Schism

A Self-Hosting Scheme to WebAssembly Compiler

ERIC HOLK, Google, Inc., USA

Schism is a small, self-hosting compiler from a subset of R6RS Scheme to WebAssembly, a new portable low-level binary format primarily targeting Web applications. The compiler was under one thousand lines of code when it first became self-hosting, and has since grown to support additional Scheme features. While currently far from a complete Scheme, Schism supports basic control flow, most basic data types and first class procedures. Schism provides an example of a small implementation of a language targeting WebAssembly and demonstrates techniques that may be useful to other languages implementors. As a dynamically typed functional programming language, Scheme is markedly different than languages with good WebAssembly support, like C and C++, and thus shows that WebAssembly has achieved its goal of being able to support a variety of languages. Still, Schism would greatly benefit from new capabilities in WebAssembly such as a proper tail call instruction and garbage collection. Given Schism's small size, it is well-positioned to provide early implementation experience to the WebAssembly standardization process for these new features. In this paper we will discuss the design and implementation of Schism, including compromises made to enable a quick and small implementation, as well as plans for future development on Schism and influence on the WebAssembly standard.

ACM Reference Format:
Eric Holk. 2018. Schism: A Self-Hosting Scheme to WebAssembly Compiler. 1, 1 (September 2018), 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Schism[1] is a small, self-hosting compiler from a subset of R6RS Scheme to WebAssembly. At the point that Schism became self-hosting, it was under one thousand lines of Scheme code. Admittedly, this version lacked important features such as first class procedures, leading one Redditer to ask "what kind of Scheme is this? [12]" Since the initial release, Schism has continued to gain features and now includes first class procedures, rudimentary garbage collection and experimental support for WebAssembly's upcoming tail call instructions. The compiler is currently about 1500 lines of code, so it remains a compact example of a standalone language implementation targeting WebAssembly.

WebAssembly is designed to be a language-independent compilation target. Most of the work so far has been towards C and C++ support, along with significant enthusiasm from the Rust community. What these languages have in common is that they have very few requirements from their host environment. Indeed, all of three of these languages can run without an operating system. Unfortunately, the first release of WebAssembly presents challenges to languages that require features such as garbage collection, dynamic typing, and stack switching. There are proposals to add support for all of these features to WebAssembly, but these would benefit greatly from implementation experience. Languages such as C# and Go are working on WebAssembly implementations, but

[1]https://github.com/google/schism

Author's address: Eric Holk, Google, Inc. 1600 Amphitheatre Parkway, Mountain View, CA, 94043, USA, eholk@google.com.

XXXX-XXXX/2018/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

they are large, mature projects and may not be well-suited for experimenting as new WebAssembly features are under development.

Schism has a role to play here. Scheme is a language that is sufficiently different from C and C++ to present interesting implementation issues. It must support features such as dynamic typing, first class procedures, proper tail calls, and first class continuations. These features can be emulated with existing WebAssembly instructions, but often at a significant performance penalty. Because of Schism's small size, it is feasible to do implementations using both existing WebAssembly features and features that are under development. This leads to early feedback into the design process as to whether a new features is usable in language implementations, and whether the benefits of adding the feature outweigh the costs of additional complexity in the WebAssembly specification and host implementations.

When Schism first became self-hosting, it supported a very small subset of Scheme. It has since grown to support more features such as first class procedures and a limited form of garbage collection. Still more features remain to be implemented, including proper tail call handling, macros and first class continuations.

This paper discusses the design and implementation of Schism. We start with a brief tour of WebAssembly (Section 2). Next we will see the initial design decisions to bootstrap Schism quickly (Section 3), followed by design changes and features that have been added since achieving self hosting (Section 4). Finally, we will see how additional Scheme features might be added and ways that Schism's implementation experience can inform further developments in WebAssembly (Section 5).

2 A TOUR OF WEBASSEMBLY

WebAssembly is a portable, load time efficient binary format for web applications. For a thorough treatment of the design of WebAssembly, see [8]. We review the relevant aspects of WebAssembly here.

WebAssembly grew out of a desire for faster web apps built from lower level code. There have been several other efforts in this area, including ActiveX [5], Native Client [2, 18], and asm.js [9]. In many ways, WebAssembly grew directly out of the work on asm.js, and in fact reuses several components such as the Emscripten toolchain [19]. WebAssembly has already gained significant support across the industry and is already included in most major web browsers. Although web browsers have been the primary use case, WebAssembly was explicitly designed to support other *embeddings*, such as on the server or even on blockchains [15]. The current version of WebAssembly features a relatively simple architecture and features a formally specified semantics and sound type system.

Modules in WebAssembly are made up of code and items such as function imports and exports, function tables, and data initializers. The code is split into separate functions, each of which carry their own type signature. Functions each take some number of parameters, declare a number of local variables, and contain a sequence of instructions. Instructions for the most part operate on specific types and perform a single operation. For example, `i32.add` does an addition of two 32-bit integers. Other instructions, such as `get_local` and `set_local`, are polymorphic and their types are determined from other sources, such as the variable declarations in the case of `get_local` and `set_local`. The type system has only four types: `i32`, `i64`, `f32`, and `f64`, although more types will probably be added in the future. Pointers are represented using the `i32` type, although 64-bit memory address support may be added in the future. WebAssembly has structured control flow, so it has control instructions such as `if` and `loop`, but no general `goto`. Thus, it is not possible to make irreducible control flow graphs.

