a machine word, meaning that it may be either 4 or 8 bytes, depending on the platform.

```
(define (add3 x y) (+ x y 3))
```

Figure 7.1: add3 example

7.4 PB Examples: Disassembler Output

7.4.1 Example 1: add3

As a first example, consider the code in Figure 7.1: a simple procedure which takes two numbers, `x` and `y`, and adds `3` to their sum.

The PB instructions for `add3` are shown in Figure 7.2. There are a variety of constructs here to explain. While the form of the PB instructions is quite similar to that of many other architectures, Chez Scheme's code generation conventions mean that the output may not be easily comprehensible.

At address `8` we check that the value `%ac0` is equal to 2. This is effectively an argument count check (recall that the procedure `add3` takes two formal arguments). In the event that `%ac0` contains the wrong count, we jump to `.l0`, where we load the relocation address of `doargerror` – the argument error handler – into `%ts`, before branching to it. A sequence of `(literal %reg) <data> (b %reg)` is extremely common in Chez Scheme-generated PB, where `<data>` is a relocation entry corresponding to a procedure whose address is a relocation.

After the argument count check, we move an immediate value into `%r11`. Chez Scheme uses a system of pointer tagging to allow for runtime type-checking, and this system also allows for a size-limited set of constants to be encoded. In this case, the value `0x18` is the Chez Scheme encoding of the value 3; it is equal to the constant 3, shifted left 3 positions. The rightmost 3-bits (all zeros in this case) can be checked as a mask

```
     0: 00013339          (subz %trap %trap (imm #x1)
                                      #:signal #t)
     4: 000034cf          (btrue (label l1 (imm 52)))
     8: 0002044b          (eq %ac0 (imm #x2))
    12: 00001ccd          (bfalse (label l0 (imm 28)))
    16: 00180b02          (mov-16 %r11 (imm #x18))
    20: 00000501          (literal %xp)
    24: 0366be0b          (data)            (relocation +)
    28: 00000001          (data)
    32: 000558b1          (ld-int64 %cp %xp (imm #x5))
    36: 00030402          (mov-16 %ac0 (imm #x3))
    40: 000d05d3          (b* %xp (imm #xd))

.l0:
    44: 00000601          (literal %ts)
    48: 03a118f0          (data)
        ... (relocation #<code doargerr>)
    52: 00000001          (data)
    56: 000600d0          (b %ts)

.l1:
    60: 00000601          (literal %ts)
    64: 03a13dc0          (data)
        ... (relocation #<code event ...
    68: 00000001          (data)
    72: 000600d0          (b %ts)
```

Figure 7.2: Disassembler output for add3

```
        (define (max a b)
            (if (> a b) a b))
```

Figure 7.3: max example

to determine the type of data. We then load the address of the procedure `+` using an argument count of 3, before branching to `+`.

7.4.2 Example 2: max

As a slightly more complex example, consider a simple definition of `max` in Figure 7.3, returning the greater of two numbers.

In this example, runtime type-checking is used to enable polymorphic behavior. The key insight here is that `a` and `b` are assumed to be numeric as they are passed as arguments to the procedure `>`. However, there is no restriction on the type of `a` and `b` beyond this. Scheme supports both fixed-size and arbitrary precision integers, fixed-size floats, and infinite precision rational numbers. Fixed size integers can be stored in a register and used with native comparison operations. Arbitrary precision integers and rationals are stored as pointers to a Scheme-specific structure that must be passed to Scheme standard library procedures such as `>`. Thus, the actual behavior of the function differs depending on the precise runtime types of a and b.

Looking to Figure 7.4, we can see how this runtime polymorphism presents itself in the PB instructions.

Instructions at addresses 8-16 combine the arguments, stored in `%r9` and `%r10`, together before checking their mask with the constant `7`. If the mask is zero, then we perform a standard, native `gt` operation in order to determine which argument is larger, and branch to either .l0 or .l1 depending on the result.

```
      0:  0002044b          (eq %ac0 (imm #x2))
      4:  0000b8cd          (bfalse (label l4 (imm 184)))
      8:  000a9f24          (ior %r15 %r9 %r10)
     12:  00070f5b          (cc %r15 (imm #x7))
     16:  000018cd          (bfalse (label l2 (imm 24)))
     20:  000a094e          (gt %r9 %r10)
     24:  000008cd          (bfalse (label l1 (imm 8)))

.l0:
     28:  0009040a          (mov %ac0 %r9)
     32:  000001d3          (b* %sfp (imm #x0))

.l1:
     36:  000a040a          (mov %ac0 %r10)
     40:  000001d3          (b* %sfp (imm #x0))

.l2:
     44:  00013339          (subz %trap %trap (imm #x1)
                                          #:signal #t)
     48:  000040cd          (bfalse (label l3 (imm 64)))
     52:  000819c5          (st-int64 %r9 (imm #x8) %sfp)
     56:  00101ac5          (st-int64 %r10 (imm #x10) %sfp)
     60:  00181117          (add %sfp %sfp (imm #x18))
     64:  00009fd7          (adr %r15 (imm #x24))
     68:  00001fc5          (st-int64 %r15 (imm #x0) %sfp)
     72:  00000601          (literal %ts)
     76:  0f231d80          (data)  (relocation #<code event>)
     80:  00000001          (data)
     84:  000600d0          (b %ts)
     88:  00000099          rp-header
     92:  00000000          (data)
     96:  0000018d          (data)
    100:  00000000          (data)
    104:  00181119          (sub %sfp %sfp (imm #x18))
    108:  000819b1          (ld-int64 %r9 %sfp (imm #x8))
    112:  00101ab1          (ld-int64 %r10 %sfp (imm #x10))
```

Figure 7.4: PB instructions for max

```
.l3:
   116: 000819c5        (st-int64 %r9 (imm #x8) %sfp)
   120: 00101ac5        (st-int64 %r10 (imm #x10) %sfp)
   124: 00181117        (add %sfp %sfp (imm #x18))
   128: 00009fd7        (adr %r15 (imm #x24))
   132: 00001fc5        (st-int64 %r15 (imm #x0) %sfp)
   136: 00000601        (literal %ts)
   140: 0f226ee0        (data)  (relocation #<code >>)
   144: 00000001        (data)
   148: 000600d0        (b %ts)
   152: 000000d9        rp-header
   156: 00000000        (data)
   160: 0000018f        (data)
   164: 00000000        (data)
   168: 00181119        (sub %sfp %sfp (imm #x18))
   172: 000819b1        (ld-int64 %r9 %sfp (imm #x8))
   176: 00101ab1        (ld-int64 %r10 %sfp (imm #x10))
   180: 0006044b        (eq %ac0 (imm #x6))
   184: ffff68cf        (btrue (label l1 (imm -152)))
   188: ffff5cd1        (b (label l0 (imm -164)))

.l4:
   192: 00000601        (literal %ts)
   196: 0f20d8f0        (data)  (relocation #<code doargerr>)
   200: 00000001        (data)
   204: 000600d0        (b %ts)
```

Figure 7.4: PB instructions for max

Notice that `%ac0` is used for storing the return value. At labels .l0 and .l1, we see the instruction `(b* %sfp (imm #x0))`. This is an important idiom to understand. `%sfp` is the *Scheme Frame Pointer*, and it holds a pointer to Chez Scheme's separate control stack (see Chapter 2). By convention, return addresses are stored at an offset of 0 from the current value of `%sfp`, so `(b* %sfp (imm #x0))` is effectively a return operation.

