Mechanising and Verifying the WebAssembly Specification

Conrad Watt
University of Cambridge
United Kingdom
caw77@cam.ac.uk

Abstract

WebAssembly is a new low-level language currently being implemented in all major web browsers. It is designed to become the universal compilation target for the web, obsoleting existing solutions in this area, such as asm.js and Native Client. The WebAssembly working group has incorporated formal techniques into the development of the language, but their efforts so far have focussed on pen and paper formal specification.

We present a mechanised Isabelle specification for the WebAssembly language, together with a verified executable interpreter and type checker. Moreover, we present a fully mechanised proof of the soundness of the WebAssembly type system, and detail how our work on this proof has exposed several issues with the official WebAssembly specification, influencing its development. Finally, we give a brief account of our efforts in performing differential fuzzing of our interpreter against industry implementations.

CCS Concepts • **Theory of computation** → *Program semantics*;

Keywords soundness, reduction, bytecode, stack machine

ACM Reference Format:
Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3167082

1 Introduction

JavaScript's continued monopoly on the modern web has caused the ecosystem to develop in unintuitive ways. Despite its original design as a high-level scripting language, a significant proportion of JavaScript currently running on the web is generated by compilation from C/C++, targetting the asm.js subset [Herman et al. 2014]. JavaScript was never designed to be a low-level compilation target, and even the severe restrictions of asm.js, disallowing nearly all of its dynamic behaviour, are not enough to save asm.js code from significant performance penalties compared to native compilation [Zakai and Nyman 2013].

Furthermore, the ubiquity of asm.js has placed unexpected and arguably inappropriate pressures on the evolution of the JavaScript specification. For example, a proposal to expose native SIMD vector instructions and types has recently reached the final candidate stage for inclusion into the JavaScript specification [TC39 2017b]. SIMD instructions are normally only explicitly utilised at the level of native assembly, or in the lowest-level C/C++ programs to eke out a final few percent of performance in arithmetic-heavy algorithms. The proposed feature is motivated by a desire to increase the efficiency of C/C++ code compiled to JavaScript, but it also bloats the specification, complicates JavaScript engine optimisations, and is practically unusable in hand-coded scripts.

Attempts to offer a more appropriate language in the browser as a compilation target, most notably Google's Native Client [Google 2017], have until now met with low adoption rates and patchwork support across different browsers. WebAssembly [WebAssembly Community Group 2017c] is the first true cross-party offering in this area. It is just as much the result of a political process as it is a technical one: the WebAssembly working group contains representatives from Google, Microsoft, Apple, and Mozilla, and each of these companies have committed to fully supporting WebAssembly in their respective browsers.

WebAssembly is designed to be embedded within a *host environment*. The canonical example is a WebAssembly implementation running inside a web browser's JavaScript engine. When a host program invokes a WebAssembly function, the function executes in a sandbox that cannot access the host's wider state. The host program can either simply be a thin wrapper around a self-contained WebAssembly function, or can choose to call out to WebAssembly only at certain points where efficiency is desired. It is anticipated that the majority of WebAssembly code will be produced through compilation from C/C++, using the official Binaryen toolchain [WebAssembly Community Group 2017a].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CPP'18, January 8–9, 2018, Los Angeles, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5586-5/18/01…$15.00
https://doi.org/10.1145/3167082

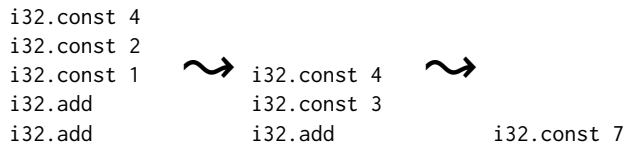

Figure 1. A trivial WebAssembly stack, reduced according to the language's small-step semantics.

```
i32.const n    ↦  EConst (ConstInt32 n)

i32.add        ↦  Unop_i T_i32 Add

loop
  e1           ↦  Loop ([] → [])
  e2                 [e1, e2, ...]
  ...
end
```

Figure 2. A selection of mappings from concrete opcodes (textual representation) to abstract operations (specification representation, in Isabelle).

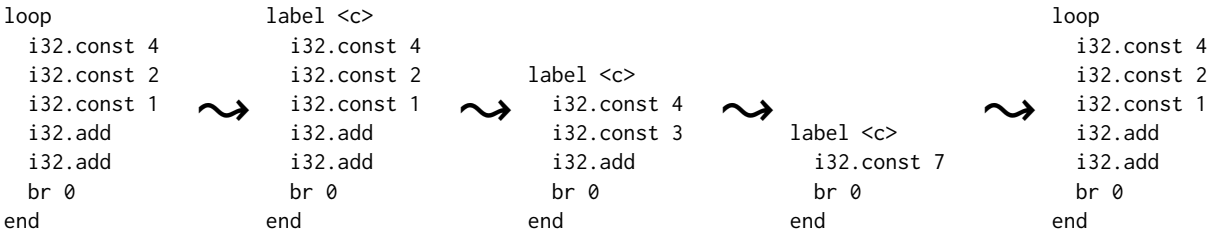

Figure 3. A WebAssembly stack illustrating the behaviour of the `loop` opcode.

The designers of WebAssembly have made it an explicit goal to specify the language in a way that makes it amenable to formal analysis and verification. The draft specification [WebAssembly Community Group 2017f] explains all semantic behaviour both in English prose, and with an accompanying natural deduction style formal rule. Moreover, members of the working group have recently produced a paper that details a non-mechanised formal semantics for a large core of the language [Haas et al. 2017a]. This paper, and the official specification, both state that the WebAssembly type system enjoys several soundness properties.

We have produced a full Isabelle mechanisation of the core execution semantics and type system of the WebAssembly language (§3). In addition, we have created a mechanised proof for the type soundness properties stated in the working group's paper. In order to complete this proof, several deficiencies in the official WebAssembly specification, uncovered by our proof and modelling work, needed to be corrected by the specification authors (§4). In some cases, these meant that the type system was originally unsound.

We have maintained a constructive dialogue with elements of the working group, mechanising and verifying new features as they are added to the specification. In particular, the mechanism by which a WebAssembly implementation interfaces with its host environment was not formally specified in the working group's original paper. Extending our mechanisation to model this feature revealed a deficiency in the WebAssembly specification that sabotaged the soundness of the type system.

We have also defined a separate verified executable interpreter (§6) and type checker (§5). Like many verified language implementations, these artefacts require integration with an external parser and linker to run as standalone programs, which introduces an untrusted interface. We use the official reference WebAssembly interpreter [WebAssembly Community Group 2017d], implemented in Ocaml, for this purpose. Our core proofs of correctness allow us to experimentally validate our mechanised specification using our executable interpreter, both by leveraging the official WebAssembly conformance test suite, and by conducting fuzzing experiments. These initial validation efforts are detailed towards the end of the paper (§7).