Modules are similar to libraries such as ELF .so files. They contain code, definitions and data, but cannot be executed without first being *instantiated*. A WebAssembly *instance* is thus similar to an operating system process. Instantiation involves loading a module, resolving any imported functions, and setting up the instance's memory. In the usual web context, imported functions may be implemented in JavaScript or by another WebAssembly instance. Other embeddings will have their own mechanism for resolving imports.

WebAssembly is specified as a stack machine. Instructions consume their arguments from the stack and leave their result on the stack. For example, the following instruction sequence adds two numbers.

```
(i32.const 5)
(i32.const 6)
i32.add
```

The standard text format for WebAssembly is an S-expression format. For easier readability, arguments can be folded into the instruction. Thus our previous example can also be written as

```
(i32.add (i32.const 5) (i32.const 6))
```

or even

```
(i32.const 5)
(i32.add (i32.const 6))
```

Most existing code generators largely ignore the stack machine nature of WebAssembly and instead use local variables as registers. Schism takes advantage of the stack in several cases.

Besides local variables and the stack, WebAssembly provides a linear memory as well. This is accessed using load and store instructions, and is simply a flat array of bytes. In the web embedding, the linear memory is backed by a JavaScript array buffer. Schism uses the linear memory for heap-allocated values. The embedder, or WebAssembly host, is responsible for bounds-checking accesses to the memory, in order to protect itself from rogue WebAssembly programs.

WebAssembly has structured control flow. Thus control is specified in terms of properly nested blocks, loops, if expressions, and other control instructions. While this can be challenging for producers coming from a less structured language, WebAssembly's structured control flow is a good fit for Scheme. For example, C to WebAssembly compilers such as Emscripten must be able to convert unstructured control flow into structured constructs such as loops and if statements, sometimes even requiring auxiliary variables to correctly implement the control flow. On the other hand, Scheme's basic control flow operators such as `if` map directly onto WebAssembly instructures.

3 BOOTSTRAPPING SCHISM

The initial goal for Schism was to develop a small, self-hosting compiler targeting WebAssembly. For each feature, we had to ask whether the effort to implement it would be less than the effort saved by using it on the way to self hosting. This led to some unexpected choices.

For example, the first self-hosting version did not support lambda. Lambda took a relatively high amount of effort to implement, but the most helpful uses for lambda in the compiler could usually be easily rewritten using a helper function. Instead of writing `(map compile-expr expr*)`, we simply did:

```
(define (compile-expr* expr*)
  (if (null? expr*)
    '()
    (cons (compile-expr (car expr*) (compile-expr* (cdr expr*))))))
```

Similarly, we were able to avoid a garbage collector by simply never freeing memory. Current WebAssembly implementations allow WebAssembly programs to use nearly 2GiB of memory, which was enough for a Schism to compile itself. Schism got surprisingly close to this limit because of an extremely inefficient representation of strings and quoted symbols (Sections 3.5, 3.6). As Schism has gained additional features, we have added a simple garbage collector, but even this is just enough to let the compiler work.

Schism is a standalone compiler. It uses nothing beyond a WebAssembly implementation and a small runtime library. The runtime is currently implemented in JavaScript, but it would be easy enough to port to other hosts, such as the WebAssembly reference interpreter [17]. Schism's output is a binary WebAssembly module that it generates directly without using external tools like Emscripten or the WebAssembly Binary Toolkit, WABT [16].

From the beginning, Schism was written in Scheme. We used an existing R6RS Scheme implementation (in our case, Chez Scheme) to develop the compiler. Once the compiler was complete enough to compile itself, we switched to using previous snapshots of Schism to develop improved versions of Schism. The first version to self-host can be seen on the Github repository under the Git tag `self-host`.[2]

The compiler is structured as a multi-pass compiler inspired by the NanoPass framework [10]. Each pass takes a higher level form and translates it to a lower level version until finally we have a form that can be directly converted to WebAssembly. Many of the passes are done as local, top-down transformations but some require an analysis step. Analysis passes produce some data structure that can be used by subsequent passes. We will see examples of this transformation process in the next few sections as we discuss how various features were implemented. As much as possible, the goal was to have at all times a complete compiler that is able to translate some subset of the input language into a correctly functioning WebAssembly module. Expanding the input language subset typically involves adding new passes to the compiler or expanding existing passes to handle more cases. Because features were added generally in order of increasing complexity, we generally added passes from back to front. As the backend of the compiler is completed, additional language can often be added simply by elaborating them into forms the compiler backend already understands.

3.1 First Steps

To first make progress, we started working on a compiler that could correctly compile the following program:

```
(library (trivial)
  (export return_5)
  (import (rnrs))
  (define (return_5) 5))
```

This is a simple R6RS library that exports a function that returns the value 5. The library, once compiled to WebAssembly, should similarly have have an exported function called `return_5` that when called by the host returns 5, or at least Schism's representation of 5.