The labels .l2 and .l3 handle the case where the passed arguments are Scheme objects. `.l2` contains a call to an "event" handler, used for various runtime purposes in Chez Scheme. We will focus on `.l3`. Instructions at addresses 116-124 store the passed arguments on the Scheme stack through `%sfp`. Notice that 128-132 store a return address at `%sfp` offset 0. A call to the procedure `>` occurs at 136.

Notice that at byte 152, we have an `rp-header` that will never be executed as code due to the call immediately before and the return address being immediately after. Instructions at addresses 168-176 are standard caller cleanup, since in this case we made a non-tail call. At the end of the function, we check the return value in `ac0` for boolean `#t`, encoded as a tag, before branching back to either `.l0` or `.l1` and resuming the same code path as the native case.

Chapter 8

PARTIAL BYTECODE COMPILATION AND PBCHUNK

8.1 Interpreter Specialization Background

The approach to compilation taken in this thesis is a partial one. It bears some resemblance to program specialization, and in particular, interpreter specialization, so it is worth introducing the concept of interpreter specialization and reviewing some existing work.

Generally speaking, program specialization applies if we have a program $prog(x, y)$ that operates on two (or more) inputs. Suppose we are operating in a situation where x is fixed, i.e., $x = x_0$. The goal of program specialization is then to create an optimized program $prog_{x_0}(y)$ that is equivalent to $prog(x_0, y)$ but is specialized to the known fixed input x_0 [31].

In the case of *interpreter specialization*, we have some interpreter program $int(p, d)$ for a bytecode language L, where p is a program to be interpreted (written in L), and d is some unknown input to the program. If we fix $p = p_0$, then we may construct a *specialized* interpreter program $int_{p_0}(d) = int(p_0, d)$, where $int_{p_0}(d)$ may have improved performance for some d.

8.1.1 Some Related Work

Interpreter Specialization may be static (ahead of time), or dynamic, based on the actual flow of execution when an interpreter is executing a program. For an example of dynamic specialization, JavaScript interpreters can be made to record traces of execution and emit type-specialized code that implements said execution in native machine instructions [21].

Direct-threading is a form of static specialization. Simple bytecode interpreters are generally implemented as a dispatch loop, where at each iteration an instruction is read and control is "dispatched" to a piece of code in which the interpreter performs the work of the instruction. Piumarta and Riccardi [34] investigate removing dispatch overhead by direct threading, or replacing each opcode with the actual address of code that performs the opcode.

Piumarta and Riccardi also experiment with creating "macro" instructions that are the concatenation of the implementations for multiple bytecode instructions. Executing a macro-instruction eliminates the overhead from dispatch **and** instruction fetching, since the implementation for a macro-opcode is simply a linear sequence of code that *performs* the work of the actual instructions. This yields performance of up to 70% that of natively compiled C in some cases [34]. The creation of macro-instructions may be performed ahead of time (statically), or dynamically based on which commonly executed sequences of instructions appear in a target program.

Interpreter specialization is effective in a variety of contexts. Other investigations, such as those from Thibault et al., show the use of interpreter specialization as a means to improve the performance of a variety of languages, including bytecode interpreters for Java, OCaml, and even a DSL for specifying networks [41].

8.1.2 Differences in our Approach

We implement the "macro-opcode" approach in this thesis. In our case, we create macro-opcodes in an ahead-of-time fashion, where each macro-opcode is the Wasm implementation of one or more instructions. Our approach does not modify the interpreter code in any fashion; rather, the interpreter has already been created with the ability to deviate from its normal execution path to call into pre-compiled functions that implement macro-opcodes. We simply create a pass to generate macro-opcodes from selected portions of bytecode.

8.2 PBChunk and its Implementation

PBChunk is a form of *partial bytecode compilation* that uses the macro-opcode approach. It statically compiles sections of PB bytecode into equivalent C-code that performs the work of the relevant PB instructions. The C-code forms the implementation for macro-opcodes. PBChunk is a system designed and implemented by Matthew Flatt, one of the core authors of the Racket compiler. As was described in the previous chapter, PB is primarily used for bootstrapping Chez Scheme. PBChunk is meant to provide a degree of speed up for this bootstrapping process.

PBChunk is a heavy inspiration for our work, and thus it is essential to have a working understanding of the PBChunk approach to provide context for our own.

PBChunk is designed to operate on *boot files*, which have been mentioned previously. When Chez Scheme is built for a new system, the executable form of the compiler along with all the supporting primitives and library functions that are implemented in Scheme, are stored in boot files. As was mentioned in Chapter 7, every Chez Scheme program will be dynamically linked with code in the boot files at load-time, meaning that nearly every program depends on the boot files implicitly.

8.2.1 Boot Files

Any effort to optimize compiled code stored in boot files is likely to have a reasonably high impact on performance, because most Scheme programs will make use of code that originates in boot files heavily. For more examples of primitives, consider the use of functions such as `list`, `car`, `cdr`, and `map`; arithmetic operations such as `+`, `-`, `*`, `/`, and vector operations such as `vector-set!` and `vector-ref`. Note that certain primitives may be inlined by the compiler, but oftentimes the use of a primitive still involves a call to a Scheme library function (consider the calls to `>` and `+` in the PB examples from the previous chapter).

Under a system built for PB, code that exists in the Chez Scheme boot files will be compiled to PB bytecode, meaning that a call to any primitive will involve *interpreting* the code for that primitive. Thus, the idea behind PBChunk is to create a chunk or *macro-opcode* for the instructions that make up a primitive, eliminating a significant amount of interpretation overhead.

For an example of a chunked primitive, consider the function `list`. `(list ...)` constructs a new list using a variable number of Scheme objects that are passed as arguments. A portion of the PB-compiled form of `list` is shown in Figure 8.2.

8.2.2 Chunk Selection

In PBChunk, chunking is done by first *identifying* ranges of instructions within code objects that will become chunks. Boot files are serialized in a binary format called FASL, and so PBChunk takes advantage of Chez Scheme's FASL-parsing utilities to parse the boot file and search through its contents for code objects. Note that there may be top-level objects in the FASL file – such as vectors and hashtables – which

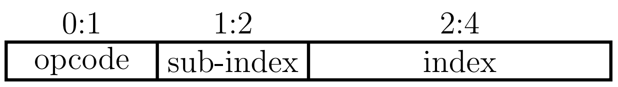


Figure 8.1: Byte-level instruction format for pbchunk instruction

are not themselves code but may contain references to code objects. Thus, the search process is recursive.

Once code objects are identified, they are split into zero or more *chunks* that represent ranges of instructions within the code object. The instructions that make up those instruction ranges are translated into equivalent C code that *performs* the underlying instructions for a chunk. The C code itself is meant to be linked with the PB-interpreter, and thus the C code will make use of C macros and types defined by the interpreter itself.

8.2.3 Chunk Instructions

A new instruction, `pbchunk`, is used to implement chunking. The instruction has a form shown in Figure 8.1.

For each selected chunk, the *first* instruction in said chunk's instruction range is overwritten with a `pbchunk` instruction. Thus, the instructions within code objects are modified to allow for chunking to occur. As a result, PBChunk produces modified boot files in addition to generated C-code.