All Isabelle and OCaml code discussed in this paper is released publicly under a BSD-style license [Watt 2017a]. At a rough count, our work contains ~11,000 non-whitespace, non-comment lines of Isabelle code, with the mechanisation of the specification itself coming to ~700 lines of code.

2 The structure of WebAssembly

WebAssembly is a stack-based, bytecode language. Its execution semantics are naturally specified using a small-step reduction relation. Figure 1 is an illustration of the execution of a simple WebAssembly program according to this relation. This example is given in (slightly simplified) WebAssembly text format, an officially supported textual representation designed to present WebAssembly bytecode in a human-readable way.

```
— values
datatype ('a,'b,'c,'d)
  v =
    ConstInt32 'a
  | ConstInt64 'b
  | ConstFloat32 'c
  | ConstFloat64 'd

— basic expressions
datatype ('a,'b,'c,'d)
  b_e =
    Unreachable
  | Nop
  | Drop
  | Select
  | Block tf "(('a,'b,'c,'d) b_e) list"
  | Loop tf "(('a,'b,'c,'d) b_e) list"
  | If tf
      "(('a,'b,'c,'d) b_e) list"
      "(('a,'b,'c,'d) b_e) list"
  | Br i
  | Br_if i
  | Br_table "i ne_list"
  | Return
  | Call i
  | Call_indirect i
  | Get_local i
  | Set_local i
  | Tee_local i
  | Get_global i
  | Set_global i
  | Load t "(tp × sx) option" a off
  | Store t "tp option" a off
  | Current_memory
  | Grow_memory
  | EConst "('a,'b,'c,'d) v" ("C _" 60)
  | Unop_i t unop_i
  | Unop_f t unop_f
  | Binop_i t binop_i
  | Binop_f t binop_f
  | Testop t testop
  | Relop_i t relop_i
  | Relop_f t relop_f
  | Cvtop t cvtop t "sx option"

— value and function types
datatype
  t = T_i32 | T_i64 | T_f32 | T_f64
datatype
  tf = Tf "t list" "t list" ("_ '→ _" 60)

— global variables (with mutability flag)
datatype
  mut = T_immut | T_mut
record ('a,'b,'c,'d) global =
  g_mut :: mut
  g_val :: "('a,'b,'c,'d) v"

— packed types and signedness bit (for memory accesses)
datatype
  tp = Tp_i8 | Tp_i16 | Tp_i32
datatype
  sx = S | U

— function closures
datatype ('a,'b,'c,'d,'host)
  cl =
    Func_native i tf "t list" "(('a,'b,'c,'d) b_e) list"
  | Func_host tf 'host

— instances
record  ('a,'b,'c,'d,'host) inst =
  types :: "tf list"
  funcs :: "i list"
  tab :: "i option"
  mem :: "i option"
  globs :: "i list"

— function tables
type_synonym ('a,'b,'c,'d,'host) tabinst =
  "(('a,'b,'c,'d,'host) cl) option) list"

— the program store
record  ('a,'b,'c,'d,'meminst,'host) s =
  inst :: "(('a,'b,'c,'d,'host) inst) list"
  funcs :: "(('a,'b,'c,'d,'host) cl) list"
  tab :: "(('a,'b,'c,'d,'host) tabinst) list"
  mem :: "'meminst list"
  globs :: "(('a,'b,'c,'d) global) list"

type_synonym i = nat
datatype unop_i = Clz | Ctz | Popcnt
datatype unop_f = Neg | Abs | Ceil | Floor | Trunc | Nearest | Sqrt
datatype binop_i = Add | Sub | Mul | Div sx | Rem sx | And | Or | Xor | Shl | Shr sx | Rotl | Rotr
datatype binop_f = Addf | Subf | Mulf | Divf | Min | Max | Copysign
datatype testop = Eqz
datatype relop_i = Eq | Ne | Lt sx | Gt sx | Le sx | Ge sx
datatype relop_f = Eqf | Nef | Ltf | Gtf | Lef | Gef
datatype cvtop = Convert | Reinterpret
```

Figure 4. Core WebAssembly AST and supporting definitions, as they appear in our Isabelle model. Values, expressions, and components of the program store are parameterised by type variables so as to abstract the underlying representation of WebAssembly types, the heap, and the host environment.

```
— administrative instructions
datatype ('a,'b,'c,'d,'host)
  e  =
    Basic "('a,'b,'c,'d) b_e" ("$_" 60)
  | Trap
  | Callcl "('a,'b,'c,'d,'host) cl"
  | Label nat "(('a,'b,'c,'d,'host) e) list" "(('a,'b,'c,'d,'host) e) list"
  | Local nat i "(('a,'b,'c,'d) v) list" "(('a,'b,'c,'d,'host) e) list"

— Definitions (type variables elided)
inductive reduce_simple :: "[e list, e list] ⇒ bool" and
reduce :: "[s, v list, e list, nat, s, v list, e list] ⇒ bool"
— Abbreviations
⦇es⦈ ↪ ⦇es'⦈ ≡ reduce_simple es es'
⦇s;vs;es⦈ ↪_i ⦇s';vs';es'⦈ ≡ reduce s vs es i s' vs' es'
C ≡ EConst
$ ≡ Basic
```

$$\frac{⦇es⦈ \hookrightarrow ⦇es'⦈}{⦇s;vs;es⦈ \hookrightarrow_i ⦇s;vs;es'⦈}\ \texttt{Reduce\_Simple}$$

$$\frac{⦇s;vs;es⦈ \hookrightarrow_i ⦇s';vs';es'⦈ \quad \texttt{Lfilled k lholed es les} \quad \texttt{Lfilled k lholed es' les'}}{⦇s;vs;les⦈ \hookrightarrow_i ⦇s';vs';les'⦈}\ \texttt{Label\_Context}$$

$$\frac{\texttt{app\_binop\_i iop c1 c2 = None}}{⦇[\$i32.const\ c1,\ \$i32.const\ c2,\ \$(Binop\_i\ T\_i32\ iop)]⦈ \hookrightarrow ⦇[Trap]⦈}\ \texttt{Binop\_i32\_None}$$

$$\frac{}{⦇s;vs;[\$(Call\ j)]⦈ \hookrightarrow_i ⦇s;vs;[Callcl\ (sfunc\ s\ i\ j)]⦈}\ \texttt{Call}$$

$$\frac{\texttt{cl = Func\_native j (t1s \_> t2s) ts es} \quad \texttt{ves = (\$\$* vcs)} \quad \texttt{length vcs = length t1s} \quad \texttt{length t2s = m} \quad \texttt{zeros ts = zs}}{⦇s;vs;ves\ @\ [Callcl\ cl]⦈ \hookrightarrow_i ⦇s;vs;[Local\ m\ j\ (vcs@zs)\ [\$(Block\ ([]\ \_>\ t2s)\ es)]]⦈}\ \texttt{Callcl}$$

Figure 5. A small selection of reduction rules as they appear in our Isabelle model, re-formatted in a natural deduction style.

