watib: An optimising toolchain for modern WebAssembly

Aghilas Y. Boussaa <aghilas.boussaa@ens.fr>

August 20, 2025

Abstract

WebAssembly (Wasm) is a code format providing portable and safe execution. The latest draft of its standard offers features making it a suitable compilation target for functional programming languages. These features led to the development of Wasm backends for compilers such as the Bigloo Scheme compiler. However, Bigloo's Wasm backend relied on binaryen, an external toolchain with user-friendliness and spec-compliance issues.

We present watib, a new WebAssembly Toolchain written in Bigloo. This optimising toolchain is integrated into the Bigloo compiler and is also available as a standalone tool. We compare watib with binaryen, showing how the former deals with the mentioned issues of the latter. On benchmarks, we have observed that watib's optimisation passes do not impact most Scheme programs except for two tests that got a noticeable improvement and one that got an unexplained regression. We conclude by presenting a promising work-in-progress adaptation of Static Basic Block Versioning to Wasm.

1 Introduction

In 2024, a Wasm backend was added to the Bigloo Scheme compiler [Ser]. This backend produced textual Wasm and relied on an external toolchain to assemble it into binary format. We developed watib as a replacement for this toolchain. It is part of the Bigloo compiler and is also available as a standalone tool with a command-line interface[1].

The rest of this section gives an overview of WebAssembly and watib. Section 2 details watib's type checker; Section 3 describes the different optimisation passes. Section 4 compares watib's and existing tools' performance, and Section 5 discusses a work-in-progress optimisation. This presentation omits the assembly phase as it is a straightforward transcription of the intermediate representation into the binary format defined in the specification.

1.1 WebAssembly

WebAssembly [Haa+17] is an assembly-like language for a stack-based virtual machine. It aims at providing a fast, safe and portable language endowed with an efficient binary format.

[1] https://www.normalesup.org/~boussaa/watib/

Unlike traditional assembly, Wasm is statically typed and only supports structured control flow. Support for Wasm in existing compilers is growing [Ems; Kot; Oca; Rub].

(1.1.1) *The standard.* The WebAssembly specification [Ros25] specifies typing rules (called validation rules), operational semantics, an abstract representation of Wasm modules, on which the previous rules are stated, and two grammars for concrete formats. The binary format represents WebAssembly modules as a sequence of bytes. Wasm virtual machines such as the ones implemented in V8 [V8] or SpiderMonkey [Spi] and toolchains such as binaryen [Bina] or wasm-tools [Was] receive this format. The textual format represents WebAssembly modules as S-expressions. Toolchains receive this format and humans (and compilers such as Bigloo) write it.

(1.1.2) *New features.* The third version of the standard adds features facilitating compiling from high-level or functional languages, such as a garbage collector, instructions for tail-calls (they are needed because the language does not support goto) and exceptions. This version is still in draft, but the mentioned features are stable and already implemented by the major Wasm engines. Bigloo's Wasm backend relies on these features.

(1.1.3) *An Example.* The example of Figure 1 demonstrates the features mentioned earlier. For a more detailed introduction to WebAssembly, see [PC+23, Section 2.1] or [Haa+17]. This program starts by defining a type of integer lists (`$pair-nil`) which is a super-type of non-empty integer lists (`$pair`). A non-empty list has a head (field `$car`), which is a 32-bit integer, and a tail (field `$cdr`), which is a list. The function `$has-zero` checks whether a given list contains 0 and returns an `i32` as Wasm has no boolean type. The function starts by testing the non-emptiness of its argument by checking that it is of type `$pair` using the instruction ref.test. Before accessing the head or the tail of the non-empty list (struct.get), the function casts it to the type `$pair`, using the instruction ref.cast. The program would be ill-typed without these two casts.

1.2 Watib

`Watib` handles type checking (see the `Val` folder of the sources), optimisation (the `Opt` folder) and assembly (the `Asm` folder) of Wasm programs given in textual format[2]. Function validation and optimisation can run in parallel. Our tool supports using an arbitrary number of threads for these passes using the `pthreads` library. This feature is optional; the Bigloo compiler cannot use it, as it cannot rely on an external library. We have set up continuous integration using tests bundled with the specification. As `watib` does not support the whole standard yet, we maintain a repository of tests patched to use only the parts of the specification covered by `watib`.

The WebAssembly platform is in constant evolution, and long-term maintenance is one of our goals. `Watib`'s design allows us to keep up with the additions made to the language.

[2] binary format is planned

```
(type $pair-nil (sub (struct)))
(type $pair (sub $pair-nil (struct (field $cdr (ref $pair-nil))
                                   (field $car i32))))
(func $has-zero
  (param $l (ref $pair-nil))
  (result i32)
  (if (ref.test (ref $pair) (local.get $l))
    (then
      (if (i32.eqz (struct.get $pair $car
                     (ref.cast (ref $pair) (local.get $l))))
        (then (return (i32.const 1)))
        (else
          (return_call $has-zero
            (struct.get $pair $cdr
              (ref.cast (ref $pair) (local.get $l)))))))))
    (else (return (i32.const 0))))
  (unreachable))
```

Figure 1: A Wasm function testing if a list of integers contains 0.