The first step was to be able to generate an empty WebAssembly module. The simplest module is, shown as a list of bytes, (`#x00 #x61 #x73 #x6d #x01 #x00 #x00 #x00`), which is just a header consisting of the null-terminated string "ASM" and the version number, 0x1. The rest of a WebAssembly module consists of sections which describe function declarations, type signatures, imported and exported functions, as well as the actual code for the functions.

Since just generating a header is boring, the next step was to add an empy types section. To do this required the ability to generate LEB128-encoded numbers [7], which are a variable length number encoding used in the DWARF debugging information format and also adopted by WebAssembly. We do this using the following function:

```
(define (number->leb-u8-list n)
  (if (and (< n #x80) (> n (- #x80)))
    (list n)
    (cons (bitwise-ior #x80 (bitwise-bit-field n 0 7))
          (number->leb-u8-list (bitwise-arithmetic-shift-right n 7)))))
```

[2]https://github.com/google/schism/tree/self-host

```
<library>   := (library (name) (export <id> ...) (import <import spec> ...) <define> ...)
<define>    := (define (<id> <id> ...) <expr> ...)
            |  (%wasm-import <string> (<id> <id> ...))
<expr>      := () | <number> | #t | #f | <char> | <string> | '<list> | `<quasiquoted list>
            |  (if <pred> <expr> <expr>) | (let ((<id> <expr>) ...) <expr>)
            |  (begin <expr> ...) | (<intrinsic> <expr> ...) | (<expr> <expr> ...)
            | (or <expr> ...) | (and <expr> ...) | (not <expr>)
            | (cond (<expr> <expr) ... (else <expr>))
<pred>      := #t | #f | (eq? <expr> <expr>) | (< <expr> <expr>) | <expr>
<intrinsic> := %read-mem | %store-mem | %get-tag | %set-tag | %as-fixnum | bitwise-and
            | bitwise-not | bitwise-ior
            | bitwise-arithmetic-shift-left | bitwise-arithemtic-shift-right | eof-object
            | + | * | - | set-car! | set-cdr!
```

Fig. 1. The grammar supported by Schism when it first self-hosted.

The ability to encode numbers gave us almost all the functionality we needed to encode the rest of a WebAssembly module. The next few steps were to fill out the rest of a WebAssembly module. This included encoding types, a function section and finally an empty function body.

At this point, it was time to compile our function body. The bulk of the work is done by a function called, surprisingly enough, `compile-expr`. Since the compiler only supported numbers at this point, the function was pretty simple:

```
(define (compile-expr expr env)
  (cond
   ((number? expr) (list 'i32.const expr))))
```

This function simply translates numbers into WebAssembly constant instructions.

A little extra bookkeeping was needed to make sure this function was exported correctly, and we added a test runner to make it easy to add and run additional tests like this one.

Now that we have a minimally functional compiler, we can get a general idea of what features are needed for this compiler to self-host. These include:

- Multiple top-level functions
- Control flow, such as `if` and `cond`
- Predicates and boolean operators, such as `eq?`, `and`, `or`, `not`, `<`, etc.
- Variables
- Basic pair support, such as `cons`, `car` and `cdr`
- Quoted lists, symbols and numbers
- Basic string support (`symbol->string` and `string->list`)
- Character support, including `char->integer`
- A reader capable of handling basic S-expressions, symbols, numbers in base 10 and base 16, and comments.

It is certainly possible to build a self-hosting compiler without many of these features, but these provide reasonable ergonomics without being overly difficult to implement. Figure 1 shows the grammar that was supported by the first self-hosting version of Schism.

This feature list served as a roadmap for further development on the compiler. Sometimes, these features added further dependencies. For example, the reader required further improvements to string handling.

3.2 Types and Data Representation

Scheme is a dynamically typed language. On the other hand, WebAssembly is statically typed. This difference required some care in structuring the compiled output, although not too much beyond what is needed when compiling Scheme to native machine code.

Schism represents all Scheme objects as 32-bit integers. The three least significant bits contain a type tag and the remaining 29 bits are the value of the object. For example, numbers are tagged with 0 and the 29 most significant bits store the value of the number. This is a standard implementation technique for dynamically typed languages.

Functions require some more care, because functions in WebAssembly are statically typed. We take advantage of two simplifications:

- The only function definition form allowed is `(define (foo a_1 a_2 ...) ...)`,
- all values are represented as `i32`,
- and all functions are assumed to return exactly one value.

These simplifications mean that for a function `(define (foo a b c) ...)`, we can assign it the type `(i32 i32 i32) -> (i32)`. Schism assumes functions are used correctly, so we do not perform any checking at the call-site. The WebAssembly validator ensures functions are called correctly. Sometimes this leads to surprising and confusing error messages, because it causes the WebAssembly stack to not match up like it should.

3.3 Control Flow

Almost all of Scheme's control flow and logical operators can be built from `if`. WebAssembly provides several control flow mechanisms. Control flow in WebAssembly is generally built out of nested, labeled blocks. There are two kinds of blocks: normal blocks and loop blocks. Normal blocks are similar to a `begin` expression in Scheme, but with the option to break out early. WebAssembly's break instruction, `br`, takes a numeric immediate value which specifies the *break depth*. The break depth specifies which nested block to break out of. Consider the following examples.

```
(block
  (block
    (br 0))
  ;; br transfers control flow to here
  ...)
```

```
(block
  (block
    (br 1)))
;; br transfers control flow to here
```

Loop blocks are similar, except loop blocks define two labels. One label transfers control to the beginning of the loop, similar to a `continue` statement in languages like C. The other label is used to exit the loop early, similar to a `break` statement in C.