WebAssembly is not a large language. Its official specification defines 171 individual opcodes [WebAssembly Community Group 2017b], but each of these can be viewed as different specialisations of the 28 abstract operations specified in the working group's paper [Haas et al. 2017a]. All WebAssembly behaviour is specified at the level of abstract operations, not concrete opcodes.

WebAssembly makes it an explicit goal to eliminate undefined behaviour from the specification [Haas et al. 2017a]. Its heap is a linear array of bytes and all accesses are bounds-checked, and there is no integer-pointer distinction when addressing into it. In addition, WebAssembly value types have no trap representations; every sequence of 32 or 64 bits can be deserialised into a valid value of any type of the appropriate length. WebAssembly requires all programs to undergo validation (type checking) before they can be executed. This allows a number of properties to be statically checked. For example, it is a type error if an operation statically accesses a local variable index that is out of bounds.

The design of the language also attempts to minimise non-determinism. There are only two sources of non-determinism in "pure" WebAssembly. Firstly, the bit representation of NaN floating point values is not precisely specified, and implementations are free to have multiple NaN representations which they may choose to use non-deterministically. Secondly, the `grow_memory` instruction, which allocates additional memory to the heap, is allowed to fail non-deterministically.

The other source of non-determinism is interaction with the host environment. A WebAssembly implementation may invoke *host functions*, effectively API calls which transfer control to the host environment. The behaviour of these host functions is specified as entirely non-deterministic, subject to certain restrictions designed to preserve the integrity of the WebAssembly state, which will be discussed in more detail in our section on the soundness proof (§4).

2.1 The semantics of WebAssembly

Throughout this section, we will use extracts from our Isabelle mechanisation of the specification to illustrate fundamental components of the language. Unless otherwise stated, these definitions correspond exactly to those of the official specification [WebAssembly Community Group 2017f], aside from the minor syntactic differences inherent in mechanisation. Our Isabelle representation of the language's core AST can be found in Figure 4, and a mapping of some pertinent concrete opcodes to abstract operations of the AST is given in Figure 2.

As the inclusion of a "loop" opcode in this mapping may have hinted, WebAssembly's approach to control flow is unusual. Unlike most other bytecode languages, WebAssembly

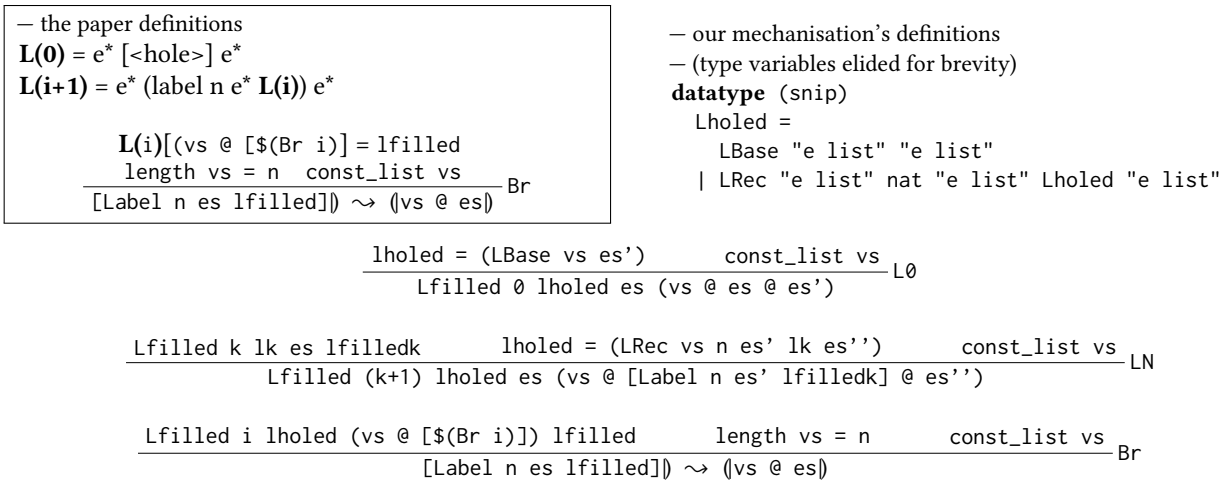

Figure 6. Comparing the paper definitions of nested execution contexts and the `Br` rule (top left) to those of our mechanisation.

does not contain any mechanism for unstructured control flow, such as branch or goto. Instead its binary format explicitly includes the `loop`, `if`, and `block` instructions, which create a form of structured control flow by bookending sections of the stack. This behaviour is illustrated in Figure 3. Notice how the original "linear" program of Figure 1 is bookended with the `loop` and `end` opcodes, which form an evaluation context. The first step of reducing the loop is to convert it to the "administrative instruction" `label`. Administrative instructions (hereafter, "administrative operations") are not part of the binary format. They are purely specification contrivances, extensions to the core abstract operations designed to keep track of intermediate state during a reduction. The conversion from loop to label effectively unrolls the loop once, with the (abbreviated) continuation element of the label, <c>, keeping track of the operations which make up the continuation of the loop. This becomes relevant when paired with the `br` instruction.

The `br` instruction is officially named "branch" in the specification, but it functions more like a combination of the JavaScript break and continue statements: `br n` transfers execution to end of the nth innermost evaluation context (zero indexed), effectively popping all intervening instructions from the stack. When a label is targetted in this way, its continuation element is then pushed onto the stack. This has the effect of bringing the loop to its next iteration. If control falls off the end of the label without executing a `br` instruction, the continuation is discarded and the loop therefore terminates. WebAssembly's unique approach to control flow is a significant component of the challenge in proving type soundness and correctness properties of implementations, especially since a single `br` instruction can break multiple loops at once (although only the outermost, "targetted" loop will have its continuation pushed).

When relating the `loop` instruction to the `Loop` abstract operation (Figure 2), notice how a function type annotation `([] → [])` is introduced. In the current WebAssembly specification, control flow opcodes are not allowed to manipulate values which occur outside their own inner context [Haas et al. 2017a]. Therefore, the type of the arguments to the abstract operation will always be the empty list. However, allowing these constructs to accept arguments is a planned future feature, and therefore the abstract operation supports this.

