Two Mechanisations of WebAssembly 1.0

Conrad Watt[1(✉)], Xiaojia Rao[2], Jean Pichon-Pharabod[1], Martin Bodin[2,3], and Philippa Gardner[2]

[1] University of Cambridge, Cambridge, UK
conrad.watt@cl.cam.ac.uk
[2] Imperial College London, London, UK
[3] Inria, Rocquencourt, France

Abstract. WebAssembly (Wasm) is a new bytecode language supported by all major Web browsers, designed primarily to be an efficient compilation target for low-level languages such as C/C++ and Rust. It is unusual in that it is officially specified through a formal semantics. An initial draft specification was published in 2017 [14], with an associated mechanised specification in Isabelle/HOL published by Watt that found bugs in the original specification, fixed before its publication [37].

The first official W3C standard, WebAssembly 1.0, was published in 2019 [45]. Building on Watt's original mechanisation, we introduce two mechanised specifications of the WebAssembly 1.0 semantics, written in different theorem provers: WasmCert-Isabelle and WasmCert-Coq. Wasm's compact design and official formal semantics enable our mechanisations to be particularly complete and close to the published language standard. We present a high-level description of the language's updated type soundness result, referencing both mechanisations. We also describe the current state of the mechanisation of language features not previously supported: WasmCert-Isabelle includes a verified executable definition of the instantiation phase as part of an executable verified interpreter; WasmCert-Coq includes executable parsing and numeric definitions as on-going work towards a more ambitious end-to-end verified interpreter which does not require an OCaml harness like WasmCert-Isabelle.

Keywords: Mechanised specification · Type soundness · WasmCert

1 Introduction

WebAssembly (Wasm) is a new bytecode language, primarily designed as a compilation target for low-level languages such as C/C++ and Rust. It is supported by all major Web browsers, allowing programs compiled to Wasm to be embedded in Web pages and executed client-side. Because a Web site may attempt to execute arbitrary Wasm code in a visitor's browser, the language is designed around strict principles of encapsulation. A Wasm program is made up of one or more modules, which can only interact with the wider system through explicitly declared imports and exports. Moreover, the language defines a strict, static type system, and all programs must be type-checked before execution.

M. Huisman et al. (Eds.): FM 2021, LNCS 13047, pp. 61–79, 2021.
https://doi.org/10.1007/978-3-030-90870-6_4

Wasm's official specification includes a formal semantics for the language, with a precise statement of the intended type soundness property. This formal approach to specification is unusual for a language created by industry. It was first published in an initial draft in 2017 [14], and then in the official standard, called WebAssembly 1.0 (Wasm 1.0), in 2019 [45].

Our goal is to develop a mechanised specification of Wasm 1.0 and verify the associated type soundness result. Many people have explored the mechanisation of real-world language specifications, especially using the Isabelle/HOL and Coq theorem provers; see Sect. 5 on related work. The accurate mechanisation of a standard can be difficult because the language may be huge and continually evolving (e.g. JavaScript [5]), underspecified (e.g. C [24]), and/or tricky to state precisely (e.g. relaxed memory concurrency [3]). Wasm 1.0 is an interesting target for mechanised specification because it is small, stable and formally specified.

We present two mechanised specifications of Wasm 1.0: WasmCert-Isabelle, written in the Isabelle/HOL theorem prover, and WasmCert-Coq, in the Coq theorem prover[1]. We prove type soundness for both specifications. Our work builds directly on Watt's mechanised specification in Isabelle/HOL of the 2017 draft semantics [37], which discovered and corrected bugs in the original specification and the statement of type soundness [37]. The Wasm 1.0 standard currently cites Watt's old work as the source of the language's type soundness proof [45]. Watt's mechanisation follows a methodology for establishing trust in mechanised specification, developed as part of the JSCert project, a Coq-mechanisation of JavaScript [5]. Watt's mechanised specification of Wasm's type checker and the runtime semantics is line-by-line close to the standard, an achievement made easier by Wasm's formal semantics. In addition, a separate type checker and executable interpreter are mechanised, with the type checker proven correct with respect to the mechanised type system, and the interpreter proven correct with respect to mechanised runtime semantics. These executable definitions are independently tested using the official test suite.

We pool our substantial experience with Wasm and mechanised language specification to develop our Wasm 1.0 mechanised specification. Our contributions are:

– WasmCert-Isabelle: our Isabelle/HOL mechanisation of the Wasm 1.0 semantics which extends and refactors the mechanisation of Watt [37], including the type soundness proof, type checker, and interpreter, to Wasm 1.0. This includes a number of editorial changes to the runtime semantics (Sect. 2), and an additional condition in the statement of type soundness (Sect. 3). We also give a verified mechanisation of the instantiation phase of the Wasm execution, which was not included in the 2017 draft semantics nor Watt's original mechanisation (Sect. 4). This work is also reported in Watt's PhD thesis [38].
– WasmCert-Coq: our fresh Coq mechanisation of the Wasm 1.0 semantics, which closely follows the structure of WasmCert-Isabelle. We include a mechanised proof of type soundness and document the common high-level proof

[1] Our mechanisations are distributed under open source licences on GitHub [36].

structure between the mechanisations (Sect. 3). We also include executable mechanisations of WebAssembly's numeric operations and binary decoding phase, which were not formalised in the 2017 draft semantics nor mechanised by Watt, and are not yet handled by WasmCert-Isabelle (Sect. 4). We also report on progress towards the verification of a more ambitious end-to-end executable interpreter which does not require an OCaml harness (see below).

We summarise our WasmCert-Isabelle and WasmCert-Coq specifications, contrasting them with Watt's original work. We mark ✗ to indicate a feature or proof which is not included, ✓ to mark a feature or proof which has been fully mechanised, ✓✓ to indicate that a feature is accompanied by a verified executable definition capable of OCaml extraction, and ✓✓+ to indicate additionally that the executable definition has been validated through full end-to-end execution of the Wasm 1.0 test suite.