To add an instruction that is not a new block, one puts a new entry in the list of opcodes (`Asm/opcodes.sch`) and in the list of validation rules (`Val/instruction-types.sch`). The case of block instructions is more involved as their representations in text format differ from those of plain instructions. This should not be a problem because the introduction of new kinds of blocks is rare. One also has to modify the optimisation passes to take into account the new instruction.

We now review some differences between watib and already existing toolchains (apart from the lack of features and maturity of the former), focusing on binaryen (and its assembler wasm-as), the toolchain most compilers use [Bina], including Bigloo before watib's development.

(1.2.1) *Fault tolerance.* For the sake of user-friendliness, watib can continue the validation of a file after encountering an error. This allows watib to report multiple errors in one pass and avoids users having to correct errors one by one. In one instance, the Bigloo Wasm backend produced a file of 1,6 million lines, and running wasm-as to find a single error took a noticeable amount of time, which complicated debugging. We also try to give informative error messages. The benefits of these features become apparent when using our toolchain to correct code written by hand, for instance, when developing a runtime.

(1.2.2) *Zealous*[3] *respect of the specification.* Wasm-as accepts files that do not conform to the specification and, in this case, outputs files that have more or less the same semantics. We collected in watib's internal documentation a list of the modifications made by wasm-as we encountered [Bou]; a copy is in Appendix A. For the sake of portability, watib follows the

specification by default. A compatibility mode with wasm-as is planned.

For instance, before the introduction of watib, Bigloo's Wasm backend was generating code ill-typed according to the standard but which was still accepted by wasm-as without any warning. In fact, the original implementation of exceptions was incorrect and a spec-compliant validation would have rejected the code. The integration of watib in Bigloo revealed the bug.

(1.2.3) *Linear Intermediate Representation (IR).* Binaryen's optimiser, wasm-opt, has a tree like IR, in part for historical reasons [Binb]. When the development of binaryen started, Wasm wasn't stack based and features such as multiple return values or block parameters weren't planned. These features remain second class citizens in binaryen[4] because they can create a data flow that cannot be reflected by classic ASTs. For instance, in Figure 2, two nodes share the results of a same function call and a node takes its two inputs from a single node.

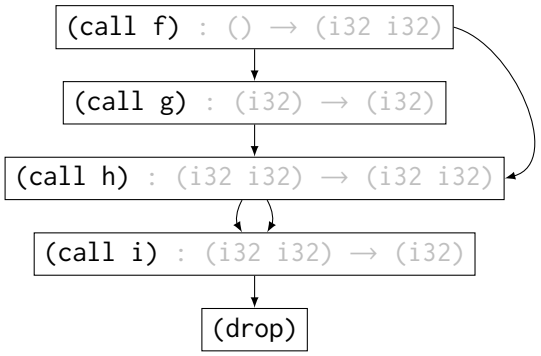

Figure 2: Non classical data flow with multiple return values.

Watib's IR is closer to modern Wasm by representing the instructions in a linear way. It allows the output of code that use block parameters without additional burden. We hope that this choice will provide more optimisations opportunities and improve the processing of code written using WebAssembly's new features.

2 Validation

Subsection 2.1 gives an overview of WebAssembly's type system, and Subsection 2.2 describes our implementation of a type checker for it. Both subsections focus on the subtyping relation and the typing of instructions as the rest of the validation is straightforward (checking that some names do not appear twice, or that global variables' initial values are constant, etc.).

[3] in fact, the proofreading induced by our careful study of the specification resulted in a dozen (minor) pull requests and issue reports to the draft git repository

[4] for instance, wasm-as assembles code using block parameters to code using local variables instead

2.1 WebAssembly's type system

We present the types used in WebAssembly and some declarative rules for subtyping and typing of instructions. For more details, Section 2 of the specification presents the abstract syntax of types, and Section 3 presents the rules of typing and subtyping.

(2.1.1) *Contexts.* We state all the following rules in an implicit *context*, which we note C in the formal rules. Contexts are records; the only relevant field in the following is types which maps type indices to *defined types* (see §2.1.6).

(2.1.2) *Value Types.* Wasm's main types are *value types*, which we will note t, t_1, t_2, etc. They are the types of values on the stack. They can be *numeric types* (i32, f64, etc.), *vector types* (v128) or *reference types*. The latter category represents pointers to functions, pointers to objects allocated on the heap (structures or arrays, called *aggregates*) or unboxed integers (using tagged pointers). Each reference type is either nullable (ref null *ht*) or not (ref *ht*), where *ht* is a *heap type*. Such a type is either an *abstract heap type*, which we do not detail here or a type index referring to one of the three following possibilities:

- *function types* are written func ${t_1}^* \rightarrow {t_2}^*$ (recall that a function can have multiple outputs).
- *Array types* are written array *mut t*, where *mut* denotes the mutability of the array. A mutability is var for a mutable array and const otherwise.
- *Structure types* are written struct $(mut\ t)^*$.