The original paper formalisation defines 46 reduction rules for the language. Our mechanisation defines 65; we type-specialise some arithmetic and bitwise operations in order to provide a cleaner interface for code extraction, and we implement host function behaviours that the paper formalisation did not support. A small selection of reduction rules and an AST extension for administrative operations can be found in Figure 5. The administrative operations are `Trap`, which represents an unrecoverable error, `Callcl` (also named `Invoke` in some versions of the specification) which represents the invocation of a function closure, `Label`, which represents a section of code that can be broken out of by `Br`, and `Local`, which represents the local context of an invoked function.

The full reduction relation for WebAssembly is defined between *configurations*, which consist of a stack of abstract operations, together with a program store and a list of local variables, and an instance index which indicates which parts of the global store are "owned" by the currently running program. For example, the `Call` rule pushes a closure object onto the stack selected using the `sfunc` operation, which looks up the appropriate closure based on the current instance index together with the call index. Parts of the reduction relation may be elided where they are not involved in a particular rule. As an example, the `Binop_i32_None` rule describes the execution of a binary arithmetic operation which results in a

```
record t_context =
  types_t :: "tf list"
  func_t :: "tf list"
  global :: "tg list"
  table :: "nat option"
  memory :: "nat option"
  local :: "t list"
  label :: "(t list) list"
  return :: "(t list) option"
```

```
— definition
inductive b_e_typing :: "[t_context, (('a,'b,'c,'d) b_e) list, tf] ⇒ bool"
— abbreviation
(C ⊢ es : tf) ≡ b_e_typing C es tf
```

$$\frac{\texttt{is\_int\_t t}}{C \vdash \texttt{[Binop\_i t \_]} : \texttt{([t,t]} \rightarrow \texttt{[t])}}\,\texttt{Binop\_i}$$

$$\frac{\texttt{tf = (tn} \rightarrow \texttt{tm)} \qquad C(\texttt{|label := ([tn] @ (label } C\texttt{))|}) \vdash \texttt{es : (tn} \rightarrow \texttt{tm)}}{C \vdash \texttt{[Loop tf es] : (tn} \rightarrow \texttt{tm)}}\,\texttt{Loop}$$

$$\frac{\texttt{i < length(label } C\texttt{)} \qquad \texttt{(label } C\texttt{)!i = ts}}{C \vdash \texttt{[Br i] : (t1s @ ts} \rightarrow \texttt{t2s)}}\,\texttt{Br} \qquad \frac{C \vdash \texttt{es : (t1s} \rightarrow \texttt{t2s)}}{C \vdash \texttt{es : (ts @ t1s} \rightarrow \texttt{ts @ t2s)}}\,\texttt{Weakening}$$

$$\frac{C \vdash \texttt{es : (t1s} \rightarrow \texttt{t2s)} \qquad C \vdash \texttt{[e] : (t2s} \rightarrow \texttt{t3s)}}{C \vdash \texttt{es @ [e] : (t1s} \rightarrow \texttt{t3s)}}\,\texttt{Composition}$$

Figure 7. A selection of typing rules and definitions as they appear in our Isabelle model.

runtime error (e.g. division by zero), and does not depend on the current instance, the store, or local variables. The `Callcl` rule describes the behaviour of invoking a function closure:a new local context is created for execution, containing the body of the function and its local variables.

Our mechanisation differs from the official specification in one minor way. The specification's reduction semantics define an execution/evaluation context structure of a single hole surrounded by nested labels. This structure is dependently typed; one of the structure's elements is a number which must be equal to the number of labels recursively nested within itself. This number is used in defining the reduction rule for a `Br i` instruction, which must break out of exactly `i` nested labels. Expressing this directly requires type-level arithmetic. Since Isabelle does not support this, we implement the dependent restrictions on the structure using a "well-formed" predicate, `Lfilled`. These two approaches are contrasted in Figure 6. Notice how the ultimate definition of the `Br` reduction rule in our mechanisation differs in its replacement of the **L(n)** execution context with an equivalent `Lfilled`. Intuitively, `Lfilled k lholed es les` can be read as "`les` is the result of filling in the `k`-nested execution context `lholed` with `es`". In practice, all semantic rules and proofs can be expressed in terms of `Lfilled`, and the underlying structure of the execution context is only inspected during our proofs.

2.2 The type system of WebAssembly

A small selection of WebAssembly's typing rules can be found in Figure 7. WebAssembly has four concrete value types, corresponding to 32- and 64-bit floats and integers. Every value in WebAssembly has one of these four types. Opcodes have a type of the form `(t* → t*)`, effectively a function from list of values to list of values. These opcode types compose as expected, and the type of a stack of instructions is therefore also `(t* → t*)`, the composition of its constituent operations. All typing derivations are with respect to a type context which represents the typing information of the instance the program is running in.

Configurations are typed with respect to one of the instances contained in their store. A configuration containing a stack of type `([] → t*)` under instance i can be said to have type t* in i. The type of a WebAssembly program is the type of its initial configuration.

Some WebAssembly typing rules are highly nondeterministic. Figure 7 gives the typing rule for `Br`. The "input" type of the `Br` operation is partially determined by the label element of the typing context, which is extended while typing the inner part of a "breakable" operation (see the typing rule for `loop`). However, the type of the result, `t2s`, is completely arbitrary. This is because the `Br` instruction guarantees that all subsequent operations in the same execution context will never be executed. However, it is still possible for this dead code to be ill-typed. Consider the stack of Figure 8. The left-hand stack is well-typed, since the type of `Br` can "fill in" the missing integer argument. However, no matter what type is picked when typing `Br` in the right-hand stack, the mismatching integer and float types of the subsequent operations cause the stack to be ill-typed. The implications of this design are relevant to our verified type checker.

The type system is claimed in the official specification to be sound with respect to the stack reduction relation, in the sense that well-typed programs enjoy progress and preservation properties during execution.

Lemma (Preservation). *Given a configuration* ⦇s;vs;es⦈ *with type* t^* *in* i, *if* ⦇s;vs;es⦈ ↪_i ⦇s';vs';es'⦈ *then* ⦇s';vs';es'⦈ *also has type* t^* *in* i.