	Watt 2018	WasmCert-Isabelle	WasmCert-Coq
Wasm 1.0 refactorings	✗	✓	✓
type system	✓✓+	✓✓+	✓✓
runtime semantics	✓✓+	✓✓+	✓✓
type soundness proof	✓	✓	✓
binary decoding	✗	✗	✓✓
numeric ops	✗	✗	✓✓
instantiation	✗	✓✓+	✓

The 2017 draft semantics and Watt's original mechanisation did not cover three main areas of the language: the binary decoding phase, where a Wasm program distributed as bytecode is decoded into the program AST; numeric operations, most notably floating-point operations; and the instantiation phase, which runs after decoding but prior to execution and performs linking and state allocation. Watt's extracted interpreter relied on an OCaml harness to fill in these three areas with unverified implementations. WasmCert-Isabelle continues to model the decoding phase and numeric operations as uninterpreted functions, but mechanises the Wasm 1.0 specification of instantiation and allocation, including a verified executable implementation, thus significantly reducing the size of the unverified OCaml harness. This implementation was non-trivial due to the standard's circular definition of instantiation (see Sect. 4). We validate WasmCert-Isabelle's extracted interpeter against the Wasm 1.0 official test suite [39].

WasmCert-Coq takes a more ambitious approach to its in-progress verified interpreter. It uses CompCert's mechanised int and float libraries [7,8] to implement Wasm 1.0 numerics, and the Parseque parser combinator library [1,2] to mechanise Wasm 1.0's binary decoding phase. It provides an executable interpreter and type checker, both proven correct with respect to the mechanised semantics. It includes a mechanisation of the instantiation phase and an associated mechanised implementation, not yet shown to be correct. We consider this proof to be high-priority future work, along with end-to-end testing using the Wasm 1.0 test suite.

2 Wasm 1.0 Core Semantics

We describe the runtime semantics and type system of Wasm 1.0, highlighting where Wasm 1.0 diverges from the draft formalisation [14] and summarising the WasmCert-Isabelle and WasmCert-Coq mechnaisations of this core semantics.

2.1 Core Concepts

Values. Wasm operations manipulate values of four fundamental *value types*: i32, i64, f32, and f64. These are 32- and 64-bit integers and floats, respectively. Every program value in Wasm has one of these types. Functions are not first-class, and must be called in a first-order, fully applied manner, with a limited mechanism for dynamic dispatch which incurs additional runtime checks.

The Stack. Wasm is a stack-based language. Each function call is associated with its own *value stack*, abstractly represented as a list of values, and operations within the call will push and pop values to/from the stack in the course of computation. Wasm's stack is governed by a course-grained dataflow type system similar to that of the Java Virtual Machine [22], which ensures that the shape of the stack is statically known at every program point. The value stack *only* contains values - the function's return address is not programmatically accessible.

Modules. The *module* is Wasm's unit of compilation. A module contains a list of Wasm function declarations, as well as declarations of "global" state which is accessible to any function (described below). Modules may share global state with each other through a system of explicit imports and exports.

Globals. Wasm modules may declare or import *global variables.* A global variable has a statically declared value type, and is accessible by any function within the module. A global variable may be exported, allowing it to be accessible through other modules. Individual Wasm functions may also declare *local variables*, which are scoped to their declaring function, and are either initialised with one of the function arguments, or default to 0 otherwise.

Memory. Wasm modules may declare or import a *memory.* A Wasm memory is a simple buffer of bytes. A value may be stored in memory: this converts the value to a list of bytes and stores it at a provided offset. Similarly, a value may be loaded from memory, by interpreting the list of bytes at a provided offset as a value of the requested type. Wasm guarantees that this process always succeeds and is stable in both directions - unlike C, values have no trap representations.

Table. Wasm modules may declare or import a function *table*. Functions stored in the table can be called by dynamic dispatch, although the function's type signature must be checked at runtime.

Control Flow. Within a function, Wasm does not provide any arbitrary `goto` instruction. Instead, Wasm provides explicit **block**, **loop**, and **if** labelled/scoped constructs, together with a **br** (break) instruction which may target any enclosing label. This style of control flow is sometimes called *semi-structured*, and is common in higher-level languages such as Java and JavaScript. Wasm's label constructs (**block**, **loop**, **if**) are explicitly type-annotated with the shape of the value stack at their beginning and end, to preserve the invariant that all control flow paths to a program point result in the same stack shape, and to simplify type checking. Control may only be transferred out of a function through a **call** to another function, or through a **return** to the calling function.

2.2 Abstract Syntax

We give a brief overview of Wasm's abstract syntax. In formal definitions, some components are greyed out (here is an example) to indicate that we do not describe them in detail. While we must partially elide some definitions for space reasons, they are included in full in our mechanisations, and the exhaustive pen-and-paper formalism can be found in the Wasm 1.0 specification [45].

Figure 1 contains the formal definitions of Wasm's core abstract syntax. In the abstract syntax it suffices to model static *immediates* as natural numbers, although they are often concretely restricted to a 32-bit range. Many parts of the Wasm state are referenced through immediates representing De Bruijn indices, rather than through explicit names. The four value types are self-explanatory. Function types are used by control constructs and functions in Wasm, which must be explicitly type-annotated to describe how the shape of the value stack changes across their execution. The pre-execution validation pass checks that these annotations are correct. An annotation ft describes the shape of the top of the stack before and after execution of the construct/function.

We briefly describe Wasm instructions. The (t.**const** c) instruction pushes the value c of type t onto the value stack. The *stackop* instructions provide pure operations for pushing and popping values to and from the value stack. For example, the **i32.add** instruction pops two i32 values from the stack and pushes the i32 result of their unsigned wrap-around addition. The validation pass ensures that instructions only pop values which are guaranteed to be on the stack at that program point.

The **local** and **global** instructions deal with accessing local and global variables. For example, (**local.get** i) pushes the current value of the i-th declared local variable of the function onto the stack, while (**global.set** i) pops a value from the stack and assigns it to the i-th declared global variable of the module.

