

Bringing the Web up to Speed with WebAssembly [Draft]

Andreas Haas Andreas Rossberg Derek Schuff Ben Titzer Dan Gohman Luke Wagner Alon Zakai JF Bastien Michael Holman
{ahaas,rossberg,dschuff,titzer}@google.com {sunfishcode,luke,azakai}@mozilla.com jfbastien@apple.com michael.holman@microsoft.com

Abstract

The rapid improvement in JavaScript virtual machines combined with maturation of Web platform tools has given rise to sophisticated and demanding web applications such as interactive 3D maps, audio and video software, and games. The increasing ambition of these applications has made the efficiency and security of mobile code on the web more important than ever. Yet JavaScript has inconsistent performance and a number of other pitfalls, especially as a compilation target. Engineers from all major browser vendors have risen to the challenge and collaboratively designed a new low-level byte code for the web called WebAssembly. It offers compact representation, fast and simple validation and compilation, low to no-overhead safe execution, and easy interoperability with the web platform, including direct access to JavaScript and Web APIs. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it both language- and platform-independent. We describe the motivation, design and formal semantics of WebAssembly and provide some preliminary experience with implementations.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Macro and assembly languages; D.3.4 [Programming Languages]: Processors—Compilers, Run-time environments

1. Introduction

The Web began as a simple document network but has now become the widest available 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, its ubiquity unmatched by other technologies available only via plugins like ActiveX, Java or Shockwave Flash. Because of its ubiquity and improved performance, and perhaps through sheer necessity, JavaScript has become a compilation target for other languages. Through Emscripten [40], even C and C++ programs can be compiled to a stylized low-level subset of JavaScript called asm.js [4].

WebAssembly addresses the problem of portable, fast, and safe low-level code on the Web. Previous attempts at solving this problem, from ActiveX to Native Client, to asm.js, have fallen short of the design goals that a low-level compilation target 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 and universal interoperability with the web platform
- Safe and efficient *representation*:
 - compact and easy to decode
 - easy to 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 system such as the browser’s native JavaScript virtual machine or a language plugin. Managed languages enforce *memory safety*, preventing programs from accessing or compromising the embedder’s internals, user data, or system state. However, managed languages have traditionally been mismatched with portable low-level code, such as compiled C/C++ applications, which do not need garbage collection and are inherently fast but memory unsafe.

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 a highly 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.

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 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 as compact as possible to reduce load times, save potentially expensive 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.

1.1 Prior attempts at low-level code on the Web

Microsoft’s ActiveX[1] was a technology for code-signing x86 binaries to run on the Web. It relied entirely upon code signing and thus did not achieve safety through technical construction, but through a trust model.

Native Client [39, 8] was the first system to introduce a sandboxing technique for machine code on the Web that runs at near native speed. It relies on static validation of x86 machine code, requiring code generators to follow certain patterns, such as bitmasks before memory accesses and jumps. While the sandbox model allows NaCl code in the same process with sensitive data, the challenges of doing so in the browser process led to an out-of-process implementation where NaCl code cannot synchronously access JavaScript or Web APIs. Because NaCl is a subset of a particular architecture’s machine code, it is inherently not portable. Portable Native Client (PNaCl) [15] builds upon the sandboxing techniques of NaCl and uses a stable subset of LLVM bitcode [21] as an interchange format, which addresses ISA portability. However, it is not a significant improvement in compactness and still exposes compiler- or platform-specific details such as the layout of the call stack. NaCl and PNaCl are integrated into Chrome, but other browser vendors have not adopted them, in part because they require separate implementations of bindings to all Web APIs.

Emscripten [40] is a framework for compiling mostly unmodified C/C++ applications to JavaScript and linking them with a JavaScript execution environment. Emscripten compiles to a specialized subset of JavaScript that later evolved into the assembly-like asm.js [4], an embedded domain specific statically-typed language. Asm.js eschews the dynamic type system of JavaScript through additional type coercions coupled with a module-level validation of interprocedural invariants. Since asm.js is a proper subset of JavaScript, it runs on all JavaScript execution engines, generally benefiting from sophisticated JIT compilers. However, it runs much faster in browsers with dedicated support. Being a subset inherently ties it to JavaScript semantics, and therefore extending asm.js with new features such as int64 requires first extending JavaScript and then blessing the extension in the asm.js subset. Even then it can be difficult to make the feature efficient. Asm.js is also not a significant compactness improvement over native code, as the type annotations consume space even when minified and compressed.

While Java and Shockwave Flash [2] were early comers to the Web and offered managed runtimes via plugins, neither support high-performance low-level code, and usage of both is declining due to security and performance issues. We discuss the differences between the JVM and WebAssembly in Section 8.

1.2 Contributions

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 major browser vendors and an online community group to build a common solution for high-performance applications. To our knowledge, it is the first industrial-strength VM that has been designed with a formal semantics from the start. In this paper we provide

- an overview of WebAssembly as a language that is the first truly cross-browser solution for fast low-level code,
- an in-depth discussion of its design, including insight into novel design decisions such as structured control flow,
- a complete yet concise formal semantics of both execution and validation, including a proof of soundness,
- a report on implementation experience from developing several production implementations available in 3 major browsers, including novel techniques such as for memory protection.

This paper focuses on design and formalization. We leave an empirical evaluation to future work.¹

While the Web is the primary motivation for WebAssembly, nothing in its design depends on the Web or a JavaScript environment. It is a completely open standard, and we expect that stand-alone implementations will become available in the future. The initial version primarily focuses on supporting low-level languages like C++, but we intend to grow it further in the future (Section 9).

2. Overview

Even though WebAssembly is a binary code format, we present it as a language with syntax and structure. This was an intentional design choice which makes it easier to explain and understand, without compromising compactness or ease of decoding. Figure 1 presents its structure in terms of abstract syntax.² For brevity we omit some minor features; see Section 2.6.

¹ *Reviewer Note:* Some preliminary numbers backing up the claims in this paper can be found in the Supplementary Appendix.

² WebAssembly has an S-expression text representation that closely resembles this syntax and tools for assembling binaries from it, e.g. to write tests.

```

module
  func (f64 → ε) import "console" "print"
  export "printFac" func (i32 → ε)
    (get_local 0) (i64.convert i32.u) (call 2) (f64.convert i64.u) (call 0)
  func (i64 → i64)
    local i64 ;; result
    block (ε → ε) ;; break
      loop (ε → ε) ;; continue
        (get_local 0) i64.eqz (br_if 1) ;; break
        (get_local 0) (get_local 1) i64.mul (set_local 1)
        (get_local 0) (i64.const 1) i64.sub (set_local 0)
        (br 0) ;; continue
      end
    end
  (get_local 1) ;; result

```

Figure 2. Example: Module with factorial function

2.1 Basics

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

Modules A WebAssembly binary takes the form of a *module*. It contains possibly empty sequences of definitions for *functions*, *globals*, *tables*, and *memories*. Each definition can optionally be *exported* under one or more names. Definitions can also be *imported*, specifying a module/item name pair and a suitable type. Imports can be re-exported.

While a module corresponds to the static representation of a program, an *instance* of a module corresponds to a dynamic representation, complete with mutable memory and an execution stack. The instantiation operation for modules is provided by the *embedder*, such as a JavaScript virtual machine or an operating system. Instantiating a module requires providing definitions for all imports, which may be exports from previously created WebAssembly instances. WebAssembly computation can then be initiated by invoking an exported function from this instance.

Functions The code in a module is organized into individual *functions*. Each function takes a sequence of WebAssembly values as parameters and returns a sequence of values as results as defined by its *function type*. Functions can call each other, including recursively. Functions are not first class and cannot be nested within each other. As we will see later, the contents of the call stack for execution are not exposed, and thus cannot be directly accessed by a running WebAssembly program, even a buggy or malicious one.

Instructions The computational model of WebAssembly is based on a *stack machine* in that the code for a function consists of a sequence of *instructions* that manipulate data values on an implicit *operand stack*, popping argument values and pushing result values. However, thanks to the type system (Section 4), the layout of the operand stack can be statically determined at any point in the code, so that actual implementations can compile the data flow between instructions directly without ever materializing the operand stack.³ The stack organization is merely a way to achieve a compact program representation, as it has been to be shown to be smaller than a register machine [35].⁴

³ While structured use of the stack is normally the case for Java bytecode, since it normally originates from Java source code, structured use of the stack is *enforced* for WebAssembly code, leading to a host of benefits.

