DOI:10.1145/3282510

Bringing the Web Up to Speed with WebAssembly

By Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J.F. Bastien, and Michael Holman

Abstract

The maturation of the Web platform has given rise to sophisticated Web applications such as 3D visualization, audio and video software, and games. With that, efficiency and security of code on the Web has become more important than ever. WebAssembly is a portable low-level bytecode that addresses these requirements by offering a compact representation, efficient validation and compilation, and safe execution with low to no overhead. It has recently been made available in all major browsers. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it independent of language, hardware, and platform and applicable far beyond just the Web. WebAssembly is the first mainstream language that has been designed with a formal semantics from the start, finally utilizing formal methods that have matured in programming language research over the last four decades.

1. INTRODUCTION

The Web began as a simple hypertext document network but has now become the most ubiquitous application platform ever, accessible across a vast array of operating systems and device types. By historical accident, JavaScript is the only natively supported programming language on the Web. Because of its ubiquity, rapid performance improvements in modern implementations, and perhaps through sheer necessity, it has become a compilation target for many other languages. Yet JavaScript has inconsistent performance and various other problems, especially as a compilation target.

WebAssembly (or "Wasm" for short) addresses the problem of safe, fast, portable low-level code on the Web. Previous attempts, from ActiveX to Native Client to asm.js, have fallen short of properties that such a low-level code format should have:

- Safe, fast, and portable *semantics*:
  - safe to execute
  - fast to execute
  - language-, hardware-, and platform-independent
  - deterministic and easy to reason about
  - simple interoperability with the Web platform
- Safe and efficient *representation*:
  - maximally compact
  - easy to decode, validate and compile
  - easy to generate for producers
  - streamable and parallelizable

Why are these goals important? Why are they hard?

Safe. Safety for mobile code is paramount on the Web, since code originates from untrusted sources. Protection for mobile code has traditionally been achieved by providing a managed language runtime such as the browser's JavaScript Virtual Machine (VM) or a language plugin. Managed languages enforce *memory safety*, preventing programs from compromising user data or system state. However, managed language runtimes have traditionally not offered much for low-level code, such as C/C++ applications that do not use garbage collection.

Fast. Low-level code like that emitted by a C/C++ compiler is typically optimized ahead-of-time. Native machine code, either written by hand or as the output of an optimizing compiler, can utilize the full performance of a machine. Managed runtimes and sandboxing techniques have typically imposed a steep performance overhead on low-level code.

Universal. There is a large and healthy diversity of programming paradigms, none of which should be privileged or penalized by a code format, beyond unavoidable hardware constraints. Most managed runtimes, however, have been designed to support a particular language or programming paradigm well while imposing significant cost on others.

Portable. The Web spans not only many device classes, but different machine architectures, operating systems, and browsers. Code targeting the Web must be hardware- and platform-independent to allow applications to run across all browser and hardware types with the same deterministic behavior. Previous solutions for low-level code were tied to a single architecture or have had other portability problems.

Compact. Code that is transmitted over the network should be small to reduce load times, save bandwidth, and improve overall responsiveness. Code on the Web is typically transmitted as JavaScript source, which is far less compact than a binary format, even when minified and compressed. Binary code formats are not always optimized for size either.

WebAssembly is the first solution for low-level code on the Web that delivers on all of the above design goals. It is the result of an unprecedented collaboration across all

[a] WebAssembly engines are not assumed to spend time on sophisticated optimizations, because producers usually can take care of that more cheaply offline. Hence WebAssembly does not magically make code faster. But it allows other languages to bypass the cost and complexity of JavaScript.

The original version of this paper was published in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain, June 18–23, 2017), 185–200.

major browser vendors and an online community group to build a common solution for high-performance applications.[a]

While the Web is the primary motivation for WebAssembly, its design—despite the name—carefully avoids any dependencies on the Web. It is an open standard intended for embedding in a broad variety of environments, and other such embeddings are already being developed.

To our knowledge, WebAssembly also is the first industrial-strength language that has been designed with a formal semantics from the start. It not only demonstrates the "real world" feasibility of applying formal techniques, but also that they lead to a remarkably clean and simple design.

2. A TOUR OF THE LANGUAGE

Even though WebAssembly is a binary code format, we define it as a *programming language* with syntax and structure. As we will see, that makes it easier to explain and understand and moreover, allows us to apply well-established formal techniques for defining its semantics and for reasoning about it. Hence, Figure 1 presents WebAssembly in terms of a grammar for its *abstract syntax*.

2.1. Basics

Let us start by introducing a few unsurprising concepts before diving into less obvious ones in the following.

Modules. A WebAssembly binary takes the form of a *module*. It contains definitions for *functions*, *globals*, *tables*, and *memories*.[b] Definitions may be *exported* or *imported*.

While a module corresponds to the static representation of a program, a module's dynamic representation is an *instance*, complete with its mutable *state*. Instantiating a module requires providing definitions for all imports, which may be exports from previously created instances. Computations is initiated by invoking an exported function.

Modules provide both *encapsulation* and *sandboxing*: because a client can only access the exports of a module, other internals are protected from tampering; dually, a module can only interact with its environment through its imports which are provided by a client, so that the client has full control over the capabilities given to a module. Both these aspects are essential ingredients to the safety of WebAssembly.

Functions. The code in a module is organized into individual *functions*, taking parameters and returnng results as defined by its *function type*. Functions can call each other, including recursively, but are not first class and cannot be nested. The call stack for execution is not exposed, and thus cannot be directly accessed by a running WebAssembly program, even a buggy or malicious one.

Instructions. WebAssembly is conceptually based on a *stack machine*: code for a function consists of a sequence of *instructions* that manipulate values on an implicit *operand stack*. However, thanks to the type system (Section 3.2), the layout of the operand stack can be statically determined at any point in the code, so that implementations can compile the data flow between instructions directly without ever materializing the operand stack. The stack organization is merely a way to achieve a compact program representation, as it has been shown to be smaller than a register machine.

Traps. Some instructions may produce a *trap*, which immediately aborts the current computation. Traps cannot (currently) be handled by WebAssembly code, but an embedder will typically provide means to handle this condition, for example, by reifying them as JavaScript exceptions.

