Internship report

Compilation of Scheme to WebAssembly

Implementation of a WebAssembly backend in the Bigloo compiler

Hubert Gruniaux

August 2024

Keywords: *wasm, compilation, scheme, bigloo, backend.*
Supervised by **Manuel Serrano** and **Erven Rohou**.
Carried out at **Inria de Sophia-Antipolis** in the **SPLITS** team.

Contents

I Introduction

I did an internship at INRIA Sophia Antipolis in the SPLITS team with the aim of adding a new backend to WebAssembly to the existing Bigloo Scheme compiler.

The code of this project can be found here: https://github.com/hgruniaux/bigloo-wasm.

I.1 Bigloo

Bigloo [1] is a compiler and interpreter for the Scheme language with its own sets of extensions: classes, C interface, modules, threads, etc. It has been freely distributed since the 90s. Today, it contains around 100,000 lines of Scheme code, and the runtime library contains around 70,000 lines of Scheme and 20,000 lines of C code (without the GC).

It is a mature compiler that has many optimizations for dynamic functional languages like Scheme. It is therefore a good target to compile ML and functional languages other than Scheme. For example, it is used as a backend for the Hop Javascript-to-Scheme compiler.

The compiler has actually multiple backends – one classic that generates C code directly from the AST and another one called SAW, which uses basic blocks as the intermediate representation and generates C and JVM bytecode [2], [3]. The WASM backend of this internship is based on this second SAW backend because it is more recent and has the most optimizations (like basic block versioning) and because its basic block structure makes it easier.

I.2 WebAssembly

WebAssembly, often shortened to **WASM**, is a new technology that allows high performance code execution inside web browsers. It is intended to provide a better alternative to `asm.js` and JavaScript as a target for native code (like C or Rust).

Since its introduction, WebAssembly has been implemented in all major web browsers. A fairly mature ecosystem is beginning to emerge. Thanks to the Emscripten project and LLVM, it is possible to compile C, C++ and Rust code (and part of their ecosystem) into WASM. It has also been used to port game engines such as Unreal Engine and Unity to the web.

WebAssembly takes the form of a typed bytecode which is executed by a stack-based virtual machine. The bytecode is statically typed and verified by the assembler and the virtual machine before execution. When required for ensuring safety, runtime checks are automatically injected by the implementations. Most of the provided instructions are based on real hardware instructions: arithmetic on 32/64-bits integers (types `i32` and `i64`) or on IEEE-754 32/64-bits floating points (types `f32` and `f64`). More complex operations like trigonometrics functions are not present in the specification. Programs have access to a linear memory, which can be seen as an array of bytes. This also includes the concept of local variables that act as pseudo-registers for temporary values inside functions.

Recently, the existence and maturity of certain extensions have increased interest in compiling dynamic and functional languages into WASM. For instance, it is now possible to guarantee optimization of terminal recursive calls and a garbage collector integrated into the language and functions as first-order values are now supported.

I.2.1 Special case of using WebAssembly outside of Web

Although the name suggests otherwise, WebAssembly can be used outside of a web browser. In particular, there are several implementations of the standalone WebAssembly virtual machine. These include Wasmer (https://wasmer.io/), Wasmtime (https://wasmtime.dev/), and the reference interpreter.

However, WebAssembly's world-isolating philosophy somewhat limits its use cases. By default, WASM code cannot communicate with the outside world. In particular, its memory is isolated. Currently, the only standardised way to interact with the operating system is through the WebAssembly System Interface (WASI) specification. This only offers a limited set of functions for manipulating the file system and sockets, and retrieving program arguments and environment variables.

There is no mechanism for dynamic linking or calling external code (without using JavaScript). It is also impossible to open an external program, map regions of memory, and so on. In other words, neither WebAssembly nor WASI offers sufficient functionality to perfectly implement the POSIX layer or the Win32 API. This makes complete interaction with the OS impossible.

For these reasons, and because of the lack of a standard WASM library, we have decided to support WASM only in a JavaScript environment (web browsers, NodeJS, Deno, etc.), as a first step.

I.3 Scheme

Scheme is a dynamically typed and functional language based on Lisp. There are several implementations of it: Bigloo (on which I worked on), Gambit, GNU Guile, Racket, etc.

In recent years, there have been some efforts to compile Scheme and its environment for WebAssembly. There are projects to implement a Scheme to WASM compiler, such as Google's Schism (now abandoned). Or to implement a backend on an existing compiler, such as Hoot (based on GNU Guile).