⁴ We also explored compressed, byte-encoded ASTs for WebAssembly, first with a pre-order encoding and then later with a post-order encoding, even going so far as to field full-scale production prototypes and development tools for both representations. We found that post-order ASTs decode and

(value types)	$t ::= i32 \mid i64 \mid f32 \mid f64$	(instructions)	$e ::= \text{unreachable} \mid \text{nop} \mid \text{drop} \mid \text{select} \mid$
(packed types)	$tp ::= i8 \mid i16 \mid i32$		$\text{block } tf \ e^* \ \text{end} \mid \text{loop } tf \ e^* \ \text{end} \mid \text{if } tf \ e^* \ \text{else } e^* \ \text{end} \mid$
(function types)	$tf ::= t^* \rightarrow t^*$		$\text{br } i \mid \text{br_if } i \mid \text{br_table } i^+ \mid \text{return} \mid \text{call } i \mid \text{call_indirect } tf \mid$
(global types)	$tg ::= \text{mut}^? \ t$		$\text{get_local } i \mid \text{set_local } i \mid \text{tee_local } i \mid \text{get_global } i \mid$
			$\text{set_global } i \mid t.\text{load } (tp_sx)^? \ a \ o \mid t.\text{store } tp^? \ a \ o \mid$
			$\text{current_memory} \mid \text{grow_memory} \mid t.\text{const } c \mid$
			$t.\text{unop}_t \mid t.\text{binop}_t \mid t.\text{testop}_t \mid t.\text{relop}_t \mid t.\text{cvtop } t_sx^?$
$unop_{iN}$	$::= \text{clz} \mid \text{ctz} \mid \text{popcnt}$	(functions)	$f ::= ex^* \ \text{func } tf \ \text{local } t^* \ e^* \mid ex^* \ \text{func } tf \ im$
$unop_{fN}$	$::= \text{neg} \mid \text{abs} \mid \text{ceil} \mid \text{floor} \mid \text{trunc} \mid \text{nearest} \mid \text{sqrt}$	(globals)	$glob ::= ex^* \ \text{global } tg \ e^* \mid ex^* \ \text{global } tg \ im$
$binop_{iN}$	$::= \text{add} \mid \text{sub} \mid \text{mul} \mid \text{div}_{sx} \mid \text{rem}_{sx} \mid$	(tables)	$tab ::= ex^* \ \text{table } n \ i^* \mid ex^* \ \text{table } n \ im$
	$\text{and} \mid \text{or} \mid \text{xor} \mid \text{shl} \mid \text{shr}_{sx} \mid \text{rotr} \mid \text{rotr}$	(memories)	$mem ::= ex^* \ \text{memory } n \mid ex^* \ \text{memory } n \ im$
$binop_{fN}$	$::= \text{add} \mid \text{sub} \mid \text{mul} \mid \text{div} \mid \text{min} \mid \text{max} \mid \text{copysign}$	(imports)	$im ::= \text{import } \text{"name"}^? \ \text{"name"}$
$testop_{iN}$	$::= \text{eqz}$	(exports)	$ex ::= \text{export } \text{"name"}$
$relop_{iN}$	$::= \text{eq} \mid \text{ne} \mid \text{lt}_{sx} \mid \text{gt}_{sx} \mid \text{le}_{sx} \mid \text{ge}_{sx}$	(modules)	$m ::= \text{module } f^* \ glob^* \ tab^? \ mem^?$
$relop_{fN}$	$::= \text{eq} \mid \text{ne} \mid \text{lt} \mid \text{gt} \mid \text{le} \mid \text{ge}$		
$cvtop$	$::= \text{convert} \mid \text{reinterpret}$		
sx	$::= \text{s} \mid \text{u}$		

Figure 1. WebAssembly abstract syntax

Traps Some instructions may produce a *trap* under some conditions, 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. Embedded in JavaScript, a WebAssembly trap will create and throw a JavaScript exception, complete with a WebAssembly-level stack trace, that can be caught by the invoking JavaScript code.

Machine Types WebAssembly programs have only four basic *value types* t , all of which are available in common hardware. These are integers and IEEE-754 floating point numbers, each in 32 and 64 bit width. 32 bit integers also function as Booleans, as addresses in the linear memory (Section 2.2), and indexes into function tables (Section 2.4). Most WebAssembly instructions simply execute various *operators* on these basic data types. The grammar in Figure 1 conveniently distinguishes between several categories, such as *unary* and *binary* operators, *tests* and *comparisons*. WebAssembly provides the full matrix of *conversions* between the four types, and the ability to *reinterpret* the bits of values across equally-sized types. Like common hardware, WebAssembly has no distinction between signed and unsigned integer types. Instead, when the signedness of values matters to an instruction, a *sign extension* suffix $_u$ or $_s$ selects either unsigned or 2’s complement signed behavior.

Local Variables Functions may declare mutable *local variables* (of types t^*) that are usable as virtual registers, in order to express imperative variables or more complex data flow. Locals are zero-initialized and read or written by index via the `get_local` and `set_local` instructions, respectively; `tee_local` allows writing a local variable while leaving the input value on the operand stack, which is very common in real code. The index space for local variables starts with and includes the function parameters, meaning that function parameters are also mutable.

Global Variables A module may also declare typed *global variables* that are accessed with the `get_global` and `set_global` instructions to read or write individual values. Globals can be either mutable or immutable and require an initializer which must be a *constant expression* that evaluates without access to any function, table, memory, local or mutable global. Importing globals and initializer expressions allow a limited form of configurability, e.g. for linking.

verify faster than pre-order ASTs, but that the stack machine, which can be seen as a generalization of the post-order format, more easily extended to multi-value support and allowed even more space optimizations.

So far so boring. In the following sections we turn our attention to more interesting features of the WebAssembly semantics.

2.2 Linear Memory

The main storage of a WebAssembly program is a large array of bytes referred to as a *linear memory* or simply *memory*.

Creation and Growing Each module can define at most one **memory**, which may optionally be shared with other instances via import/export. Memory is created with an initial size but may be dynamically grown with the `grow_memory` instruction. Growing may fail with an out-of-memory condition, which is indicated by `grow_memory` returning -1 to be handled by the program.⁵ The size can be queried with the `current_memory` instruction. The unit of size and growth is a *page*, which is defined to be 64 KiB, the least common multiple of minimum page sizes on modern virtual memory hardware. The page size allows reusing virtual memory hardware for bounds checks (Section 7). Page size is fixed instead of being system-specific to prevent a common portability hazard.

Access Memory is accessed with `load` and `store` instructions that take a static alignment exponent a , a positive static offset o , an optional static width expressed as a *packed type* tp , and the dynamic `i32` address. Addresses are simply unsigned integers starting at 0. The *effective address* of an access is the sum of the static offset o (32 bits) and the dynamic `i32` address as a 33 bit address (i.e. no wraparound), which allows specific optimizations shown in Section 7. Memory access is always dynamically checked against the memory size, and out of bounds access results in a trap. Memory can be accessed with 8, 16, 32, or 64-bit wide loads and stores, with *packed* integer loads performing a zero or sign extension sx to either 32 or 64 bits. Unaligned access, where 2^a is smaller than the (packed) type’s width, is supported, e.g. accessing a 32 bit integer on an odd address. Such access may be slow on some platforms, but always produces the same unexciting results.

Endianness Byte order in memory is observable to programs that load and store to aliased locations with different types or widths. Contemporary hardware seems to be converging on little-endian byte order, either being natively little-endian (x86, x86-64, ARM) or having optional endian conversion included in memory access (Sparc, PowerPC, PA-RISC, IA-64), or being architecturally neutral with both variants available (MIPS). Recognizing this convergence, we chose to define WebAssembly memory to have little-

⁵ To support additional optimizations, WebAssembly also allows declaring an upper limit for each memory’s size, which we omit in this presentation.

endian byte order. Of course, that entails that big-endian platforms require explicit endian conversions. However, these conversions can be subjected to classical compiler optimizations such as redundancy elimination and code motion by the WebAssembly engine. Inheriting host system endianness would also be a portability risk, since languages like C++ may depend on it statically. Thus the semantics of memory access is completely deterministic and portable across all engines and platforms, even for unaligned accesses and unrestricted type-punning.

Security Unlike CCured [31] or other systems for securing C/C++ code, WebAssembly does not enforce memory safety at the granularity of language objects, but only at the granularity of a whole memory. 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 exploited WebAssembly program can make a mess of the data in its own memory. This means that even untrusted modules can be safely executed in the same address space as other code. Achieving fast in-process isolation was a necessary design constraint for interacting with untrusted JavaScript and the full complement of Web APIs in a high-performance way. It also allows a WebAssembly engine to be embedded into any other managed language runtime without violating memory safety, as well as enabling programs with many independent instances with their own memory to exist in the same process.

2.3 Structured Control Flow

WebAssembly represents control flow differently from most stack machines. It does not offer simple 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, or even transformed to an SSA-form intermediate form in a single pass. Structured control flow disassembled to a text format is also easier to read, an often overlooked but important human factor on the web, where users are accustomed to inspecting the code of webpages to learn and share in an open manner.

Control Constructs and Blocks 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 in the instruction stream 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. The **if** construct encloses two blocks separated by an **else** opcode. The **else** can be omitted if the second block is empty. Executing an **if** pops an *i32* operand off the stack and executes one of the blocks depending on whether the value is non-zero.

Branches and labels Branches have “label” immediates that do not reference positions in the instruction stream but instead reference outer control constructs by relative nesting depth. That means that labels are effectively *scoped*: branches can only reference constructs in which they are nested. Taking a branch “breaks from” that construct’s block;⁶ the exact effect depends on the target construct: in case of a **block** or **if** it is a *forward* jump, resuming execution after the matching **end** (like a break statement); with a **loop** it is a *backward* jump, restarting the loop (like a continue statement).

Besides unconditional branches, **br_if** conditionally branches if its stack operand is non-zero, and **br_table** selects a target from

⁶The instruction name **br** can also be read as “break” wrt. to a block.

an entire list of label immediates based on an index operand, with the last label being the target for all out-of-bounds index values; having these two instructions allows minimal code that avoids any jumps-to-jumps.

Block Signatures and Unwinding Every control construct is annotated with a function type $t_f = t_1^* \rightarrow t_2^*$ that describes how it changes the stack.⁷ Conceptually, blocks execute like function calls. Each block pops its argument values t_1^* off the stack, creates a new stack, pushes the arguments onto the new stack, executes its instructions, pops its results off the internal stack, and then pushes its results t_2^* onto the outer stack upon completion. Since the beginning and end of a block represent control join points, all branches must also produce compatible stacks. Consequently, branch instructions themselves expect operands, depending on whether they jump to the start or end of the block, i.e., with types t_1^* for **loop** targets and t_2^* for **block** or **if**.