(2.1.3) *Instruction types.* The specification states inference rules to give *instruction types* to instructions. These types indicate which values on the stack an instruction pops, which values it pushes and which local variables it sets[5]. For the sake of simplicity, we omit this last piece of information in this presentation. We write instruction types like function types: ${t_1}^* \twoheadrightarrow {t_2}^*$ for an instruction that pops ${t_1}^*$ and pushes ${t_2}^*$[6]. These types obey the classical rule of subtyping for functions as presented in [Car88]. An instruction can accept smaller types as inputs and produce bigger types as outputs. An additional subtyping rule reflects the stack-based semantics of Wasm: an instruction can add arbitrary value types before the input and output types as long as it adds the same to both. Figure 3 presents a formal version of these rules.

A sequence of instructions can then be typed by the composition of types derived for each of its instructions. For instance, the sequence `(i32.const 0) (i32.const 0) i32.add` can be typed $() \twoheadrightarrow (\text{i32})$, by giving the type $(\text{i32}) \twoheadrightarrow (\text{i32 i32})$ to the second i32.const, showing the need for the second subtyping rule.

[5] to forbid, at the type level, using a variable that has not been initialised yet

[6] the WebAssembly standard uses the same arrow for instruction and function types, we distinguish them for improved readability

$$\frac{(C \vdash t_1 \leq t_1)^* \qquad (C \vdash t_2 \leq t_2)^*}{C \vdash {t_1}^* \twoheadrightarrow {t_2}^* \leq {t'_1}^* \twoheadrightarrow {t'_2}^*} \qquad\qquad \frac{}{C \vdash {t_1}^* \twoheadrightarrow {t_2}^* \leq t^* {t_1}^* \twoheadrightarrow t^* {t_2}^*}$$

Figure 3: Subtyping rules for instruction types.

(2.1.4) *Function and aggregate types.* We now give details on the definitions and subtyping of function types and aggregate types. As shown in Figure 4, Wasm allows the definition of mutually recursive types. Moreover, it supports *declared and mutually iso-recursive subtyping*.

```
(rec
  (type $a (array (ref $b)))
  (type $b (struct (field (ref null $c))))
  (type $c (func (param (ref $a)) (result (ref $c)))))
```

Figure 4: Some type definitions.

Function types are endowed with the same first rule described for instruction types (the contravariant/covariant one). An array type $\mathsf{array}\, mut\ t$ is a subtype of $\mathsf{array}\, mut'\ t'$ if, and only if, $mut\ t$ is a subtype of $mut'\ t'$. This means that mut is equal to mut' and that t is a subtype of t'. When both types are mutable we also require that t' is a subtype of t, i.e., that both types are equivalent. A structure type $\mathsf{struct}\,(mut_i\ t_i)_{i=1}^n (mut\ t)^*$ is a subtype of $\mathsf{struct}\,(mut'_i\ t'_i)_{i=1}^n$ if, and only if, $mut_i\ t_i$ is a subtype of $mut'_i\ t'_i$ for all $1 \leq i \leq n$.

(2.1.5) *Declared subtyping.* Wasm's subtyping is *declared*; this means that a subtyping relation between two types defined in a program has to be specified as part of the subtype's definition to be used. The validity of the subtyping relations can be checked with the type definition's validation. For example, the definition of `$pair` in Figure 1 states that it is a subtype of `$pair-nil`. If the `sub $pair-nil` part wasn't in the definition, the type tests and the casts would be ill-typed, as a value can be cast or tested against a type rt if and only if rt is a subtype of the value's static type.

Each type definition can specify a supertype (the current specification only allows one). A type can be a supertype if and only if it is not *final*. Types declared in a program are final by default. The `sub` keyword in the definition of `$pair-nil` of Figure 1 makes it non-final.

(2.1.6) *Mutually iso-recursive subtyping.* We now give the intuition behind *mutually iso-recursive subtyping*; see [Ros23] for a full discussion. It is a way to support subtyping in presence of mutually recursive type definitions, while avoiding size blowups. When checking a block of such type definitions, we build an internal representation where indices referring to types of the block have been replaced by special indices of the form $\mathsf{rec}\, i$ where i is the position of the corresponding index in the block. For instance, the internal representation of the block

of Figure 4 is:

$$\begin{array}{l}\mathsf{rec}\ ((\mathsf{array\ const\ ref\ rec}\ 1)\\ \qquad(\mathsf{struct}\ (\mathsf{const\ ref\ null\ rec}\ 2))\\ \qquad(\mathsf{func}\ (\mathsf{ref\ rec}\ 0)\ \rightarrow\ (\mathsf{ref\ rec}\ 2))).\end{array}$$

A defined type is then a vector of type definitions and an index in this vector. The types field of the context then maps each type index with a corresponding defined type. For instance, for Figure 4, `$a` has index 0, `$b` has index 1, etc.

With this representation, a defined type t will be a subtype of another defined type t' if they are equal or if the program declares t as a subtype of a subtype of t'.

(2.1.7) *Unrolling.* Defined types only give a block of definitions and an index. To retrieve the corresponding concrete type, one uses *unrolling*. It consists of replacing a defined type of the form $(\mathsf{rec}\, st^*).i$ by the i^{th} type of st where the indices of the form $\mathsf{rec}\ j$ have been replaced by $(\mathsf{rec}\, st^*).j$.

2.2 A type checker for WebAssembly

We now describe watib's type checker for instructions and sequences of instructions. The main difficulty encountered was implementing the transitivity rule for instructions.