It is also possible to reuse the existing backends of current Scheme compilers like the C backend of Bigloo, and then compile the C code to WebAssembly. This last step is already well known with projects like Emscripten (a C/C++ compiler to WASM) [4]. This possibility was explored by Igalia [5]. However, Emscripten is intended to compile C-like code, so does not use the last proposals of WASM for dynamic and functional languages. This has two effects: the optimizer doesn't have access to certain high-level information, and the runtime must embed a complete implementation of a GC (which can be heavy).

The aim of this internship is to explore the possibility of directly generating WASM code in Bigloo using the latest specification proposals and perhaps compare our results with Guile Hoot, or with similar projects for other languages, such as Wasocaml [6] or the Glasgow Haskell Compiler (GHC). It will be also interesting to compare it to the Igalia project (compilation of the C output to WASM using Emscripten) [5].

II Representation of types

II.1 Common types

Because of the full polymorphism of Scheme, most values are boxed and allocated by the garbage collector. Generally, to represent such values and avoid too many allocations, we use techniques like pointer tagging (the technique used in the C backend by Bigloo) or NaN tagging. However, these techniques cannot be used in WASM. The GC reference type available in WASM is an opaque pointer, with no bit manipulation allowed. The GC proposal only supports two kinds of values: structures and arrays.

This happened to be sufficient because the WASM implementation may optimize it and choose an efficient representation. Second, Bigloo uses a custom dialect of Scheme that allows typing and the compiler uses these types to use unboxed representations of objects as much as possible [7].

To represent any boxed value, which corresponds to the Scheme type `obj`, we initially used the WASM type `anyref` (see Figure 1 for the hierarchy of GC types). However, we were faced with a problem in implementing reference equality efficiently (function `eq?` in Scheme) as the `ref.eq` instruction expects values of type `eqref`. A paper from Wasocaml [6] showed us that `eqref` can be used directly in source code (the GC proposal is not always clear) and is sufficient to represent any boxed type. Thus, we used `eqref` to represent the Scheme `obj` type.

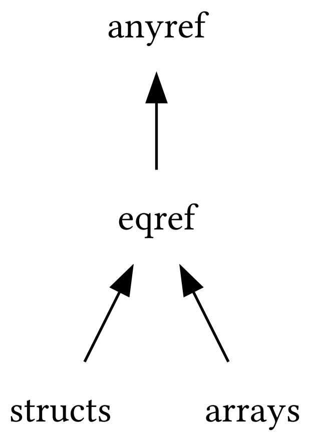

$$\begin{tikzcd}
 & \text{anyref} & \\
 & \text{eqref} \arrow[u] & \\
\text{structs} \arrow[ur] & & \text{arrays} \arrow[ul]
\end{tikzcd}$$

Figure 1: Hierarchy of GC types in WASM (arrow is subtyping)

For primitive types (integers, reals, etc.) that must be boxed, we introduce a structure with a single field of the primitive type. But because the type system of WASM is based on structural equivalence, two Scheme primitive types that are based on the same WASM primitive type (like `bint64` and `buint64` that are both based on the WASM `i64` type) will be considered the same. This is not desired, as `ref.is` will then be incorrect in regard to the Scheme semantics. To avoid that, we enclose these problematic types inside a WASM `rec` group. See Code 1 for an example of boxed Scheme types implemented in WASM.

```
;; A boxed 64-bit integer (signed or unsigned)
(rec (type $bint (struct (field i64))))
;; A boxed IEEE-754 64-bits floating-point
(rec (type $real (struct (field f64))))
;; A byte string (no encoding specified in Scheme, but generally UTF-8)
;; Strings are mutable, thus the 'mut' keyword
(rec (type $bstring (array (mut i8))))
;; Pairs which are just couples of Scheme obj (eqref in WASM)
(rec (type $pair (struct (field $car eqref) (field $cdr eqref))))
```

Code 1: Representation of some Scheme boxed types

Another interesting case is the implementation of special constants in Scheme. This includes the two booleans `#t` (true) and `#f` (false), `#unspecified`, etc. They are represented using the WASM `i31ref` type, which is a non-allocated reference that stores the data in its first 31-bits. This avoids GC allocations. However, in practice, this doesn't matter much, as special constants are created once at the start of the program and are accessed using globals after that. Still, using `i31ref` avoided some linker problems that duplicated their global definition and therefore created multiple instances of `#t` (so it was possible that true $\neq$ true).

It was not possible to use `i31ref` to represent Scheme integers, as Bigloo had no type for small integers. Most of the integer types of Bigloo expect at least 32-bits. However, `i31ref` could be used to implement boxed characters or boxed 8/16-bit integers, but this is not done in the current implementation.