8.2.4 Chunk Structure

The chunks themselves take the form of C functions with the signature:

```c
static uptr chunk_n(MACHINE_STATE ms, uptr ip, int sub_index) {...}
```

where ms is a pointer to a structure that holds the PB interpreter state, seen in Figure 8.4, ip represents the value of the instruction pointer *on entry* to a chunk, and sub_index is the index of a *chunklet*, or a single label within the chunk.

A chunk function's return value is the address of the next instruction to execute upon exit from the chunk. This may be statically known when the chunk is created, or (in the case of conditional branch instructions) the chunk code itself may determine this based on its execution of instructions.

8.2.5 Chunk Contents

Each line within a chunk is a statement or C macro that implements the actual instruction in question. For example, do_pb_mov16_pb_zero_bits_pb_shift0(4, 38) corresponds to the mov-16 instruction at address 8, and expands to the C statement: regs[4] = (uptr)38; it is the equivalent interpreter code for moving the immediate value of 38 into the 5th PB register, with no shift applied.

For instructions that set a flag, such cmp variants, a local flag variable is used to hold the flag state.

8.2.6 Chunk Returns and Trampolining

For an example of when a chunk function might return, observe the b* (branch-indirect) operation at address 0xC in Figure 8.2. Looking to the PBChunk translation in Figure 8.3, this instruction is implemented by the macro get_pb_bs_op_immediate_addr applied to sfp and the immediate offset 0x0; This performs the address calculation (uptr)ms->regs[1]+0x0, where ms->regs[1] represents the virtual PB register for sfp.

In order to perform the branch, we return this address back to the interpreter loop, so that the value of the instruction pointer for the next iteration is set to the address returned by this chunk. This is a form of "trampolining." Note that the reason trampolining is necessary in this case is because the branch target comes from a register and could not be statically known.

8.2.7 Chunklets and Local Labels

As can be seen in Figure 8.3, constructed PB chunks make use of C labels and associated `goto` statements. One reason for this is to support "chunklets", where each internal label in a chunk function corresponds to a chunklet, and a chunklet corresponds to a basic block in the original PB. A sub-index then allows all the converted basic blocks in a PB function to exist in the same PBChunk, leading to a more efficient C representation. The chunk function must be passed the `sub_index` to determine which basic block to execute.

Another extremely important reason for the local labels is to enable localized branching within a chunk. In PB, a branch to a local label within a function will be translated to a C-level `goto`, with the target being an equivalent C-level label. For example, the instruction at address 8 in Figure 8.2, `(bfalse (label 10 (imm 8)))`, is translated into the statement `if (!flag) goto label_10` in Figure 8.3. This avoids the overhead involved with returning to the interpreter loop and re-entering the chunk function at a different entry point.

8.2.8 Chunk Registration

Pointers to each `chunk` function are placed in an array of function pointers within each generated C file. Each C file exports a registration function that copies the chunk

```
    0: 0000044b          (eq %ac0 (imm #x0))
    4: 000008cd          (bfalse (label l0 (imm 8)))
    8: 00260402          (mov-16 %ac0 (imm #x26))
   12: 000001d3          (b* %sfp (imm #x0))

.l0:
   16: 00044429          (lsl %ac0 %ac0 (imm #x4))
   20: fff92517          (add %xp %ap (imm #xfff9))
   24: 00042230          (add %ap %ap %ac0 #:signal #t)
   28: 0000bccf          (btrue (label l7 (imm 188)))
   32: 00680fb1          (ld-int64 %r15 %tc (imm #x68))
   36: 00020f56          (bl %r15 %ap)
   40: 0000b0cf          (btrue (label l7 (imm 176)))

.l1:
   ...
```

Figure 8.2: Portion of PB instructions for "list"

pointers within the table into a global array. When the PB interpreter is initialized, it calls an external function that calls all the individual chunk registration functions in order to initialize the global chunk array.

```
static uptr chunk_722(MACHINE_STATE ms, uptr ip, int sub_index) {
  int flag;
  switch (sub_index) {
    case 0: ip -= 0x0; goto label_0;
    case 1: ip -= 0x10; goto label_10;
    case 2: ip -= 0x2C; goto label_2C;
    case 3: ip -= 0x48; goto label_48;
    case 4: ip -= 0x6C; goto label_6C;
    case 5: ip -= 0x90; goto label_90;
    case 6: default: ip -= 0xDC; goto label_DC;
  }
label_0:
/* 0x0 */   do_pb_cmp_op_pb_eq_pb_immediate(4, 0); /* r4 <- 0x0 */
/* 0x4 */   if (!flag) goto label_10; /*
↪  pb_b_op_pb_fals_pb_immediate: 0x8 */
/* 0x8 */   do_pb_mov16_pb_zero_bits_pb_shift0(4, 38); /* r4 <- 0x26
↪  */
/* 0xC */   return get_pb_bs_op_pb_immediate_addr(1, 0); /* r1 + 0x0
↪  */
label_10:
/* 0x10 */   do_pb_bin_op_pb_no_signal_pb_lsl_pb_immediate(4, 4, 4);
↪  /* r4 <- r4, 0x4 */
/* 0x14 */   do_pb_bin_op_pb_no_signal_pb_add_pb_immediate(5, 2, -7);
↪  /* r5 <- r2, 0x-7 */
/* 0x18 */   do_pb_bin_op_pb_signal_pb_add_pb_register(2, 2, 4); /*
↪  r2 <- r2, r4 */
/* 0x1C */   if (flag) goto label_DC; /*
↪  pb_b_op_pb_true_pb_immediate: 0xBC */
/* 0x20 */   do_pb_ld_op_pb_int64_pb_immediate(15, 0, 104); /* r15 <-
↪  r0, 0x68 */
/* 0x24 */   do_pb_cmp_op_pb_bl_pb_register(15, 2);  /* r15 <- r2 */
/* 0x28 */   if (flag) goto label_DC; /*
↪  pb_b_op_pb_true_pb_immediate: 0xB0 */
...
```

Figure 8.3: PB chunk representing “list”

```
typedef struct machine_state {
  ptr machine_regs[pb_reg_count];
  double machine_fpregs[pb_fpreg_count];
  ptr machine_call_arena[pb_call_arena_size];
} machine_state;
```

Figure 8.4: PB interpreter state structure

Chapter 9

WASM-PBCHUNK IMPLEMENTATION

9.1 Rationale

As has already been described, Chez Scheme's backend interface has proven difficult to adapt to Wasm. Semantically speaking, Wasm is rather different than other instruction set architectures that are already supported, and thus the interface is not readily applicable – especially where control flow is concerned.

Guile's Hoot project is taking the approach of building out support for WebAssembly more directly, and the work is *highly* involved, given specific features of Scheme that complicate compilation (continuations and garbage collected objects being the primary examples) [51]. Further, we wish to avoid re-implementing parts of the compiler in a Wasm-specific context, especially with respect to compiling features such as tail calls and continuations.

9.1.1 A level below the backend: PBChunk Inspiration

One alternative is to go a level *below* Chez Scheme's backend, and instead work with the output of Chez Scheme directly. Taking significant inspiration from the PBChunk approach outlined in the previous chapter, we might create a system that operates on compiled PB, and which splits off fragments of PB into compiled Wasm chunks.

The presence of the PB interpreter allows support to be developed gradually. Any constructs that are difficult to translate from PB into Wasm need not be included in chunks, and can simply be executed in the interpreter. We can circumvent the problem of control flow almost entirely by using the interpreter as a trampoline for branching, including only basic blocks in chunks.