(2.2.1) *The problem of transitivity.* The typing rules given by the third section of the specification are purely declarative. They give more than one type to many instructions and rely on the transitivity rule of subtyping, which cannot be implemented as is. To check that $t_1 \leq t_2$ in presence of transitivity, one can choose a third type t_3 to compare it to t_1 and t_2, which complicates subtyping as t_3 is arbitrary. We thus needed to adapt the rules to implement a type checker.

(2.2.2) *Our solution.* Watib's type checker is similar to the one shipped with the specification. For the subtyping of *value types*, we rewrote the rules to have equivalent ones without the transitivity rules[7], and for *instruction types*, we rely on a stack and most general types, as defined in §2.2.3.

(2.2.3) *Most general instruction types.* Most instructions have a most general type (meaning an admissible type for the instruction that is a subtype of all its types) or do not return (like unconditional branches). The typing rules for the latter are of the form $i : {t_1}^* t^* \twoheadrightarrow {t_2}^*$ for all t_1 and t_2, but where t^* is fixed. We represent it as $t^* \twoheadrightarrow (\mathsf{poly})$, where poly is a special symbol indicating that the stack can be whatever we need it to be. It can be considered as a type variable on which we do not perform explicit unification. When we refer to the type one of those instruction, we mean its most general type or the type with the poly symbol — these types are contained in the file `Val/instruction-types.sch`. The instructions that do not fall in either category (for instance, ref.as_non_null has type $\mathsf{ref\ null}\, ht \rightarrow \mathsf{ref}\, ht$ for all heap

[7] they are implemented by the function `<ht=` of the `Type/match.scm` and reproduced in Appendix B

type *ht*) are treated in a separate function: they peek on the stack to compute their output type, avoiding the use of type variables.

(2.2.4) *The algorithm.* To check a sequence of instructions we maintain an internal state recording the types of the elements present on the stack and then check instruction in order, instead of analysing each instruction individually and checking that the types compose well.

We describe how one instruction i modifies the state of this stack. Either we treat the instruction in an ad-hoc way (function `adhoc-instr`), or we apply the following procedure (function `valid-instr`) if the instruction falls in one of the two categories of §2.2.3:

- Let $t_n \dots t_1{}^* \twoheadrightarrow t'^*$ be the type of i.
- Check that the j^{th} type on the stack is a subtype of t_j and pop it, for j from 1 to n. If the stack is equal to (poly), for all j, consider that the j^{th} type on the stack is $\bot$, making the previous test successful.
- If t'^* is equal to (poly), replace the stack by (poly), otherwise push the return types on top of the stack.

(2.2.5) *Type checking blocks.* To check a block against a given type, set the parameter types as initial state of the stack and compare the final state of the stack with the expected type. For that final check, pop types from the stack and check that they are subtypes of the expected ones. End by checking that the resulting stack is empty or equal to (poly).

3 Optimisation

Watib performs several optimisation passes that all maintain type information. Our optimiser is transformation-based: it applies sequentially small indepedent code transformations. As noted in the introduction of [JS98], this design increases modularity of the toolchain and facilitates experimenting with the order of the passes.

3.1 Generic optimisations

We start by reviewing optimisation passes which benefit the other passes, by giving more precise information or removing generated useless code. They are generic in the sense that they are applied in most optimising compilers [Muc97].

(3.1.1) *Copy Propagation.* When a variable x receives the value of another variable y, one can replace references to x with references to y until the next assignment to x or y. The Copy Propagation pass performs such replacements. We took inspiration from Muchnick's [Muc97, Section 12.5] approach to local copy propagation for our implementation.

We go through a sequence of instruction while maintaining a table `acp` which indicates if a variable contains the value of another one: if x is a variable, then $\texttt{acp}[x]$ contains either another variable or a constant indicating that the value of x is not bound to another variable.

We also associate to each label a list of tables, which all have the same type as acp: they associate to a label l all the possible states of the acp when the program reaches l. We then apply the following procedure when encountering an instruction i:

- If i assigns the value of a variable y to another variable x, update acp with x pointing to y. Replace each occurence of x in acp by y. Otherwise, unset acp$[x]$ and unset each cell of acp that contains x.
- If i accesses a variable x and that acp$[x]$ contains a varibale y, modify the instruction to access y.
- If i can branch to a label l, append the current table to the list associated to l.
- If i is a block of label l, associate an empty list to l. The table after the block corresponds to the upper-bound of all the tables associated to l and acp at the end of the block. The upper bound of a list of tables assigns y to x if all of its tables do and gives no information on x otherwise.
- If i is a loop, reset acp to remove all the assignments as we may jump to the beginning of the loop from later in the block[8]. Then treat the loop's body.
- If i is an if statement, treat each branch as a block and take the upper bound of the two resulting tables.
- If protecting a block with some catch clauses, append the table indicating no assignment to all the labels these clauses can jump to. Once again this is coarse.

At the end of each block, remove all the assignments in acp whose right-hand side is uninitialised in the parent block.

(3.1.2) *Pure Drops Elimination.* When an instruction produces a single value that is then popped by a drop, both instructions are useless except for the side effects of the former. The Pure Drops Elimination pass removes such patterns that have no side effects apart from popping values on the stack. When a removed instruction popped values of the stack, each popped value gets a corresponding drop that can be removed by the pass.