Machine types. WebAssembly has only four basic *value types t* to compute with. These are integers and IEEE 754 floating point numbers, each with 32 or 64 bits, as available in common hardware. Most WebAssembly instructions provide simple operators on these data types. The grammar in Figure 1 conveniently distinguishes categories such as *unary* and *binary* operators, *tests*, *comparisons*, and *conversions*. Like hardware, WebAssembly makes no distinction between signed and unsigned integer types. Instead, where it matters, a *sign extension* suffix _u or _s to an instruction selects either unsigned or two's complement signed behavior.

Variables. Functions can declare mutable *local variables*, which essentially provides an infinite set of zero-initialized virtual registers. A module may also declare typed *global variables* that can be either mutable or immutable and require an explicit initializer. Importing globals allows a limited

[b] WebAssembly's text format closely resembles this syntax. For brevity we omit minor features regarding initialization of modules.

Figure 1. WebAssembly abstract syntax.

```
(value types)     t   ::= i32 | i64 | f32 | f64
(packed types)    pt  ::= i8 | i16 | i32
(function types)  ft  ::= t* → t*
(global types)    gt  ::= mut? t

unop_iN    ::= clz | ctz | popcnt
unop_fN    ::= neg | abs | ceil | floor | trunc | nearest | sqrt
binop_iN   ::= add | sub | mul | div_sx | rem_sx |
               and | or | xor | shl | shr_sx | rotl | rotr
binop_fN   ::= add | sub | mul | div | min | max | copysign
testop_iN  ::= eqz
relop_iN   ::= eq | ne | lt_sx | gt_sx | le_sx | ge_sx
relop_fN   ::= eq | ne | lt | gt | le | ge
cvtop      ::= convert | reinterpret
sx         ::= s | u

(instructions)  e ::= unreachable | nop | drop | select |
                      block ft e* end | loop ft e* end | if ft e* else e* end |
                      br i | br_if i | br_table i+ | return | call i | call_indirect ft |
                      get_local i | set_local i | tee_local i | get_global i |
                      set_global i | t.load (pt_sx)? a o | t.store pt? a o |
                      memory.size | memory.grow | t.const c |
                      t.unop_t | t.binop_t | t.testop_t | t.relop_t | t.cvtop t_sx?

(functions)  func ::= ex* func ft local t* e* | ex* func ft im
(globals)    glob ::= ex* global gt e* | ex* global gt im
(tables)     tab  ::= ex* table n i* | ex* table n im
(memories)   mem  ::= ex* memory n | ex* memory n im
(imports)    im   ::= import "name" "name"
(exports)    ex   ::= export "name"
(modules)    mod  ::= module func* glob* tab? mem?
```

form of configurability, for example, for linking. Like all entities in WebAssembly, variables are referenced by integer indices.

So far so boring. In the following sections we turn our attention to more unusual features of WebAssembly's design.

2.2. Memory

The main storage of a WebAssembly program is a large array of raw bytes, the *linear memory* or simply *memory*. Memory is accessed with load and store instructions, where addresses are simply unsigned integer operands.

Creation and growing. Each module can define at most one memory, which may be shared with other instances via import/export. Memory is created with an initial size but may be dynamically grown. The unit of growth is a *page*, which is defined to be 64KiB, a choice that allows reusing virtual memory hardware for bounds checks on modern hardware (Section 5). Page size is fixed instead of being system-specific to prevent portability hazards.

Endianness. Programs that load and store to aliased locations with different types can observe byte order. Since most contemporary hardware has converged on little endian, or at least can handle it equally well, we chose to define WebAssembly memory to have little endian byte order. Thus the semantics of memory access is completely deterministic and portable across all engines and platforms.

Security. All memory access is dynamically checked against the memory size; out of bounds access results in a trap. Linear memory is disjoint from code space, the execution stack, and the engine's data structures; therefore compiled programs cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined behavior. At worst, a buggy or malicious WebAssembly program can make a mess of the data in its own memory. Consequently, even untrusted modules can be safely executed in the same address space as other code. Achieving fast in-process isolation is necessary for interacting with untrusted JavaScript and the various Web Application Programming Interfaces (APIs) in a high-performance way. It also allows a WebAssembly engine to be safely embedded into other managed language runtimes.

2.3. Control flow

WebAssembly represents control flow differently from most stack machines. It does not offer arbitrary jumps but instead provides *structured control flow* constructs more akin to a programming language. This ensures by construction that control flow cannot form irreducible loops, contain branches to blocks with misaligned stack heights, or branch into the middle of a multi-byte instruction. These properties allow WebAssembly code to be validated in a single pass, compiled in a single pass, and even transformed to an SSA-form intermediate representation in the same single pass.

Control constructs. As required by the grammar in Figure 1, the **block**, **loop** and **if** constructs must be terminated by an **end** opcode and be properly nested to be considered well-formed. The inner instruction sequences e^* in these constructs form a *block*. Note that **loop** does not automatically iterate its block but allows constructing a loop manually with explicit branches. Every control construct is annotated with a function type $ft = t_1^* \rightarrow t_2^*$ describing its effect on the stack, popping values typed t_1^* and pushing t_2^*.

Branches. Branches can be unconditional (**br**), conditional (**br_if**), or indexed (**br_table**). They have "label" immediates that do not denote positions in the instruction stream but reference outer control constructs by relative nesting depth. Hence, labels are effectively *scoped*: branches can only reference constructs in which they are nested. Taking a branch "breaks from" that construct's block;[c] the exact effect depends on the target construct: in case of a **block** or **if** it is a *forward* jump to its end (like a break statement); with a **loop** it is a *backward* jump to its beginning (like a continue statement). Branching *unwinds* the operand stack by implicitly popping all unused operands, similar to returning from a function call. This liberates producers from having to track stack height across sub-expressions and adding explicit drops to make them match.

Expressiveness. Structured control flow may seem like a severe limitation, but most high-level control constructs are readily expressible with the suitable nesting of blocks. For example, a C-style switch statement with fall-through,

```
switch (x) {                          block block block block
  case 0: ...A...                       br_table 0 1 2
  case 1: ...B... break;    becomes    end ...A...
  default: ...C...                      end ...B... br 1
}                                       end ...C...
                                      end
```

Slightly more finesse is required for fall-through between unordered cases. Various forms of loops can likewise be expressed with combinations of **loop**, **block**, **br** and **br_if**.