A major benefit of this approach is the breadth of feature support it enables; *all* of Racket's features and its runtime – including garbage collection and delimited continuations – are already supported by PB, so by porting PB to Wasm, we essentially have full support under Wasm.

The problem then becomes achieving reasonable performance, and this is where a PBChunk-like system comes into play.

9.2 System Overview

At a high-level, we developed a version of PBChunk for Wasm, named Wasm-PBChunk for the time being. The system takes as input a boot file for a Scheme program containing Scheme code compiled to PB, and outputs extracted Wasm chunks. Since the PB interpreter is written in C, the PB interpreter, along with Chez Scheme's runtime, can be compiled into Wasm. We can then link the Emscripten-produced Wasm with our generated Wasm chunks, to obtain a Wasm-compatible version of Chez Scheme that can run our specific program.

9.3 Extracting Chunks from a Boot File

The procedure for extracting chunks from an existing boot file under Wasm-PBChunk is almost identical to the original PBChunk; we must recursively search through all objects in the FASL file, including those that are directly listed as code, and those that may contain references to code objects.

```
(func $chunk_n (export "chunk_{n}")
  (param $ms i32)
  (param $ip i32)
  (result i32)
  (local $flag i32)
  ;; code ...
)
```

Figure 9.1: Wasm chunk function stub

9.4 Chunk Selection

Pseudocode for the chunk selection algorithm will be given. The chunk selection algorithm is highly similar to that of the original PBChunk, but adapted for chunking only basic blocks and those instructions that are supported under Wasm.

Given a start index, chunk selection attempts to select some range $[start_i, end_i]$ that can be compiled into a single Wasm chunk. The chunk selection procedure shown in Algorithm 1 demonstrates high-level pseudo-code for selecting a chunk. The cases for processing instructions are outlined in Algorithm 2. We notably skip over rp-headers when creating chunks, and end a chunk at any label or branch instruction.

Note that the described procedure should run in $O(n)$ time, where n is the number of instruction words in a code object. At each step, we are processing a single instruction, and we process each instruction exactly once. If a chunk ends before the end of a code object, chunk selection is simply restarted at the next instruction.

9.5 Structure of Chunks

The structure for Wasm chunks is quite similar to the C chunks in the original PBChunk. Referring to Figure 9.1, we see a chunk stub function. The parame-

```
Input: $i$: start search index
Input: $headers$: list of headers
Input: $labels$: list of labels
Input: $len$: length of code object
$chunk \leftarrow null$;
$start_i \leftarrow null$;
while $i < len$ and $chunk = null$ do
    if isHeaderStart(i, first(headers)) then
        if $start_i \neq null$ then
            $chunk \leftarrow (start_i, i)$;
        else
            $i \leftarrow i +$ size(first(headers));
            $headers \leftarrow$ rest(headers);
            $relocs \leftarrow$ advanceRelocsPast(relocs, i);
        end
    else if isLabel(i, first(labels)) then
        if $start_i \neq null$ then
            $chunk \leftarrow (start_i, i)$;
        else
            $labels \leftarrow$ rest(labels);
        end
    else if $i \geq$ offset(first(relocs)) then
        signal error;
    else
        Handle Instruction, see instruction handling cases
    end
end
if $chunk = null$ then
    if $start_i \neq null$ then
        $chunk \leftarrow (start_i, i)$
    else
        $chunk \leftarrow (i, i)$
    end
end
```

Algorithm 1: Chunk Selection Algorithm

```
$instr \leftarrow instrs(i)$;
if $Unsupported(instr)$ then
    if $start_i \neq null$ then
        $chunk \leftarrow (start_i, i)$
    else
        $i \leftarrow i + size_{instr}$
    end
else if $isBranch(instr)$ then
    $end_i \leftarrow i + instr_{size}$;
    if $start_i \neq null$ then
        $chunk \leftarrow (start_i, end_i)$
    else
        $chunk \leftarrow (i, end_i)$
    end
else if $isLiteral(instr)$ then
    if $start_i = null$ then
        $start_i \leftarrow i$
    end
    if $i \neq offset(first(relocs))$ then
        signal error (relocation must follow literal);
    end
    $relocs \leftarrow rest$(relocs);
    $i \leftarrow i + size_{data}$
else
    if $start_i = null$ then
        $start_i \leftarrow i$;
    end
    $i \leftarrow i + instr_{size}$;
end
```

Algorithm 2: Instruction Handling Cases

ters `$ms` and `$ip`, which represent the machine state and current instruction pointer value respectively, are simply encoded as `i32` types; Wasm only supports a 32-bit address space. Similarly, a chunk's return value, which represents the next address for the instruction pointer, is an `i32`. Note that chunks are explicitly exported from the Wasm module. There is no sub-index included for simplicity; a chunk is comprised of a single basic block at most.

A `$flag` local is declared for cases in which we are compiling instructions such as comparisons that set the interpreter's flag.

9.6 Chunk Contents

Each chunk is comprised of the Wasm implementations for the instructions in said chunk. The core computations in many PB instructions map to Wasm fairly directly. For example, Table 9.1 shows the Wasm equivalents for binary operations in 64-bit PB. Similarly, PB's various `mov` operations (many of which include type casts) have direct Wasm equivalents as well, shown in Table 9.2.

That being said, significant boilerplate needs to be generated in order to actually modify the state of the PB interpreter from within a Wasm chunk.

For an example, consider the PB instructions listed in Figure 9.2. The instructions comprise a small basic block, the purpose of which is to check that the argument count in `ac0` is equal to 1, branching to an error label further along in the function if not.

The Wasm chunk corresponding to this basic block is listed in Figure 9.3. To briefly explain the contents: Lines 6-10 declare the `$flag` variable and temporary locals.

Lines 12-29 implement the `eq` instruction seen at address 0 in Figure 9.2:

- Lines 12-17 are an address computation; they calculate the offset into `$ms` for `ac0`, the 5th register. Note that in this configuration, each register is 64-bits. Emscripten places the register array (that corresponds to the first field of the machine state structure) at the lowest address within the memory allocated for the structure.
- Lines 18-22 perform a load from the virtual register `ac0`, and store the value in a temporary. Lines 23-25 load the constant `0x1`, and sign-extend the value to 64-bits, again storing in a temporary.
- Finally, lines 26-29 compare the value loaded from `ac0` to the constant value stored in a temporary, using the `i64.eq` instruction, placing the value into the `$flag` local.

Lines 31-37 implement the `bfalse` instruction. It is here that we can see our trampoline-based branching strategy come into play:

- The `i32.eq` instruction on line 31 checks for equality with 0, given that the branch is meant to be executed if the flag is set to *false*.
- The `then` portion of the `if` statement on line 32 represents the case in which the test succeeds. In this case, we return `$ip + 484`, the branch target address, as an `i32`.
- The `else` portion of the `if` represents the case in which we do *not* branch. In this case, we return the address of the next instruction. Since this chunk is two instructions in length, and PB instructions are 4 bytes in width, we must move the instruction pointer forward 8 bytes from its value on entry to this chunk. Thus, we return the address `$ip + 8`.

```
0:        0001044b          (eq %ac0 (imm #x1))
4:        0001dccd          (bfalse (label l11 (imm 476)))
```

Figure 9.2: Simple PB basic block. The first two instructions of a compiled Scheme factorial function

pb64	Wasm Equivalent
`add`	`i64.add`
`sub`	`i64.sub`
`mul`	`i64.mul`
`div`	`i64.div`
`and`	`i64.and`
`or`	`i64.or`
`xor`	`i64.xor`
`lsl`	`i64.shl`
`lsr`	`i64.shr_u`
`asr`	`i64.shr_s`

Table 9.1: Binops in 64-bit PB and Wasm Equivalents

9.7 Trampolining for Branching

As can be seen in Figure 9.3, conditional branches in PB are compiled into wasm `if` statements that return an address in each case. This address may be a constant offset from `$ip`, or it may be loaded from a register. Unconditional branches are simply an address computation, followed by a `return`.