(3.1.3) *Constant Folding.* The Constant Folding pass performs computations whose results are known at compile time, for instance, if statements on a constant value can be replaced by the corresponding branch.

(3.1.4) *Unreachable Code Elimination.* The Unreachable Code Elimination pass eliminates some instructions that are known to never execute (the ones after a return for instance). However, some dead code is needed for validation. For instance, removing the unreachable instruction at the end of the function of Figure 1 makes it ill-typed, even though both branches of the if statement return. The if instruction puts nothing on the stack while the function expects an i32 on top of the stack as it is its declared return type. To avoid such problems, an

[8] this is suboptimal for global copy propagation, but it is sufficient for the code we get, and this optimisation could be replaced by the optimisation we describe in Section 5

unreachable instruction replaces the unreachable code when it is not preceded by an instruction of polymorphic return type. The unreachable code can be deleted otherwise.

(3.1.5) *Peephole Optimisation.* Some small patterns in Wasm code are equivalent to smaller or faster code. For instance a sequence of local.set and local.get with the same variable is equivalent to a local.tee which assigns to a variable the value on top of the stack without popping it. The Peephole Optimisation pass recognises such patterns and replaces them.

3.2 Wasm specific optimisations

We now review more Wasm-specific optimisations and detail in which way they use the previous ones.

(3.2.1) *Cast Elimination.* The Cast Elimination pass removes redundant casts and replaces type tests by constants if their result can be determined. Such casts and tests could be introduced by copy-propagation as when we may replace a variable with another one of a smaller type or by the next optimisation. To stay conservative about side effects, eliminated tests are replaced by a drop and the result of the test, which is a constant. Pure Drop Elimination will remove some useless computation. Constant folding will propagate the result of the test.

To check for type tests to eliminate, we compare the argument's actual type (which may be smaller than its static type) against the type it is tested for. For casts, we only check against the static type as the cast can be necessary to maintain well-typedness.

(3.2.2) *Type-Dependent Control Flow Rewriting.* A common pattern in the Wasm code generated during the compilation of dynamically typed programming languages, such as Bigloo, is a branching on a type test followed by the use of casts in the branch where the test is successful, like in Figure 1. If Wasm wasn't statically typed, these casts could be removed by an optimising compiler that would determine that they are redundant.

The instruction br_on_cast l rt_1 rt_2, where l is a label expecting a value of type rt_2 as parameter and $rt_1 \geq rt_2$ are reference types, can avoid useless type tests, by combining branching and casting. While ref.cast traps when the type test fails, br_on_cast allows arbitrary (structured) jumps. A call to this instruction with a value of type rt_1 on top of the stack applies the following procedure:

- Check this value against the type rt_2.
- If the type test succeeded, branch to label l with the value on top of the stack as a parameter.
- Else, continue the execution without modifying the stack.

This pass replaces if instructions which branch on a (ref.test rt (local.get x)) by calls to br_on_cast, as shown in Figure 5. It adds a local variable y of type rt and creates a new block labelled l with a parameter of type rt. The else branch follows the br_on_cast which branches to l. The block labeled by l puts its parameter in the local variable y and replaces x by y in the then branch following the same procedure as Copy Propagation. This whole code is contained

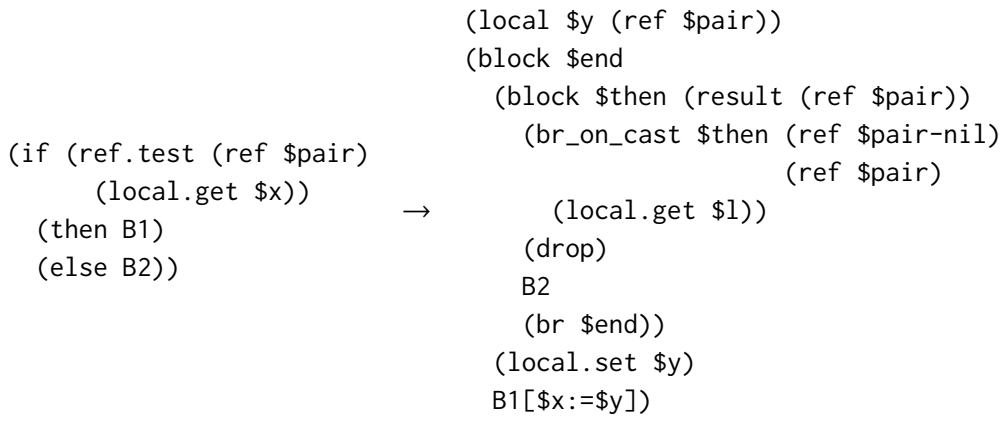

Figure 5: General form of type-dependent control flow rewriting.

in a block which the else branch jumps to at the end. Cast Elimination can then remove tests made useless.

While the code after the transformation is bigger, some instructions (`br $end`) can be removed on real examples.

4 Benchmarks

(4.0.1) *The methodology.* We tested our toolchain by running, on Bigloo's Wasm backend, a subset of the R7RS benchmarks [R7r], the standard benchmark suite for Scheme. We compare the execution time using watib with all of its optimisation passes enabled with the previous backend using wasm-as with no optimisation enabled.