In addition to unconditional breaks, WebAssembly provides `br_if`. This instruction consumes the value on the top of the WebAssembly stack. If that value was non-zero, then `br_if` breaks out to the specified block depth. Otherwise, control continues with the following instruction.

While these control flow structures are used heavily by the LLVM-based C/C++ toolchain, WebAssembly also provides an `if` instruction that is roughly equivalent to Scheme's `if` expression. Thus, Schism compiles Scheme `if` expressions to WebAssembly `if` instructions in the straightforward way.

At this point, we must discuss the representation of conditions and booleans. WebAssembly's control flow mirrors C, where 0 is considered to be false and any nonzero value is true. On the other hand, in Scheme only `#f` is false, and any other value is true. One obvious choice, then, is to represent `#f` as the constant 0. Using our three-bit tagging representation, this would require `#f` to have the tag 0. This conflicts with a previous design choice, however, where fixnums are tagged with `0` to allow some arithmetic operations to avoid any untagging and tagging steps. Given that conditionals in Scheme also tend to be common, it would be worth measuring whether the performance is better when `#f` has the same representation as a WebAssembly false value.

Instead, Schism reserves the tag 1 for constants such as `#f`, `#t`, and `#<void>`. We assign the value 0 to `#f`, so `#f` is represented as the i32 constant 0x00000001. Using this representation, we can see how a simple `if` expression is translated to WebAssembly. Consider the following example.

```
(if (check-condition)
    (if-true)
    (if-false))
```

In WebAssembly, this expression would be translated roughly to the following.

```
(if (i32.ne (call check-condition) (i32.const 0x00000001))
    (call if-true)
    (call if-false))
```

This explicit comparison against `#f` can be avoided in many cases. Schism's internal grammar makes a distinction between expressions and expressions in predicate position (called preds from now on). For example, the grammar for `if` is:

```
(if <pred> <expr> <expr>)
```

The `<pred>` nonterminal then has productions for some primitive comparison operators. For example, we can recognize equality comparisons by having a production from `<pred>` to `(eq? <expr> <expr>)`. Thus an expression such as

```
(if (eq? 1 2)
    (if-true)
    (if-false))
```

is compiled to

```
(if (i32.eq (i32.const 0x00000008) (i32.const 0x00000010))
    (call if-true)
    (call if-false))
```

rather than the more general version shown below.

```
    (if (i32.ne
          (i32.eq (i32.const 0x00000008) (i32.const 0x00000010))
          (i32.const 0x00000001))
        (call if-true)
        (call if-false))
```

There are other control flow structures that Schism supports, but these are all implemented as macro expansions to `if` expressions (Section 3.7).

3.4 Cons, Car and Cdr

Scheme programs build many of their data structures out of pairs. In Schism, pairs are also used to build other data types. We will see in the subsequent sections how to implement strings and symbols by building on the functionality provided by pairs.

Pairs are different from the data types we have seen so far because they require allocation. For WebAssembly, this means we must also declare a memory. The memory declaration specifies minimum and maximum sizes of the memory in multiples of WebAssembly's 64KiB page size. Memories can either be defined internally to the module, or imported from outside. Internally-defined memories can also be exported, which allows host code to access

WebAssembly's memory and also allows other WebAssembly instances to import the same memory. Schism modules always import their memory. The reason for this is to simplify the way the runtime can coordinate memories between different Schism modules.

Initially, Schism did not make any attempt at garbage collection. Since the goal was to self-host as quickly and easily as possible, we instead chose to declare an extremely large memory and hope that the compiler can complete without running out of memory. This strategy was enough for Schism to self-host, but did not last long afterwards. Schism now has an ad hoc collector (Section 4.4) that runs between certain compiler passes. A proper garbage collector is planned, as the current strategy will not last indefinitely.

Allocation is accomplished by a simple bump allocator. The first two 32-bit words in memory are reserved as the *system pair*, which contains both the allocation pointer and a pointer to the symbol table. We will discuss Schism's handling of symbols in Section 3.6. These two pointers could also be represented as WebAssembly globals, but storing them in memory allows us to avoid any handling of globals at all in Schism. We assume that allocations always consist of an even number of words. In other words, allocations are always aligned to eight byte boundaries. This means to go from a tagged `ptr` to a memory address, all that is needed is to mask off the three low tag bits and avoid doing any additional shifting.

The `cons` procedure is implemented in terms of a new `%alloc` compiler instrinsic, as well as `set-car!` and `set-cdr!`. Below is its initial implementation.

```
(define (cons a d)
  (init-pair (%alloc (pair-tag) 2) a d))
(define (init-pair p a d)
  (set-car! p a)
  (set-cdr! p d)
  p))
```

Here, the `%alloc` function takes a tag to apply to the newly allocated pointer and the number of words to allocate. In this case, we allocate two words, one for the car and one for the cdr of the pair. At the time this was written, Schism did not support `let`. The usual workaround is the so-called "left-left-lambda" expansion, which rewrites `(let ((x e) b)` as `((lambda (x) b) e)`. Unfortunately, Schism also did not support `lambda` at this time. Instead, we use a new top-level function as an imitation `let`-binding to hold the result of the allocation so that we can assign both the car and cdr before returning.