It is the responsibility of producers to transform unstructured and irreducible control flow into structured form. This is the established approach to compiling for the Web, where JavaScript is also restricted to structured control. In our experience building an LLVM backend for WebAssembly, irreducible control flow is rare, and a simple restructuring algorithm[18] is sufficient to translate any Control Flow Graph (CFG) to WebAssembly. The benefit of the restriction is that many algorithms in engines are much simpler and faster.

2.4. Function calls and tables

A function body is a block. Execution can complete by either reaching the end of the block with the function's result values on the stack, or by a branch exiting the function block, with the branch operands as the result values; the **return** instruction is simply shorthand for the latter.

Calls. Functions can be invoked *directly* using the **call** instruction which takes an immediate identifying the function to call. Function pointers can be emulated with the **call_indirect** instruction which takes a runtime index into a *table* of functions defined by the module. The functions in this table are not required to have the same type. Instead, the type of the function is checked dynamically against an

[c] The name **br** can also be read as "break" wrt. a block.

expected type supplied to the **call_indirect** instruction and traps in case of a mismatch. This check protects the integrity of the execution environment. The heterogeneous nature of the table is based on experience with asm.js's multiple homogeneous tables; it allows more faithful representation of function pointers and simplifies dynamic linking. To aid dynamic linking scenarios further, exported tables can be grown and mutated dynamically through external APIs.

Foreign calls. Functions can be imported into a module. Both direct and indirect calls can invoke an imported function, and through export/import, multiple module instances can communicate. Additionally, the import mechanism serves as a safe *Foreign Function Interface* (FFI) through which a WebAssembly program can communicate with its embedding environment. For example, on the Web imported functions may be *host* functions that are defined in JavaScript. Values crossing the language boundary are automatically converted according to JavaScript rules.

2.5. Determinism

WebAssembly has sought to provide a portable target for low-level code without sacrificing performance. Where hardware behavior differs it usually is corner cases such as out-of-range shifts, integer divide by zero, overflow or underflow in floating point conversion, and alignment. Our design gives deterministic semantics to all of these across all hardware with only minimal execution overhead.

However, there remain three sources of implementation-dependent behavior that can be viewed as non-determinism.

NaN payloads. WebAssembly follows the IEEE 754 standard for floating point arithmetic. However, IEEE does not specify the exact bit pattern for NaN values in all cases, and we found that CPUs differ significantly, while normalizing after every numeric operation is too expensive. Based on our experience with JavaScript engines, we picked rules that allow the necessary degree of freedom while still providing enough guarantees to support techniques like NaN-tagging.

Resource exhaustion. Available resources are always finite and differ wildly across devices. In particular, an engine may be *out of memory* when trying to grow the linear memory—semantically, the **memory.grow** instruction can non-deterministically return −1. A call instruction may also trap due to *stack overflow*, but this is not semantically observable from within WebAssembly itself since it aborts the computation.

Host functions. WebAssembly programs can call host functions which are themselves non-deterministic or change WebAssembly state. Naturally, the effect of calling host functions is outside the realm of WebAssembly's semantics.

WebAssembly does not (yet) have threads, and therefore no non-determinism arising from concurrent memory access. Adding threads is the subject of ongoing work.

2.6. Binary format

WebAssembly is transmitted as a binary encoding of the abstract syntax presented in Figure 1. This encoding has been designed to minimize both size and decoding time. A binary represents a single module and is divided into sections according to the different kinds of entities declared in it. Code for function bodies is deferred to a separate section placed after all declarations to enable *streaming compilation* as soon as function bodies begin arriving over the network. An engine can also *parallelize* compilation of function bodies. To aid this further, each body is preceded by its size so that a decoder can skip ahead and parallelize even its decoding.

The format also allows user-defined sections that may be ignored by an engine. For example, a custom section is used to store debug metadata such as source names in binaries.

3. SEMANTICS

The WebAssembly semantics consists of two parts: the *static semantics* defining *validation*, and the *dynamic semantics* defining *execution*. In both cases, the presentation as a language allowed us to adopt off-the-shelf formal methods developed in programming language research over the past decades. They are convenient and effective tools for declarative specifications. While we can not show all of it here, our specification is precise, concise, and comprehensive—validation and execution of all of WebAssembly fit on just two pages of our original paper.[4]

Furthermore, these formulations allow effective *proofs* of essential properties of this semantics, as are standard in programming language research, but so far have rarely ever been done as part of an industrial-strength design process.

Finally, our formalization enabled other researchers to easily *mechanize* the WebAssembly specification with theorem provers, thereby machine-verifying our correctness results as well as constructing a provably correct interpreter.

3.1. Execution

We cannot go into much detail in this article, but we want to give a sense for the general flavor of our formalization (see Haas et al.[4] for a more thorough explanation).

Reduction. Execution is defined in terms of a standard *small-step reduction* relation,[13] where each step of computation is described as a *rewrite rule* over a sequence of instructions. Figure 2 gives an excerpt of these rules.

For example, the instruction sequence

(**i32.const** 3) (**i32.const** 4) **i32.add**

is reduced to the constant (**i32.const** 7) according to the fourth rule. This formulation avoids the need for introducing the operand stack as a separate notion in the semantics—that stack simply consists of all leading *t*.**const** instructions in an instruction sequence. Execution terminates when an instruction sequence has been reduced to just constants, corresponding to the stack of result values. Therefore, constant instructions can be treated as *values* and abbreviated v.

To deal with control constructs, we need to squint a little and extend the syntax with a small number of auxiliary *administrative instructions* that only occur temporarily during reduction. For example, **label** marks the extent of an active block and records the continuation of a branch to it, while **frame** essentially is a call frame for function invocation. Through nesting these constructs, the intertwined nature of operand and control stack is captured, avoiding the need for separate stacks with tricky interrelated invariants.