Lemma (Progress). *If a configuration* ⦇s;vs;es⦈ *has type* t^* *in* i, *then either* es *is a bare* Trap *representing an exception, or a list of constant values, or there exists* ⦇s';vs';es'⦈ *such that* ⦇s;vs;es⦈ ↪_i ⦇s';vs';es'⦈

WebAssembly's type system is very similar (most likely unwittingly) to the "stack effect calculus" [Poial 1990], a decades-old type system initially proposed as part of an effort to formalise the Forth language. This is mostly a historical curiosity, as despite Forth's similarity to WebAssembly as another stack-based language with structured control flow, most existing formal work on Forth is not applicable to WebAssembly due to the decision to model control flow using a separate control stack [Power and Sinclair 2004] [Knaggs 1993] , whereas in WebAssembly all values, operations and control instructions are held (at least abstractly) in the same stack. However, recent discussions about the typing ramifications of adding a hypothetical "dup" opcode to WebAssembly dovetail neatly with existing theory on more polymorphic variants of the stack effect calculus [Poial 2002].

3 The model

We have built a full mechanisation of the core WebAssembly specification as it appears in the working group's original paper [Haas et al. 2017a], extended with features and behaviour added to the official draft specification [WebAssembly Community Group 2017f] after its publication. This mechanisation adheres as strictly as possible to the ideals of "eyeball closeness", first explicitly advocated for by JSCert [Bodin et al. 2014], a mechanisation of the ES5 JavaScript specification. Eyeball closeness is a design principle of the formal model such that there is a line to line textual correspondence between the official specification and the mechanisation. In the ideal case, someone familiar with the official specification should be able to read an eyeball close mechanisation as though it is re-stating the specification in an unfamiliar pseudocode. Compared to JSCert, our model enjoys a significant advantage in preserving eyeball closeness in that all WebAssembly reduction and typing rules in the specification already include a definition in formal notation. This means that our definitions can be eyeball close at the level of specification logical sentence to mechanisation logical sentence, not merely specification English sentence to mechanisation logical sentence.

The core of the mechanisation is our definition of two inductive relations, which correspond to the WebAssembly specification's reduction and typing rules. These relations are not directly executable, but we define separate executable functions for an interpreter and type checker, and prove them correct with respect to their corresponding relation.

```
block                    block
  br 0                     br 0
  i32.const 1              i32.const 1
  i32.add                  f32.const 0
  drop                     i32.add
end                        drop
                         end
```

Figure 8. Two WebAssembly stacks illustrating the typing behaviour of Br. Only the left-hand stack is well-typed.

Our work does not formalise the mapping of concrete opcodes to abstract operations, as we consider this to be part of work of the parser, which we do not model. This is consistent with the way the official specification specifies the structure of the binary format. Because of this, our model already has full support for loops with non-empty arguments, even though this feature is not available yet due to the restrictions in place on the binary format.

Similarly, as previously mentioned, we do not model the instantiation process of a WebAssembly module. This is a linking and allocation phase that must be carried out before a WebAssembly program can be executed. Instead, our mechanisation deals purely with the WebAssembly execution environment in its post-instantiation form, often referred to as an *instance*. Within the official specification, both instantiation and parsing are self-contained sections and therefore our decision not to support them had no negative effect on the rest of our mechanisation.

There are a small number of situations where the original WebAssembly paper and the draft specification document differ in their representation of certain specification artefacts. We try to primarily follow the paper's representation, since it is more directly designed as a formal specification. In some cases, however we must adopt the draft specification's representation in order to model a feature which the paper did not support, or to improve compatibility with the untrusted parser/linker that we use to make our verified interpreter executable as a standalone program, since its internal state is more heavily based on the draft specification.

In particular, the paper formalisation stores functions declared within WebAssembly programs within each instance itself. This is possible because the formalisation gives only a sketch description of the instantiation process. In the full draft specification, multiple instances may share the same store, and one may export a function that is imported by another. Instances cannot directly access each other, and therefore a single copy of the function can be held directly in the store, with each instance maintaining a reference to it. Using this arrangement instead of the paper's (which effectively makes a copy of an imported function every time a new instance is created) more accurately represents the instantiated state in real implementations, and allows our

verified interpreter to interface with the official reference interpreter's instantiation mechanism, since we do not model it ourselves.

A perennial concern when defining a mechanised specification is accurately handling arithmetic, especially floating point calculations. We use Isabelle's locale mechanism to abstract the implementation of arithmetic and the heap as parameters. Any proofs completed within the locale give results that are implicitly quantified over all possible implementations, so long as they satisfy the locale assumptions, which encode the properties of the heap that we rely on.

We also use locales to abstract the behaviour of the host environment, allowing our proofs to range over all possible hosts. We slightly restrict the behaviour of host functions compared to the full behaviour allowed by the latest official specification. We allow host functions to arbitrarily mutate the heap, but not the function table or the list of declared global variables. We have found that this behaviour is sufficient to pass all available WebAssembly conformance tests, and execute all WebAssembly programs encountered "in the wild" so far.

Another advantage of locales is that they force us to be explicit in the assumptions we make about the behaviour of untrusted code interfacing with our verified interpreter. For example, when carrying out code extraction for integration with the official reference interpreter, we must explicitly axiomatise our assumption that the native OCaml code we interface with has well-behaved host functions.

4 Soundness

We have produced a fully mechanised proof of both soundness properties (Figure 9) as they are stated in the working group's original paper. Prior to our work, no proof of soundness for the type system, mechanised or otherwise, was available. Each property was proven by induction, over either the reduction or the typing relation.

To prove each of these properties, a large number of auxiliary lemmas needed to be established. This was most notable when dealing with the inductive cases that involved recursive execution contexts mixed with control flow. Figure 10 shows one of these lemmas, which must be proven by induction over the definition of the `Lfilled` predicate. The lemma is a generalised version of a property that is required to prove the progress property; a `Br n` instruction is only well-typed if it is surrounded by at least `n+1` breakable labels.

In the course of conducting this proof, we identified several errors in the official specification which were acknowledged and fixed by members of the working group. In some cases, these errors meant that the type system was originally unsound. We detail the most significant examples here.

```
theorem preservation:
  assumes "⊢_i s;vs;es : ts"
          "(|s;vs;es|) ↝_i (|s';vs';es'|)"
  shows "⊢_i s';vs';es' : ts"

theorem progress:
  assumes "⊢_i s;vs;es : ts"
  shows "const_list es ∨
         es = [Trap] ∨
         (∃s' vs' es'. (|s;vs;es|) ↝_i (|s';vs';es'|))"
```

Figure 9. The preservation and progress properties, as they appear in our Isabelle proof.

```
lemma progress_LN1:
  assumes "(Lfilled j lholed [$Br (j+k)] es)"
          "S·C ⊢ es : (ts → ts')"
  shows "length (label C) > k"
```

Figure 10. An auxiliary lemma in the progress property proof.