II.2 Strings literals

The implementation of strings literals requires special attention. They are represented as globals of type `bstring` (GC array of bytes). To initialize the arrays, we need to store the actual string data. Usually, this is done using data segments, which are also supported in WASM. However, in the current version of the GC proposal, there are some limitations in the use of data segments to initialize arrays. The instruction `array.new_data` used for that is not considered constant and, therefore, cannot be used in globals initialization. So, we need to explicitly initialize them at program startup in the Bigloo auto-generated constant initialization functions. The general pattern is shown in Code 2.

```
;; Declaration in the module preamble.
(data $my-data-section "my string data")
;; Instruction 'array.new_data' not allowed in a global declaration.
(global $my-string-constant (ref null $bstring) (ref.null none))

;; In the constant initialization module function:
;; This is called at module import or at the start of the program.
(func $my-module-initialize-constants
  (; ... ;)
  (global.set $my-string-constant
    (array.new_data $bstring $my-data-section
      (i32.const 0 (; offset in data section ;))
      (i32.const 14 (; string length ;))))
  (; ... ;))
```

Code 2: Handling of strings

Requiring string literals to be initialized (which is not the case in the C and JVM backends) may cause some initialization problems. For example, if two modules have cyclic dependencies, what module should we initialize first? If we choose an incorrect order, we may call a function that uses uninitialized data. This use case appears in the runtime library but it is carefully handled. However, with the introduction of uninitialized string literals, the previous cases can be now erroneous. In the future, some work must be done to fix these problems.

Another possibility is to use the `array.new_fixed` instruction, which takes the array data inline. This instruction is considered constant and can therefore be used in global initializations. However, the syntax is really verbose. See, for example, the code required for the ASCII string "Hey":

```
(array.new_fixed 3 (i32.const 0x48) (i32.const 0x65) (i32.const 0x79))
```

This makes the code less readable, but also takes up more space. The last part is quite important, as the textual form of `array.new_fixed` is really heavyweight, and actually we use the textual form as an intermediate representation of WASM for linking purposes. Another problem arises from the constraints imposed by current implementations. For example, WASM engines embedded in browsers impose a maximum size of 10000 for arrays created with `array.new_fixed`.

II.3 Mutually dependent classes

In Bigloo Scheme, it is possible to create classes. They can inherit from another classes (simple inheritance), and their fields can reference other classes instances or any Scheme value. In WASM, we represent them using GC structures and the WASM subtyping:

```
;; Scheme code
(class parent
  (x::obj))
(class child::parent
  (y::obj))
```

```
;; Generated WASM code
(type $parent (sub
  (struct
    (field $x eqref))))
(type $child (sub $parent
  (struct
    ;; we must repeat parent fields
    (field $x eqref)
    (field $y eqref))))
```

WASM imposes two constraints:

- Parent types must be defined before children types.
- Types in fields references must be defined before references.

We consider the dependency graph of classes where a class A is connected to a class B if either A inherits from B or A has field of type B. Then, if we have no cycles in this graph WASM constraints are easy to satisfy, we only need to emit the WASM types in a topological order.

However, if there is a cycle, e.g. if there is mutually dependent classes (such as in Figure 2), then we must work harder. WASM has a simple solution for mutually dependent types: `rec` clauses. However, this causes problems at linking stage if not used correctly.

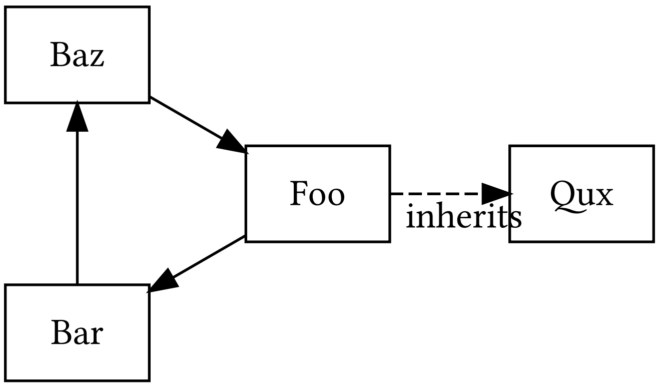


Figure 2: Example of dependency graph of mutually dependent classes

Indeed, the type system of WASM is iso-recursive and its based on structural equivalence. As a consequence, in the following two codes, type `A` on the left-hand side is not the same as the one on the right-hand side even if they have the same name. This is because the `rec` clause is not the same on the two codes, thus the children types are not equivalent because their parent `rec` clause is not equivalent. So, we can not emit all classes found in a module inside a `rec` clause. As modules do not have access to the same set of classes and will generate different `rec` clauses when using different imports. Then at link stage, the class types will not be considered the same from two different modules and will not be merged (either link time errors because functions do not have the correct signature, or function duplication). To solve this problem, we need to be sure to always generate the same `rec` clauses.

```
(rec
  (type (sub (struct $A))))
```

Types for module Foo

```
(rec
  (type (sub (struct $A)))
  (type (sub (struct $B))))
```

Types for module Bar

Code 3: The types `$A` in Foo and Bar are not equivalent.