A `return` instruction with the next address after the end of a chunk is added as the last instruction in every chunk as a fallthrough, to simplify compilation. This can be seen on line 38 in Figure 9.3.

pb64	Wasm Equivalent
`mov-i-d`	`f64.convert_i64_s`
`mov-d-i`	`i64.trunc_f64_s`
`mov-i-bits-d-bits`	`f64.reinterpret_i64`
`mov-d-bits-i-bits`	`i64.reinterpret_f64`
`mov-s-d`	`f64.promote_f32`

Table 9.2: Move operations in 64-bit PB and Wasm equivalents

```
(func $chunk_0 (export "chunk_0")
  (param $ms i32)
  (param $ip i32)
  (result i32)

    (local $flag i32)
    (local $g2 i64)
    (local $g3 i32)
    (local $g0 i64)
    (local $g1 i64)
    ;;0: r4 <- 0x1
    (i32.const 4)
    (i32.const 3)
    (i32.shl)
    (local.get $ms)
    (i32.add)
    (local.set $g3)
    (local.get $g3)
    (i64.load)
    (local.set $g2)
    (local.get $g2)
    (local.set $g1)
    (i32.const 1)
    (i64.extend_i32_s)
    (local.set $g0)
    (local.get $g1)
    (local.get $g0)
    (i64.eq)
    (local.set $flag)
    ;;4 b 476
    (i32.eq (local.get $flag) (i32.const 0))
    (if (then (return
                  (i32.add (local.get $ip)
                  (i32.const 484))))
        (else (return
                  (i32.add (local.get $ip)
                  (i32.const 8)))))
    (return (i32.add
              (local.get $ip)
              (i32.const 8)))
)
```

Figure 9.3: Wasm chunk implementing small PB block

9.7.1 When is trampolining necessary?

Local branching, or branching to an address within the confines of a given code object, can certainly be implemented in Wasm without trampolining. This is accomplished in the C version of PBChunk by making use of C labels and `goto` statements. With Wasm's structured control flow restriction, an algorithm such as Emscripten's Relooper [54] could be employed to appropriately nest basic blocks in order to emulate PB's branching semantics *within a function*. This approach will be discussed more as a piece of future work, but it was not implemented for this thesis due to time constraints.

For non-local branching – branching outside the confines of a given function – trampolining is likely to be necessary, short of a radical change in compilation approach. Most non-local branching will make use of a branch or branch indirect with a target address stored in a register. The value itself may be loaded from memory. This target address cannot be statically compiled with the current approach, since the address to branch to simply will not be known until runtime.

Wasm's only mechanism for inter-procedural control flow is a `call`. Assuming we could compile entire PB functions into Wasm, it might be possible to transform sequences of PB instructions that represent calls into native Wasm `call` operations in some cases. However, this was an avenue that was not explored and was deemed to be beyond the scope of the project.

9.8 Chunk Registration and Linking

Wasm-PBChunk performs chunk registration in a similar manner to the original, but with some slight differences.

```
(table $chunks 26 funcref)
(elem (i32.const 0) $chunk_0)
(elem (i32.const 1) $chunk_1)
(elem (i32.const 2) $chunk_2)
(elem (i32.const 3) $chunk_3)
(elem (i32.const 4) $chunk_4)
(elem (i32.const 5) $chunk_5)
...
```

Figure 9.4: A fragment of a generated table of chunk functions within Wasm-PBChunk module

```
(func $wasm_pbchunk_register (export "wasm_pbchunk_register")
    (table.set 0 (i32.const 0) (ref.func $chunk_0))
    (table.set 0 (i32.const 1) (ref.func $chunk_1))
    (table.set 0 (i32.const 2) (ref.func $chunk_2))
    (table.set 0 (i32.const 3) (ref.func $chunk_3))
    (table.set 0 (i32.const 4) (ref.func $chunk_4))
    (table.set 0 (i32.const 5) (ref.func $chunk_5))
 ...
 )
```

Figure 9.5: A fragment of the Wasm PBChunk registration function

Once a series of chunk functions has been generated, it must be exported such that the PB interpreter, compiled with Emscripten, can access it.

Once generated, each chunk is placed in a Wasm table, as seen in Figure 9.4. Due to some complications with static initialization, a Wasm registration function, `$wasm_pbchunk_register`, is used to manually set each slot in the table (seen in Figure 9.5)

Exporting the table itself from Wasm such that it was accessible within the Emscripten compiled PB interpreter proved tricky. Conceptually speaking, a Wasm table is a statically declared array of function pointers. When linking Wasm with Emscripten compiled C code, however, Wasm tables do not directly unify with `extern` function pointer arrays declared in C. This is unexpected behavior from Emscripten's

```
(func $wasm_do_jump (export "wasm_do_jump")
    (param $idx i32) (param $ms i32) (param $ip i64)
    (result i64)
  (i64.extend_i32_u
      (call_indirect (type $_chunksig)
          (local.get $ms)
          (i32.wrap_i64 (local.get $ip))
          (local.get $idx))))
```

Figure 9.6: Wrapper for performing a call from Emscripten compiled C into a generated Wasm chunk

linker, but it is understandable, given that the linker was created to link Emscripten-compiled code together and support may be more limited for hand-written Wasm.

To get around this behavior for now, our generated Wasm module exports another procedure, `$wasm_do_jump`, seen in Figure 9.6. Given a chunk index and the parameters for a chunk function, it indexes into the table and invokes the appropriate chunk with the given parameters.

9.9 Build System Integration

It is worth providing a brief overview of how generated Wasm chunks are linked with the existing Chez Scheme runtime (referred to as the "kernel") in order to provide chunking behavior.

The high level build steps are the following:

1. Create a build of Chez Scheme that targets PB. A configuration option for targeting PB already exists.

2. Compile a desired Scheme program to boot file form, using a PB build of Chez Scheme. This step is needed because Wasm-PBChunk is designed to operate on a boot file, like the original PBChunk.

3. Run the Wasm-PBChunk script to extract Wasm chunks from the program's boot file. Wasm-PBChunk places the generated chunks in a single separate Wasm module, and also outputs a modified boot file.

4. Rebuild the kernel of Chez Scheme – which is written in C and includes the PB interpreter – with **Emscripten**, to allow the kernel and interpreter to run in a Wasm environment. As part of this step, we *link* the Wasm module containing the generated Wasm chunks. We must also include the actual boot file code for the Scheme program, since the boot file has been modified to contain `pbchunk` instructions.