4.1 Exception propagation

As previously mentioned, the WebAssembly semantics has a `Trap` administrative operation which is used to model an unrecoverable exception. A `Trap` value is generated by runtime errors such as division by `0` or out-of-bounds memory accesses. Once it is generated, all other execution ceases, and the `Trap` value propagates through all function calls and nested control structures, terminating the program when it reaches the top of the stack. However, the original draft specification we based our model on did not allow the `Trap` value to propagate as intended, and the reduction could incorrectly become stuck before `Trap` reached the top of the stack, violating the progress property.

We discovered this issue while attempting to prove the progress property, and were able to communicate a counterexample to the specification authors, as well as a suggested solution, which was ultimately adopted into the specification.

4.2 Return

Originally, the `Return` operation was simply specified as a "maximal" break. That is, it would reduce to `Br n`, where `n` was the number of nested labels within the current function call. However, we discovered that the typing rule given was incorrect, with the effect that a `Return` operation could occur outside a function call and still be well typed. The informal intention of the specification was that such a program should be rejected by validation. Determining the correct fix was an extended process, and we assisted a member of the working group by modelling several possible solutions. The final solution involved changing the structure of the `Label` and `Local` operations to keep track of the arity of the value to be returned from their inner context, as well as altering

the semantics of `Return` so that it now breaks directly to the outside of the function call instead of being defined in terms of `Br`.

A former draft of the semantics where these two issues are still present can be found on the official GitHub repository [Haas et al. 2017b].

4.3 Host functions

Since host functions may behave in arbitrary, nondeterministic ways that fall outside the space of possible WebAssembly behaviours, they are required to preserve certain invariants on the runtime state, so that they do not dynamically invalidate an assumption that would otherwise always hold throughout "normal" execution. For example, a host function may deallocate portions of memory, or change the type of an immutable global variable. We discovered that the invariants as they were stated in the specification were too weak, so that a host function could obey them yet still cause a well-typed program to crash [Rossberg 2017a], in violation of the progress property. Ultimately, this section of the specification was entirely re-written to fix this problem [Rossberg 2017b].

5 An executable type checker

We have defined, separate from our model, an executable type checker in Isabelle, and proven it sound and complete with respect to the inductive typing relation of our mechanisation. We did not use Isabelle's tools to extract executable code from the original typing relation (a process known as animation), because we desired our type checker to match the behaviour of industry implementations by running in a single pass.

As previously mentioned, one quirk of WebAssembly's type system is that it requires the full stack to be typed, even if a control instruction guarantees that a certain portion of the stack is dead code. This is accomplished by giving the section of stack terminated by the control instruction an arbitrary type, but it is still possible for the remainder of the stack to be ill-typed. This has implications for a concrete type checking algorithm. WebAssembly's type system is intended to be checkable in one pass over the stack. However, this requires a richer representation of the types of intermediate sections of the stack than is possible using the normal WebAssembly syntax. Upon scanning to, for example, a `br` instruction, a type checker with no polymorphic symbols does not have the information it needs to pick the arbitrary concrete type that will allow the rest of the stack to be well-typed. If, instead, the type checker is allowed access to some symbol representing an unconstrained type, it can progress, constraining the type of the stack as it encounters further instructions.

```
datatype ct =
    TAny
  | TSome t
```

```
datatype checker_type =
    TopType "ct list"
  | Type "t list"
  | Bot
```

Figure 11. An extended type syntax, for use with the executable type checker.

```
theorem b_e_typing_equiv_b_e_type_checker:
  shows "(C ⊢ es : (tn → tm)) =
           (b_e_type_checker C es (tn → tm))"
```

Figure 12. Equivalence of our type checker with our inductive typing relation.

We implement a single-pass type checker which internally types the stack using an extended version of the WebAssembly type syntax which includes polymorphic symbols. We introduce no more polymorphism than is necessary to facilitate single-pass typing, so as to simplify the proofs. The definition of our extended type syntax can be found in Figure 11.

Polymorphism occurs at two levels. First, an individual value type on the stack can be polymorphic. This is represented by the `TAny` type. Second, the stack itself can be polymorphic in its contents, including length. The only way the stack itself can become polymorphic is by executing an unconditional control flow instruction, which makes the whole stack unconstrained for the purposes of typing. `TopType` represents this unconstrained stack, which may include certain constrained types appended to its head, including polymorphic value types, by subsequent instructions. Currently, the only way to generate a single polymorphic value type is to inspect an unconstrained stack with the `Select` operation. Therefore, a stack that does not have an unconstrained base cannot include polymorphic value types, and therefore the `Type` stack type, representing the type of an exact stack, is a list of `t` rather than `ct`. Finally, `Bot` represents a stack with no valid type.

The type checking algorithm walks the stack from top to bottom, operation by operation. Each operation produces and consumes a certain number of (potentially polymorphic) type symbols, or makes the entire stack type `TopType []`, the entirely unconstrained stack. `Bot` is produced if the correct type cannot be consumed, or if some other condition is not satisfied, such as a `Br` operation attempting to break out of more labels than are present.

This extended type syntax can be viewed as a constraint system with the solutions being types in the form of the original syntax. We prove that, for all WebAssembly programs, the types that satisfy our model's typing relation are exactly the solutions to the constraints our type checker generates. Finally, we define a top level function which takes a type-annotated WebAssembly program, and runs our type

checker, checking that the type annotations satisfy the resulting constraints. This function therefore returns true if and only if the program is well-typed, and can be automatically used by Isabelle's code generation tools as an efficient, executable version of our model's typing relation.

Our type checker does not perform full type inference, since we are allowed to assume during implementation that the "initial" stack type, at the start the typing pass, is an exact type. However, this could be an interesting future extension.

6 An executable interpreter

We have implemented an executable interpreter as an Isabelle function, and proven it sound with respect to the mechanised specification. Again, we decided against attempting to directly animate the reduction relation for several reasons. Firstly, the definition of reduction given in the specification makes use of inductively defined evaluation contexts. Our mechanisation stays faithful to this in order to maximise eyeball closeness. However, this arrangement does not lend itself to direct animation, since there is no syntax-directed way of determining which part of a given WebAssembly stack is the evaluation context and which part takes the place of the "hole". Secondly, the semantics for exception propagation contain a significant amount of trivial, confluent nondeterminism. A direct animation would require handling this; for example, by representing the possible reductions in a choice monad, despite the different choices having no effect on the final result. Finally, our relational definition of reduction makes use of a number of highly inefficient list manipulations, which even a "naive" interpreter would not carry out.