Figure 2. Small-step reduction rules (Excerpt).

$$
\begin{array}{lll}
\text{(store)} & s & ::= \{\mathsf{func}\ fi^*,\ \mathsf{global}\ v^*,\ \mathsf{table}\ ti^*,\ \mathsf{mem}\ mi^*\} \\
\text{(frames)} & f & ::= \{\mathsf{module}\ m,\ \mathsf{local}\ v^*\} \\
\text{(module instances)} & m & ::= \{\mathsf{func}\ a^*,\ \mathsf{global}\ a^*,\ \mathsf{table}\ a^?,\ \mathsf{mem}\ a^?\} \\
\text{(function instances)} & fi & ::= \{\mathsf{module}\ m,\ \mathsf{code}\ func\}\ \text{(where } func \text{ is not an import and has all exports } ex^* \text{ erased)} \\
\text{(table instances)} & ti & ::= (a^?)^* \\
\text{(memory instances)} & mi & ::= b^* \\
\text{(values)} & v & ::= t.\mathbf{const}\ c \\
\text{(administrative operators)} & e & ::= \dots \mid \mathbf{trap} \mid \mathbf{call}\ fi \mid \mathbf{label}_n\{e^*\}\ e^*\ \mathbf{end} \mid \mathbf{frame}_n\{f\}\ e^*\ \mathbf{end}
\end{array}
$$

$$
\begin{array}{rcll}
\mathbf{nop} & \hookrightarrow & \epsilon & \\
v\ \mathbf{drop} & \hookrightarrow & \epsilon & \\
(t.\mathbf{const}\ c)\ t.unop & \hookrightarrow & t.\mathbf{const}\ unop_t(c) & \\
(t.\mathbf{const}_{c_1})\ (t.\mathbf{const}_{c_2})\ t.binop & \hookrightarrow & t.\mathbf{const}\ binop_t(c_1, c_2) & \\
v^n\ \mathbf{block}\ (t_1^n \rightarrow t_2^m)\ e^*\ \mathbf{end} & \hookrightarrow & \mathbf{label}_m\{\epsilon\}\ v^n\ e^*\ \mathbf{end} & \\
v^n\ \mathbf{loop}\ (t_1^n \rightarrow t_2^m)\ e^*\ \mathbf{end} & \hookrightarrow & \mathbf{label}_n\{\mathbf{loop}(t_1^n \rightarrow t_2^m)\ e^*\ \mathbf{end}\}\ v^n\ e^*\ \mathbf{end} & \\
\mathbf{label}_n\{e^*\}\ v^*\ \mathbf{end} & \hookrightarrow & v^* & \\
\mathbf{label}_n\{e^*\}\ L^i\ [v^n\ (\mathbf{br}\ i)]\ \mathbf{end} & \hookrightarrow & v^n\ e^* & \text{where } L^0 ::= v^*\ [_]\ e^* \text{ and } L^{k+1} ::= v^*\ \mathbf{label}_n\{e^*\}\ L^k\ \mathbf{end}\ e^* \\
f; (\mathbf{get_local}\ i) & \hookrightarrow & v & \text{if } f_{\mathsf{local}}(i) = v \\
f; v\ (\mathbf{set_local}\ i) & \hookrightarrow & f'; \epsilon & \text{if } f' = f \text{ with } \mathsf{local}(i) = v \\
s; f; (\mathbf{call}\ i) & \hookrightarrow & \mathbf{call}\ s_{\mathsf{func}}(f_{\mathsf{func}}(i)) & \\
s; f; (\mathbf{i32.const}\ i)\ (\mathbf{call_indirect}\ ft) & \hookrightarrow & \mathbf{call}\ s_{\mathsf{func}}(s_{\mathsf{table}}(f_{\mathsf{table}})(i)) & \text{if } s_{\mathsf{func}}(s_{\mathsf{table}}(f_{\mathsf{table}})(i))_{\mathsf{code}} = (\mathbf{func}\ ft\ \mathbf{local}\ t^*\ e^*) \\
s; f; (\mathbf{i32.const}\ i)\ (\mathbf{call_indirect}\ ft) & \hookrightarrow & \mathbf{trap} & \text{otherwise} \\
v^n\ (\mathbf{call}\ fi) & \hookrightarrow & \mathbf{frame}_m\{\mathsf{module}\ fi_{\mathsf{module}},\ \mathsf{local}\ v^n\ (t.\mathbf{const}\ 0)^k\}\ e^*\ \mathbf{end}\ \dots & \\
\mathbf{frame}_n\{f\}\ v^n\ \mathbf{end} & \hookrightarrow & v^n & \mid\ \dots \text{where } fi_{\mathsf{code}} = (\mathbf{func}\ (t_1^n \rightarrow t_2^m)\ \mathbf{local}\ t^k\ e^*) \\
\mathbf{frame}_n\{f\}\ L^k\ [v^n\ \mathbf{return}]\ \mathbf{end} & \hookrightarrow & v^n & \\
s; f_0; \mathbf{frame}_n\{f\}\ e^*\ \mathbf{end} & \hookrightarrow & s'; f_0; \mathbf{frame}_n\{f'\}\ e'\ \mathbf{end} & \text{if } s; f; e^* \hookrightarrow s'; f'; e'^*
\end{array}
$$

Configurations. In general, execution operates relative to a global *store* s as well as the current function's *frame* f. Both are defined in the upper part of Figure 2.

The store models the global state of a program and is a record of the lists of function, global, table and memory *instances* that have been allocated. An index into one of the store components is called an *address* a. We use notation like $s_{\mathsf{func}}(a)$ to look up the function at address a. A module instance then maps static indices i that occur in instructions to their respective dynamic addresses in the store.

To that end, each frame carries—besides the state of the function's local variables—a link to the module instance it resides in, applying notational short-hands like f_{table} for $(f_{\mathsf{module}})_{\mathsf{table}}$. Essentially, every function instance is a *closure* over the function's module instance. An implementation can eliminate these closures by specializing generated machine code to a module instance.

For example, for a direct **call** i, the respective function instance is looked up in the store through the frame of the caller. Similarly, for an indirect call, the current table is looked up and the callee's address is taken from there (if the function's type ft does not match the expected type, then a trap is generated). Both kinds of calls reduce to a common administrative instruction **call** fi performing a call to a known function instance; reducing that further creates a respective frame for the callee and initializes its locals.