To always generate the same `rec` clauses, we generate `rec` clauses only for classes in elementary cycles in the dependency graph. That way, we are sure that the generated `rec` clauses are always minimal and unique. In Bigloo, we use the following algorithm:

```
1 Compute the dependency graph of classes used in the module.
2 for each strongly connected component of the graph
3     for each elementary cycle of the SCC
4         Merge all classes inside the cycle into a WASM rec clause.
5         Also merges the vertices in the dependency graph.
  The dependency graph does not contain any cycle now.
6 Emits the class types and rec clauses in WASM in a topological order of the dependency graph.
```

On the Figure 2 case, the algorithm gives the WASM code shown in Code 6.

```
;; All WASM constraints are respected. Struct fields are omitted.
(type $Qux (sub (struct $Qux (; ... ;))))
(rec
  (type $Foo (sub $Qux (struct (; ... ;))))
  (type $Bar (sub (struct (; ... ;))))
  (type $Baz (sub (struct (; ... ;)))))
```

Code 6: Example of generated WASM for Figure 2 case

Recently, the above algorithm (or similar) was proposed to be merged into WebAssembly/binaryen (a toolchain for WebAssembly). See the pull request #6832, *"Add a pass for minimizing recursion groups"*. If this PR is merged, then the implementation inside Bigloo may be removed.

III Control flow in WebAssembly

Unlike low-level assembler, control flow is more limited in WebAssembly. It is mainly based on the notion of `block` and `loop` and the `br`, `br_if` and `br_table` instructions. Branching instructions can only be nested in `block`/`loops` and can only jumps (conditionally or not) to one of the parent `block`/`loop`. That is, it only supports structured control flow. In particular, the absence of a goto instruction to an arbitrary address in the bytecode complicates the compilation process. The Code 7 shows the different WASM control flow instructions.

```
;; Semantic of blocks
(block $block-label
  (; code... ;)
  (br $block-label) ;; Jumps to <TARGET BLOCK>, like a 'break' in C
  (; code... ;))
;; <TARGET BLOCK>

;; Semantic of loops
(loop $loop-label
  ;; <TARGET LOOP>
  (; code... ;)
  (br $loop-label) ;; Jumps to <TARGET LOOP>, like a 'continue' in C
  (; code... ;))

;; Same as 'br' but conditionally on the top value in the stack.
(br_if $some-loop-or-block-label)
;; We can also implement something like a 'jump table'. Otherwise, same
;; semantics and constraints as `br`.
(br_table $label1 $label2 (; ... ;) $default-label (; Jump index here ;))

;; There is also a high level conditional structure.
(if (; condition ;)
  (then (; code ;))
  (else (; code ;)))

;; Some extras
(return)
(unreachable)
;; ...
```

Code 7: Example of control flow in WASM

The WASM backend for bigloo is based on the SAW backend [2], [3], which is based around the notion of basic blocks. Generally, basic blocks are translated into assembler using arbitrary `gotos`. Thus, we need to somewhat emulate gotos or translate them into a WASM friendly structured control flow.

As a first solution, we could emulate merely `gotos` by introducing a local variable `label` that store the ID of the current basic block, and then jump to the basic block code using a switch (using the `br_table` instruction), see Code 8.

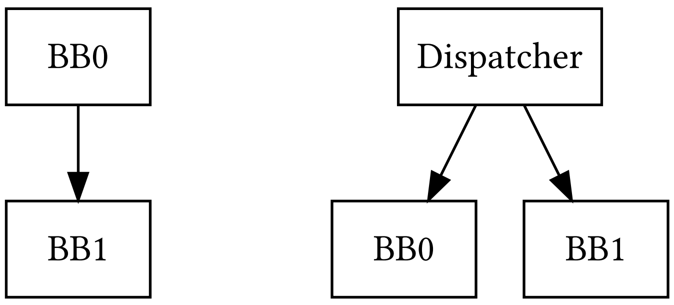
```
;; Function entry point
(local $label i32)
(local.set $label (i32.const 0))
(loop $dispatcher
  (block $bb_n
    (...
      (block $bb_0
        (br_table $bb_0 ... $bb_n (local.get $label))
      )
      ;; Code for basic block 0...
      ;; Jump to basic block 3 (for example)
      (local.set $label (i32.const 3))
      (br $dispatcher)
    )
    ;; Other basic blocks...
  )
  ;; Code for basic block n...
  ;; Jump to basic block 0
  (local.set $label (i32.const 0))
  (br $dispatcher)
)
```

Code 8: The *dispatcher* pattern to emulate `gotos`.