The (t.**load**) instruction pops an i32 value from the stack to use as an index into the module's memory, and pushes a value of type t that is deserialised from the bytes at that location. The (t.**store**) instruction pops a value of type t and

$$
\begin{array}{lrcl}
\text{(immediates)} & i, min, max & ::= & nat \\
\text{(value types)} & t & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\text{(func/block types)} & ft & ::= & t^* \rightarrow t^*
\end{array}
$$

$$
\begin{array}{lrcl}
\text{(instructions)} & e & ::= & t.\textbf{const}\ c \mid \textbf{i32.add} \mid \textit{other stackops} \mid \\
 & & & \textbf{local.}\{\textbf{get/set}\}\ i \mid \textbf{global.}\{\textbf{get/set}\}\ i \mid \\
 & & & t.\textbf{load}\ \textit{flags} \mid t.\textbf{store}\ \textit{flags} \mid \\
 & & & \textbf{memory.size} \mid \textbf{memory.grow} \mid \\
 & & & \textbf{block}\ ft\ e^* \mid \textbf{loop}\ ft\ e^* \mid \textbf{if}\ ft\ e^*\ e^* \mid \\
 & & & \textbf{br}\ i \mid \textbf{br\_if}\ i \mid \textbf{br\_table}\ i^+ \mid \\
 & & & \textbf{call}\ i \mid \textbf{call\_indirect}\ i \mid \textbf{return}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{(functions)} & func & ::= & \textbf{func}\ i\ \ t^*\ e^* \\
\text{(globals)} & glob & ::= & \textbf{glob}\ \mathsf{mutable}^?\ t\ \ e_{\mathsf{init}} \\
\text{(memories)} & mem & ::= & \textbf{mem}\ min\ max \\
\text{(tables)} & tab & ::= & \textbf{tab}\ min\ max
\end{array}
$$

$$
\begin{array}{l}
\text{(modules)}\ \ m\ ::= \\
\quad \{\ \text{types} :: ft^*,\ \text{funcs} :: func^*,\ \text{globs} :: glob^*,\ \text{mems} :: mem^*,\ \text{tabs} :: tab^*, \\
\quad\ \ \text{data} :: \ldots\ ,\ \text{elem} :: \ldots\ ,\ \text{imports} :: \ldots\ ,\ \text{exports} :: \ldots\ \ \}
\end{array}
$$

WasmCert-Isabelle: `wasm_ast.thy` and `wasm_module.thy`
WasmCert-Coq: `datatypes.v`

Fig. 1. Wasm abstract syntax

an i32 index, and serialises the value into bytes at that location in memory. Both of these instructions have slight variant behaviours which are configured using intra-instruction flags, the details of which we elide. The indices provided to these instructions are dynamically bounds-checked against the size/length of the module's memory. The current size of memory can be explicitly checked through **memory.size**, and grown through **memory.grow**.

Wasm provides only *semi-structured* control flow constructs. Its **block**, **loop**, and **if** instructions define break targets. Code within the body of one of these constructs can target the construct with one of the **br** family of instructions, which works like the `break` instruction of a higher-level language, with **br_if** and **br_table** being forms of conditional break. Multiple nested constructs are allowed, with (**br** k) instruction breaking to the k-th enclosing label.

The (**call** k) instruction calls the k-th function declared or imported by the current module. The (**call_indirect** k) instruction is Wasm's dynamic dispatch call. It pops an i32 index j from the stack, and the function stored in the table at index j is called. The static index k references a type annotation declared by the module, and the dynamically indexed function is checked to have this type annotation. Execution is halted with an error if the check fails. The **return** instruction ends the current call, and returns control to the caller, possibly pushing values onto the caller's stack corresponding to the function's output type.

The top-level Wasm module contains the declarations of globally-scoped state (functions, globals, memories, tables). Functions must carry an explicit type annotation, which is encoded as an immediate i referencing a canonical list of function types held by the module. This indirection to a canonical list was newly introduced in Wasm 1.0 to shift the abstract syntax of the language to more closely mirror its bytecode format, which has a distinguished "types" declaration section. Globals must declare whether they are mutable, their value type, and an optional initialiser expression which is executed at start-up. Memories and tables must declare their minimum and maximum sizes (although currently only the memory can be grown, the table size will be used by future features [41]). Note that in Wasm 1.0 a module may only declare/import at most one memory and one table in total, although this restriction will be lifted in future.

2.3 Runtime Semantics

WebAssembly's runtime semantics is specified in terms of a small-step reduction of *configurations*. A Wasm configuration is of the form $S; F; e^*$, where S is the execution-wide *store*, F is the *frame* of the current function, and e^* is the code fragment comprising the list of intructions currently under execution.

$$\begin{array}{ll} \text{(store)} & S ::= \{\ \text{funcs} :: \mathit{finst}^*,\ \ \text{globs} :: \mathit{ginst}^*,\ \ \text{mems} :: \mathit{minst}^*,\ \ \text{tabs} :: \mathit{tinst}^*\ \} \\ \text{(frame)} & F ::= \{\ \text{locs} :: v^*,\quad \text{inst} ::= \mathit{inst}\ \} \end{array}$$

The store contains all the module state which has been created in the course of execution. Its fields hold the runtime representations of the globally-scoped state declared by the constituent modules of the executing program (see Sect. 2.2). We elide the precise definitions of these components, but their structures are runtime versions of the static declarations made by the module, as shown in Fig. 1. The frame holds information relevant to the currently executing function. It holds the current values of local variables (in a list representation), and the *instance*. The instance tracks which components of the store are in scope for the current function (because they were declared/imported by the function's enclosing module). We elide the further details of its formal structure.

The value stack is not given explicitly as part of Wasm's runtime configuration. Instead, the value stack is represented in each reduction rule through a leading list of **const** instructions. For example, the reduction rule for **i32.add** is:

$$S; F; (\textbf{i32.const}\ j)(\textbf{i32.const}\ k)(\textbf{i32.add}) \hookrightarrow S; F; (\textbf{i32.const}\ (j + k))$$

This reduction rule represents the consumption of two stack values j and k by the **i32.add** instruction, and the production of the stack value $j + k$. This computation leaves the store S and the frame F unchanged.