(4.0.2) *The results.* Figure 6 shows the results of our benchmarks. The red line indicates the performance whith all of watib's optimisation passes enabled, and the blue bars indicates the performance of the previous backend. Our optimisation passes do not impact running time of most benchmarks. There is a noticeable negative impact on only `conform` which runs 10% slower with optimisations. Two benchmarks benefit from the optimisations: `early` and `fib` with a respective speedup of 17% and 10%. The overall lack of impact of our optimisations on these benchmarks is not surprising: the benchmark mode of the Bigloo compiler already removes type tests. Using Bigloo's safe mode shows a better performance gain.

We are investigating the reasons of the slowdown on `conform`; we found that:

- Disabling the Cast Elimination pass solves the issue. In fact, removing three useless casts from the original program is enough to have a program 10% slower. We do not remove this optimisation as it reduces code size and does not cause the same issues in other conditions.
- The SpiderMonkey engine does not suffer from this performance penalty.

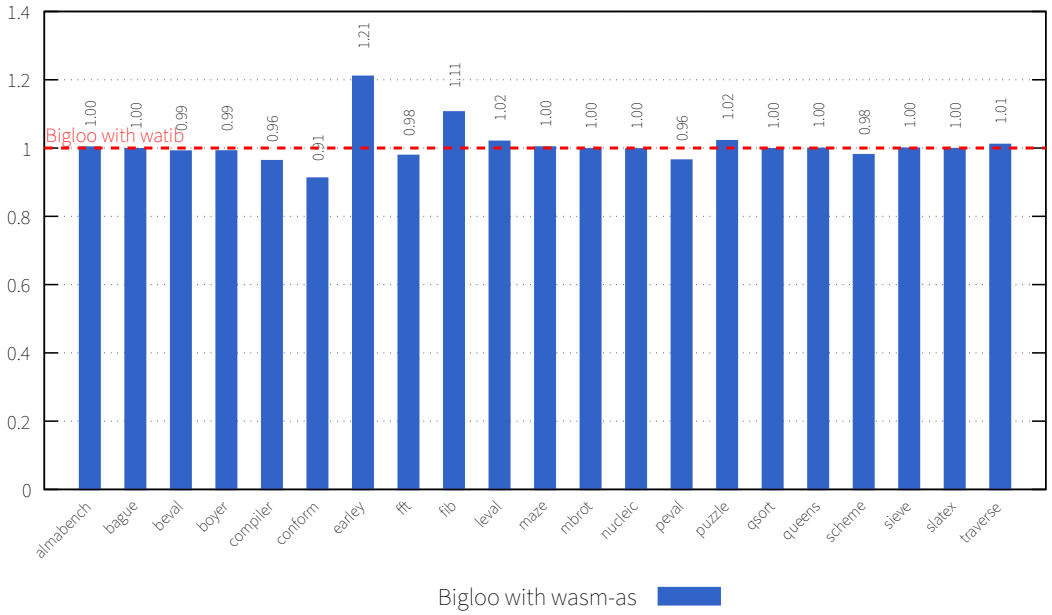

Figure 6: Benchmarks against the a subset of the R7RS benchmarks suite. The blue bars represent the ratio between the execution times after and before the introduction of watib (higher is better)

- V8's TurboFan, a multi-pass compiler, does not suffer either from this performance penalty.
- Other programs do not benefit from disabling cast elimination.

5 Static Basic Block Versioning

We present here a work-in-progress pass. This pass is not complete but works on toy programs.

(5.0.1) *Control-Flow Graphs.* Some optimisations and code transformations manipulate a *Control-Flow Graph* (CFG). It is a graph whose nodes are *basic blocks* (sequences of instructions that do not affect the control flow, here Wasm instructions) and arcs represent potential jumps. To reflect the typing discipline of Wasm, we consider CFGs whose basic blocks have instruction types.

(5.0.2) *The optimisation.* We are working to adapt Static Basic Block Versioning or SBBV [MFS24] (which works on CFGs) to WebAssembly. This optimisation associates to each basic block a context which contains information on the program at the beginning of the block (for instance wether some reference is null). This pass then duplicates some blocks

to reflect for each context. Each version can specialise its instructions to take into account the context (by removing redundant type tests for instance). The same basic block can be duplicated up to a fixed limit. When this limit is reached, the pass merges two versions of the block: it replaces two different context by a single one which contains less information than both.

In watib, SSBV could subsume Copy Propagation and Cast Elimination.

(5.0.3) *Construction of a control-flow graph.* Constructing a CFG from a Wasm program is straightforward except for try_table which watib does not support yet. One only needs to follow the program's control flow, while keeping a state of the types on the stack to annotate the basic blocks with types.

(5.0.4) *Reconstruction of Wasm.* Absence of a goto instruction complicates the translation of a CFG to a valid Wasm program. To reconstruct structured control flow, we implement the algorithm described in [Ram22]. This algorithm relies on the *dominance tree* of the CFG, which we compute using the algorithm of [CHK01]. We adapted Ramsey's algorithm to support br_on_cast as a way to jump out of a block and to use the type annotations on the CFG to type the Wasm blocks. The latter addition allows preserving block parameters and output values in the input program.