Instead, we chose to define a separate executable interpreter as a function from WebAssembly configuration to result, and prove it sound with respect to the reduction relation. In doing this, we enjoy the dual advantages of having a specification in a form conducive to proofs, and a potentially optimising interpreter which avoids the performance pitfalls of a direct animation of the reduction relation. Indeed, our interpreter implements a more efficient representation of constant values on the stack, which reduces the frequency with which the stack (as a list of operations) needs to be split and concatenated during execution.

The core of the executable interpreter is a one step evaluation function which mirrors the one step reduction relation of the specification. For most operations, the function proceeds exactly as in the reduction relation. The reduction rules focussing on execution contexts (see the `Label_Context` rule of Figure 5) translate naturally into a simple recursive definition. The main complications in this strategy are the control flow instructions `Br` and `Return`. For example, the `Br` instruction is defined as breaking out of all enclosing labels in a single step. We therefore extend the evaluation function to return a `res_step` (Figure 13) , which can either be a stack

```
datatype res_crash =
  CError
| CExhaustion

datatype ('a,'b,'c,'d) res =
  RCrash res_crash
| RTrap
| RValue "(('a,'b,'c,'d) v) list)"

datatype ('a,'b,'c,'d,'e) res_step =
  RSCrash res_crash
| RSBreak nat "(('a,'b,'c,'d) v) list)"
| RSReturn "(('a,'b,'c,'d) v) list)"
| RSNormal "(('a,'b,'c,'d,'e) e) list)"
```

Figure 13. Interpreter result types.

after one step of reduction, or a special "control result" which signals either that an outer call of the evaluation function is for a label which is being broken to, or that an error has occured.

Invocations of the evaluation function which result in a new stack can be directly proven to be sound with respect to the reduction relation. In order to prove the behaviour of the evaluation function sound for control flow instructions which involve one of these control results as a return value, certain auxiliary lemmas must be established. Figure 14 shows one of these lemmas. This lemma relates an `RSBreak` control result to the underlying structure of the evaluated code. In particular, it encodes that an `RSBreak n res` control result must have originated from a `Br n'` operation nested `(n'-n)` labels deep, with exactly the constant values `res` directly preceding it on the stack. `Lfilled_exact` is a tweaked definition of `Lfilled` used exclusively during the soundness proof. Unlike `Lfilled`, `Lfilled_exact` exposes all constant values existing on the stack at the same level as the "hole", in this case, `res`, allowing them to be explicitly referred to during the proof.

At the top level, our interpreter repeatedly calls the one step evaluation function until either a result or error is reached. We turn "getting stuck" (having no reduction according to the semantics) into an explicitly signalled error. Because all functions in Isabelle require a termination proof, we augment this interpreter with a standard fuel value which decreases each iteration. The WebAssembly specification requires all implementations to gracefully implement a limit on the level of nested function calls, so we separately keep track of a depth parameter which decreases every time a function context is entered. A function executed with a remaining depth of 0 results in a simulated stack exhaustion error.

Our executable interpreter cannot run as a stand-alone function, since, as previously explained, we do not model instantiation, or decoding of the binary format. Therefore, to obtain a complete WebAssembly engine, we use Isabelle's

```
lemma run_step_break_imp_lfilled:
  assumes "run_step d i (s,vs,es) = (s', vs', RSBreak n res)"
  shows "s = s' ∧
         vs = vs' ∧
         (∃n' lfilled es_c. n' ≥ n ∧
                            Lfilled_exact (n'-n) lfilled ((vs_to_es res) @ [$Br n'] @ es_c) es)"
```

Figure 14. Relating the `RSBreak` control result to an evaluation context of nested labels.

extraction mechanism to extract our interpreter function to OCaml, and integrate it with the working group's reference WebAssembly implementation, also written in OCaml. This requires that we implement an untrusted interface with the reference implementation's AST. For convenience, we also make use of the reference interpreter's internal implementation/representation of integers, floating point, and the heap. In principle, these could be replaced with Isabelle defined and extracted definitions, but the correctness of these definitions is entirely orthogonal to our soundness results, and the reference implementation makes use of Ocaml-native types that are more efficient.

7 Validation and fuzzing

Our executable interpreter, suitably augmented with the reference parser and linker, successfully passes all core language conformance tests available in the WebAssembly repository [WebAssembly Community Group 2017e]. Due to the soundness result we have with respect to our mechanised specification, these tests also serve to validate our model.

In addition to this, we have conducted differential testing of our executable interpreter against several major WebAssembly engines. This was done both with the purpose of validating our interpreter, and potentially discovering semantic bugs in commercial WebAssembly engines. Tests were generated using the CSmith tool [Yang et al. 2011], combined with the official Binaryen toolchain [WebAssembly Community Group 2017a] to convert the generated C tests into WebAssembly. This mimics how most WebAssembly programs will be produced "in the wild". No errors were found either in our implementation or any commercial engine, although a crash bug was discovered within the Binaryen toolchain itself, which was reported and fixed by the developers [Watt 2017b].

8 Related work

Our mechanisation draws heavily from the original formalisation by the WebAssembly working group [Haas et al. 2017a]. However, this formalisation is purely handwritten, contains no support for interaction with the host environment, and offers no proof of the two soundness properties it claims the type system enjoys. To the best of our knowledge, our work represents the first mechanised formalisation of the complete WebAssembly core language, as well as the first full proof, mechanised or otherwise, of the soundness of the WebAssembly type system.

The structure and organisation of our proofs and executable artefacts owe a great debt to the JSCert project [Bodin et al. 2014], which mechanised the ES5 JavaScript specification. JSCert separated its specification and executable implementation in order to offer a specification which could be used to build proofs about language properties without sacrificing eyeball closeness, which its contributors argued was necessary to relate proven properties back to the original textual specification. We consider our project to have benefited significantly from following JSCert's lead in this arrangement: our work in proving WebAssembly's type soundness properties identified several important issues with the official specification which we could not have discovered without embarking on such a "deep" proof of a language property, and might not have been so immediately actionable by the official specification authors had we not maintained eyeball closeness. Furthermore, we can now guarantee, through our proofs, that the type system is sound in a way that would not be possible for a "light-weight" specification.

The CakeML project [Kumar et al. 2014] includes formal models of several intermediate, assembly-like languages. Due to WebAssembly's positioning as a web-compatible, platform-independant compilation target, it may be fruitful to investigate a WebAssembly backend for the CakeML compiler. Indeed, its maintainers acknowledge this as a possible direction for the project [CakeML project 2017].

The Java Virtual Machine and bytecode have been extensively formalised, with a mechanisation existing in Isabelle [Klein and Nipkow 2006]. As previously discussed, the Java bytecode does not share WebAssembly's approach to control flow, and therefore the comparisons we can make to their model are limited. An extension to this work [Lochbihler and Bulwahn 2011] uses locales to abstract over memory consistency models, similar to the way we use the same feature to abstract over different implementations of the WebAssembly heap.