This configuration was refactored in the move from the draft specification to the Wasm 1.0 standard. Originally, the frame was not an explicit component of the configuration. Instead, all executing instances were held as an additional list field in the store, and reduction was parameterised by an integer indexing this list, denoting the instance used by the current executing code fragment.

Mechanisation. Our mechanisations of the Wasm 1.0 runtime semantics, the executable interpreter and the correctness proof can be found as follows:

Mechanisation of the reduction rules which define the runtime semantics:

WasmCert-{Isabelle/Coq}: `reduce` in {`wasm.thy`/`opsem.v`}

Mechanisation of an executable interpreter:

WasmCert-Isabelle:	`wasm_interpreter.thy`
WasmCert-Coq:	`interpreter_func.v`

Proof that the interpreter is sound with respect to the reduction rules:

WasmCert-Isabelle:	`wasm_interpreter_properties.thy`
WasmCert-Coq:	`interpreter_func_sound.v`

WasmCert-Isabelle's interpreter definitions and proofs are based on those of Watt [37]. We refactor the interpreter to use the Wasm 1.0 definition of configuration, as discussed above. Orthogonally, we significantly simplify the Isabelle/HOL proof of interpreter soundness, removing ∼800 lines of code from the original proof due to better use of high-level Isabelle tactics.

2.4 Validation

Wasm programs must be *validated* before they can be executed. The validation involves a type-checking pass which checks the correctness of function and block type annotations, and enforces the following properties for code in the module:

- Operations which pop from the value stack (such as **i32.add**) are guaranteed that the value stack will contain the values necessary to allow the pop;
- Operations which access state using a static index are checked to ensure that the index is in the bounds, e.g. every (**global.get** i) instruction is checked to ensure that at least $i + 1$ global variables have been declared/imported;
- **br** instructions must target an enclosing label construct, and the shape of the value stack at the point of the **br** must match construct's type annotation.

The typing judgement for a Wasm code fragment has the shape $C \vdash e^* : \mathit{ft}$, associating a list of Wasm expressions e^* with a function type ft in typing context C. The definition of the typing context and some selected typing rules are shown in Fig. 2. The typing context C tracks the types of state (e.g. global variables) which have been declared by the enclosing module and are available in the current environment, as well as currently in-scope label (for **br**) and return (for **return**) targets. We elide the full definitions of some fields of the typing context which are not required to understand the examples in this paper. The local component of the context C holds the types of the declared local variables as a list which is indexed by instructions such as **local.get**. The label component of C holds the list of break targets currently defined by the enclosing program context. Its structure is a list of stack types (list of list of value types). Each stack type represents the required shape of the stack at the point the break target is broken to. The syntax $\{\text{label } t_2^*\} \oplus C$ in the typing of **block** describes the addition of the entry t_2^* to the left of C's label list. When a **block** is targetted by **br**, execution jumps to the end of the block, so the block's output type is inserted into the label context. The **br** instruction counts outwards through enclosing contexts to determine its break target, and therefore requires its input type to match the required type of the k-th enclosing label. The return component of C functions similarly. When typing a function, the return component is set to the output type of the function, and is used for typing the **return** instruction. Note though that the return component is optional. When typing a top-level configuration, the return type is set to empty, to denote that the code inside cannot return out of the top level.

Mechanisation. Our mechanisations of the inductive typing rules of the Wasm type system (see Fig. 2) can be found as follows:

WasmCert-Isabelle:	`b_e_typing` in `wasm.thy`
WasmCert-Coq:	`be_typing` in `typing.v`

Mechanisation of an executable type checker:

WasmCert-Isabelle:	`wasm_checker.thy`
WasmCert-Coq:	`type_checker.v`

Proof that the type checker is correct with respect to the typing rules:

WasmCert-Isabelle:	`wasm_checker_properties.thy`
WasmCert-Coq:	`type_checker_reflects_typing.v`

$$C \;::=\; \left\{ \begin{array}{l} \text{type} :: \mathit{ft}^*, \text{func} :: \mathit{ft}^*,\ \text{table} :: \mathit{tt}^*,\ \text{memory} :: \mathit{mt}^*,\ \text{global} :: \mathit{gt}^*, \\ \text{local} :: t^*,\ \text{label} :: (t^*)^*,\ \text{return} :: (t^*)^? \end{array} \right\}$$

$$\frac{}{C \vdash t.\textbf{const}\ c : \epsilon \to t} \qquad \frac{}{C \vdash \textbf{i32.add} : \mathsf{i32}\ \mathsf{i32} \to \mathsf{i32}} \qquad \frac{C.\text{local}[i] = t}{C \vdash \textbf{local.get}\ i : \epsilon \to t}$$

$$\frac{\mathit{ft} = t_1^* \to t_2^* \qquad \{\text{label}\ t_2^*\} \oplus C \vdash e^* : \mathit{ft}}{C \vdash \textbf{block}\ \mathit{ft}\ e^*\ \textbf{end} : \mathit{ft}} \qquad \frac{C.\text{label}[i] = t^*}{C \vdash \textbf{br}\ i : t_1^*\ t^* \to t_2^*}$$

Fig. 2. Selected typing rules.

3 Wasm 1.0 Type Soundness

The Wasm typing judgement described above is used by the standard as the basis of a standard statement of *syntactic type soundness* [46]. This involves defining an extended typing rule for runtime configurations [30] of the form $\vdash_{\mathsf{c}} S; F; e^* : t^*$. This judgement associates a configuration with a stack type (using the typing judgement of Sect. 2.4 as we will see below), and the type soundness properties state that execution will either diverge, terminate with a runtime error (such as division by zero), or terminate with a value stack corresponding to the type t^*. Judgement definitions are found in our mechanisations here:

WasmCert-Isabelle: `wasm.thy` WasmCert-Coq: `typing.v`

We give the high-level structure of important judgements, and the type soundness theorems. As previously mentioned, the formal definitions of the configuration and frame were refactored as part of the move from the draft specification to Wasm 1.0, with knock-on effects for the definitions of the associated judgements.