Together, the triple $s; f; e^*$ of store, frame, and instruction sequence forms a *configuration* that represents the complete state of the WebAssembly abstract machine at a given point in time. Reduction rules, in their full generality, then rewrite configurations not just instruction sequences.

3.2. Validation

On the Web, code is fetched from untrusted sources and must be *validated*. Validation rules for WebAssembly are defined succinctly as a *type system*. This type system is, by design, embarrassingly simple, and designed to be efficiently checkable in a single linear pass.

Typing rules. Again, we utilize standard techniques for defining our semantics declaratively, this time via a system of *natural deduction* rules.[12] Figure 3 shows an excerpt of the rules for typing instructions. They collectively define a *judgement* $C \vdash e : ft$, which can be read as "instruction e is valid with type ft under the assumptions embodied in the *context* C." The context records the types of all declarations that are in scope at a given point in a program. The type of an instruction is a function type that specifies its required *input* stack and the provided *output* stack.

Each rule consists of a conclusion (the part below the bar) and a possibly empty list of premises (the pieces above the bar). It can be read as a big implication: the conclusion holds if all premises hold. One rule exists for each instruction, defining when it is well-typed. A program is valid if and only if the rules can inductively derive that it is well-typed.

For example, the rules for constants and simple numeric operators are trivial *axioms*, since they do not even require a premise: an instruction of the form $t.binop$ always has type $t\ t \rightarrow t$, that is, consumes two operands of type t and pushes one. The rules for control constructs require that their type matches the explicit annotation ft, and they extend the context with a local label when checking the inner block. Label types are looked up in the context when typing branch

Figure 3. Typing rules (Excerpt).

$$\text{(contexts)} \quad C \quad ::= \quad \{\mathsf{func}\ ft^*,\ \mathsf{global}\ gt^*,\ \mathsf{table}\ n^?,\ \mathsf{memory}\ n^?,\ \mathsf{local}\ t^*,\ \mathsf{label}\ (t^*)^*,\ \mathsf{return}\ (t^*)^?\}$$

$$\frac{}{C \vdash t.\mathbf{const}\ c : \epsilon \to t} \qquad \frac{}{C \vdash t.unop : t \to t} \qquad \frac{}{C \vdash t.binop : t\ t \to t} \qquad \frac{}{C \vdash \mathbf{nop} : \epsilon \to \epsilon} \qquad \frac{}{C \vdash \mathbf{drop} : t \to \epsilon}$$

$$\frac{ft = t_1^n \to t_2^m \qquad C, \mathsf{label}(t_2^m) \vdash e^* : ft}{C \vdash \mathbf{block}\ ft\ e^*\ \mathbf{end} : ft} \qquad \frac{ft = t_1^n \to t_2^m \qquad C, \mathsf{label}(t_1^n) \vdash e^* : ft}{C \vdash \mathbf{loop}\ ft\ e^*\ \mathbf{end} : ft} \qquad \frac{C_{\mathsf{label}}(i) = t^*}{C \vdash \mathbf{br}\ i : t_1^*\ t^* \to t_2^*}$$

$$\frac{C_{\mathsf{func}}(i) = ft}{C \vdash \mathbf{call}\ i : ft} \qquad \frac{ft = t_1^* \to t_2^* \qquad C_{\mathsf{table}} = n}{C \vdash \mathbf{call_indirect}\ ft : t_1^*\ \mathsf{i32} \to t_2^*} \qquad \frac{C_{\mathsf{return}} = t^*}{C \vdash \mathbf{return} : t_1^*\ t^* \to t_2^*}$$

$$\frac{C_{\mathsf{local}}(i) = t}{C \vdash \mathbf{get_local}\ i : \epsilon \to t} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \mathbf{set_local}\ i : t \to \epsilon}$$

instructions, which require suitable operands on the stack to match the stack at the join point.

3.3. Soundness

The WebAssembly type system enjoys standard *soundness* properties.[16] Soundness proves that the reduction rules actually cover all execution states that can arise for valid programs. In other words, it proves the absence of undefined behavior. In particular, this implies the absence of *type safety* violations such as invalid calls or illegal accesses to locals, it guarantees *memory safety*, and it ensures the inaccessibility of code addresses or the call stack. It also implies that the use of the operand stack is structured and its layout determined statically at all program points, which is crucial for efficient compilation on a register machine. Furthermore, it establishes memory and state *encapsulation*—that is, abstraction properties on the module and function boundaries, which cannot leak information.

Given our formal definition of the language, soundness can be made precise as a fairly simple theorem:

THEOREM 3.1. (SOUNDNESS). *If* $\vdash s; f; e^* : t^n$ *(i.e., configuration* $s; f; e^*$ *is valid with resulting stack type* t^n*), then:*

- *either* $s; f; e^* \hookrightarrow {}^* s'; f'; (t.\mathbf{const}\ c)^n$ *(i.e., after a finite number of steps the instruction sequence has been reduced to values of the correct types),*
- *or* $s; f; e^* \hookrightarrow {}^* s'; f'; \mathbf{trap}$ *(i.e., execution traps after a finite number of steps),*
- *or execution diverges (i.e., there is an infinite sequence of reduction steps it can take).*

This formulation uses a typing judgement generalized to configurations whose definition we omit here. The property ensures that all valid programs either diverge, trap, or terminate with values of the expected types. The proofs are completely standard (almost boring) induction proofs like can be found in many text books or papers on the subject.

3.4. Mechanization

For our paper, we have done the soundness proofs by hand, on paper. We also have implemented a WebAssembly reference interpreter in OCaml that consists of a direct transliteration of the formal rules into executable code (Section 4.1). While both tasks were largely straightforward, they are always subject to subtle errors not uncovered by tests.

Fortunately, over the last 15 years, methodologies for *mechanizing* language semantics and their meta-theory in theorem provers have made significant advances. Because our formalization uses well-established techniques, other researchers have been able to apply mechanization to it immediately. As a result, there are multiple projects for mechanizing the semantics—and in fact, the full language definition (Section 4)—in three major theorem provers and semantics tools, namely Isabelle, Coq, and K. Their motivation is in both verifying WebAssembly itself as well as providing a foundation for other formal methods applications, such as verifying compilers targeting WebAssembly or proving properties of programs, program equivalences, and security properties.