These high-level steps yield a build of Chez Scheme that is compiled to WebAssembly, and which contains, in its kernel, a version of the PB interpreter that can run a particular Scheme program under Wasm. This build of Scheme can be executed using any compatible Wasm runtime, though V8 has been used exclusively for testing thus far, both under Node.js and the Chrome Web Browser.

9.10 Current Limitations

Table 9.3 summarizes support for instructions that can currently be translated into Wasm. Notable exceptions are FFI operations (arena-in/arena-out, stack-call), and threading operations (lock, cas, etc.). Branch support is listed as partial, since local branch reconstruction is still not implemented in the places where it is possible and trampolining is instead used in those instances. As was previously mentioned, for non-

Instruction	Support (yes/no/partial)
mov*	**yes**
mov-16*	**yes**
binop (no signal)	**yes**
binop (signal)	**yes**
fp-binop	**yes**
un-op (not)	**no**
fp-un-op (sqrt)	**yes**
cmp*	**yes**
ld*	**yes**
st*	**yes**
b	*partial*
b-indirect	**yes**
btrue	*partial*
bfalse	*partial*
return	no
interp	no
call	no
adr	**yes**
inc*	no
lock	no
cas	no
fence	no
call-arena*	no
fp-call-arena*	no
stack-call	no

Table 9.3: Summary of instruction support for Wasm-PBChunk

local branching, trampolining is likely to be a continued necessity, so branch-indirect is completed.

Chapter 10

EVALUATION

The primary goal of this thesis was to take a step towards native Wasm compilation for Chez Scheme, and by extension, Racket. We must then validate that this approach is correct, and assess its performance. In particular, we wish to determine whether generating Wasm chunks from compiled PB actually produces any noticeable performance benefit.

10.1 Methods

We chose to utilize a subset of the Larceny Project R6RS benchmarks [30]. Larceny is primarily a compiler research project. It has enabled research into novel compiler optimizations through its TwoBit compiler [9], as well as research on garbage collection [10].

Given Larceny's goal to be a test bed for compiler optimization, Larceny features a large suite of benchmarks that are specifically designed to stress-test particular features of Scheme – for example, tail calls and first-class continuations.

We would certainly expect that our Wasm-PBChunk system will eliminate the interpreter overhead of instruction fetching and dispatch for sections of PB code. It is also hopefully the case that the Wasm translations of PB-compiled Scheme constructs – such as calls, numerical operations, and continuations – provide some performance benefits. The best way to test this is through diverse benchmarks that leverage a variety of Scheme features.

10.2 Benchmark Results

10.2.1 Benchmarking Suite

The “Gabriel” benchmarks, a subset of the full Larceny suite, were chosen for evaluation. The Gabriel benchmarks consist of 13 different Scheme programs that are designed to stress-test different aspects of code generation, including tail calling behavior, continuation capture, and use of primitives. Several benchmarks have both iterative, recursive, and continuation-based versions. They vary widely in size, from 10 lines up to over 100.

10.2.2 System Configuration

All benchmarks were performed on a MacBook Pro 2018 model, running OSX 11.4, with an 2.2 GHz 6-Core Intel Core i7 and 16GB of memory.

The Chez Scheme sub-repository of a Racket fork from March 10, 2023 was used for testing.

Node.js version 20.2 was used to execute the Wasm-compiled versions of Scheme. Thus, the chosen runtime for Wasm was V8, and includes the Turbofan JIT compiler for Wasm [43].

We tested three different configurations:

1. **Native**. Chez Scheme built to target x86_64, the native architecture of the host machine.

2. **NPB**. PB build of Chez Scheme, running on the host machine.

Benchmark	WPBChunk vs. WPB	WPB vs. NPB	WPBChunk vs. Native
browse	1.11	8.25	0.44
deriv	**1.48**	3.12	0.17
dderiv	**1.34**	4.32	0.20
destruc	**1.25**	7.00	0.33
diviter	**1.64**	5.43	0.36
divrec	**1.21**	4.29	0.21
puzzle	0.98	6.78	0.27
triangl	1.06	7.22	0.29
tak	**1.22**	5.56	0.28
takl	1.02	9.07	0.38
ntakl	0.96	9.02	0.36
cpstak	**1.23**	12.70	**0.73**
ctak	**1.31**	3.62	0.19

Table 10.1: Speedup numbers from the Gabriel benchmarks, for each described machine configuration. X vs. Y means "Speedup of X with respect to Y". See Appendix for full benchmark timings

3. **WPB**. PB build of Chez Scheme, compiled with Emscripten and running on Wasm

4. **WPBChunk**. PB Build of Chez Scheme running on Wasm, **with chunking enabled** for each specific benchmarked program.

Our primary goal was to compare the **WPB** and **WPBChunk** configurations, and it is this comparison that we will analyze in the most detail. The other configurations are primarily for reference.

10.3 Analysis

10.3.1 Wasm PB vs. Native PB

One surprising result is the *immense* difference in performance between PB running in a Wasm-compiled interpreter, and PB running in a native-compiled interpreter.

These speedup numbers are in the second column of Table 10.1, demonstrating a 3x speedup at minimum, and a 12x speedup at maximum. The likely reason for this is the presence of a Wasm JIT compiler – namely, V8's Turbofan. Turbofan performs background re-compilation of Wasm functions, meaning that the PB interpreter loop – one of the most-called functions in the program – is likely re-compiled many times into a heavily optimized state [43]. The JIT may be able to perform further optimizations on the PB interpreter code based on runtime profiling.

10.3.2 Wasm-PBChunk vs. Wasm-PB

Looking again to Table 10.1, we see moderate to significant speedups when comparing execution time for Wasm-PB and Wasm-PBChunk. In several other cases such as triangl, puzzle, takl, and ntakl, we see negligible speedup, or even a slight decrease (of no more than 4 percentage points).

The largest speedup is seen in the `diviter` benchmark, followed by the `deriv` benchmark. Analyzing the two benchmarks, we see a common pattern: slightly longer basic blocks that are amenable to chunking and are on *heavily utilized* code paths. The `diviter` benchmark divides a natural number n (represented as a list of length n) in two. It consists of a tight loop that skips over elements of the list, two at a time, incrementing a counter at each step. Two core sections of the compiled loop are split into chunks of approximately equal length, together totaling around 50% of the loop's instructions. Each of these chunks eliminates dispatch overhead for around a quarter of the instructions in the loop. This is before even considering possibility efficiency gains from the Wasm-PBChunk generated implementations of the instructions in each chunk.

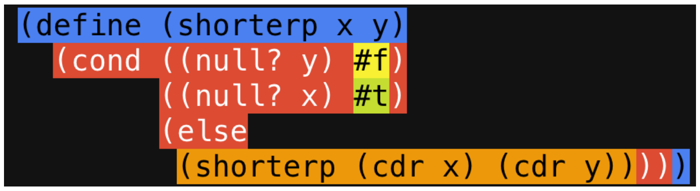
The `deriv` benchmark is a similar case. It is not iterative and makes non-tail calls, but two sections of the code implementing recursive cases that are likely to be commonly hit each include a chunk of 8 instructions in length.

Comparatively speaking, the `ntakl` benchmark had the worst performance relative to non-chunked PB running under Wasm (speedup of 0.96). The benchmark computes a recursive function that operates on lists, and uses a supporting procedure, `shorterp`, for comparing list lengths. `shorterp` is the most-called procedure in the benchmark, as is confirmed by profiling. Looking to Figure 10.1, we see that the most-run lines, highlighted in red, are the `null?` checks in the `cond` form. Each `null?` check compiles into two PB instructions that are seen in Figure 10.2: a comparison with a constant pointer mask `0x26` that encodes `'()`, and a branch to a block that performs a return.