Unfortunately, avoiding the native control flow instructions of WASM and using the dispatcher pattern makes the code slower and less readable. Moreover, emulating `gotos` also destroys the CFG of the final generated WASM code (see Figure 3). As a consequence, the WASM assembler cannot correctly construct the dominator tree and thus prove that each local non-nullable reference is defined before its uses, which is required by the specification (see Code 9 for example). So, if using the dispatcher pattern, we must mark all local references into the GC as possibly null, even if Bigloo can guarantee that it is not possible. This penalises performance, as null pointer tests must be introduced at runtime.

```
(local $label i31)
(local $ptr (ref $bstring))
(local.set $label (i32.const 0))
(loop $dispatcher
  (block $bb1
    (block $bb0
      (br_table $bb0 $bb1 (local.get $label)))
    ;; Basic block 0
    (local.set $ptr (some function that creates a bstring))
  ;; Basic block 1
  ;; ERROR: $ptr may be null, use not dominated by a def.
  (drop (local.get $ptr))))
```

Code 9: Problematic code when using the dispatcher pattern



CFG before dispatcher pattern.
BB0 dominates BB1.

CFG after dispatcher pattern
BB0 do not dominates BB1.

Figure 3: Impact of the dispatcher pattern on the CFG

As the problem is not new, extensive research has been conducted and fortunately several solutions or improvements have been proposed. B. S. Baker described an algorithm to transform the control flow into a program using `if-then-else`, `loop`, `while`, `break` and `continue` statements in 1977 [8]. Similarly in 2011, A. Zakai proposed a new algorithm called Relooper that uses a heuristic to reconstruct some high-level control flow structures in LLVM when compiling to Javascript [4]. In Bigloo, we used an algorithm based on dominator trees and reverse postorder numbers that is deviced by N. Ramsay [9]. His implementation is clear and adapted to functional programming languages.

In all cases, these algorithms only work for a reducible control flow graph. The edges of a reducible graph split into two distinct sets such that:

- Its forward edges form a directed acyclic graph, with all nodes reachable from the entry node.
- For all back edges (A, B), node B dominates node A.

If needed, a preliminary work could be done to convert an irreducible CFG to a reducible one (possibly duplicating some basic nodes, [9] has a section dedicated to it).

However, in the case of Scheme, most CFGs are reducible. Indeed, Scheme has no construction for arbitrary `goto`s, and, in this internship, we don't support `call/cc`. However, non-reducible CFGs are sometimes produced:

- due to certain optimizations (such as basic block versioning in Bigloo), or
- due to regular expression compilation (see `regular-grammar` in Bigloo Scheme).

In such case, we must resort to use the dispatcher pattern for these functions. Later iterations of this implementation should convert the irreducible CFG to a reducible one to take full advantage of this optimization.

IV Implementation of the runtime library

One of the major parts of this internship was to port the Scheme standard library to WASM. Hopefully, most of it was written in Scheme and could be compiled to WASM with the new backend. But some other parts were written directly in C or Java, depending on the target language. These parts needed to be rewritten either in pure Scheme when possible, or in WASM.

Unlike C or Java, WASM is an assembly language, and writing complex control flow or algorithms is not a realistic option. Implementing the runtime functions would be a bug-prone and time-consuming job that would result in thousands of lines of handwritten WebAssembly code. That's why we opted for rewriting some parts in Scheme so that other backends can also benefit in the future.

The dependence to JavaScript also helped implementing more complex features like regexes and date and time management.

IV.1 Interfacing with JavaScript

WebAssembly offers the possibility of importing external functions (defined by other WASM modules or in JavaScript). In this section, we show how Bigloo interfaces with JavaScript to implement simple IOs. The JavaScript part is shown in Code 10, whereas the WASM code is shown in Code 11.

JavaScript cannot directly interact with the WASM GC. However, Bigloo strings are implemented as arrays of bytes allocated in the GC. So JavaScript can't directly create, manipulate, or access Scheme strings. To do so, we need to go through an intermediate data structure, which is the linear memory. To pass a string to JavaScript, we need to perform the following steps:

- Copy the Scheme string to the WASM linear memory at some fixed address.
- Call the JS function with the memory address and the string length.
- The JS function stores the new string into the memory address at the specified address and returns the string length.
- Then Scheme creates a new GC byte array and loads the data from the memory at the correct memory address.