The Isabelle mechanization was completed first and has already been published.[14] It not only contains a machine-verified version of our soundness proof, it also includes a machine-verified version of a validator and interpreter for WebAssembly that other implementations and our reference interpreter can be compared against. Moreover, in the process of mechanizing the soundness proof this work uncovered a few minor bugs in the draft version of our formalization and enabled us to fix it in a timely manner for publication.

4. STANDARDIZATION

The previous section reflects the formalization of WebAssembly as published in our Programming Language Design and Implementation (PLDI) paper[4] with a few minor stylistic modifications. However, our reason for developing this semantics was more than producing a paper targeted at researchers—it was meant to be the basis of the official language definition.[15] This definition, which is currently under review as a standard for the W3C, contains the complete formalization of the language.

4.1. Core language

The language definition follows the formalization and specifies abstract syntax, typing rules, reduction rules, and an abstract store. Binary format and text format are given as attribute grammars exactly describing the abstract syntax they produce. As far as we are aware, this level of rigor and precision is unprecedented for industrial-strength languages.

Formalism. Although the formal methods we use are standard in academic literature and computer science (CS)

curricula, a widely consumed standard cannot (yet) assume that all its readers are familiar with formal notation for semantics (unlike for syntax). Next to the formal rules, the specification hence also contains corresponding prose. This prose is intended to be a one-to-one "text rendering" of the formal rules. Although the prose follows the highly verbose "pseudo-COBOL" style of other language definitions, its eyeball proximity to a verified formalism aids spotting bugs. Having the formal rules featured centrally in the standard document hence benefits even readers that do not read them directly.

Reference interpreter. Along with the formalization and production implementations in browsers, we developed a reference interpreter for WebAssembly. For this we used OCaml due to the ability to write in a high-level stylized fashion that closely matches the formalization, approximating an "executable specification." The interpreter is used to develop the test suite, test production implementations and the formal specification, and to prototype new features.

Proposal process. To maintain the current level of rigor while evolving WebAssembly further, we have adopted a multi-staged proposal process with strong requirements. At various stages of a proposal, its champions must provide (1) an informal description, (2) a prose specification, (3) a prototype implementation, (4) a comprehensive test suite, (5) a formal specification, (6) an implementation in the reference interpreter, and (7) two implementations in independent production systems.

The process is public on the working group's Git repository, where specification, reference interpreter, and test suite are hosted. Creating a proposal involves asking the group to create a fork of the main "spec" repository and then iterating and reviewing all required additions there.

Obviously, a formal semantics is not straightforward in all cases. Where necessary, the working group is collaborating with research groups for non-trivial features, such as a suitable weak memory model for the addition of threads.

4.2. Embedding

WebAssembly is similar to a virtual Instruction Set Architecture (ISA) in that it does not define how programs are loaded into the execution engine or how they perform I/O. This intentional design separation is captured in the notion of *embedding* a WebAssembly implementation into an execution environment. The embedder defines how modules are loaded, how imports and exports are resolved, how traps are handled, and provides foreign functions for accessing the environment.

To strengthen platform-independence and encourage other embeddings of WebAssembly, the standard has been layered into separate documents: while the core specification only defines the virtual ISA, separate *embedder specifications* define its interaction with concrete host environments.

JavaScript and the web. In a browser, WebAssembly modules can be loaded, compiled and invoked through a JavaScript API. The rough recipe is to (1) acquire a binary module from a given source, for example, as a network resource, (2) instantiate it providing the necessary imports, and (3) call the desired export functions. Since compilation and instantiation may be slow, they are provided as asynchronous methods whose results are wrapped in promises. The JavaScript API also allows creating and initializing memories or tables externally, or accessing them as exports.

Interoperability. It is possible to link multiple modules that have been created by different producers. However, as a low-level language, WebAssembly does not provide any built-in object model. It is up to producers to map their data types to memory. This design provides maximum flexibility to producers, and unlike previous VMs, does not privilege any specific programming paradigm or object model.

Interested producers can define common ABIs *on top of* WebAssembly such that modules can interoperate in heterogeneous applications. This separation of concerns is vital for making WebAssembly universal as a code format.

5. IMPLEMENTATION

A major design goal of WebAssembly has been high performance without sacrificing safety or portability. Throughout its design process, we have developed independent implementations of WebAssembly in all major browsers to validate and inform the design decisions. This section describes some points of interest of those implementations.

Implementation strategies. V8 (Chrome), SpiderMonkey (Firefox) and JavaScriptCore (WebKit) reuse their optimizing JavaScript compilers to compile WebAssembly modules ahead-of-time. This achieves predictable high performance and avoids the unpredictability of warmup time which has often been a problem for JavaScript. Chakra (Edge) instead lazily translates individual functions to an interpreted internal bytecode format upon first execution, and later Just In Time (JIT)-compiles the hottest functions. The advantage is faster startup and potentially lower memory consumption. We expect more strategies to evolve over time.

Validation. In the four aforementioned implementations, the same algorithmic strategy using abstract control and operand stacks is used. Validation of incoming bytecodes occurs in a single pass during decoding, requiring no additional intermediate representation. We measured single-threaded validation speed at between 75MB/s and 150MB/s on a suite of representative benchmarks on a modern workstation. This is approximately fast enough to perform validation at full network speed.

Baseline JIT compiler. The SpiderMonkey engine includes two WebAssembly compilation tiers. The first is a fast baseline JIT that emits machine code in a single pass combined with validation. The JIT creates no Intermediate Representation (IR) but does track register state and attempts to do simple greedy register allocation in the forward pass. The baseline JIT is designed only for fast startup while an optimizing JIT is compiling the module in parallel in the background. V8 includes a similar baseline JIT in a prototype configuration.