Configuration Validity. This is the top-level judgement in defining type soundness.

$$\frac{\vdash_{\mathsf{s}} S : \mathsf{ok} \quad S; \epsilon \vdash_{\mathsf{loc}} F; e^* : t^*}{\vdash_{\mathsf{c}} S; F; e^* : t^*}$$

Premise $\vdash_{\mathsf{s}} S : \mathsf{ok}$ tracks well-formedness conditions on the store that must be preserved as language invariants (e.g. memories may not exceed their max size).

Local Validity. This types an instruction sequence under a given function frame.

$$\frac{S \vdash_{\mathsf{f}} F : C \qquad S; C \vdash e^* : \epsilon \to t^*}{S; \epsilon \vdash_{\mathsf{loc}} F; e^* : t^*}$$

The typing context C under which the instruction sequence is typed is determined by the frame validity judgement defined below. Note that instruction sequence typing is defined via a slightly extended version of the judgement shown in Sect. 2.4, which is also parameterised by the store. This extension is necessary to type certain intermediate reducts which appear during execution.

Frame Validity. This judgement associates a frame with a type context.

$$\frac{(\ \mathrm{typeof}(v) = t_v\ )^n \qquad F.\mathrm{locs} = v^n \qquad S \vdash_{\mathsf{i}} F.\mathrm{inst} : C}{S \vdash_{\mathsf{f}} F : C[\mathrm{local}\ :=\ t_v^n]}$$

The premise $S \vdash_{\mathsf{i}} F.\mathrm{inst} : C$ builds an initial type context with the parts of the store that are in scope according to the instance component of frame F (full details elided). The context associated with the frame in the conclusion of the frame validity judgement is built from this initial type context, extended with the types of the local variables held by the frame. The premise $(\ \mathrm{typeof}(v) = t_v\ )^n$ abuses superscript notation to indicate that v^n is related to t_v^n by an element-wise mapping of the typeof relation.

Theorem 1 (preservation).
If $\vdash_{\mathsf{c}} S; F; e^* : t^*$ and $S; F; e^* \hookrightarrow S'; F'; e'^*$, then $\vdash_{\mathsf{c}} S'; F'; e'^* : t^*$ and $S \prec_{\mathsf{s}} S'$
WasmCert-Isabelle: `preservation` in `wasm_soundness.thy`
WasmCert-Coq: `t_preservation` in `type_preservation.v`

Theorem 2 (progress).
If $\vdash_{\mathsf{c}} S; F; e^* : t^*$, then $\mathsf{is\text{-}terminal}(e^*) \vee \exists S' F' e'.\ S; F; e^* \hookrightarrow S'; F'; e'^*$.

WasmCert-Isabelle: `progress` in `wasm_soundness.thy`
WasmCert-Coq: `t_progress` in `type_progress.v`

The $\prec_{\mathsf{s}}$ relation used when stating preservation is called *store extension*, and is an additional strengthening of the type soundness statement in the 1.0 specification, compared to that of Haas et al. [14]. Its presence in the preservation property enforces that the store cannot have elements removed as a result of execution (can only grow), and that previously allocated global state cannot change its type (even if the configuration would remain well-typed overall).

We first address the proof of the preservation property. In order for the induction to succeed, the inductive hypothesis must be strengthened so that instead of considering only the type preservation of a top-level configuration, we consider the type preservation of an arbitrary program fragment.

Lemma 1 (fragment preservation).

Assuming $S; F; e^* \hookrightarrow S'; F'; e'^*$
$\vdash_{\mathsf{s}} S : \mathit{ok}$
$S \vdash_{\mathsf{f}} F : C$
$S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash e^* : \mathit{tf}$

we have $S \prec_{s} S'$
$\vdash_{\mathsf{s}} S' : \mathit{ok}$
$S' \vdash_{\mathsf{f}} F' : C$
$S'; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash e'^* : \mathit{tf}$

WasmCert-Isabelle: `types_preserved_e2` in `wasm_properties.thy`

WasmCert-Coq: `t_preservation_e`, `reduce_inst_unchanged`, `store_extension_reduce` in `type_preservation.v`

Note the inclusion of $S \prec_{\mathsf{s}} S'$ in the conclusion, which as discussed is a new proof obligation introduced as part of the move to Wasm 1.0. The proof proceeds by induction on the definition of the reduction relation $\hookrightarrow$. The arbitrary label component l_{arb} appended to the type context C indicates that the code fragment e^* is potentially only well-typed when embedded inside some larger context of labelled control flow constructs (i.e. **block**, **loop**, **if**). The arbitrary return component r_{arb} indicates that the code is potentially only well-typed when embedded within a function definition with some arbitrary return type. We can then show that this stronger property implies the top-level preservation property. A similar generalisation must be made in proving the progress property:

Lemma 2 (fragment progress).

Assuming $S; C[\text{label} := l_{arb}, \text{return} := r_{arb}] \vdash e^* : t^* \rightarrow t'^*$
$C \vdash v^* : \epsilon \rightarrow t^*$
$\forall L^k.\ e^* \neq L^k[\mathbf{return}]$
$\forall i\ L^k.\ e^* = L^k[\mathbf{br}\ i] \Longrightarrow i < k$
$\forall v'^*.\ e^* \neq v'^*$
$e^* \neq \mathbf{trap}$
$\vdash_{\mathsf{s}} S : \mathsf{ok}$
$S \vdash_{\mathsf{f}} F : C$

we have $\exists S'\ F'\ e'^*.\ S; F; v^*\ e^* \hookrightarrow S'; F'; e'^*$

WasmCert-Isabelle: `progress_e` in `wasm_properties.thy`
WasmCert-Coq: `t_progress_e` in `type_progress.v`

The proof proceeds by induction on the definition of expression typing. A number of restricting assumptions must be included, beyond those necessary for preservation, for the induction to succeed. Assumption $\forall L^k.\ e^* \neq L^k[\textsf{return}]$ restricts the induction to only consider program fragments which are returning to a calling context. Assumption $\forall i\ L^k.\ e^* = L^k[\textsf{br}\ i] \implies i < k$ similarly restricts the induction to only consider program fragments which are breaking to a label within the program fragment itself, and not the label of any enclosing context. The L^k symbol denotes an evaluation context made up of k label constructs (**block**, **loop**, **if**). Failing to disregard these cases would make the induction hypothesis too weak, so they must be handled through separate proofs.