As previously discussed, some formal work has been done on the Forth language [Knaggs 1993]. To the best of our knowledge, no mechanisation work exists.

9 Future work

To reduce our reliance on the official reference interpreter in making our verified interpreter executable, it would be

valuable to model the instantiation phase of WebAssembly program execution. This is an area of the specification undergoing active re-writing [Rossberg 2017c], so it would be challenging to keep the model abreast of ongoing changes.

A major incoming feature for WebAssembly is integration with the SharedArrayBuffer proposal [TC39 2017a], which adds a weak memory semantics to both WebAssembly and JavaScript. Correct specification of weak memory models is a major open problem. The models of many languages have been shown to have fundamental deficiencies [Batty et al. 2015], including the C++11 memory model, which WebAssembly must be "compatible" with in some sense, if it is intended to be a compilation target. This is an area where timely input from the formal verification community could have a tangible impact on the health of the language. In particular, it is important to work out to what extent the memory model of the SharedArrayBuffer proposal falls foul of known problems in the field, such as the issue of "thin air" executions.

Finally, Ethereum developers have announced eWASM [Ethereum Group 2017], a proposal to use WebAssembly as the bytecode representation of programs running on the Ethereum virtual machine. Our existing model needs almonst no extension to be a faithful model of this new virtual machine, and there is scope for useful verification work, particularly in verifying translations from the old bytecode representation, which has also been recently modelled [Hirai 2017], to WebAssembly.

10 Summary

We have presented a full mechanisation of the core WebAssembly language, together with several proofs of soundness, and verified implementations of a type checker and interpreter. In the course of conducting these proofs, we have identified and assisted in fixing several errors in the official WebAssembly specification.

Acknowledgments

Conrad Watt is supported by an EPSRC Doctoral Training award, and the REMS EPSRC program grant (EP/K008528/1). We thank Peter Sewell and Andreas Rossberg for their advice during this work.

References

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. *The Problem of Programming Language Concurrency Semantics.* Springer Berlin Heidelberg, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14).* ACM, New York, NY, USA, 87–100. https://doi.org/10.1145/2535838.2535876

CakeML project. 2017. CakeML Projects. (2017). Retrieved October 7, 2017 from https://cakeml.org/projects

Ethereum Group. 2017. eWASM. (2017). Retrieved October 7, 2017 from https://github.com/ewasm

Google. 2017. Welcome to Native Client. (2017). Retrieved October 7, 2017 from https://developer.chrome.com/native-client

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017a. 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.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017b. Bringing the Web Up to Speed with WebAssembly. (March 2017). Retrieved October 7, 2017 from https://github.com/WebAssembly/spec/blob/bbb26c42b62096baff86089767531c3b1f108a85/papers/pldi2017.pdf

David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js. (August 2014). Retrieved October 7, 2017 from http://asmjs.org/spec/latest

Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *1st Workshop on Trusted Smart Contracts.*

Gerwin Klein and Tobias Nipkow. 2006. A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.* 28, 4 (July 2006), 619–695. https://doi.org/10.1145/1146809.1146811

Peter J Knaggs. 1993. Towards a Formal Forth. (September 1993). Retrieved October 7, 2017 from http://www.rigwit.co.uk/papers/formal2.pdf

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14).* ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Andreas Lochbihler and Lukas Bulwahn. 2011. Animating the Formalised Semantics of a Java-like Language. In *Proceedings of the Second International Conference on Interactive Theorem Proving (ITP'11).* Springer-Verlag, Berlin, Heidelberg, 216–232. http://dl.acm.org/citation.cfm?id=2033939.2033958

Jaanus Poial. 1990. Algebraic Specification of Stack Effects for Forth Programs. In *Forml Conference Proceedings (Forml 1990).* The Forth Interest Group, San Jose, CA, USA, 282–290.

Jaanus Poial. 2002. Stack effect calculus with typed wildcards, polymorphism and inheritance. In *Proc. 18-th EuroForth Conference*.

James F. Power and David Sinclair. 2004. A Formal Model of Forth Control Words in the Pi-Calculus. *J. UCS* 10, 9 (2004), 1272–1293. https://doi.org/10.3217/jucs-010-09-1272

Andreas Rossberg. 2017a. [spec] Fix and clean up invariants for host functions. (September 2017). Retrieved October 7, 2017 from https://github.com/WebAssembly/spec/pull/563

Andreas Rossberg. 2017b. [spec] Fix and clean up invariants for host functions. (September 2017). Retrieved October 7, 2017 from https://github.com/WebAssembly/spec/commit/772d87705ea4786c0d44d41902097e91cf31f82b

Andreas Rossberg. 2017c. [spec] Fix and clean up invariants for host functions. (September 2017). Retrieved October 7, 2017 from https://github.com/WebAssembly/spec/pull/563

TC39. 2017a. Memory Model. (September 2017). Retrieved October 7, 2017 from https://tc39.github.io/ecma262/#sec-memory-model

TC39. 2017b. SIMD.js specification v0.9. (April 2017). Retrieved October 7, 2017 from http://tc39.github.io/ecmascript_simd/

Conrad Watt. 2017a. Conrad Watt. (October 2017). Retrieved November 27, 2017 from http://www.cl.cam.ac.uk/~caw77/

Conrad Watt. 2017b. Crash in wasm-opt on certain optimisation levels. (August 2017). Retrieved October 7, 2017 from https://github.com/WebAssembly/binaryen/issues/1149

WebAssembly Community Group. 2017a. Binaryen. (October 2017). Retrieved October 7, 2017 from https://github.com/WebAssembly/binaryen

WebAssembly Community Group. 2017b. Instructions. (September 2017). Retrieved October 7, 2017 from https://webassembly.github.io/spec/binary/instructions.html

WebAssembly Community Group. 2017c. WebAssembly. (2017). Retrieved October 7, 2017 from http://webassembly.org

WebAssembly Community Group. 2017d. WebAssembly. (October 2017). Retrieved October 7, 2017 from https://github.com/WebAssembly/spec/tree/master/interpreter

WebAssembly Community Group. 2017e. WebAssembly. (October 2017). Retrieved October 7, 2017 from https://github.com/WebAssembly

WebAssembly Community Group. 2017f. WebAssembly Specification. (September 2017). Retrieved October 7, 2017 from https://webassembly.github.io/spec/

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Alon Zakai and Robert Nyman. 2013. Gap between asm.js and native performance gets even narrower with float32 optimizations. (December 2013). Retrieved October 7, 2017 from https://hacks.mozilla.org/2013/12/20/