Optimizing JIT compiler. All four engines include optimizing JITs for their top-tier execution of JavaScript and reuse them for WebAssembly. Both V8 and SpiderMonkey use SSA-based intermediate representations. As such, it was important that WebAssembly can be decoded to SSA form in a single pass. This is greatly helped by WebAssembly's structured control flow, making the decoding algorithm simpler

and more efficient and avoiding the limitation of JITs that usually do not support irreducible control flow. Reusing the advanced JITs from four different JavaScript engines has been a resounding success that allowed all engines to achieve high performance in a short time.

Bounds checks. By design, all memory accesses in WebAssembly can be guaranteed safe with a single dynamic bounds check, which amounts to checking the address against the current size of the memory. An engine will allocate the memory in a large contiguous range beginning at some (possibly non-deterministic) *base* in the engine's process, so that all access amounts to a hardware address *base+addr*. While *base* can be stored in a dedicated machine register for quick access, a more aggressive strategy is to *specialize* the machine code for each instance to a specific base, embedding it as a constant directly into the code, freeing a register. Although the base may change when the memory is grown dynamically, it changes so infrequently that it is affordable to *patch* the machine code when it does.

On 64 bit platforms, an engine can make use of virtual memory to eliminate bounds checks for memory accesses altogether. The engine simply reserves 8GB of virtual address space and marks as inaccessible all pages except the valid portion of memory near the start. Since WebAssembly memory addresses and offsets are 32 bit integers plus a static constant, by construction no access can be further than 8GB away from *base*. Consequently, the JIT can simply emit plain load/store instructions and rely on hardware protection mechanisms to catch out-of-bounds accesses.

Parallel and streaming compilation. With ahead-of-time compilation it is a clear performance win to parallelize compilation of WebAssembly modules, dispatching individual functions to different threads. For example, both V8 and SpiderMonkey achieve a 5-6× improvement in compilation speed with eight compilation threads. In addition, the design of the WebAssembly binary format supports *streaming* where an engine can start compilation of individual functions before the full binary has been loaded. When combined with parallelization, this minimizes cold startup.

Code caching Besides cold startup, warm startup time is important as users will likely visit the same Web pages repeatedly. The JavaScript API for the IndexedDB database allows JavaScript to manipulate and compile WebAssembly modules and store their compiled representation as an opaque blob. This allows a JavaScript application to first query IndexedDB for a cached version of their WebAssembly module before downloading and compiling it. In V8 and SpiderMonkey, this mechanism can offer an order of magnitude improvement of warm startup time.

5.1. Measurements

Execution. Figure 4 shows the execution time of the PolyBenchC benchmark suite running on WebAssembly on both V8 and SpiderMonkey normalized to native execution.[d] Times for both engines are shown as stacked bars, and the results show that there are still some differences between them due to different code generators.[e] We measured a VM startup time of 18 ms for V8 and 30ms for SpiderMonkey. These times are included along with compilation times as bars stacked on top of the execution time of each benchmark. Overall, the results show that WebAssembly is competitive with native code, with seven benchmarks within 10% of native and nearly all of them within 2× of native.

We also measured the execution time of the PolyBenchC benchmarks running on asm.js. On average, WebAssembly is 33.7% faster than asm.js. Especially validation is significantly more efficient. For SpiderMonkey, WebAssembly validation takes less than 3% of the time of asm.js validation. In V8, memory consumption of WebAssembly validation is less than 1% of that for asm.js validation.

Code size. Figure 5 compares code sizes between WebAssembly, minified asm.js, and ×86-64 native code. For the asm.js comparison we use the Unity benchmarks, for the native code comparison the PolyBenchC and SciMark benchmarks. For each function in these benchmarks, a yellow point is plotted at $\langle size_{asmjs}, size_{wasm} \rangle$ and a blue point at $\langle size_{x86}, size_{wasm} \rangle$. Any point below the diagonal represents code for which WebAssembly is smaller than the corresponding other representation. On average, WebAssembly code is 62.5% the size of asm.js, and 85.3% of native ×86-64 code.

6. RELATED WORK

Microsoft's ActiveX was a technology for code-signing ×86

Figure 4. Relative execution time of the Poly-BenchC benchmarks on WebAssembly normalized to native code.

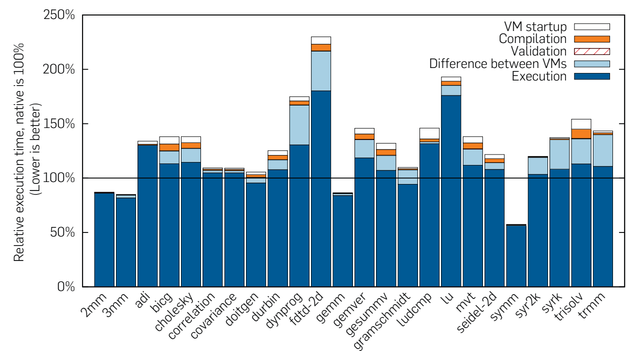
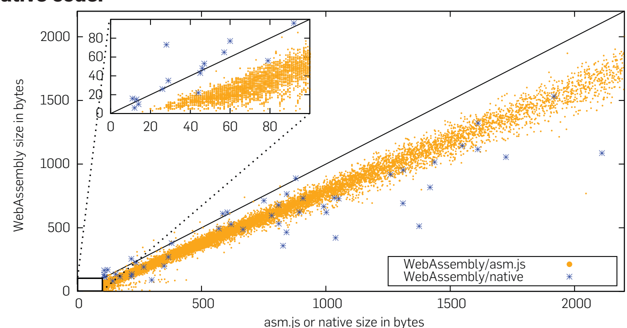


Figure 5. Binary size of WebAssembly in comparison to asm.js and native code.



[d] See Haas et al.[4] for details on the experimental setup for these measurements.

[e] V8 is faster on some benchmarks and SpiderMonkey on others. Neither engine is universally faster than the other.

binaries to run on the Web. It relied entirely upon trust and thus did not achieve safety through technical construction.

Native Client[17] was the first system to introduce a sand-boxing technique for machine code on the Web that runs at near native speed. It relies on static validation of ×86 machine code, requiring code generators to follow certain patterns, such as bit masks before memory accesses and jumps. Instead of hardware-specific ×86 code, Portable Native Client (PNaCl) uses a stable subset of LLVM bitcode[6] as an interchange format.