Branching *unwinds* a block’s local operand stack by implicitly popping all remaining operands from it, similar to returning from a function call. When a branch crosses several block boundaries, all respective stacks up to and including the target block’s are unwound. This liberates producers from having to track stack height across sub-expressions in order to make stack heights match up at branches by adding explicit drops.

Production implementations perform register allocation and compile away the operand stack when generating machine code. However, the design makes sure that interpretation is still relatively easy, e.g., to implement debuggers. An interpreter can represent the individual block stacks as contiguous and just remember the *height* upon entry to each block in a separate *control stack*. Further, an interpreter can make a prepass to construct a mapping from branches to instruction position and avoid dynamically searching for **end** opcodes, making all interpreter operations constant-time.⁸

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

<pre>switch (x) { case 0: ...A... case 1: ...B... break; default: ...C... }</pre>	becomes	<pre>block block block block br_table 2 1 0 end ...A... end ...B... br 1 end ...C... end</pre>
---	---------	--

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

By design, unstructured and irreducible control flow using **goto** is impossible in WebAssembly. 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. It was pioneered by tools like Emscripten [40] which uses the Relooper algorithm to target *asm.js*, handling both unstructured and irreducible control flow patterns. In our experience building an LLVM back-end targeting WebAssembly, irreducible control flow is rare, and a simple restructuring algorithm for introducing blocks is all that is necessary to compile any reducible CFG to WebAssembly. The benefit of requiring reducible control flow by construction is that many algorithms in consumers are much simpler and faster.

2.4 Function Calls and Tables

A function body is a block (Section 2.3) whose signature maps the empty stack to the function’s result. The arguments to a function

⁷In the initial version of WebAssembly, t_1^* must be empty and $|t_2^*| \leq 1$.

⁸That is the approach V8 takes in its debugging interpreter.

are stored in the first local variables of the function. Execution of a function can complete in one of three ways: (1) by reaching the end of the block, in which case the operand stack must match the function’s result types; (2) by a branch targeting the function block, with the result values as operands; (3) by executing **return**, which is a shorthand for a branch that targets the function’s block.

Direct Calls Functions can be invoked directly using the **call** instruction which takes a static index immediately identifying the function to call. The call instruction pops the required function arguments from the operand stack and pushes the function’s return values upon return.

Indirect Calls Function pointers and method tables can be emulated with the **call_indirect** instruction, which takes a runtime index into a global *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 expected type supplied to the **call_indirect** instruction. The dynamic signature check protects integrity of the execution environment, since native calls pass arguments in registers and on the stack; a successful signature check ensures that a single machine-level indirect jump to the compiled code of the target function is safe. In case of a type mismatch or an out of bounds table access, a trap occurs. 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.

External and Foreign Calls Functions can be imported to a module and are specified by name and signature. 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* through which a WebAssembly program can communicate with its embedding environment. For example, when WebAssembly is embedded in JavaScript, imported functions may be *host* functions that are not defined by WebAssembly code, but are defined in JavaScript. Values crossing the language boundary as arguments or results are automatically converted according to JavaScript rules.⁹

2.5 Determinism

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

However, there remain just three sources of implementation-dependent behavior which are best explained as non-determinism:

NaN Payloads WebAssembly follows the IEEE-754 standard for floating point arithmetic, which is generally fast on all modern CPUs. However, IEEE-754 does not specify the exact bit pattern for NaN values in all cases, and we found that CPUs differ with regard to the sign bit of *canonical NaNs* and the rules for *NaN propagation*. While programming languages generally make no specific guarantees in these cases and programs don’t normally observe NaN bit patterns, a low-level target like WebAssembly should give tighter guarantees.

We found that normalizing after every numeric operation is too expensive. However, based on our experience with JavaScript

⁹ Where trying to communicate an i64 value produces a JavaScript type error, because JavaScript cannot yet represent such values adequately.

engines, we established rules that are sufficient for allowing NaN-boxing [18]: (1) instructions only output canonical NaNs with a nondeterministic sign bit, unless (2) if an input is a non-canonical NaN, then the output NaN payload bits are nondeterministic.

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, which can be viewed semantically as a **grow_memory** instruction non-deterministically returning -1 . A **call** or **call_indirect** instruction may also experience *stack overflow*, but this is not semantically observable inside of WebAssembly, since it results in a trap.

Calling Host Functions WebAssembly programs can behave non-deterministically if they call host functions which are themselves non-deterministic or change WebAssembly state. This is expected since the effect of calling host functions is outside the realm of the WebAssembly semantics.

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

2.6 Restrictions

WebAssembly as presented in this paper is almost complete except some minor omitted features related to module initialization:

- Tables can be partially initialized, separately from their definition, and initialization can be applied to imported tables.
- Memories can be pre-initialized with given data at given offsets.
- A module can specify a designated *startup* function that is automatically executed upon instantiation.
- Tables and memories can have an optional maximum size that limits how much they can be grown.

Relative to this presentation, the initial release of WebAssembly also imposes a few restrictions, likely lifted in later versions:

- Blocks and functions may produce at most one result value.
- Blocks may not consume outer operands.
- Constant expressions for globals may only be of the form $(t.const\ c)$ or $(get_global\ i)$, where i refers to an import.

Modules may also define multiple tables and memories in the future.

3. Execution

Presenting WebAssembly as a language provides us with convenient and effective formal tools for specifying and reasoning about its semantics very precisely. In this section we define execution in terms of a standard *reduction* relation and in the next section we will define *validation* as a standard typing relation.

3.1 Stores and Instances

Execution operates relative to a global *store* s . The upper part of Figure 3 defines syntax for representations of stores and other runtime objects. A store is represented as a record of the lists of module instances, tables and memories that have been allocated in it. Indices into these lists can be thought of as addresses, and “allocation” simply appends to these lists.

As described in Section 2.1, a module must be *instantiated* before it can be used. The result is an initialized *instance*. Figure 3 represents such an instance as a record of the entities it defines. Tables and memories reside in the global store and are only referenced by address, since they can be shared between multiple instances.

(store)	s	::=	$\{\text{inst } \text{inst}^*, \text{tab } \text{tabinst}^*, \text{mem } \text{meminst}^*\}$
(instances)	inst	::=	$\{\text{func } \text{cl}^*, \text{glob } v^*, \text{tab } i^?, \text{mem } i^?\}$
	tabinst	::=	cl^*
	meminst	::=	b^*
(closures)	cl	::=	$\{\text{inst } i, \text{code } f\}$ (where f is not an import and has all exports ex^* erased)
(values)	v	::=	$t.\text{const } c$
(administrative operators)	e	::=	$\dots \mid \text{trap} \mid \text{call } \text{cl} \mid \text{label}\{t^*; e^*\} e^* \text{ end} \mid \text{local}\{i; v^*\} e^* \text{ end}$
(local contexts)	L^0	::=	$v^* [-] e^*$
	L^{i+1}	::=	$v^* \text{label}\{t^*; e^*\} L^i \text{end } e^*$