These steps impact the performance, so we need to avoid passing character strings between the two worlds. In general, we should minimize back-and-forth between JavaScript and Scheme WASM. It can be costly to make conversations between the different representations and to call functions from different worlds.

```javascript
function read_file(fd, offset, length) {
  if (fd < 0)
    throw WebAssembly.RuntimeError("invalid file descriptor");

  // Treat the WASM memory as a byte array.
  const memory = new Uint8Array(
    // The WASM memory buffer
    instance.exports.memory.buffer,
    offset,
    length);

  // We use the NodeJS fs sync API which writes directly
  // to the specified byte buffer (in our case, the WASM memory).
  const readBytes = readSync(fd, memory);
  return readBytes;
}
```

Code 10: JavaScript implementation of `read_file`

```
(import "js" "read_file" (func $js_read_file (param i32 i32 i32) (result i32)))
(func $myfunction (result (ref $bstring))
  (local $nbread i32)
  ;; Call the JS function.
  (local.set $nbread (call $js_read_file
    (i32.const 0) ;; File descriptor (0 = stdin)
    (i32.const 128) ;; Memory address where the result is stored
    (i32.const 10))) ;; Number of bytes to read
  ;; We need to load the bytes from the linear memory into a
  ;; GC array of bytes which is our representation for Scheme strings.
  (call $load_string
    (i32.const 128)
    (local.get $nbread)))
```

Code 11: WASM code using the JS function `read_file`

The real code is more complex. We need to take care of runtime errors, disparities between the different JavaScript implementations, encoding issues, etc.

As a first step, we wanted our WASM to be executed by NodeJS and Deno, two JavaScript runtime environments outside the Web that are cross-platforms. These two environments supports the POSIX API and can communicate with the OS. They are perfect to initially port the C runtime as they have all the required features. But, NodeJS and Deno, even though they are mostly compatible, have some weird difference in their API. For example, their handling of `readSync` function that emulates the POSIX `read` syscall but synchronously (by default NodeJS and Deno are highly asynchronous). Indeed, the behavior is not the same on the POSIX file descriptor 0 that corresponds to `stdin`. On NodeJS, calling this function with `stdin` descriptor results in an error (`EAGAIN` POSIX error) whereas on Deno this is fully supported. So, our runtime implementation must have a special case for `stdin` on NodeJS.

There are also problems with the way string encodings are managed. In Scheme, strings are just array of bytes like in C. The encoding is implicit, and depends on the underlying environment (in particular the C environment and locale). To keep compatibility with the C backend, programs must keep the same behavior. Therefore, when converting a Scheme string to a JavaScript string (required to access the filesystem API for example), we need to carefully decode the string with the correct encoding. Actually, in our implementation, the current encoding is a global variable that can be configured by the user.

IV.2 Small WASI example

As we said at the beginning, because WASI is not yet mature and complete enough, we have decided to use the JavaScript environment. Even if this means a dependency on Web browsers or JavaScript runtime environments.

However, to access the file system, WASI can already be used although, depending on the implementation, it may only be possible to access an isolated sandbox file system. An example of its use is given in this section in the following code:

```
(import "wasi_snapshot_preview1" "fd_write"
  (func $wasi_fd_read (param i32 i32 i32 i32) (result i32)))

(func $write_file (param $str (ref $bstring))
  (i32.store (i32.const 0) (i32.const 8))
  (i32.store (i32.const 4) (array.len (local.get $str)))

  (; Copy $str content into memory at address 8 ;)
  ;; ...

  ;; Ignore return code, we omit error handling.
  (drop
    (call $wasi_fd_read
      (i32.const 1) ;; file handle
      (i32.const 0) ;; mem address of iovec array
      (i32.const 4) ;; mem address of buffer length
      (i32.const 0) ;; mem address of the count of written bytes
    )))
```

We have used the WASI `fd_write` function that emulates the POSIX `writev` syscall. The semantics of the functions are not clearly stated in the specification and seems to rely on the POSIX function semantics. So we expect file handle 1 to define `stdout`, likewise we expect the string encoding to be one that POSIX wants.