Lemma 3 (return progress).
Assuming $S; \epsilon \vdash_{\mathsf{loc}} F; e^* : t^*$ *we have* $\forall L^k.\ e^* \neq L^k[\textsf{return}]$
WasmCert-Isabelle: `progress_e1` in `wasm_properties.thy`
WasmCert-Coq: `s_typing_lf_br` in `type_progress.v`

Lemma 4 (br progress).
Assuming $S; \epsilon \vdash_{\mathsf{loc}} F; e^* : t^*$ *and* $e^* = L^k[\textsf{br}\ i]$ *we have* $i < k$
WasmCert-Isabelle: `progress_e2` in `wasm_properties.thy`
WasmCert-Coq: `s_typing_lf_return` in `type_progress.v`

These two lemmas are proven by induction on k, the label depth of the current evaluation context. The first lemma states that if the current frame has no return type set (denoted by ϵ), then the code fragment may not contain a **return** instruction. The second lemma states that a **br** instruction cannot attempt to jump outside the current frame, and so must target one of the labels inside the frame. These two lemmas show that the cases not handled by **fragment progess** are prevented by the type system from occurring at the top level of a program execution, so the lemmas together imply the top-level progress property.

4 Wasm 1.0 Full Semantics

The core runtime semantics and type checker for the W3C Wasm 1.0 standard, described in Sect. 2, is a refactoring of the 2017 draft semantics [14]. The full semantics of Wasm 1.0 significantly extends this draft semantics to include formal specifications of the binary decoding, the numerics and the instantiation phase. We describe our two mechanisations of these extensions: in Isabelle/HOL, we mechanise the instantiation using an OCaml harness for the binary encoding and numerics; and, in Coq, we mechanise the extentions in full, using established Coq libraries for the binary encoding and numerics. In this way, for the first time, we present a fully mechanised specification of the Wasm 1.0 standard.

Binary Decoding. Wasm modules are distributed in a bytecode format. Web browsers type check, compile, and instantiate Wasm code in a streaming manner as the files are downloaded to the user's browser [9]. Abstractly, it is specified that the Wasm binary format can be decoded into the module AST of Fig. 1, and subsequent phases of execution are defined over that AST [43]. In WasmCert-Coq, we make use of the Parseque [1] Coq library to mechanise the binary decoding phase of Wasm in an executable way. Parseque is more powerful than strictly necessary for our purposes. It is designed for parsing "complex recursive grammars" [2] whereas Wasm's binary grammar is fairly flat. However, using Parseque's *alternative* combinator gives us an off-the-shelf way to build an executable definition of the binary format which has close line-by-line correspondence to the Wasm 1.0 formal specification. Our Coq definitions (not including library code) come to ∼800 lines of non-comment, non-whitespace code (`binary_format_parser.v`). We have not attempted an analogous mechanisation in WasmCert-Isabelle, although it would be interesting future work.

Numeric Operations. We provide executable numeric definitions in WasmCert-Coq by linking with CompCert's integer and float libraries. The Wasm specification defers many of the definitions of its floating point operations to the IEEE 754 floating point standard, which is also the basis of the CompCert mechanisation. Wasm 1.0 does, however, define its integer operations directly. We prove a number of "sanity lemmas" for the integer operations which check that CompCert's definitions match those of the Wasm 1.0 specification. Our Coq numeric definitions and proofs together (excluding library code) come to ∼1500 lines of non-comment, non-whitespace code (`numerics.v`). As future work, we might exhaustively mechanise Wasm's integer operation specification, and prove it equivalent to the definitions of CompCert. WasmCert-Isabelle instead abstracts its numeric operations, and relies on an OCaml implementation of numerics provided by the WebAssembly Community Group [40] when extracting its interpreter.

Instantiation. Instantiation is a phase in the execution of a Wasm program which takes place after type-checking but before runtime evaluation. During instantiation, module imports are satisfied, and the state corresponding to module declarations (e.g. new memories, global variables) are created in the global store.

In the Wasm 1.0 standard [45], instantiation is fully formalised. The definition of instantiation is given by a large collection of inductive rules which do not directly describe an algorithm for instantiation. In essence, the specification defines a relational predicate which takes an initial state, a module to be instantiated, its provided imports, and an output state, and evaluates to true if and only if an instantiation operation in the initial state results in the output state. It does not give an algorithmic procedure for building the output state from the initial state. In fact, the standard's definition of instantiation contains deliberate circularities, which make direct execution of the definition unlikely.

An explanatory note in the standard indicates that a concrete implementation is expected to perform an additional pre-processing pass over the module to break this circularity [44].