Emscripten[18] is a framework for compiling C/C++ applications to a specialized subset of JavaScript that later evolved into asm.js,[1] an embedded domain specific language that serves as a statically-typed assembly-like language and eschews the dynamic types of JavaScript through additional type coercions coupled with a module-level validation of interprocedural invariants.

Efficient memory safety is a hard design constraint of WebAssembly. Previous systems such as CCured[11] and Cyclone[5] have imposed safety at the C language level, which generally requires program changes. Other attempts have enforced it at the C abstract machine level with combinations of static and runtime checks, sometimes assisted by hardware. For example, the Secure Virtual Architecture[2] defines an abstract machine based on LLVM bitcode that enforces the SAFECode[3] properties.

We investigated reusing other compiler IRs that have a binary format, such as LLVM. Disadvantages with LLVM bitcode in particular are that it is not entirely stable, has undefined behavior, and is less compact than a stack machine. Furthermore, it requires every consumer to either include LLVM, which is notoriously slow, or reimplement a fairly complex LLVM IR decoder/verifier. In general, compiler IRs are better suited to optimization and transformation, and not as compact, verifiable code formats.

In comparison to safe "C" machines, typed intermediate languages, and typed assembly languages,[9] WebAssembly radically reduces the scope of responsibility for the VM: it is not required to enforce the type system of the original program at the granularity of individual objects; instead it must only enforce memory safety at the much coarser granularity of a module's memory. This can be done efficiently with simple bounds checks or virtual memory techniques.

The speed and simplicity of bytecode validation is key to good performance and high assurance. Our work was informed by experience with stack machines such as the Java Virtual Machine (JVM)[8], Common Intermediate Language (CIL)[10], and their validation algorithms. It took a decade of research to properly systematize the details of correct JVM verification,[7] including the discovery of vulnerabilities. By designing WebAssembly in lockstep with a formalization we managed to make its semantics drastically simpler: for example, instead of 150 pages for JVM bytecode verification, just a single page of formal notation.

7. FUTURE DIRECTIONS

The initial version of WebAssembly presented here consciously focuses on supporting *low-level* code, specifically compiled from C/C++. A few important features are still missing for fully comprehensive support of this domain and will be added in future versions, such as *exceptions*, *threads*, and *Single Instruction Multiple Data (SIMD)* instructions. Some of these features are already being prototyped in implementations of WebAssembly.

Beyond these, we intend to evolve WebAssembly further into an attractive target for *high-level* languages by including relevant primitives like *tail calls*, *stack switching*, or *coroutines*. A highly important goal is to provide access to the advanced and highly tuned *garbage collectors* that are built into all Web browsers, thus eliminating the main shortcoming relative to JavaScript when compiling to the Web.

In addition to the Web, we anticipate that WebAssembly will find a wide range of uses in other domains. In fact, multiple other embeddings are already being developed: for sandboxing in content delivery networks, for smart contracts or decentralized cloud computing on blockchains, as code formats for mobile devices, and even as mere stand-alone engines for providing portable language runtimes.

Many challenges lie ahead in supporting all these features and usage scenarios equally well while maintaining the level of precision, rigor, and performance that has been achieved with the initial version of WebAssembly. **C**

References

1. asm.js. http://asmjs.org. Accessed: 2016-11-08.
2. Criswell, J., Lenharth, A., Dhurjati, D., Adve, V. Secure virtual architecture: a safe execution environment for commodity operating systems. *Operating Systems Review 41*, 6 (Oct. 2007), 351–366.
3. Dhurjati, D., Kowshik, S., Adve, V. SAFECode: enforcing alias analysis for weakly typed languages. In *Programming Language Design and Implementation (PLDI)* (2006).
4. Haas, A., Rossberg, A., Schuff, D., Titzer, B., Gohman, D., Wagner, L., Zakai, A., Bastien, J. Bringing the web up to speed with WebAssembly. In *Programming Language Design and Implementation (PLDI)* (2017).
5. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference (ATEC)* (2002).
6. Lattner, C., Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)* (2004).
7. Leroy, X. Java bytecode verification: algorithms and formalizations. J. *Automated Reason. 30*, 3–4 (Aug. 2003), 235–269.
8. Lindholm, T., Yellin, F., Bracha, G., Buckley, A. The Java Virtual Machine Specification (Java SE 8 Edition). Technical report, Oracle, 2015.
9. Morrisett, G., Walker, D., Crary, K., Glew, N. From system F to typed assembly language. *ACM Trans. Program. Lang. Sys. (TOPLAS) 21*, 3 (May 1999), 527–568.
10. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W. CIL: intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction (CC)* (2002).
11. Necula, G.C., McPeak, S., Weimer, W. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)* (2002).
12. Pierce, B. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, USA, 2002.
13. Plotkin, G. A structural approach to operational semantics. J. *Logic and Algebraic Program.* (2004), 60–61:17–139.
14. Watt, C. Mechanising and verifying the WebAssembly specification. In *Certified Programs and Proofs (CPP)* (2018).
15. WebAssembly Community Group. WebAssembly Specification, 2018. https://webassembly.github.io/spec/.
16. Wright, A., Felleisen, M. A syntactic approach to type soundness. *Inf. Comput. 115*, 1 (Nov. 1994), 38–94.
17. Yee, B., Sehr, D., Dardyk, G., Chen, B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N. Native client: a sandbox for portable, untrusted ×86 native code. In *IEEE Symposium on Security and Privacy* (2009).
18. Zakai, A. Emscripten: an LLVM-to-JavaScript compiler. In *Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA)* (2011).

Andreas Rossberg (rossberg@mpi-sws.org), Dfinity Stiftung, Germany.

Ben L. Titzer and Andreas Haas ([titzer, ahaas]@google.com), Google GmbH, Germany.

Derek L. Schuff, (dschuff@google.com), Google Inc, USA.

Dan Gohman, Luke Wagner, and Alon Zakai ([sunfishcode, luke, azakai]@mozilla.com), Mozilla Inc, USA.

JF Bastien (jfbastien@apple.com), Apple Inc, USA.

Michael Holman (michael.holman@microsoft.com), Microsoft Inc, USA.

Copyright held by authors/owners.
Publication rights licensed to ACM. $15.00.