In the early experiments, I found the WASI documentation to be hideous and not clear at all. For example, the signature of the `fd_write` as in the specification is provided below. There is no real explanation how to translate it to the WASM way. There are specification on the binary layout of the used structures, but not how the array is passed in arguments in WASM.

```
fd_write(fd: fd, iovs: ciovec_array) -> Result<size, errno>
```

V Benchmarks

Due to lack of time and various technical problems I can not include relevant benchmarks at the moment. Currently, preliminary tests inform of a slowdown of a factor of 4 for WASM-generated code compared to Bigloo-generated C code (for the computation of naive recursive fibonacci). I am investigating to improve or confirm these results. I hope to be able to provide better and more complete results for the oral presentation.

There is two sets of benchmark for Scheme that we could use. One from Gambit and one from Bigloo (see bglstone).

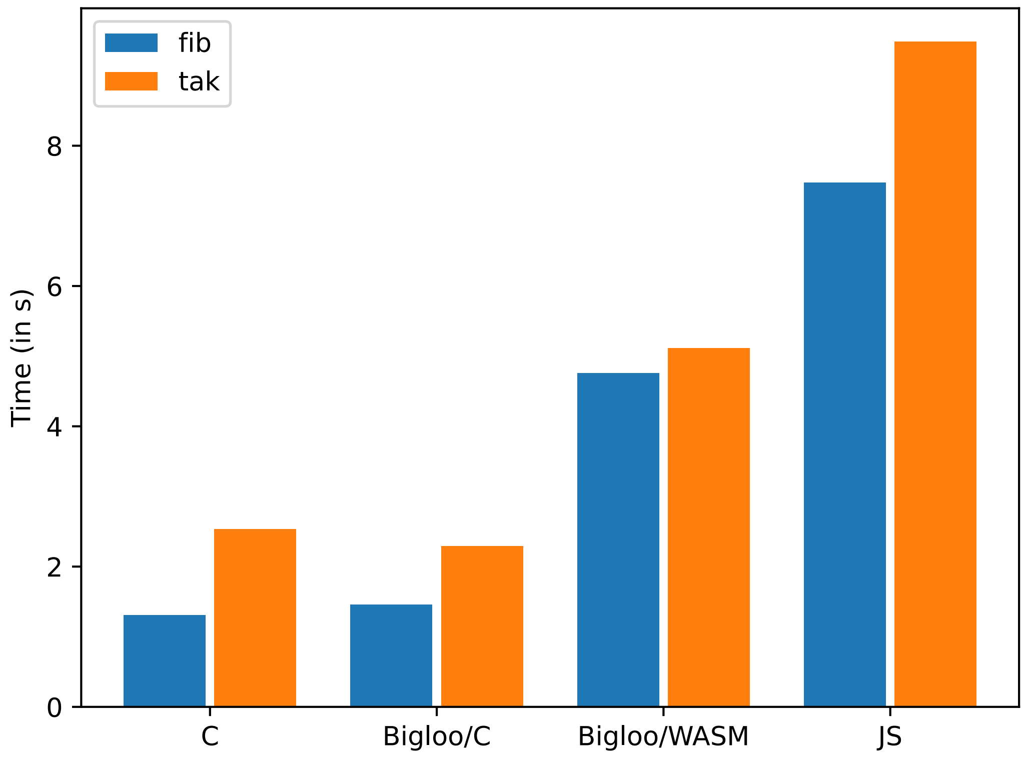


Figure 4: Results of benchmarks

The `fib` benchmark is 10 iterations of `fib(40)` using the naive recursive implementation. The `tak` benchmark is 100 iterations of `tak(30, 22, 12)`. The handwritten C and Bigloo/C implementations are compiled using GCC 11.4 with `-O3`. The JS and Bigloo/WASM implementations are executed using NodeJS 22.2. The Bigloo/C and Bigloo/WASM were both compiled using `-Obench`. For the Scheme code, we used the unboxed operations available in Bigloo (`+fx`, `<fx` etc.). The benchmarks were executed on a Intel i5-1240P 2.112GHz with 16 cores under Ubuntu 22.04.4 LTS on Windows 10 x86_64.

We observe that the C implementations and the Bigloo/C compilations gives the same performance. Whereas, the Bigloo/WASM has a slowdown of a factor of 2 to 4 compared to the C implementation. The handwritten JavaScript implementations are slower in these benchmarks as compared to C and Bigloo Scheme because they have no type information.

VI Conclusion

VI.1 Is WASM mature enough?

WebAssembly is a recent technology, first introduced in 2015. The proposals about garbage collection, exceptions, tail calls, and typed function references are even more recent. They are not yet officially part of the WebAssembly specification, but their inclusion is planned for version 2.