Based on current chunking rules, these two empty list checks would each become separate chunks, meaning that the most run section of the program (hit approximately 470 million times, according to profiling) is split into chunks with only a single instruction. It is not surprising that the overhead from calling into these 1-instruction chunks is **not** offset by a potentially faster implementation for a single instruction. Instead, the overhead of calling into the chunk likely imposes a performance penalty that accumulates and accounts for the performance differential.

10.4 Takeaways

It is fairly intuitive to assume that smaller chunks are more likely to have a negligible or even negative performance impact due to the overhead of calling the chunk, and this is borne out in the `ntakl` benchmark in a fairly extreme case. The benchmark provides an empirical basis for setting a minimum chunk size, something that we resisted doing before analysis could be done.



```
(define (shorterp x y)
  (cond ((null? y) #f)
        ((null? x) #t)
        (else
         (shorterp (cdr x) (cdr y)))))
```

Figure 10.1: Hot code in ntakl [30] benchmark, profiled with Chez Scheme

Conversely, as chunk size increases, the overhead imposed by calling into a chunk seems to be amortized, suggesting that we should continue to increase chunk size as much as possible. This provides a fairly compelling case for implementing local branching within chunks, though more testing is certainly needed.

As a general rule it appears to be the case that sequences of instructions that are run more frequently are likely to be the best candidates for chunking, but only if they would be compiled to chunks that are sufficiently long.

10.5 Caveats

For this experiment, we performed chunking on entire programs, but notably *not* on supporting boot files (unlike the original PBChunk, which would only chunk supporting boot files). Thus, supporting procedures such as `+`, `car`, `vector-ref`, and the like did not benefit from any chunking, meaning that if test programs made heavy use of primitives, we may not have unlocked the maximum speedup. Similarly, if chunking were to cause a significant slowdown in primitive procedures, this also would not be borne out in our results. That being said, even in the worst case, we observed only a minor slowdown, so we would expect that chunking supporting boot files would be more likely to improve the speedup numbers.

```

; -------------------------------- begin
    (eq %r10 (imm #x26)) ; (null? y)
    (bfalse (label l0 (imm 8)))
; -------------------------------- end
; -------------------------------- begin
    (mov-16 %ac0 (imm #x6))
    (b* %sfp (imm #x0)) ; return #f
; -------------------------------- end
.l0:
; -------------------------------- begin
    (eq %r9 (imm #x26)) ; (null? x)
    (bfalse (label l1 (imm 8)))
; -------------------------------- end
; -------------------------------- begin
    (mov-16 %ac0 (imm #xe))
    (b* %sfp (imm #x0)) ; return #t
; -------------------------------- end
```

Figure 10.2: PB instructions for null? checks, with chunks commented

Chapter 11

FUTURE WORK

In this chapter, we provide some remarks on how our approach might be extended and improved. We also comment on the ways in which the Wasm compilation landscape may be changing in the near future – changes that may warrant a shift in compilation approach.

11.1 Rounding out instruction support

Referring back to Table 9.3, there are still a handful of core instructions that remain to be implemented. In particular, FFI and threaded instructions, which were deemed outside the scope of this project, could feasibly be implemented under Wasm-PBChunk. This has the potential to provide some level of speedup for PB programs that are threaded, and PB programs that make foreign calls.

There are currently some complications involved in compiling Racket to Wasm with full threading support, so supporting threaded instructions in PB would be predicated on the resolution of those issues.

11.2 Enabling Local Branching

One of the most appealing directions for future work is in enabling local branching for Wasm-PBChunk. This would bring Wasm-PBChunk much closer to feature parity with the original PBChunk. However, the mapping between PB and Wasm labels, as well as the strategy for compilation of PB's local branches, would have to be properly dealt with. A transformation such as Emscripten's Relooper [54] would need to be used in order to nest generated Wasm basic blocks in such a way that the original PB branching semantics could be emulated. This is likely to be doable, but is non-trivial.

If local branching were enabled, this could allow for a significant speedup. Chunks could be made much longer, and thus we would further amortize the overhead imposed by calling chunks and trampolining back to the interpreter loop.

11.3 Associated Optimizations

11.3.1 Lazy Register Writeback

If local branching were enabled, this would allow chunks to encompass large sections of PB functions. As a result, many virtual registers (stored in the PB interpreter's memory) would be written to and read again within the same chunk. Instead of writing and reading virtual registers from PB's interpeter state, we could instead create locals for every live register in a chunk and lazily "write back" the locals to the underlying interpreter state only when returning to the interpreter loop (at the end of a chunk). The use of Wasm locals for registers might provide a hint to the Wasm JIT to allocate said locals in *native* registers, providing a 1-1 mapping between PB and native registers when executing a procedure. However, experimentation would be needed to see if this would be the case.

11.4 Cross-procedure branching

As was previously mentioned, trampolining is likely to be necessary to implement cross-procedure (non-local) branching for the foreseeable future. This is because the addresses involved are often relocated, and may not be statically known.

However, assuming we are able to create chunks that encompass entire procedures, we may be able to eliminate the trampolining in some cases. Even if the procedure *address* cannot be known until load time, we could potentially determine whether a code object calls other procedures that may be compiled to Wasm already by inspecting the relocation information attached to a code object. If a code object does contain relocated procedure addresses, we might attempt to translate the relocation loading instructions, along with the following branch, into a Wasm call to a chunked function. There would be significant details to work out here, however.

11.5 Utilizing Upcoming Wasm Features

Numerous Wasm features are likely to change the landscape for compilers targeting Wasm. In particular, Wasm's upcoming GC proposal [49] may well land later this year. This would provide Wasm-level support for heap allocated, garbage collected structures, and the in-engine GC would provide inter-operation with the existing JavaScript heap.

Two tail call instructions – `return_call`, and `return_call_indirect` – are slated for addition to Wasm as well. These instructions would have the familiar semantics of a "call by jump," re-using the existing stack frame for a procedure. In effect, tail call support for Wasm would make it far easier to emulate Chez Scheme's execution model, which uses "call by jump" almost exclusively. This is also Guile's Wasm compilation approach – performing all calls via a tail-call, and maintaining a separate virtualized stack that can be sliced into pieces to implement continuation capture.

Utilizing these new native Wasm capabilities may in fact prove to be a more performant strategy, particularly for garbage collection, as the overhead of compiling the existing collector with Emscripten would be removed. Further, it would be a more "Wasm-native" approach and may perhaps allow for better inter-operation with JavaScript and other code running in the same environment.

Chapter 12

CONCLUSION

In this thesis, we present a new approach to the problem of Racket-to-Wasm compilation. First, we seek to describe the fundamental execution model behind Chez Scheme's generated code, an understanding of which is crucial to conceptualize the difficulties in porting the system to run under WebAssembly.

We then *reduce* the problem of compiling the full Racket language into Wasm to the problem of compiling an interpreter for the PB bytecode to Wasm and partially compiling sections of PB programs to Wasm chunks ahead of time.