The full definitions of module allocation and instantiation are too large and interconnected to other areas of the specification to summarise here, but we sketch the main source of definitional circularity in Fig. 3. The conclusion of the rule states that instantiating the *module* in global store S with the provided *imports* results in a global store S' (among other outputs which we elide for brevity). The store S' is obtained from the allocmodule abstract operation in the premise, which additionally requires the input v^*_{inits}, the values obtained by evaluating the global variable initialisers (see Sect. 2.2). However, the global initialisers are specified as being evaluated in the context of the global store S'. Therefore, the value of v^*_{inits} is defined as depending on the result of allocmodule, which itself takes v^*_{inits} as input—a circularity! In reality, the evaluation of the global initialisers only depends on a subset of the effects of allocmodule, in such a way that a concrete algorithm can pre-process the module to identify the parts of module allocation that global initialiser evaluation will depend on, perform part of the abstract allocmodule operation, evaluate global initialisers using the partial result, and then finish off the rest of the operation. This pre-pass is made simpler by the fact that the evaluation of a global initialiser is only allowed to depend on the values of imported (hence previously initialised) global variables. The specification deliberately chooses not to define this tiered process concretely.

$$\frac{\begin{array}{c}\mathrm{initialisers}(\mathit{module}.\mathrm{globs}) = e^*_{\mathsf{inits}} \qquad\qquad S'; F; e^*_{\mathsf{inits}} \hookrightarrow S'; F; v^*_{\mathsf{inits}} \\ \mathrm{allocmodule}(S, \mathit{module}, \mathit{imports}, v^*_{\mathsf{inits}}) = S' \ldots\end{array}}{\mathrm{instantiate}(S, \mathit{module}, \mathit{imports}) = S' \ldots}$$

Fig. 3. Illustrating a circularity in the Wasm 1.0 instantiation definition.

In both WasmCert-Isabelle and WasmCert-Coq, we mechanise the standard's inductive definition of instantiation, and create an executable definition of instantiation which performs the pre-processing pass sketched by the standard's explanatory note to break the circularity of the instantiation definition. In WasmCert-Isabelle, we prove these two definitions equivalent. By integrating the executable definition with our verified interpreter, we eliminate a significant amount of Watt's original unverified OCaml harness [37]. The WasmCert-Isabelle definitions and proofs represent ∼1650 lines of non-comment, non-whitespace code. WasmCert-Coq's definitions (currently without correctness proof) represent ∼800 lines of non-comment, non-whitespace code.

WasmCert-Isabelle:	`wasm_module.thy`, `wasm_module_checker.thy`, and `wasm_instantiation.thy`
WasmCert-Coq:	`instantiation.v`

We also validate the WasmCert-Isabelle interpreter against the official Wasm 1.0 end-to-end test suite, containing ∼17,810 tests, which we pass without error [39]. Testing the Coq-extracted interpreter end-to-end is left for future work.

5 Related Work

There is a wide body of existing work on the mechanised specification of programming language semantics [31]. It is usual for such specificication based on an interactive theorem prover to target an interesting language core or abstraction for mechanisation, due to the ambiguity, complexity, or size of the full language definition. In contrast, we are able to closely follow the *whole* of the Wasm 1.0 semantics directly, as stated in the standard, a task made tractable by its compact design and official formalisation. Xuan presents a partial Coq-mechanisation of Wasm in his M.Sc project, independently named WasmCert [15]. We have been given permission to use the WasmCert name. As already discussed, we build on Watt's Isabelle/HOL mechanisation [37].

Norrish presents a mechanisation of a fragment of C in HOL [26]. The CompCert project [23] has a mechanisation of a large fragment of C in Coq, called Clight [4], making simplifying assumptions regarding some details of the C memory model which are not relevent to the CompCert compilation correctness proof. Lee et al. mechanise in Twelf [29] an "internal language" with "equivalent expressive power" to Standard ML, the semantics of which they formalise via elaoration [21]. CakeML [19,35] includes a mechanised semantics in HOL4 for a large fragment of Standard ML (minus functors). OCaml Light [27] is a mechanised semantics in HOL4 of a core subset of OCaml. Jinja [18] and Java$_S$ [34] provide mechanised fragments of Java. JSCert [5] is a Coq-mechanised specification of a large subset of ECMAScript 5, handling all core constructs but leaving out "library objects" such as `Array` and `Number`. Guha et al. give a JavaScript semantics through elaboration to a mechanised semantics in Coq of a core calculus [13].

More lightweight approaches are possible: e.g. the K framework [32] used to define term-rewriting models of significant fragments of C [10], Java [6], JavaScript [28], and PHP [11]; Cerberus [24] which defines an elaboration semantics for a large fragment of C with a core calculus defined in the Lem specification language [25]; and JaVerT, an analysis tool for JavaScript programs [12,33], where the semantics of JavaScript is defined by a elaboration to a simpler intermediate language.

6 Conclusion and Future Work

We hope our mechanisation of Wasm 1.0 will replace Watt's mechanisation of the original Wasm draft [37] as the canonical source for the Wasm 1.0 type soundness proof. We also hope for further adoption of our work by the WebAssembly Community Group, with our mechanisations endorsed in the same way that certain developer tools and compilers for Wasm are already hosted under their official banner. Wasm 1.0 is in the process of being extended with a number of new feature proposals. We intend to keep our mechanisations abreast of these changes, and hope that early mechanisation will be valuable for in-progress features.

Our immediate future priority for WasmCert-Coq is to complete the final proofs and engineering to enable us to extract a verified end-to-end interpreter which does not use an unverified OCaml harness as is currently used in WasmCert-Isabelle. This would require us to complete the verification of the WasmCert-Coq instantiation implementation. We would also like to do independent testing of the end-to-end interpreter using the Wasm 1.0 test suite. Our priority for WasmCert-Isabelle is to mechanise Wasm's binary decoding phase and numeric operations, again with the aim of providing a verified end-to-end interpreter.

Our work opens the door to a number of applications. We are currently investigating the integration of WasmCert-Coq with the Iris framework [17], developing a higher-order mechanised program logic for a Wasm host language to explore language-interoperable reasoning. WasmCert-Coq could be linked to CompCert's IRs [8], to provide verified compilation both to and from Wasm. WasmCert-Isabelle could be linked to the Isabelle/HOL port of the CakeML verified compiler [16,20]. It could also be linked with a Java mechanisation such as Jinja [18], in order to investigate the expressivity of WebAssembly's hotly-debated in-progress extension of Garbage-Collected Types [42]. In summary, we believe that WasmCert-Isabelle and WasmCert-Coq each have the potential to become the foundation of many different mechanisation projects for Wasm 1.0.