Later in the compiler, `%alloc`, `set-car!` and `set-cdr!` are replaced by direct WebAssembly memory instructions. As an example, here is the expander for `set-car!`:

```
(cond
 ((eq? tag 'set-car!)
  (let ((p (compile-expr (cadr expr) env))
        (x (compile-expr (caddr expr) env)))
   `(i32.store (offset 0) (i32.and ,p (i32.const -8)) ,x)))
  ...)
```

The `car` and `cdr` procedures are similar, except that they translate to WebAssembly memory read instructions.

3.5 Strings

In the same way that pairs can be used to build lists, trees and other data structures, Schism uses pairs to implement some Scheme's other primitive data types. Note that this is not the highest performance implementation, but it was sufficient to get the compiler to self-host. This implementation should be replaced by a more efficient one in the future.

Strings in Schism are simply represented as a list of characters, but the first `ptr` in the list is tagged with the string tag rather than the pair tag. This representation makes the implementation of functions like `list->string` and `string->list` trivial, as they only needed to changed the tag on a list of characters. This representation also leads to some surprising handling of string literals:

```
(cond
  ((string? expr)
   (list 'call 'list->string (parse-expr (cons 'quote (string->list expr)))))
  ...)
```

In other words, any string literal is expanded into a quoted list of characters at compile time. This quoted list is then passed at run time to `list->string`. Written more compactly using quasiquote, this is essentially replacing a string literal `s` with:

```
`(list->string ,(string->list s))
```

It is almost as if strings are implemented in terms of themselves.

3.6 Symbols

Schism's support for symbols builds on the theme of making simple representations from simpler, existing structures. At first glance, it may seem that we could do a similar trick to what we did for strings and represent symbols as lists of characters with a symbol tag instead of a pair tag. Such a representation destroys a key aspect of symbols, which is that it must be possible to compare them with `eq?`. In other words, by testing for reference equality. The expression `(eq? (string->symbol "a") (string->symbol "a"))` needs to evaluate to `#t`. Unfortunately, this does not hold for strings:

```
> (eq? (list->string '(#\a)) (list->string '(#\a)))
#f
```

In order to facilitate proper comparison for symbols, Schism keeps a symbol table that is simply a list of all symbols in the program. The symbol table is reachable from the symbol table entry of the system pair. Each entry in the list is a string and a pointer to the rest of the list. If a program executed `(list ’a ’b ’c)` then the symbol table would be something like `("a" "b" "c")`. Symbols are then represented as a pointer to its entry in the list, but tagged with the symbol tag instead of the pair tag.

When converting a string to a symbol, Schism searches the symbol table for an entry matching the same string. If an entry is found, then this is returned as the symbol. Otherwise, a new entry is added.

A useful feature in writing compilers is `gensym`, which creates a unique symbol that is used nowhere else. These symbols are useful, for example, for naming temporary variables that are introduced during the course of compilation. Schism supports these by adding an entry to the symbol table where the name is `#f`. The code for searching for a symbol by string ignores these, so no gensym will ever match a symbol typed or generated programmatically. These symbols currently are just opaque tokens, but it is common in other Schemes to add a descriptive label. For example, `(gensym "t")` might generate `t.0`, then `t.1`, and so on. This behavior could be added to Schism by augmenting the symbol table with a field for a name on generated symbols.

3.7 Macros and Quasiquote

Scheme macros are extremely powerful and can simplify the developer's life in many ways. For example, many of Scheme's seemingly primitive logical operators and control structures such as `and`, `or` and `cond` can be implemented using macros. On the other hand, correctly implementing Scheme's macro system and especially macro hygeine is non-trivial. In order to strike a balance between making the Schism implementors' lives easier and not complicating the compiler more than is necessary, Schism supports several hard coded macros. By implementing operators such as `and`, `or` and `cond` as macros, we can use these in Schism's own code. These operators do not have to be supported by all passes of the compiler, however, because they are eliminated at macro expansion time. Macros are expanded even before parsing Scheme code into an abstract syntax tree, making it easy to add more forms as needed.

Quote and quasiquote are also supported for similar reasons and by a similar mechanism. For example, the quoted list `'(1 2 3)` expands into direct calls to `cons`:

```
(cons 1 (cons 2 (cons 3 '())))
```

Although it would be possible to write the compiler entirely with `cons`, being able to write patterns using quasiquote not only makes the code shorter but also easier to read. Thus overall it seems adding support for quasiquote saved more effort than was needed to implement it.

3.8 Recursion Limits

We have touched in several places so far on steps that were taken to avoid limits in either Schism or in WebAssembly. One particular case is in the use of recursion. Schism supports only recursion as its means of doing any kind of iteration. WebAssembly implementations, however, normally have a limited amount of space for the program stack. Furthermore, Schism does not yet properly handle tail calls, so even tail-recursive loops will eventually run out of stack space.