Reduction	$s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$	$s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$	$s; v^*; e^* \hookrightarrow_i s; v^*; e^*$
	$s; v^*; L^k[e^*] \hookrightarrow_i s'; v'^*; L^k[e'^*]$	$s; v_0^*; \text{local}\{i; v^*\} e^* \text{end} \hookrightarrow_j s'; v_0'^*; \text{local}\{i; v'^*\} e'^* \text{end}$	
	$(t.\text{const } c) t.\text{unop}$	\hookrightarrow	$t.\text{const } \text{unop}_t(c)$
	$(t.\text{const } c_1) (t.\text{const } c_2) t.\text{binop}$	\hookrightarrow	$t.\text{const } c$ if $c = \text{binop}_t(c_1, c_2)$
	$(t.\text{const } c_1) (t.\text{const } c_2) t.\text{binop}$	\hookrightarrow	trap otherwise
	$(t.\text{const } c) t.\text{testop}$	\hookrightarrow	i32.const testop $_t(c)$
	$(t.\text{const } c_1) (t.\text{const } c_2) t.\text{relop}$	\hookrightarrow	i32.const relop $_t(c_1, c_2)$
	$(t_1.\text{const } c) t_2.\text{convert } t_1\text{-sx}^?$	\hookrightarrow	$t_2.\text{const } c'$ if $c' = \text{cvt}_{t_1, t_2}^{\text{sx}}(c)$
	$(t_1.\text{const } c) t_2.\text{convert } t_1\text{-sx}^?$	\hookrightarrow	trap otherwise
	$(t_1.\text{const } c) t_2.\text{reinterpret } t_1$	\hookrightarrow	$t_2.\text{const } \text{const}_{t_2}(\text{bits}_{t_1}(c))$
	unreachable	\hookrightarrow	trap
	nop	\hookrightarrow	ϵ
	$v \text{ drop}$	\hookrightarrow	ϵ
	$v_1 v_2 (\text{i32.const } 0) \text{ select}$	\hookrightarrow	v_2
	$v_1 v_2 (\text{i32.const } i + 1) \text{ select}$	\hookrightarrow	v_1
	$v^n \text{ block } (t_1^n \rightarrow t_2^m) e^* \text{ end}$	\hookrightarrow	$\text{label}\{t_2^m; \epsilon\} v^n e^* \text{ end}$
	$v^n \text{ loop } (t_1^n \rightarrow t_2^m) e^* \text{ end}$	\hookrightarrow	$\text{label}\{t_1^n; \text{loop } (t_1^n \rightarrow t_2^m) e^* \text{ end}\} v^n e^* \text{ end}$
	$(\text{i32.const } 0) \text{ if } \text{tf } e_1^* \text{ else } e_2^* \text{ end}$	\hookrightarrow	block $\text{tf } e_2^* \text{ end}$
	$(\text{i32.const } i + 1) \text{ if } \text{tf } e_1^* \text{ else } e_2^* \text{ end}$	\hookrightarrow	block $\text{tf } e_1^* \text{ end}$
	$\text{label}\{t^*; e^*\} v^* \text{ end}$	\hookrightarrow	v^*
	$\text{label}\{t^*; e^*\} \text{ trap end}$	\hookrightarrow	trap
	$\text{label}\{t^n; e^*\} L^i[v^n (\text{br } i)] \text{ end}$	\hookrightarrow	$v^n e^*$
	$(\text{i32.const } 0) (\text{br_if } i)$	\hookrightarrow	ϵ
	$(\text{i32.const } j + 1) (\text{br_if } i)$	\hookrightarrow	br } i
	$(\text{i32.const } k) (\text{br_table } i_1^k i i_2^*)$	\hookrightarrow	br } i
	$(\text{i32.const } k + j) (\text{br_table } i_1^k i)$	\hookrightarrow	br } i
	$s; \text{call } j$	\hookrightarrow_i	$\text{call } s_{\text{func}}(i, j)$
	$s; (\text{i32.const } j) \text{ call_indirect } \text{tf}$	\hookrightarrow_i	$\text{call } s_{\text{tab}}(i, j)$ if $s_{\text{tab}}(i, j)_{\text{code}} = (\text{func } \text{tf } \text{local } t^* e^*)$
	$s; (\text{i32.const } j) \text{ call_indirect } \text{tf}$	\hookrightarrow_i	trap otherwise
	$v^n (\text{call } \text{cl})$	\hookrightarrow	$\text{local}\{cl_{\text{inst}}; v^n (t.\text{const } 0)^k\} \text{block } (\epsilon \rightarrow t_2^m) e^* \text{end end } \dots$
	$\text{local}\{i; v_i^*\} v^* \text{ end}$	\hookrightarrow	v^* \dots if $cl_{\text{code}} = (\text{func } (t_1^n \rightarrow t_2^m) \text{local } t^k e^*)$
	$\text{local}\{i; v_i^*\} \text{ trap end}$	\hookrightarrow	trap
	$\text{local}\{i; v_i^*\} L^{j+1}[\text{return}] \text{ end}$	\hookrightarrow	$\text{local}\{i; v_i^*\} L^{j+1}[\text{br } j] \text{ end}$
	$v_1^i v v_2^j; \text{get_local } i$	\hookrightarrow	v
	$v_1^i v v_2^j; v' (\text{set_local } i)$	\hookrightarrow	$v_1^i v' v_2^j; \epsilon$
	$v (\text{tee_local } i)$	\hookrightarrow	$v v (\text{set_local } i)$
	$s; \text{get_global } j$	\hookrightarrow_i	$s_{\text{glob}}(i, j)$
	$s; v (\text{set_global } j)$	\hookrightarrow_i	$s'; \epsilon$ if $s' = s$ with $\text{glob}(i, j) = v$
	$s; (\text{i32.const } k) (t.\text{load } a o)$	\hookrightarrow_i	$t.\text{const } \text{const}_t(b^*)$ if $s_{\text{mem}}(i, k + o, t) = b^*$
	$s; (\text{i32.const } k) (t.\text{load } \text{tp}\text{-sx } a o)$	\hookrightarrow_i	$t.\text{const } \text{const}_t^{\text{sx}}(b^*)$ if $s_{\text{mem}}(i, k + o, tp) = b^*$
	$s; (\text{i32.const } k) (t.\text{load } \text{tp}\text{-sx}^? a o)$	\hookrightarrow_i	trap otherwise
	$s; (\text{i32.const } k) (t.\text{const } c) (t.\text{store } a o)$	\hookrightarrow_i	$s'; \epsilon$ if $s' = s$ with $\text{mem}(i, k + o, t) = \text{bits}_i^{ t }(c)$
	$s; (\text{i32.const } k) (t.\text{const } c) (t.\text{store } \text{tp } a o)$	\hookrightarrow_i	$s'; \epsilon$ if $s' = s$ with $\text{mem}(i, k + o, tp) = \text{bits}_i^{ tp }(c)$
	$s; (\text{i32.const } k) (t.\text{const } c) (t.\text{store } \text{tp}^? a o)$	\hookrightarrow_i	trap otherwise
	$s; \text{current_memory}$	\hookrightarrow_i	i32.const $\lfloor s_{\text{mem}}(i, *) \rfloor / 64 \text{ Ki}$
	$s; (\text{i32.const } k) \text{ grow_memory}$	\hookrightarrow_i	$s'; \text{i32.const } \lfloor s_{\text{mem}}(i, *) \rfloor / 64 \text{ Ki}$ if $s' = s$ with $\text{mem}(i, *) = s_{\text{mem}}(i, *) (0)^{k \cdot 64 \text{ Ki}}$
	$s; (\text{i32.const } k) \text{ grow_memory}$	\hookrightarrow_i	i32.const (-1)

Figure 3. Small-step reduction rules

The representation of instantiated tables is simply a list of *closures* cl and memories a list of bytes b .

A closure is the runtime representation of a function, consisting of the actual function definition and a reference to the instance from which it originated. The instance is needed to access stateful objects such as the globals, tables, and memories, since functions can be imported from a different instance. A typical implementation will eliminate closures by specializing generated machine code to an instance.

Globals are represented by the values they hold. Since mutable globals cannot be aliased, they reside in their defining instance. Values are simply represented by a $t.\mathbf{const}$ instruction, which is convenient for the presentation of the reduction rules.

When dealing with the records introduced here, we use notation like s_{func} to refer to the func component of a store record s ; similarly for other records. Indexing $xs(i)$ denotes the i -element in a sequence xs . We extend it to stores with instance addresses i as a major dimension by the following short-hands:

$$\begin{aligned} s_{\text{func}}(i, j) &= s_{\text{inst}}(i)_{\text{func}}(j) & s_{\text{tab}}(i, j) &= s_{\text{tab}}(s_{\text{inst}}(i)_{\text{tab}})(j) \\ s_{\text{glob}}(i, j) &= s_{\text{inst}}(i)_{\text{glob}}(j) & s_{\text{mem}}(i, j) &= s_{\text{mem}}(s_{\text{inst}}(i)_{\text{mem}})(j) \end{aligned}$$

For memories, we generalize indexing notation to slices, i.e., $s_{\text{mem}}(i, j, k)$ denotes the byte sequence $s_{\text{mem}}(i, j) \dots s_{\text{mem}}(i, j + k - 1)$; plus, $s_{\text{mem}}(i, *)$ is the complete memory in instance i . Finally, we write “ s with $\text{glob}(i, j) = v$ ” for the store s' that is equal to s , except that $s'_{\text{glob}}(i, j) = v$; similarly for updating memories.

3.2 Instantiation

Instantiating a module $m = (\mathbf{module} \ f^* \ \mathit{glob}^* \ \mathit{tab}^? \ \mathit{mem}^?)$ in a store s requires providing external function closures cl_0^* , global values v_0^* , table index $i^?$ and memory index $k^?$, corresponding to the respective imports declared by the module in order of appearance. Their types must match the import declarations; for functions and globals that means an exact match, and for tables or memories, a reference to an instance with a size at least as large.

The new instance then is assigned a fresh address $i = |s_{\text{inst}}|$ and defined as follows:

$$\mathit{inst} = \{\text{func } cl^*, \text{ glob } v^*, \text{ tab } j^?, \text{ mem } k^?\}$$

where cl^* is the list of closures such that each $cl = \{\text{inst } i, \text{ code } f\}$ if it corresponds to a definition f in f^* that is not an import, or the corresponding import from cl_0^* otherwise. Each value in v^* is the result of evaluating the initializer expressions of a global definition, or the respective import from v_0^* otherwise. The indices i and j are either the imports, respectively, or $j = |s_{\text{tab}}|$ and $k = |s_{\text{mem}}|$ in case the table or memory are defined in m itself as (**table** $n_t \ i_t^{n_t}$) or (**memory** n_m). New table or memory instances are created from such definitions as:

$$\begin{aligned} \mathit{tabinst} &= (\mathit{inst}_{\text{func}}(i_t))^{n_t} \\ \mathit{meminst} &= (0)^{n_m \cdot 64 \text{Ki}} \end{aligned}$$

Instantiation results in a new store s' which is s with inst and possibly $\mathit{tabinst}$ and $\mathit{meminst}$ appended to the respective lists.

3.3 Reduction

The lower part of Figure 3 specifies WebAssembly execution in terms of a small-step reduction relation [33], which enables very compact rules and avoids the need for introducing separate notions of operand or control stacks – the operand “stack” simply consists of all $t.\mathbf{const}$ instructions left of the next redex. Reduction is defined over *configurations* $s; v^*; e^*$ that consist of a global store s , local variable values v^* , and the instruction sequence e^* to execute. In the reduction rules, we omit parts of the initial configuration that are not used by a rule, and parts of the resulting configuration that remain unchanged. Moreover, reduction is indexed by the address

i of the “current” instance it executes in, which we also omit where it is not relevant.