These proposals are supported by Google Chrome and Firefox. However, Apple Safari does not yet support garbage collection or tail calls. Likewise, many tools around WebAssembly still lack support for these proposals or have gained support in very recent versions. As an example, during my work on the Bigloo backend, I have found bugs in Binaryen (see issues #6655, #6708, #6738, and #6739), a toolchain for WebAssembly, used in particular by LLVM, Clang, and Rust. But also in Deno, a JS runtime based on V8 (see issue #24990).

Standalone implementations of WebAssembly, such as wasmtime or wasmer, also do not yet support the garbage collector proposal, even if it is planned. And the code generated by wasmedge (a ahead-of-time compiler for WASM) for our generated WASM code crashed at execution (free pointer error, … but we are only using the GC, so probably an error in wasmedge).

As a conclusion, it appears that WebAssembly with the above-mentioned proposals (required by Bigloo and for efficient implementation of dynamic and function languages) is **still experimental**. Its support is not complete, either in the tools or the virtual machine implementations. But we can hope for improvements in this area. The Wasocaml [6] and Guile Hoot [10] projects are also vectors for improvement and proof of **the potential of these proposals and WASM**.

VI.2 The future of Bigloo WASM

The implementation of the WASM backend is not yet complete. Some library functions are still missing. In addition, Scheme features such as call/cc are missing (but they are not required for compiling JavaScript using the Hop compiler).

There are still many Scheme programs that can not be compiled with the new backend. In particular, the test suite of Bigloo can not be yet executed. Further fixes must be implemented to complete this work, but, the biggest part has been accomplished. Generally, the problem is an unsupported type, or a missing instruction, and in most cases can be solved in no time. Ideally, we should aim to be able to fully compile the testsuite and the benchmarks in the future.

There are also probably still opportunities for optimization in the generated code. For example, caching the boxed values of small integers or improve errors implementation. The way strings are handled can maybe also be improved. Irreducible control flow graphs are not yet handled and use the naive dispatcher pattern. These are just a few ideas among many.

VI.3 Acknowledgements

I would like to thank Manuel Serrano for his supervision, help, and support during all this internship. Marc Feeley deserves thanks for his help and advice, especially on benchmarks. I also appreciate Erven Rohou and Olivier Melancon for their questions and conversations.

On a more general note, I did also like to thank the Inria center for its warm welcome, my friends for their proofreading and everyone I met at the lab for being so welcoming and pleasant.

Bibliography

[1] M. Serrano and P. Weis, “Bigloo: A Portable and Optimizing Compiler for Strict Functional Languages,” in *Proceedings of the Second International Symposium on Static Analysis*, in SAS '95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 366–381.

[2] B. P. Serpette and M. Serrano, “Compiling scheme to JVM bytecode: : a performance study,” in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, M. Wand and S. L. P. Jones, Eds., ACM, 2002, pp. 259–270. doi: 10.1145/581478.581503.

[3] Y. Bres, B. P. Serpette, and M. Serrano, “Bigloo.NET: compiling Scheme to .NET CLR,” *Journal of Object Technology*, vol. 3, 2004, doi: 10.5381/jot.2004.3.9.a4.

[4] A. Zakai, “Emscripten: an LLVM-to-JavaScript compiler,” in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, in OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 301–312. doi: 10.1145/2048147.2048224.

[5] A. Takikawa, “Compiling Bigloo Scheme to WebAssembly,” May 2023, [Online]. Available: https://blogs.igalia.com/compilers/2023/05/10/compiling-bigloo-scheme-to-webassembly/

[6] L. Andrès, P. Chambart, and J.-C. Filliâtre, “Wasocaml: compiling OCaml to WebAssembly,” in *IFL 2023 - The 35th Symposium on Implementation and Application of Functional Languages*, Braga, Portugal, Aug. 2023. [Online]. Available: https://inria.hal.science/hal-04311345

[7] M. Serrano and M. Feeley, “Storage use analysis and its applications,” in *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, in ICFP '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 50–61. doi: 10.1145/232627.232635.

[8] B. S. Baker, “An Algorithm for Structuring Flowgraphs,” *J. ACM*, vol. 24, no. 1, pp. 98–120, Jan. 1977, doi: 10.1145/321992.321999.

[9] N. Ramsey, “Beyond Relooper: recursive translation of unstructured control flow to structured control flow (functional pearl),” *Proc. ACM Program. Lang.*, vol. 6, no. ICFP, Aug. 2022, doi: 10.1145/3547621.

[10] F. R. Farmer, “Guile on WebAssembly project underway!,” Feb. 2023, [Online]. Available: https://spritely.institute/news/guile-on-web-assembly-project-underway.html