For the most part, this was not a problem. Schism's data structures do not get too deep, so a straightforward recursive traversal is generally not a problem. One case where we did run into trouble was in generating the final output. At first, Schism generated a list of bytes and then iterated over these bytes, writing each out to the output file. Before long, the size of Schism's generated programs grew too large and this ran into recursion limits. Instead, Schism now gathers the generated bytes into a tree instead of a list, simply by replacing calls to `append` on the bytes of two sub expressions with a call to `cons`.

3.9 Standard Library

Schism implements some of the R6RS standard libary. Currently this is done by prepending any compiled program with the code for the standard library. This library is implemented in terms of low level compiler intrinsics. For example, `cons`, `car` and `cdr` are implemented in terms of functions that directly allocate and manipulate memory. These intrinsics are then replaced by sequences of WebAssembly instructions late in the compiler. This style of implementation keeps the core compiler simpler and makes it easy to quickly add library features. In the future, Schism should support propery library loading so that the standard library could be compiled and linked separately from the programs that use it.

3.10 JavaScript Runtime

WebAssembly programs are meant to be embedded in a host environment. This is most often a web browser, though other environments such as node.js or even native environments are possible. A goal for Schism is to implement as much as possible in the language, without relying on an extensive runtime library. Still, interfacing with the outside world requires some glue code written in JavaScript. Schism's requirements are small, consisting of little more than reading a byte of input and writing a byte of output. By contrast, the Emscripten runtime library includes JavaScript implementations of most of the Linux system calls.

The runtime library also provides functions to compile and run Scheme programs in JavaScript. This functionality is packaged as a Schism engine. Below is an example of how to use the engine to run a compiled Scheme program.

```
async function runScheme(module_bytes, fn_name) {
  const engine = new Schism.Engine;
  const wasm = await engine.loadWasmModule(module_bytes);
  wasm.exports[fn_name]();
}
```

This code creates a new instance of the Schism engine, which encapsulates all the state needed to load and run Schism programs. Like many new Web technologies, the WebAssembly interface is based around JavaScript's asynchronous Promises, so this code makes heavy use of `async` and `await`. The `loadWasmModule` function imports a compiled Wasm module containing Scheme code into the engine instance, returning a reference to this Scheme module. The resulting reference provides an array of functions exported from Scheme. Looking up the function by name in this array returns a function that is then callable by JavaScript to execute the exported Scheme code. Schism's runtime library also provides simple operations for converting between Schism and JavaScript's representation of values.

4 SUBSEQUENT IMPROVEMENTS

Since successfully becoming a self-hosting compiler, Schism has gained several significant improvements.

4.1 Error Checking, Runtime Type Safety, and Debuggability

As is often the case, early Schism development assumed all of its input was correct. Unfortunately, completely correct inputs is a luxury that is rarely afforded. This is especially true when one of the compiler's inputs is itself an in-development compiler. While limited error handling was sufficient to achieve self-hosting, further progress was definitely aided by more defensive coding.

Most of this functionality consists of simple type checks as part of the various constructors and accessors. Once something goes wrong, it is useful to be able to see where things went wrong. Schism, and WebAssembly in general, do not yet have in-depth debugging features such as being able to inspect local variables, but most WebAssembly implementations are able to provide stack traces. Schism generates a Name Section in its output binaries which allows WebAssembly stack traces to have human-readable function names. Even this little bit of information makes it easier to diagnose problems in the compiler.

Finally, Schism also has basic support for `printf`-style debugging by including a limited version of `display`.

4.2 Local Variables

Although we saw in Section 3.4 that the lack of `let`-binding can be worked around, supporting proper local variables makes the language much more usable. One option is to first implement support for closures and then replace all variable bindings with immediately applied `lambda` expressions. This is sub-optimal for several reasons. First, Schism did not support `lambda` when `let`-binding support was added. Furthermore, having the ability for the compiler to create temporary variables would be useful to implement closures. Second, local variables should be inexpensive and not require a heap-allocated closure and an indirect function call.

Instead, we choose to map `let`-bound variables to WebAssembly locals. This approach avoids the two downfalls of a `lambda` expansion. In fact, most WebAssembly implementations will try to store local variables in registers.

Schism creates WebAssembly local variables for any `let`-bound variables and tracks a mapping from the Scheme variable to the WebAssembly variable. The right hand sides of the `let` expression are evaluated and the result is then assigned to the correct variable. Variable references are then replaced with `get_local` instructions.

The obvious way to assign Scheme variables to WebAssembly variables would be to create a unique WebAssembly variable for every Scheme binding. Instead, Schism uses something more like lexical address [1], which allows Schism to assign Scheme variables to WebAssembly variables by only keeping track of the number of bindings passed through while traversing the syntax tree. There is no need to consider sibling expressons with this strategy. As an example, consider this Scheme program.

```
(let ((a 1))
  (let ((b 2))
    (display b))
  (let ((c 3))
    (let ((d 4))
      (+ c d)))))
```

For this expression Schism would declare three variables because the deepest chain of bindings is of length three: a-c-d. The compiler would assign a to variable index 0, b and c to variable index 1, and d to variable index 2. Schism translates the above expression into WebAssembly resembling the following:

```
(block
  (i32.const 1)                           ;; evaluate RHS of a
  (set_local 0)                           ;; store value of a
  (i32.const 2)                           ;; evaluate RHS of b
  (set_local 1)                           ;; store value of b
  (get_local 1)                           ;; put argument to display on stack
  (call $display)
  (i32.const 3)                           ;; evaluate RHS of c
  (set_local 1)                           ;; store value of c, clobbering b
  (i32.const 4)                           ;; evaluate RHS of d
  (set_local 2)                           ;; store value of d
  (i32.add (get_local 1) (get_local 2)))    ;; add the top two values on the stack
```

Because WebAssembly implementations do their own register allocation, either approach to assigning Scheme variables to WebAssembly variables should be roughly equivalent in the end.