Administrative Syntax To deal with control constructs and functions, however, the syntax of instructions must be extended with a number of *administrative operators*: **trap** signifies that a trap has occurred, **call** cl is a call directly applied to a closure, **label** marks the label and extent of an active control construct, and **local** essentially is a call frame for function invocation. We will explain these constructs in more detail later. To aid the formulation of rules dealing with enclosing labels we additionally define *local contexts* L^k that represent k nested labels.

We abuse notation slightly and let x^n range over *sequences* of different x ’s with length n in the rules. We also write x^* where the length doesn’t matter; ϵ denotes the empty sequence.

Numerics The first group of rules handles the various numeric instructions. We assume a number of auxiliary operators:

$$\begin{aligned} \mathit{unop}_t &: t \rightarrow t & \mathit{cvt}_{t_1, t_2}^{sx} &: t_1 \rightarrow t_2 \\ \mathit{testop}_t &: t \rightarrow \text{i32} & \mathit{bits}_t^n &: t \rightarrow b^n \quad (n \leq |t|) \\ \mathit{binop}_t &: t \times t \rightarrow t & \mathit{const}_t &: b^n \rightarrow t \quad (n = |t|) \\ \mathit{relop}_t &: t \times t \rightarrow \text{i32} & \mathit{const}_t^{sx} &: b^n \rightarrow t \quad (n < |t|) \end{aligned}$$

The left group abstracts the actual numerics of those operators, which have the “obvious” semantics. The others implement conversions and reinterpretation from and to sequences of raw bytes b^n in little endian with optional sign extension. Some of the binary operators and conversions may trap for some inputs and thus are partial (written \rightarrow). Notably, div_{sxin} , rem_{sxin} , and float-to-int conversions trap when the results are unrepresentable.

Control Control constructs reduce into the auxiliary **label** construct. It represents a block along with its label’s signature and the continuation with which the block is replaced when branching to the label. This continuation is empty in the case of **block** and **if**, terminating the block, and the original expression in the case of **loop**, thereby restarting the loop.

Via the definition of local contexts L^k , evaluation proceeds inside a **label** block until the **label** construct itself can be reduced in one of several ways. When the **label** block has only values remaining, it is exited, leaving those values as results. Similarly in case of a trap has occurred. When a **label** is targeted by a **br**, execution keeps as many values v^n on the stack as prescribed by the targeted label signature t^n and reduces to the label’s continuation. The remaining values of the local and all intermediate stacks are thrown away, implementing unwinding.

Conditional and indexed branches reduce to regular branches first. If a conditional branch is not taken, however, it just reduces to the empty instruction sequence – note that this leaves any potential branch operands on the operand stack, and they may have to be dropped explicitly.

Calls Performing calls is a bit more involved. Both **call** and **call_indirect** reduce to the auxiliary form (**call** cl) that takes a closure as immediate. To find that in the store, the instructions have to know their own instance i and look it up.

The rule for calling closures looks at its type $t_1^n \rightarrow t_2^m$ to determine how many operands n to pop off the stack as arguments. It creates a new **local** block, which is another auxiliary form similar to **label** but representing a call frame. The frame holds a reference i to the instance of the invoked closure and a list of values as the state of the function’s local variables. This list is initialized with the call arguments v^n followed by the zero-initialized locals with types t^k . The signature of the block is the return type t_2^m of the function.

Like a **label** block, a **local** block can be exited by either reducing to a sequence of result values, a trap, or by a branch targeting it. In all cases, the construct – and thereby the call that created it –

is reduced to its result values. In addition, it can also be targeted by a **return**, which is just a branch to the nested block.

Variables The values of local variables are part of the configuration, and simply accessed there. In the case of **set_local**, the respective variable is updated with the new value.

Global variables are stored in the instance, so **get_global** and **set_global** use their instance’s address i to access it; **set_global** mutates the store by updating the global with the new value.

Memory Load instructions look up the current instance’s memory and extract a slice of raw bytes of the appropriate length from it. We write $|t|$ or $|tp|$ for the byte width of value types or packed types. They then apply the suitable reinterpretation function to create a constant value from it. Inversely, stores convert values into a sequence of raw bytes and write them into the appropriate memory. In either case, if the access is out of bounds, a trap is generated.

Finally, **current_memory** measures the size of its memory in page units (i.e., dividing by 64 Ki) and pushes it. **grow_memory** grows the respective memory by appending the correct amount of zero bytes to it. We assume here that this can fail as an out-of-memory condition, in which case the instruction returns -1 .

4. Validation

On the web, code is usually fetched from untrusted sources. Before any code can be executed safely, it must be *validated*. Validation rules for WebAssembly are defined succinctly as a *type system*. This type system is, by design, embarrassingly simple. It is designed to be efficiently checkable in a single linear pass, possibly interleaved with binary decoding and compilation.

Type-correctness not only guarantees the absence of certain safety violations and other undefined behavior, such as invalid calls or illegal accesses to locals, it also establishes various invariants that are central to efficient compilation of WebAssembly. For example, the typing rules statically determine the operand stack, such that the data flow between all instructions is known at compile time.

4.1 Typing Rules

Figure 4 defines the WebAssembly type system declaratively via the usual system of deduction rules [32]. The upper part of the figure defines a judgement $C \vdash e^* : tf$ assigning a function type to every instruction sequence that is valid under a *context* C . The lower part gives specialized rules defining validity of complete modules.

Contexts A context C records lists of all declared entities accessible at a given point in a program and their respective types. When spelling out a context record we omit components that are empty. Similar to Section 3, we write C_{func} to access the func component of C and $C_{\text{func}}(i)$ to access the i -th function in it; similarly for other components. In the case of $C_{\text{label}}(i)$ we make a special case and define this to mean indexing *from the end* of the list, because labels are referenced relatively, with the last being referenced as 0 (a form of de Bruijn indexing [11]). For extension, $C, \text{local } t^*$ denotes the same context as C , but with the sequence t^* appended to the list of locals. In the case of $C, \text{label } (t^*)$ the parentheses indicate that C ’s labels are extended with a single entry that is itself a list.

Instructions The function type $tf = t_1^* \rightarrow t_2^*$ assigned to instructions specifies their required *input* stack t_1^* and the provided *output* stack t_2^* . Most of the typing rules for individual instructions are straightforward. We focus on specific points of interest.

The rules for control constructs require that their type matches the explicit annotation tf , and they extend the context with a local label. As explained in Section 2, the label’s operand types are the instruction’s output types for forward labels (**block**, **if**) and the

instruction’s input types for backward labels (**loop**). Label types are used in the typing of branch instructions, which require suitable operands on the stack to match the stack at the join point.

The side condition $C_{\text{table}} = n$ in the rule for **call_indirect** ensures that this instruction can only be used when a table is present. Similar side conditions exist for all memory-related instructions, that ensure that a memory has been declared. The other side conditions for load and store instructions make sure that alignment is no larger than the type’s width, which we call the *natural alignment*, and that packed access happens only with a packed type tp narrower than the computation type t .

When typing sequences of expressions, the rules require that the input stack t_2^* of the last instruction e_2 matches the output stack of the preceding instruction sequence. However, most rules describe only the top of the stack as far as they affect it. Composing instructions may require extending these types to deeper stacks so that they match up, which is achieved by the last rule. It allows weakening the type of any instruction by assuming additional values of types t^* further down the stack which are untouched between input and output. For example, when typing the sequence (**i32.const** 1) (**i32.const** 2) **i32.add**, this rule is invoked to weaken the type of the middle instruction from $\epsilon \rightarrow i32$ to $i32 \rightarrow i32$, as needed to compose with the other two.

Polymorphism Some instructions are *polymorphic* in one of two ways. The simple case is in the rules for **drop** and **select**, that can pick an arbitrary operand type t locally. In most circumstances, the choice will be determined by program context, i.e., preceding or succeeding instructions and the constraints imposed by the rule for instruction sequences. In the few remaining cases the choice does not matter and a compiler could pick any type.

A second degree of polymorphism extends to the entire stack. It occurs with all instructions that perform an unconditional control transfer (**unreachable**, **br**, **br_table**, **return**). Since control never proceeds to the instructions following these (they are dead code) no requirements exist for the stack “afterwards”. That is, the rules can assume whatever fits the next instruction. That is expressed by allowing arbitrary excess input type t_1^* and arbitrary output types t_2^* in the types of these instructions; the former are virtually consumed, the latter are virtually produced, leaving any possible stack.

The outcome of this is that producers do not need to worry about reachability of code and about manually adjusting the stack before a control transfer. For example, a simple compiler can compositionally compile an expression like “ $a + b$ ” into “**compile**(a) **compile**(b) **i32.add**” regardless of whether the recursive compilations of a or b produced code that ends in a branch or an unconditional runtime error. It also ensures that various program transformations are always valid, e.g. inlining and other forms of partial evaluation.

Modules A module is a closed definition, so no context is required for validation. It is valid when all the individual definitions are valid and all export names are different. Note that the premises are recursive with respect to C ; in particular, all functions are mutually recursive. Globals on the other hand are not recursive, and the incremental contexts C_i used for checking individual global declarations do only contain references to previous globals, nor any other declarations, because they may not be used in the initializer.

A function definition is valid when its body is a block of suitable type. Similarly, for globals, the initialization expression needs to have the right type; moreover, it may only be exported if immutable. A table definition is valid if it lists the right number of (valid) functions. Imports are only supplied at instantiation time, so no static constraints apply to their declarations, except that imported globals may not be mutable.