This reduces the complexity of the problem dramatically, as the PB bytecode and interpreter, combined with the existing runtime system (both written in C) already provide for the full functionality of Chez Scheme, and by extension most of the Racket language. Further, we can already run this combined system under Wasm by compiling it with Emscripten, and so the challenge mostly comes from improving performance. The PB semantics are far easier to translate to WebAssembly, and the execution of any operations for which translation is *more* difficult can be left to the PB interpreter for the time being.

We evaluate our system using standard benchmarks, and show that it is indeed effective – both in its ability to run a variety of test programs, and in its performance, as compared to a baseline with no extra bytecode compilation.

We believe we have taken a unique approach in developing our solution to this problem; we are performing partial bytecode compilation with the *end goal* of building a PB compiler that targets Wasm and sits at a level below the existing Racket implementation. Our development process is incremental in that the presence of the interpreter allows us to develop feature support more gradually. For example, the presence of the interpreter allowed us to focus only on compiling basic blocks to WebAssembly, removing the complexity involved in handling calls and branching.

Our work represents a promising, fairly feature-complete effort to bring Racket support for Wasm, and we hope that our approach may be informative for similar projects that are hoping to add Wasm support.

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[2] Adobe Flash Player. `https://get.adobe.com/reader/flashplayer/about/index.html`.

[3] The History of Adobe Flash Player: From Multimedia to Malware, Dec. 2020. `https://www.intego.com/mac-security-blog/the-history-of-adobe-flash-player-from-multimedia-to-malware/`.

[4] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[5] asm.js. `http://asmjs.org/`.

[6] I. R. Atol. Proving correctness of a chez scheme compiler pass. *Master's Theses*, June 2021. `https://digitalcommons.calpoly.edu/theses/2336/`.

[7] 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, pages 22–34, New York, NY, USA, Aug. 2015. Association for Computing Machinery.

[8] W. D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 174–185, 1998.

[9] W. D. Clinger and L. T. Hansen. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, LFP '94, pages 128–139, New York, NY, USA, July 1994. Association for Computing Machinery.

[10] W. D. Clinger and L. T. Hansen. Generational garbage collection and the radioactive decay model. *ACM SIGPLAN Notices*, 32(5):97–108, May 1997.

[11] A. Donovan, R. Muth, B. Chen, and D. Sehr. Pnacl: Portable native client executables, 2011.

[12] R. K. Dybvig. The development of Chez Scheme. *ACM SIGPLAN Notices*, 41(9):1–12, Sept. 2006.

[13] R. K. Dybvig, D. Eby, and C. Bruggeman. Don't stop the bibop: Flexible and efficient storage management for dynamically typed languages. Technical report, Technical Report 400, Indiana University Computer Science Dept, 1994.

[14] ECMA-262. `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`.

[15] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4):365–378, July 2004.

[16] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to design programs: an introduction to programming and computing.* MIT Press, 2018.

[17] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. The Racket Manifesto. In T. Ball,

R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[18] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, Mar. 2002.

[19] M. Flatt. Binding as sets of scopes. *ACM SIGPLAN Notices*, 51(1):705–717, Jan. 2016.

[20] 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). *Proceedings of the ACM on Programming Languages*, 3(ICFP):78:1–78:15, July 2019.

[21] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. *ACM SIGPLAN Notices*, 44(6):465–478, June 2009.

[22] GNU's programming and extension language — GNU Guile. `https://www.gnu.org/software/guile/`.

[23] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on*

Programming Language Design and Implementation, pages 185–200, Barcelona Spain, June 2017. ACM.

[24] C. Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.

[25] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. *ACM SIGPLAN Notices*, 25(6):66–77, June 1990.

[26] spritely / Guile Hoot · GitLab. `https://gitlab.com/spritely/guile-hoot`.

[27] JavaScriptCore. `https://docs.developer.apple.com/documentation/javascriptcore`.

[28] 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*, pages 343–350, 2013.

[29] S. Klabnik. Is WebAssembly the return of Java Applets & Flash?, July 2018.

[30] The Larceny Project – R6RS Benchmarks. `http://www.larcenists.org/benchmarksAboutR6.html`.

[31] A. Lisitsa and A. P. Nemytykh. A note on specialization of interpreters. In *CSR*, volume 4649, pages 237–248. Springer, 2007.

[32] G. Matejka. Rasm: Compiling Racket to WebAssembly. *Master's Theses*, June 2022.

[33] Chez Scheme PB Backend. `https://github.com/racket/ChezScheme/blob/master/s/pb.ss`.

[34] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 291–300, New York, NY, USA, May 1998. Association for Computing Machinery.

[35] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, Mar. 1999.

[36] C. Sabharwal. Java, java, java. *IEEE Potentials*, 17(3):33–37, 1998.

[37] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, Jan. 2008.

[38] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. The Revised6 Report on the Algorithmic Language Scheme, Sept. 2007.

[39] SpiderMonkey — Firefox Source Docs documentation. `https://firefox-source-docs.mozilla.org/js/index.html`.

[40] G. J. Sussman and G. L. Steele. Scheme: A Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, Dec. 1998.

[41] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller. Static and Dynamic Program Compilation by Interpreter Specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, Sept. 2000.

[42] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, Jan. 2008.

[43] WebAssembly compilation pipeline · V8.
https://v8.dev/docs/wasm-compilation-pipeline.

[44] Standardizing WASI: A system interface to run WebAssembly outside the web – Mozilla Hacks - the Web developer blog.
https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface.

[45] WASI. https://wasi.dev/.

[46] Wasmer - The Universal WebAssembly Runtime. https://wasmer.io/.

[47] Wasmtime. https://wasmtime.dev/.

[48] WebAssembly. https://webassembly.org/.

[49] GC Extension, Mar. 2023. https://github.com/WebAssembly/gc/blob/main/proposals/gc/Overview.md.

[50] Tail Call Extension, June 2022. https://github.com/WebAssembly/tail-call/blob/main/proposals/tail-call/Overview.md.

[51] A. Wingo. a world to win: webassembly for the rest of us – wingolog, Mar. 2023. https://wingolog.org/archives/2023/03/20/a-world-to-win-webassembly-for-the-rest-of-us.

[52] V. Yadav. RacketScript, 2016.

[53] 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. *Communications of the ACM*, 53(1):91–99, Jan. 2010.

[54] A. Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *OOPSLA '11*, pages 301–312, Portland, Oregon, Oct. 2011. ACM.

APPENDICES

Appendix A

FULL BENCHMARK TIMINGS

Benchmark	WPBChunk	WPB	NPB	Native
browse	00:18:00	00:19.8	02:44.8	00:07.6
deriv	00:29.3	00:43.5	02:14.5	00:05.0
dderiv	00:44.4	00:59.4	04:15.2	00:08.9
destruc	00:11.7	00:14.7	01:44.6	00:04.4
diviter	00:28.1	00:45.7	04:10.3	00:09.5
divrec	00:42.1	00:51.3	03:39.5	00:08.9
puzzle	00:54.7	00:54.2	06:05.5	00:15.2
triangl	01:09.8	01:14.2	08:54.0	00:19.6
tak	00:31.6	00:39.0	03:37.2	00:08.8
takl	00:53.1	00:53.9	08:09.9	00:19.8
ntakl	00:55.4	00:53.4	07:57.9	00:19.8
cpstak	00:22.0	00:27.0	05:43.1	00:15.8
ctak	00:15.9	00:21.4	01:15.9	00:03.3

Table A.1: Full execution timings for Gabriel benchmarks. Format is mm:ss.0