Sharing storage for variables still requires care in the compiler to implement it correctly. Consider the following example.

```
(let ((a 1)
      (b (let ((c 2))
           (+ c 1))))
  a)
```

This expression should evaluate to 1. Earlier versions of Schism instead produced code that returned 2 instead. Originally Schism evaluated the right hand side of a and then immediately assigned it to its variable slot. Then, in evaluating the right hand side for b, Schism had to choose a location for c. Because a is not visible at that point in the program, Schism assigned both a and c to WebAssembly variable 0. The value of c was assigned after a's binding was already fully evaluated, which is why the value 2 overwrote the value that had been stored in a.

The solution is to evaluate all of the right hand sides and store them in a temporary location, and then assign the values to their final location after all the right hand sides are fully evaluated. While this could be done with still more temporary variables, WebAssembly's stack gives a convenient temporary location. Thus, Schism generates code that first fully evaluates all the right hand sides of a `let` binding, leaving the results on the WebAssembly stack, and then assigns all of these values to local variables.

4.3 Lambda

First class procedures were a somewhat late addition to Schism. Although they were not necessary to make a self-hosting compiler, first class procedures have been useful since their introduction. For example, Schism contains many functions like this one:

```
(define (parse-body* body*)
  (if (null? body*)
    '()
    (cons (parse-expr (car body*)) (parse-body* (cdr body*)))))
```

With first class procedures, that function can be eliminated and calls to `parse-body*` can be replaced by `(map (lambda (body) (parse-body body)) body*)`. In the future this expression could be further reduced to `(map parse-body body*)`, but at the moment top level function in Schism are not first class.

First class procedures in Schism are implemented using standard techniques [3]. Schism eliminates `lambda` expressions fairly early in the compiler, instead translating them to more primitive forms. At first, Schism annotates all `lambda` expressions with a list of their free variables. Then the bodies of these expressions are lifted into a top-level function. The top level function gains a new argument representing the closure. Before executing the body, all free variables are unpacked from the closure and `let`-bound so that the body can execute as though it were running in its normal context. As an example, consider the following function.

```
(define (foo)
  (let ((a 5))
    (let ((add-a (lambda (b) (+ a b))))
      (add-a 6))))
```

After `lambda` elimination, this program would be translated into something like:

```
(define (lambda-0 closure b)
  (let ((a (read-closure closure 0))
    (+ a b))))
(define (foo)
  (let ((a 5))
    (let ((add-a (make-closure 0 a)))
      (call-closure add-a 6))))
```

Here, `make-closure` would allocate two words on the heap. One of these words is for the function index, and the other is to hold the captured value of `a`.

Unlike native code targets, WebAssembly does not allow direct access to function addresses. Instead, WebAssembly modules include a function table and a `call_indirect` instructions that allows dynamically calling into the function table. The `call_indirect` instructions works like the `call` instruction, except that it takes an additional index argument that specifies which entry of the function table to call. As with the `call` instruction, `call_indirect` has a type signature. At runtime, WebAssembly implementations must ensure that the signature matches the target function, as there is only one function table rather than one per signature. As with normal procedure calls, Schism constructs the signature based on how the function is used. Note that this means variable length argument lists are not currently supported.

As mentioned earlier in this section, Schism does not currently allow top-level functions to be passed by value. One easy way to support this in the future would be to detect references to these functions in value position and wrap them in a suitable `lambda` expression. Effectively, the compiler would perform the inverse-η transformation that we had to do manually above. Alternatively, instead of mapping top level functions to bare WebAssembly functions, these could instead be closures that are stored in global variables. In practice, some combination of the two approaches may be best. This remains a task for future work.

4.4 Garbage Collection

Schism does not yet have a complete garbage collector, but unfortunately the compiler itself generates enough garbage that some rudimentary collection is needed. The garbage collection that currently exists is implemented in JavaScript as a function that takes a list of roots and uses that to collect garbage. The collector does this by allocating a new JavaScript array buffer, doing a deep copy of all the roots into the new array buffer and then copying the new array buffer over the previous WebAssembly memory for the Schism module. Garbage collection does not happen on demand, but instead the compiler is split in two. The JavaScript runtime calls the first half of the compiler, which returns the state necessary to continue with the rest of the compilation process. This intermediate state then becomes the root for a collection, and is passed back into the second half of the compiler.

This basic collector is enough to continue making progress on Schism, but clearly a fully automatic collector will be needed before long.

5 FUTURE WORK

Although Schism supports several key features of the Scheme programming language, it is still an early project. We have already seen a number of features that need to be filled out. There are some missing Scheme features, such as the macro system, support for variable argument functions, proper tail call handling, and proper garbage collection. Additionally, it would be good to support some of Scheme's more uncommon features, such as `call/cc`.