(contexts) $C ::= \{\text{func } tf^*, \text{global } tg^*, \text{table } n^?, \text{memory } n^?, \text{local } t^*, \text{label } (t^*)^*\}$

Typing Instructions

$C \vdash e^* : tf$

$$\begin{array}{c}
\frac{}{C \vdash t.\text{const } c : \epsilon \rightarrow t} \quad \frac{}{C \vdash t.\text{unop} : t \rightarrow t} \quad \frac{}{C \vdash t.\text{binop} : t t \rightarrow t} \\
\frac{}{C \vdash t.\text{testop} : t \rightarrow \text{i32}} \quad \frac{}{C \vdash t.\text{relop} : t t \rightarrow \text{i32}} \quad \frac{t_1 \neq t_2 \quad sx^? = \epsilon \Leftrightarrow (|t_1| > |t_2| \wedge t_1 = \text{in})}{C \vdash t_1.\text{convert } t_2\text{-sx}^? : t_2 \rightarrow t_1} \quad \frac{t_1 \neq t_2 \quad |t_1| = |t_2|}{C \vdash t_1.\text{reinterpret } t_2 : t_2 \rightarrow t_1} \\
\frac{}{C \vdash \text{unreachable} : t_1^* \rightarrow t_2^*} \quad \frac{}{C \vdash \text{nop} : \epsilon \rightarrow \epsilon} \quad \frac{}{C \vdash \text{drop} : t \rightarrow \epsilon} \quad \frac{}{C \vdash \text{select} : t t \text{ i32} \rightarrow t} \\
\frac{tf = t_1^n \rightarrow t_2^m \quad C, \text{label}(t_2^m) \vdash e^* : tf}{C \vdash \text{block } tf \ e^* \ \text{end} : tf} \quad \frac{tf = t_1^n \rightarrow t_2^m \quad C, \text{label}(t_1^n) \vdash e^* : tf}{C \vdash \text{loop } tf \ e^* \ \text{end} : tf} \\
\frac{tf = t_1^n \rightarrow t_2^m \quad C, \text{label}(t_2^m) \vdash e_1^* : tf \quad C, \text{label}(t_2^m) \vdash e_2^* : tf}{C \vdash \text{if } tf \ e_1^* \ \text{else } e_2^* \ \text{end} : t_1^n \text{ i32} \rightarrow t_2^m} \\
\frac{C_{\text{label}(i)} = t^*}{C \vdash \text{br } i : t_1^* t^* \rightarrow t_2^*} \quad \frac{C_{\text{label}(i)} = t^*}{C \vdash \text{br.if } i : t^* \text{ i32} \rightarrow t^*} \quad \frac{(C_{\text{label}(i)} = t^*)^+}{C \vdash \text{br.table } i^+ : t_1^* t^* \text{ i32} \rightarrow t_2^*} \\
\frac{C_{\text{label}(|C_{\text{label}}| - 1)} = t^*}{C \vdash \text{return} : t_1^* t^* \rightarrow t_2^*} \quad \frac{C_{\text{func}(i)} = tf}{C \vdash \text{call } i : tf} \quad \frac{tf = t_1^* \rightarrow t_2^* \quad C_{\text{table}} = n}{C \vdash \text{call.indirect } tf : t_1^* \text{ i32} \rightarrow t_2^*} \\
\frac{C_{\text{local}(i)} = t}{C \vdash \text{get.local } i : \epsilon \rightarrow t} \quad \frac{C_{\text{local}(i)} = t}{C \vdash \text{set.local } i : t \rightarrow \epsilon} \quad \frac{C_{\text{local}(i)} = t}{C \vdash \text{tee.local } i : t \rightarrow t} \quad \frac{C_{\text{global}(i)} = \text{mut}^? t}{C \vdash \text{get.global } i : \epsilon \rightarrow t} \quad \frac{C_{\text{global}(i)} = \text{mut } t}{C \vdash \text{set.global } i : t \rightarrow \epsilon} \\
\frac{C_{\text{memory}} = n \quad 2^a \leq (|tp| <)^? |t| \quad (tp\text{-sz})^? = \epsilon \vee t = \text{in}}{C \vdash t.\text{load } (tp\text{-sz})^? \ a \ o : \text{i32} \rightarrow t} \quad \frac{C_{\text{memory}} = n \quad 2^a \leq (|tp| <)^? |t| \quad tp^? = \epsilon \vee t = \text{in}}{C \vdash t.\text{store } tp^? \ a \ o : \text{i32 } t \rightarrow \epsilon} \\
\frac{C_{\text{memory}} = n}{C \vdash \text{current.memory} : \epsilon \rightarrow \text{i32}} \quad \frac{C_{\text{memory}} = n}{C \vdash \text{grow.memory} : \text{i32} \rightarrow \text{i32}} \\
\frac{}{C \vdash \epsilon : \epsilon \rightarrow \epsilon} \quad \frac{C \vdash e_1^* : t_1^* \rightarrow t_2^* \quad C \vdash e_2 : t_2^* \rightarrow t_3^*}{C \vdash e_1^* e_2 : t_1^* \rightarrow t_3^*} \quad \frac{C \vdash e : t_1^* \rightarrow t_2^*}{C \vdash e : t^* t_1^* \rightarrow t^* t_2^*} \\
\text{Typing Modules} \\
\frac{tf = t_1^* \rightarrow t_2^* \quad C, \text{local } t_1^* t^*, \text{label}(t_2^*) \vdash e^* : \epsilon \rightarrow t_2^*}{C \vdash \text{ex}^* \ \text{func } tf \ \text{local } t^* \ e^* : \text{ex}^* \ tf} \quad \frac{tg = \text{mut}^? t \quad C \vdash e^* : \epsilon \rightarrow t \quad \text{ex}^* = \epsilon \vee tg = t}{C \vdash \text{ex}^* \ \text{global } tg \ e^* : \text{ex}^* \ tg} \\
\frac{(C_{\text{func}(i)} = tf)^n}{C \vdash \text{ex}^* \ \text{table } n \ i^n : \text{ex}^* \ n} \quad \frac{}{C \vdash \text{ex}^* \ \text{memory } n : \text{ex}^* \ n} \\
\frac{}{C \vdash \text{ex}^* \ \text{func } tf \ im : \text{ex}^* \ tf} \quad \frac{tg = t}{C \vdash \text{ex}^* \ \text{global } tg \ im : \text{ex}^* \ tg} \quad \frac{}{C \vdash \text{ex}^* \ \text{table } n \ im : \text{ex}^* \ n} \quad \frac{}{C \vdash \text{ex}^* \ \text{memory } n \ im : \text{ex}^* \ n} \\
\frac{(C \vdash f : \text{ex}_f^* \ tf)^* \quad (C_i \vdash \text{glob}_i : \text{ex}_g^* \ tg_i)_i^* \quad (C \vdash \text{tab} : \text{ex}_t^* \ n)^? \quad (C \vdash \text{mem} : \text{ex}_m^* \ n)^?}{(C_i = \{\text{global } tg_j^{i-1}\})_i^* \quad C = \{\text{func } tf^*, \text{global } tg^*, \text{table } n^?, \text{memory } n^?\} \quad \text{ex}_f^* \ \text{ex}_g^* \ \text{ex}_t^* \ \text{ex}_m^* \ \text{distinct}}{\vdash \text{module } f^* \ \text{glob}_i^* \ \text{tab}^? \ \text{mem}^?}
\end{array}$$

Figure 4. Typing rules

4.2 Soundness

Typing WebAssembly enjoys standard *soundness* properties [38]. Soundness proves that the reduction rules from Section 3 actually cover all execution states that can arise for valid programs. In other words, it proves that Figure 3 specifies a complete execution semantics for WebAssembly and leaves no room for undefined behavior, assuming the auxiliary numeric primitives are well-defined.

Store Typing Before we can state the soundness theorems concretely, we must extend typing to stores and configurations as defined in Figure 3. These rules, shown in Figure 5 are not required for validation, but for generalizing to dynamic computations. They use an additional *store context* S to classify the store. The typing

judgement for instructions in Figure 4 is extended to $S; C \vdash e^* : tf$ by implicitly adding S to all rules – it is never modified or used by those rules, but is accessed by the new rules for **call** cl and **local**.

Theorems With the help of these auxiliary judgements we can now formulate the relevant properties:

PROPOSITION 4.1 (Preservation). *If $\vdash_i s; v^*; e^* : t^*$ and $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$, then $\vdash_i s'; v'^*; e'^* : t^*$.*

PROPOSITION 4.2 (Progress). *If $\vdash_i s; v^*; e^* : t^*$, then either $e^* = v'^*$, or $e^* = \text{trap}$, or $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$.*

These properties ensure that all valid programs either diverge, trap, or terminate with values of the correct types.