Acknowledgement. Watt, Pichon-Pharabod, Bodin, and Gardner were supported by the EPSRC grant *REMS: Rigorous Engineering for Mainstream Systems* (EP/K008528/1). Watt and Pichon-Pharabod were supported by the VeTSS/NCSC grant *Mechanising Concurrent WebAssembly* (G105616 RI2).

Watt was supported by a Peterhouse Research Fellowship and a Google PhD Fellowship in Programming Technology and Software Engineering. Rao was supported by an Imperial College London, Department of Computing, Doctoral Scholarship Award. Pichon-Pharabod was supported by the ERC grant *Engineering with Logic and Verification: Mathematically Rigorous Engineering for Safe and Secure Computer Systems* (789108). Bodin and Gardner were supported by the EPSRC fellowship *VeTSpec: Verified Trustworthy Software Specification* (EP/R034567/1).

References

1. Allais, G.: Parseque (2017). https://github.com/gallais/parseque
2. Allais, G.: Agdarsec - total parser combinators. In: JFLA 2018 (2018)
3. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 283–307. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_12
4. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. J. Autom. Reason. **43**(3), 263–288 (2009). https://doi.org/10.1007/s10817-009-9148-3. https://hal.inria.fr/inria-00352524

5. Bodin, M., et al.: A trusted mechanised JavaScript specification. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 87–100. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2535838.2535876
6. Bogdanas, D., Roşu, G.: K-Java: a complete semantics of Java. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 445–456. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2676726.2676982
7. Boldo, S., Jourdan, J.H., Leroy, X., Melquiond, G.: Verified compilation of floating-point computations. J. Autom. Reason. **54**(2), 135–163 (2015). http://xavierleroy.org/publi/floating-point-compcert.pdf
8. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic, pp. 243–252 (2011). https://doi.org/10.1109/ARITH.2011.40
9. Bynens, M.: Loading webassembly modules efficiently (2018). https://developers.google.com/web/updates/2018/04/loading-wasm
10. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, pp. 533–544. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2103656.2103719
11. Filaretti, D., Maffeis, S.: An executable formal semantics of PHP. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 567–592. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_23
12. Fragoso Santos, J., Maksimović, P., Sampaio, G., Gardner, P.: Javert 2.0: compositional symbolic execution for javascript. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290379
13. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 126–150. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_7
14. Haas, A., et al.: Bringing the web up to speed with WebAssembly. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (2017)
15. Huang, X.: A mechanized formalization of the webassembly specification in coq. In: RIT Computer Science (2019)
16. Hupel, L., Zhang, Y.: Cakeml. Archive of Formal Proofs, March 2018. https://isa-afp.org/entries/CakeML.html. Formal proof development
17. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28** (2018)
18. Klein, G., Nipkow, T.: A machine-checked model for a java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst. **28**(4), 619–695 (2006). https://doi.org/10.1145/1146809.1146811
19. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 179–191. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2535838.2535841
20. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Principles of Programming Languages (POPL), pp. 179–191. ACM Press (2014). https://doi.org/10.1145/2535838.2535841. https://cakeml.org/popl14.pdf

21. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of standard ML. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, pp. 173–184. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1190216.1190245
22. Leroy, X.: Java bytecode verification: an overview. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 265–285. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_26
23. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
24. Memarian, K., et al.: Into the depths of C: elaborating the de facto standards. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, pp. 1–15. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2908080.2908081
25. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: reusable engineering of real-world semantics. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ICFP 2014, pp. 175–188. ACM, New York (2014). https://doi.org/10.1145/2628136.2628143
26. Norrish, M.: C formalised in HOL. Technical report (1998)
27. Owens, S.: A sound semantics for OCamllight. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_1
28. Park, D., Stefănescu, A., Roşu, G.: KJS: A complete formal semantics of JavaScript. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 346–356. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2737924.2737991
29. Pfenning, F., Schürmann, C.: System description: twelf — a meta-logical framework for deductive systems. In: CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_14
30. Pierce, B.C.: Types and Programming Languages, 1st edn. The MIT Press, Cambridge (2002)
31. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: QED at large: a survey of engineering of formally verified software. Found. Trends Program. Lang. **5**(2-3), 102–281 (2019). https://doi.org/10.1561/2500000045
32. Rou, G., erbănută, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012. http://www.sciencedirect.com/science/article/pii/S1567832610000160. Membrane computing and programming
33. Santos, J.F., Maksimović, P., Grohens, T., Dolby, J., Gardner, P.: Symbolic execution for JavaScript. In: Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3236950.3236956
34. Syme, D.: Proving Java type soundness. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. LNCS, vol. 1523, pp. 83–118. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48737-9_3
35. Tan, Y.K., Owens, S., Kumar, R.: A verified type system for CakeML. In: Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages. IFL 2015. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2897336.2897344

36. WasmCert: WasmCert (2021). https://github.com/WasmCert
37. Watt, C.: Mechanising and verifying the WebAssembly specification. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, pp. 53–65. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3167082
38. Watt, C.: Mechanising and evolving the formal semantics of WebAssembly: The Web's new low-level language (2021, not yet published)
39. WebAssembly Community Group: tests (2020). https://github.com/WebAssembly/spec/tree/704d9d9e9c861fdb957c3d5e928f1d046a31497e/test
40. WebAssembly Community Group: Webassembly (2020). https://github.com/WebAssembly/spec/tree/704d9d9e9c861fdb957c3d5e928f1d046a31497e/
41. WebAssembly Community Group: bulk-memory-operations (2021). https://github.com/WebAssembly/bulk-memory-operations
42. WebAssembly Community Group: GC (2021). https://github.com/WebAssembly/gc
43. WebAssembly Working Group: Binary format (2019). https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/#binary-format%E2%91%A0
44. WebAssembly Working Group: Instantiation (2019). https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/#instantiation%E2%91%A1
45. WebAssembly Working Group: Webassembly core specification (2019). https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/
46. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994). https://doi.org/10.1006/inco.1994.1093