These features can all be implemented in the current WebAssembly specification, although perhaps at a performance penalty. There are a number of proposals to WebAssembly, such as tail call instructions, garbage collection, and exception handling. All of these would simplify and improve Schism's ability to fully support the Scheme language. Having a proper tail call instruction would avoid the need for unpleasant full program transformations to support tail calls. WebAssembly garbage collection proposal includes a suite of richer types and checked casts between references of different types. Schism may be able to use these instead of its own tagging system for dynamic types, and by using fully garbage collected types Schism could avoid using WebAsembly's linear memory entirely. The current proposal for exception handling in WebAssembly is designed to be easily generalizable to one-shot delimited continuations, so this may also be able to cover some uses of `call/cc` in Scheme. Given Schism's small size, it would be easy to adopt these proposals in Schism and this should be considered as an option to gain early stage feedback on the usability of new WebAssembly features.

6 RELATED WORK

One of WebAssembly's goals is to enable a variety of languages to target the Web with high performance. It is encouraging to see that a number of languages are experimenting with WebAssembly. Besides C and C++, perhaps the language best supported on WebAssembly is Rust [14]. A particularly powerful feature of Rust is `wasm-bindgen`, which can automatically generate JavaScript and Rust interoperability code [4]. Other languages with experimental WebAssembly support include Haskell [6] and the .NET languages [13]. These languages require more from their runtime system than lower level languages like Rust, and thus have their own challenges. Future improvements to WebAssembly may greatly improve the support for these other languages.

Scheme 48 [11] has some similarity to Schism. Scheme 48 is a compiler built in layers with each lower layer being a more restricted version of Scheme. The lowest layer, Pre-Scheme, is designed to be easily translated to idiomatic and efficient C code. Schism's layers are more ad-hoc and being more careful about layering in the future would help manage the complexity as Schism grows to support more of Scheme.

7 CONCLUSION

Schism is a small, self-hosting compiler for WebAssembly. It is a from-scratch implementation, which makes it an ideal example of the experience of bringing up a new language for WebAssembly. One of WebAssembly's goals is to make it easy to support other programming languages on the Web as first class citizens. The author's hope is that Schism will be a demonstration of useful techniques for other language implementors targeting WebAssembly, as well as provide a vehicle for quickly experimenting with new WebAssembly features.

REFERENCES

[1] Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd editon ed.). MIT Press/McGraw-Hill, Cambridge.

[2] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. 2011. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 355–366. https://doi.org/10.1145/1993498.1993540

[3] Luca Cardelli. 1984. Compiling a Functional Language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 208–217. https://doi.org/10.1145/800055.802037

[4] Alex Chrichton. 2018. JavaScript to Rust and Back Again: A wasm-bindgen Tale. (4 April 2018). Retrieved 2018-07-12 from https://hacks.mozilla.org/2018/04/javascript-to-rust-and-back-again-a-wasm-bindgen-tale/

[5] Microsoft Corporation. 2017. ActiveX Controls. (15 Aug. 2017). Retrieved 2018-07-12 from https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa751968(v=vs.85)

[6] Dfinity Haskell Compiler [n. d.]. ([n. d.]). Retrieved 2018-07-12 from https://github.com/dfinity/dhc

[7] Free Standards Group. 2005. *DWARF Debugging Information Format, Version 3*.

[8] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[9] David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js. (18 Aug. 2014). Retrieved 2018-07-12 from http://asmjs.org/spec/latest/

[10] Andrew W. Keep and R. Kent Dybvig. 2013. A Nanopass Framework for Commercial Compiler Development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 343–350. https://doi.org/10.1145/2500365.2500618

[11] Richard A. Kelsey and Jonathan A. Rees. 1994. A tractable Scheme implementation. *LISP and Symbolic Computation* 7, 4 (01 Dec 1994), 315–335. https://doi.org/10.1007/BF01018614

[12] lngnmn. 2018. (28 Feb. 2018). Retrieved 2018-07-12 from https://www.reddit.com/r/programming/comments/81243z/schism_a_selfhosting_scheme_to_webassembly/dv040t7/

[13] Daniel Roth. 2018. Get started building .NET web apps that run in the browser with Blazor. (22 March 2018). Retrieved 2018-07-12 from https://blogs.msdn.microsoft.com/webdev/2018/03/22/get-started-building-net-web-apps-in-the-browser-with-blazor/

[14] Rust + WebAssembly Team [n. d.]. ([n. d.]). Retrieved 2018-07-12 from https://github.com/rustwasm/team

[15] Jason Teutsch and Christian Retwießner. 2017. A scalable verification solution for blockchains. (16 Nov. 2017). Retrieved 2018-07-12 from https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf

[16] WABT: The WebAssembly Binary Toolkit [n. d.]. ([n. d.]). Retrieved 2018-07-12 from https://github.com/webassembly/wabt

[17] WebAssembly Community Group. [n. d.]. WebAssembly Interpreter. ([n. d.]). Retrieved 2018-07-23 from https://github.com/WebAssembly/spec/tree/master/interpreter

[18] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Commun. ACM* 53, 1 (Jan. 2010), 91–99. https://doi.org/10.1145/1629175.1629203

[19] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/2048147.2048224