(store context) $S ::= \{\text{inst } C^*, \text{tab } n^*, \text{mem } n^*\}$

$$\begin{array}{c}
\frac{S = \{\text{inst } C^*, \text{tab } n^*, \text{mem } m^*\} \quad (S \vdash \text{inst} : C)^* \quad ((S \vdash \text{cl} : \text{tf})^*)^* \quad (n \leq |cl^*|)^* \quad (m \leq |b^*|)^*}{\vdash \{\text{inst } \text{inst}^*, \text{tab } (cl^*)^*, \text{mem } (b^*)^*\} : S} \\
\\
\frac{S_{\text{inst}}(i) \vdash f : \text{tf}}{S \vdash \{\text{inst } i, \text{code } f\} : \text{tf}} \quad \frac{(S \vdash \text{cl} : \text{tf})^* \quad (\vdash v : t)^* \quad (S_{\text{tab}}(i) = n)^? \quad (S_{\text{mem}}(j) = m)^?}{S \vdash \{\text{func } cl^*, \text{glob } v^*, \text{tab } i^?, \text{mem } j^?\} : \{\text{func } \text{tf}^*, \text{global } t^*, \text{table } n^?, \text{memory } m^?\}} \\
\\
\frac{\vdash s : S \quad S \vdash_i v^*; e^* : t^*}{\vdash_i s; v^*; e^* : t^*} \quad \frac{(\vdash v : t_v)^* \quad S; S_{\text{inst}}(i), \text{local } t_v^* \vdash e^* : \epsilon \rightarrow t^*}{S \vdash_i v^*; e^* : t^*} \quad \frac{}{\vdash t.\text{const } c : t} \\
\\
\frac{}{C \vdash \text{trap} : \text{tf}} \quad \frac{C \vdash e_0^* : t^* \rightarrow t_2^* \quad C, \text{label } (t^*) \vdash e^* : \epsilon \rightarrow t_2^*}{C \vdash \text{label}\{t^*; e_0^*\} e^* \text{end} : \epsilon \rightarrow t_2^*} \quad \frac{S \vdash \text{cl} : \text{tf}}{S; C \vdash \text{call } cl : \text{tf}} \quad \frac{S \vdash_i v^*; e^* : t^*}{S; C \vdash \text{local}\{i; v^*\} e^* \text{end} : \epsilon \rightarrow t^*}
\end{array}$$

Figure 5. Store and configuration typing and rules for administrative instructions

4.3 Lazy Validation

While the rules in Figure 4 define when a WebAssembly module is valid, and producers are required to generate code according to this semantics, engines are not required to fully check it in all cases.

To minimize start-up latency and accommodate for different implementation strategies, such as just-in-time lazy compilation, validation requirements are more lenient: it is not required that a complete module is validated upfront. Instead, validation of individual functions may be deferred to the point when they are first called (if at all).

5. Binary Format

The actual format in which WebAssembly is transmitted over the wire is a binary encoding of the abstract syntax presented in Figure 1. This encoding has been designed to minimize both size and decoding time. For space reasons, and because much of the format is rather straightforward, we only give a brief summary here.

A binary represents a single module and is divided into sections according to the different kinds of entities declared in it, plus a few auxiliary sections. Function types tf are collected in their own section to allow sharing. Code for function bodies is deferred to a separate section that is placed after all declarations. This way, a browser engine can minimize page-load latency by starting *streaming compilation* as soon as the first function body has been received over the wire. It can also *parallelize* compilation of consecutive 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.

Instructions are represented with simple byte codes as one would expect; in the future, sentinel opcodes may support variable lengths. All integral numbers, including opcode immediates, are encoded in LEB128 format [6]. The binary format strives for overall simplicity and is regular enough that it can be expressed (and theoretically, parsed) by a simple grammar.

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

6. Embedding and Interoperability

WebAssembly is similar to a virtual ISA in that it does not define how programs are loaded into the execution engine or how they perform input and output. 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 between modules are resolved, provides foreign functions to accomplish I/O and timers,

and specifies how WebAssembly traps are handled. While the execution semantics of WebAssembly as presented should be considered independent of a particular embedding, the primary use case so far has been the Web and JavaScript embedding.¹⁰

JavaScript API In the 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, such as network or disk, (2) instantiate it providing the necessary imports, (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. This pattern may be expressed by the following JavaScript code embedded into a web page:

```

let imports = {console: {print: (x) => console.log(x)}}
fetch("fac.wasm")
  .then(response => response.arrayBuffer())
  .then(buffer => WebAssembly.instantiate(buffer, imports))
  .then(instance => instance.exports.printFac(20))

```

This tries to load the module from Figure 2 remotely and instantiates it. As import, it supplies one module "console" with one function "print", implemented in JavaScript. Once instantiation is completed, it invokes the function printFac provided by the module, which will print the result to the browser console.

Linking An embedder can instantiate multiple modules and use exports from one as imports to the other. That allows instances to call each other's functions, share memory, or share function tables. Imported globals can serve as configuration parameters for linking. In the browser, the JavaScript API also allows creating and initializing memories or tables externally, or accessing exported memories and tables. They are represented as objects of dedicated JavaScript classes, and each memory is backed by a standard ArrayBuffer.

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; modules can only communicate through numbers, which may represent addresses in a common memory. It is up to producers to map other data types to number or memory. This design is fully intentional. It provides maximum flexibility to producers, and unlike previous VMs, does not privilege any specific programming or object model while penalizing others. WebAssembly is an abstraction over hardware, not over a programming language.

¹⁰ Members of the community group are also developing independent stand-alone implementations that run in other environments.

However interested producers can define a common ABI *on top of* WebAssembly such that modules can interoperate more closely in heterogeneous applications, and particularly, use a common memory. This separation of concerns is vital for making WebAssembly universal as a code format.

7. Implementation

A major design goal of WebAssembly has been high performance execution without sacrificing safety or portability. Throughout the collaborative design process of WebAssembly, representatives of major browser vendors have developed parallel and independent implementations of WebAssembly to validate and inform the design decisions. This section describes some points of interest of those implementations. While the goal of this paper is not a thorough comparison of different implementation techniques for WebAssembly, the different design choices in four different implementations is worth mentioning.¹¹

V8 (the JavaScript engine in Google’s Chrome), SpiderMonkey (the JavaScript engine in Mozilla’s Firefox) and JavaScriptCore (the JavaScript engine in WebKit) reuse their optimizing JIT compilers to compile WebAssembly modules ahead-of-time before instantiation. This achieves predictable and high peak performance and avoids the unpredictability of warmup time which has often been a problem for JavaScript.

However, other implementation strategies also make sense. Chakra (the JavaScript engine in Microsoft Edge) instead lazily translates WebAssembly functions to an internal bytecode upon first execution, first interpreting them and then JIT compiling the hottest functions. The main advantage of this approach is faster startup and potentially lower memory consumption.

Validation A key design goal of WebAssembly has been fast validation of code. In the three implementations developed independently for V8, SpiderMonkey, and Chakra, the same basic strategy is taken where an abstract control stack, an abstract operand stack with types, and a forward program counter are maintained. Validation proceeds by in-place abstract interpretation of the incoming bytecodes, with no intermediate representation being constructed.¹² While the main point of this paper is not a thorough performance evaluation, we measured single-threaded validation speed at between 75 MiB/s and 150 MiB/s on a suite of representative benchmarks on a modern workstation. This is approximately fast enough to perform validation at full network speed of 1 Gib/s.

Baseline JIT Compiler The implementation of WebAssembly in Mozilla’s SpiderMonkey JavaScript engine includes two compilation tiers. The first is a WebAssembly-specific fast baseline JIT that emits machine code in a single pass that is combined with validation. The baseline JIT creates no internal IR during compilation 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 the Ion optimizing JIT is compiling the module in parallel in the background. The Ion JIT is also used by SpiderMonkey as its top tier for JavaScript.

Optimizing JIT Compiler V8, SpiderMonkey, JavaScriptCore, and Chakra all include optimizing JITs for their top tier execution of JavaScript and reuse them for maximum peak performance of WebAssembly. Both V8 and SpiderMonkey top-tier JITs use SSA-based intermediate representations. As such, it was important to

validate that WebAssembly could be decoded to SSA form in a single linear pass to be fed to these JITs. Although details¹³ are beyond the scope of this paper, both V8 and SpiderMonkey implement direct-to-SSA translation in a single pass during validation of WebAssembly bytecode, while Chakra implements a WebAssembly-to-internal-bytecode translation to be fed through their adaptive optimization system. This is greatly helped by the structured control flow constructs of WebAssembly, making the decoding algorithm far simpler and more efficient and avoiding the limitation that many JITs have in that they do not support irreducible control flow¹⁴. In the case of V8, decoding targets the TurboFan compiler’s sea of nodes [9] graph-based IR, producing a loosely-ordered graph that is suitable for subsequent optimization and scheduling. Once decoded to an intermediate representation, compilation is then a matter of running the existing compiler backend, including instruction selection, register allocation, and code generation. Because some WebAssembly operations may not be available on all platforms, such as 64 bit integers on 32 bit architectures, IR rewriting and lowering might be performed before feeding to the backend of the compiler. Our experience reusing the advanced JITs from 4 different JavaScript engines has been a resounding success, allowing all engines to achieve high performance in a short time.¹⁵

Reference interpreter In addition to production-quality implementations for three major browsers, we implemented a reference interpreter for the entire WebAssembly language. For this we used OCaml [23] due to the ability to write in a high-level stylized way that closely matches the formalization, approximating an “executable specification”. The reference interpreter includes a full binary encoder and decoder, validator, and interpreter, as well as an extensive test suite. It is used to test both production implementations and the formal specification and to prototype new features.

7.1 Bounds checks

By design, all memory accesses in WebAssembly can be guaranteed safe with a single dynamic bounds check. Each instruction *t.load a o k* with type *t*, alignment *a*, static offset *o* and dynamic address *k* represents a read of the memory at $s_{\text{mem}}(i, k + o, |t|)$. That means bytes $k + o$ to $k + o + |t| - 1$ will be accessed by the instruction, and must be in bounds. That amounts to checking $k + o + |t| \leq \text{memsize}$. In a WebAssembly engine, the memory for an instance will be allocated in a large contiguous range beginning at some (possibly nondeterministic) *base* in the engine’s process, so the above access amounts to an access of $\text{base}[k + n]$.