(5.0.5) *Reducibility.* Ramsey's algorithm works on *reducible* CFGs. Intuitively, a CFG is reducible when each loop has a single entry point. WebAssembly's structured control flow only produces such CFGs. But the SBBV we work on can modify the CFG of a program with no guarantee on reducibility. We found some programs where the CFGs obtained after applying SBBV were not reducible anymore.

(5.0.6) *Fixing non-reducibility.* We are considering two different ways to obtain a reducible CFG after static basic block versioning.

On the one hand, the GHC compiler, which uses Ramsey's algorithm, makes CFGs reducible through *node-splitting*. This technique duplicates problematic nodes so that each copy has a unique incoming arc, until the resulting graph is reducible. With this method, the reducible graph obtained executes exactly the same instructions as the original one, at the cost of a size expansion, which can be exponential [CFT03]. GHC implements node splitting [Ram22, Appendix A] with a greedy algorithm. A more elaborate method obtains better results [JC97].

On the other hand, non-reducibility appears when (1) SBBV duplicates the head of a loop and (2) arcs between the different versions of the same loop exist. We can then make the graph reducible by creating a new node that serves as a common head for all the versions of the loop. Instead of jumping to a head, a node would branch to this new head indicating which version of the loop it targets. The new head would then branch to one of the previous heads with a br_table. This approach solves size explosion problem but adds execution overhead.

We also plan to test not applying the optimisation to some parts of the graph where the optimisation generates a non-reducible CFG.

(5.0.7) *Keeping the graph reducible.* Node splitting turns a non-reducible CFG into a reducible one by duplicating basic blocks, but SBBV already duplicates some basic blocks. We are thus investigating a way to maintain a reducible CFG at each step of the algorithm. To do so, each basic block contains, in its corresponding context, a stack of the loops it belongs to. This allows forbidding jumping inside a loop from a basic block outside the loop (unless the destination is the loop's head). When such a jump should occur, the jumps target is duplicated.

(5.0.8) *Preliminary results.* We tested our implementation of SBBV on a toy program written to benefit from SBBV (see Appendix C). Applying this pass to the toy program gives a program 28% faster with V8 and 102% faster with SpiderMonkey, on an M1 Mac.

6 Conclusion

We reviewed and motivated some of watib's design choices and we compared it to existing toolchains. We also detailed the type checking algorithm, and the optimisation passes. Benchmarks showed that these passes are not sufficient to impact most code produced by Bigloo. To address this issue, we ended by presenting our ongoing implementation of SBBV for Wasm.

References

[Bina] *Binaryen*. https://github.com/WebAssembly/binaryen. Accessed: 2025-07-06.

[Binb] *Future of Binaryen in a stack machine world?* https://github.com/WebAssembly/binaryen/issues/663. Accessed: 2025-07-07.

[Bou] Aghilas Boussaa. *Wasm-as' extensions to the specification*. https://www.normalesup.org/~boussaa/watib/doc/wasm-as-spec.html. Accessed: 2025-07-9.

[Car88] Luca Cardelli. "A semantics of multiple inheritance". In: *Information and computation* 76.2-3 (1988), pp. 138–164.

[CFT03] Larry Carter, Jeanne Ferrante, and Clark Thomborson. "Folklore confirmed: reducible flow graphs are exponentially larger". In: *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2003, pp. 106–114.

[CHK01] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. "A simple, fast dominance algorithm". In: *Software Practice & Experience* 4.1-10 (2001), pp. 1–8.

[Ems] *emscripten*. https://emscripten.org/. Accessed: 2025-07-11.

[Haa+17] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 2017, pp. 185–200.

[JC97] Johan Janssen and Henk Corporaal. "Making graphs reducible with controlled node splitting". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.6 (1997), pp. 1031–1052.

[JS98] Simon L Peyton Jones and AndréL M Santos. “A transformation-based optimiser for Haskell”. In: *Science of computer programming* 32.1-3 (1998), pp. 3–47.

[Kot] *Kotlin/Wasm*. https://kotlinlang.org/docs/wasm-overview.html. Accessed: 2025-07-11.

[MFS24] Olivier Melançon, Marc Feeley, and Manuel Serrano. “Static Basic Block Versioning”. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2024, pp. 28–1.

[Muc97] Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[Oca] *Js_of_ocaml (jsoo)*. https://github.com/ocsigen/js_of_ocaml. Accessed: 2025-07-11.

[PC+23] Luna Phipps-Costin et al. “Continuing WebAssembly with effect handlers”. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA2 (2023), pp. 460–485.

[R7r] *R7RS Benchmarks*. github.com/ecraven/r7rs-benchmarks. Accessed: 2025-07-28.

[Ram22] Norman Ramsey. “Beyond Relooper: recursive translation of unstructured control flow to structured control flow (functional pearl)”. In: *Proceedings of the ACM on Programming Languages* 6.ICFP (2022), pp. 1–22.

[Ros23] Andreas Rossberg. “Mutually iso-recursive subtyping”. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA2 (2023), pp. 347–373.

[Ros25] *WebAssembly Core Specification*. Version 3.0 (Draft 2025-05-15). W3C, May 15, 2025. URL: https://webassembly.github.io/spec/versions/core/WebAssembly-3.0-draft.pdf.

[Rub] *ruby.wasm*. https://github.com/ruby/ruby.wasm. Accessed: 2025-07-11.

[Ser] Manuel Serrano. *Bigloo*. https://www-sop.inria.fr/indes/fp/Bigloo/. Accessed: 2025-07-06.

[Spi] *SpiderMonkey*. https://spidermonkey.dev/. Accessed: 2025-07-06.

[V8] *What is V8?* https://v8.dev/. Accessed: 2025-07-06.

[Was] *wasm-tools*. github.com/bytecodealliance/wasm-tools/. Accessed: 2025-07-06.

A Wasm-as's extensions to the specification

We try to list wasm-as's extensions to the spec, i.e., code modifications made by wasm-as silently that do not appear in the specification. To our knowledge, this behaviours are not documented. The list is non-exhaustive. Feel free to contact us if you find anything missing.

A.1 **Unreachable** insertion

Wasm-as replaces dead code by unreachable and add unreachable after blocks that do not exit. While semantically preserving, this transformation can transform invalid code in valid code. For instance, the following function is not well-typed according to the specification. But wasm-as inserts an unreachable at the end, making it well-typed.

```
(func (result i32)
  (if (i32.const 0)
    (then (return (i32.const 0)))
    (else (return (i32.const 0)))))
```

In the following piece of code, the br is not well-typed but wasm-as accepts it:

```
(func
  (result i32)
  (block $main (result i32)
    (i32.const 0)
    (if
      (then (return (i32.const 0)))
      (else (return (i32.const 0))))
    (br $main)))
```

A.2 Automatic function reference declaration

According to the specification, functions appearing in a ref.func have to appear outside the function bodies (in a global or in an elem section). Wasm-as puts all the undeclared function references in an elem declare func section.

A.3 Sign extension insertion for packed get

According to the specification, when using array.get or struct.get on a packed field (i8 or i16), a sign extension has to be specified. It means that either .get_u or .get_s has to be used. Wasm-as allows .get instructions on packed types, by replacing them with their .get_u version.

A.4 Replacement of block input values by locals

Binaryen's README mentions that block input values are represented in the IR by pop subexpressions for catch blocks and not supported for the others. It is not mentioned that code using block input values will be replaced by code using local variables instead. For instance, the following code:

```
(i32.const 0)
(block (param i32)
  (drop))
```

is assembled to:

```
(i32.const 0)
(local.set 0)
(local.get 0)
(drop)
```

B Matching rules for heap types

$$\frac{t_1 = t_2}{C \vdash t_1 \leq t_2} \qquad \frac{}{C \vdash \bot \leq t} \qquad \frac{C \vdash C.\mathsf{types}[x] \leq t}{C \vdash x \leq t} \qquad \frac{C \vdash t \leq C.\mathsf{types}[x]}{C \vdash t \leq x}$$

$$\frac{C \vdash t \leq \mathsf{any}}{C \vdash \mathsf{none} \leq t} \qquad \frac{C \vdash t \leq \mathsf{func}}{C \vdash \mathsf{nofunc} \leq t} \qquad \frac{C \vdash t \leq \mathsf{extern}}{C \vdash \mathsf{noextern} \leq t} \qquad \frac{C \vdash t \leq \mathsf{exn}}{C \vdash \mathsf{noexn} \leq t}$$

$$\frac{C \vdash t \leq \mathsf{any}}{C \vdash t \leq \mathsf{eq}} \qquad \frac{}{C \vdash \mathsf{i31} \leq \mathsf{eq}} \qquad \frac{}{C \vdash \mathsf{struct} \leq \mathsf{eq}} \qquad \frac{}{C \vdash \mathsf{array} \leq \mathsf{eq}}$$

$$\frac{\mathsf{unroll}(deftype) = (symbol \ldots)}{C \vdash deftype \leq symbol} \qquad \frac{\mathsf{unroll}(deftype) = (symbol \ldots) \qquad C \vdash symbol \leq \mathsf{eq}}{C \vdash deftype \leq \mathsf{eq}}$$

C A toy program to test SBBV

```
(module
  (type $boxed-int (struct (field i32)))
  (type $int-array (array i32))

  (func $f
```

```
6     (export "cfg")
7     (param $array (ref $int-array))
8     (result (ref null $boxed-int))
9     (local $i i32)
10    (local $len i32)
11    (local $acc (ref null $boxed-int))
12
13    (local.set $i (i32.const 0))
14    (local.set $len (array.len (local.get $array)))
15    (local.set $acc (ref.null $boxed-int))
16    (block $break
17       (loop $continue
18          (br_if $break (i32.ge_u (local.get $i) (local.get $len)))
19          (if $l (i32.eq (i32.const -1)
20                  (array.get $int-array (local.get $array)
21                     (local.get $i)))
22             (then (local.set $acc (ref.null $boxed-int)))
23             (else (block $br (result (ref $boxed-int))
24                     (br_on_non_null $br (local.get $acc))
25                     (local.set $acc
26                       (struct.new $boxed-int
27                         (array.get $int-array
28                            (local.get $array)
29                            (local.get $i))))
30                     (br $l))
31                  (drop)))
32          (local.set $i (i32.add (i32.const 1) (local.get $i)))
33          (br $continue)))
34    (local.get $acc)))
```