Machine code specialization While *base* can be stored in a dedicated machine register for quick access, a JIT for WebAssembly can be even more aggressive and actually *specialize* the machine code generated for a module to a specific memory base, embedding the base address as a constant directly into the code, freeing a register. First, the JIT can reduce the cost of the bounds check by reorganizing the expression $k + o + |t| \leq \text{memsize}$ to $k \leq \text{memsize} - o - |t|$ and then constant-fold the right hand side.¹⁶ Although *memsize* is not necessarily a constant (since memory can be grown dynamically) it changes so infrequently that the JIT can embed it in the generated machine code, later *patching* the ma-

¹¹ Chrome and FireFox are already poised to activate their WebAssembly implementations by default in early 2017.

¹² A linear validation algorithm is sketched at docs.google.com/document/d/1E9zNSnATBeETbIAcrwb1DrEKKaR5enjNnk4Ph1N0a4c/

¹³ Reference omitted for double-blind review.

¹⁴ Which is also the case for many JVMs, because irreducible control flow never results from Java-source-generated bytecode.

¹⁵ A comparative and absolute performance study is beyond the scope of the paper, but for the final version we will include preliminary numbers to support this statement (they are available to reviewers).

¹⁶ This identity holds because we very carefully defined that effective address calculations do not wrap around.

chine code if the memory size changes.¹⁷ Unlike other speculation techniques, the change of a constant value is controlled enough that deoptimization [19] of the code is not necessary.

Virtual memory techniques. On 64 bit platforms, the WebAssembly engine can make use of virtual memory techniques to eliminate the need to insert bounds checks for memory accesses altogether. The engine simply reserves 8 GiB of virtual address space and marks as inaccessible all pages except the valid portion of memory near the beginning. Since WebAssembly memory addresses are 32 bit integers, by definition an access of $base[n + k]$ cannot be larger than 8 GiB from the beginning of $base$. Since most 64 bit CPU architectures offer 32 bit arithmetic on general purpose registers that *clears* the upper 32 bits of the output register, the JIT can simply emit accesses to $(base + n)[k]$ and rely on the hardware protection mechanism to catch out-of-bounds accesses. Moreover, since the memory size is no longer embedded in the generated machine code, no patching is necessary if memory is grown.

7.2 Improving compile time

Parallel compilation Since both V8 and SpiderMonkey implement ahead-of-time compilation, it is a clear performance win to parallelize compilation of WebAssembly modules, dispatching individual functions to different threads. Both V8 and SpiderMonkey achieve a 5-6 \times improvement in compilation speed with 8 compilation threads.

Caching code While implementors have spent a lot of resources improving compilation speed of JITs to reduce cold startup time of WebAssembly, we expect that warm startup time will become important as users will likely visit the same webpages repeatedly. The JavaScript API for IndexedDB [5] now allows JavaScript to manipulate and compile WebAssembly modules and store their compiled representation as an opaque blob in IndexedDB. This allows a JavaScript application to first query IndexedDB for a cached version of their WebAssembly module before downloading and compiling it. This mechanism has already been implemented in V8 and SpiderMonkey and accounts for a massive startup time improvement.

8. Related Work

The most direct precursors of WebAssembly are (P)NaCl [39, 8, 15] and asm.js [4], which we discussed in Section 1.1.

Efficient memory safety is a hard design constraint of WebAssembly. Previous systems such as CCured [31] and Cyclone [20] have tried to impose safety at the C language level, which generally requires program changes. Other attempts have been to try to enforce safety at the C abstract machine level using combinations of static checks and runtime checks [7, 17, 28], sometimes assisted by hardware [13, 27]. For example, the Secure Virtual Architecture [10] introduced an abstract machine based on LLVM bitcode that enforces the SAFECode [14] properties.

Typed intermediate languages carry type information throughout the compilation process from source to target code. For example TIL [25] and FLINT [34] pioneered typed ILs for functional languages, allowing higher confidence in compiler correctness and allowing more type-based optimizations late in compilation. However typed ILs have a somewhat different purpose than a compilation target. Since typed ILs are usually only expressed as data structures in a compiler, they are typically compiler-specific and exist only as an intermediate stage of compilation, not as a storage or execution format. Typed Assembly languages [26] do serve as

a compilation target, typically taking the form of a complex type system imposed on top of an existing assembly language. Compilers that target typed assembly languages must produce well-typed (or proof-carrying [29]) code by preserving and transforming types throughout compilation. The modeling of complex types imposes a severe burden on existing compilers, requiring them to be rearchitected to not only preserve and transform quantified types throughout compilation, but avoid any optimizations that break the type system. In comparison to typed intermediate languages, typed assembly languages, and safe “C” machines, the insight of WebAssembly is to radically reduce the scope of responsibility for the virtual machine: 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.

Mu [37] is a low-level “micro virtual machine” designed to be a minimalist set of abstractions over hardware, memory management, and concurrency. It offers an object model complete with typed pointers and automatic memory management, concurrency abstractions such as threads and stacks, as well as an intermediate representation based on LLVM. However, Mu does not enforce memory safety, since it is meant more as a substrate for language implementors to build upon. The safety mechanisms are left up to higher layers of the stack, such as a trusted client language VM on top of Mu. Since the client language VM is trusted, the lack of or a bug in that layer could allow an incorrect program to read or write memory arbitrarily or exhibit other undefined behavior.

For managed language systems where bytecode is the distribution format, the speed and simplicity of validation is key to good performance and high assurance. Our work was directly informed by experience with stack machines such as the JVM [24] and CIL [30] and their validation algorithms. Because we designed WebAssembly in lock-step with a formalization, we managed to make its semantics drastically simpler. For example, JVM bytecode verification takes more than 150 pages to describe precisely in the current JVM specification, while for WebAssembly it fits on one page (Figure 4). It took a decade of research to hash out the details of correct JVM verification [22], including the discovery of inherent vulnerabilities [12, 16] – such as a potential $O(n^3)$ worst-case of the iterative dataflow approach that is a consequence of the JVM’s unrestricted *gotos* and other idiosyncracies [36] that had to be fixed with the addition of stack maps to class files. Both the JVM and the CIL, as well as Android Dalvik [3], allow bytecode to create irreducible loops and unbalanced locking structures, features which are typically not supported by optimizing JITs, normally relegating methods containing those constructs to run in an interpreter. In contrast, the structured control flow of WebAssembly makes validation and compilation fast and simple and paves the way for structured locking and exception constructs in the future.

9. Future Directions

The initial version of WebAssembly presented here focuses on supporting low-level code, specifically compiled 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 *SIMD* instructions. Some of these features are already being prototyped in implementations of WebAssembly.

Beyond that, we intend to evolve WebAssembly further into a more attractive target for high-level languages by including relevant primitives like *tail calls* and *stack switching*. A highly important goal is to provide access to the advanced *garbage collectors* built into all browsers, thus eliminating one main reason for compilation to JavaScript.

¹⁷ A concept of “pseudo constant” in the compiler allows embedding constants in the machine code without other optimizations trying to infer invariants about the current value the pseudo constant has at compile time.

References

- [1] Activex controls. [https://msdn.microsoft.com/en-us/library/aa751968\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa751968(v=vs.85).aspx). Accessed: 2016-11-14.
- [2] Adobe Shockwave Player. <https://get.adobe.com/shockwave/>. Accessed: 2016-11-14.
- [3] ART and Dalvik. <https://source.android.com/devices/tech/dalvik/>. Accessed: 2016-11-14.
- [4] asm.js. <http://asmjs.org>. Accessed: 2016-11-08.
- [5] Indexed Database API. <https://www.w3.org/TR/IndexedDB/>. Accessed: 2016-11-08.
- [6] LEB128. <https://en.wikipedia.org/wiki/LEB128>. Accessed: 2016-11-08.
- [7] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09*, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.
- [8] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pages 355–366, New York, NY, USA, 2011. ACM.
- [9] C. Click and M. Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, Mar. 1995.
- [10] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A safe execution environment for commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 41(6):351–366, Oct. 2007.
- [11] N. G. de Bruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.
- [12] D. Dean, E. Felten, and D. Wallach. Java security: from HotJava to Netscape and beyond. In *Symposium on Security and Privacy*. IEEE Computer Society Press, 1996.
- [13] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. *SIGPLAN Not.*, 43(3):103–114, Mar. 2008.
- [14] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Not.*, 41(6):144–157, June 2006.
- [15] A. Donovan, R. Muth, B. Chen, and D. Sehr. PNaCl: Portable native client executables. Technical report, 2010.
- [16] A. Gal, C. W. Probst, and M. Franz. Complexity-based denial of service attacks on mobile-code systems. Technical Report 04-09, School of Information and Computer Science, University of California, Irvine, Irvine, CA, April 2004.
- [17] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck. Memory-safe execution of C on a Java VM. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security, PLAS'15*, pages 16–27, New York, NY, USA, 2015. ACM.
- [18] D. Gudeman. Representing type information in dynamically typed languages. Technical Report 93-27, Department of Computer Science, University of Arizona, Phoenix, Arizona, October 1993.
- [19] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Not.*, 27(7):32–43, July 1992.
- [20] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings the USENIX Annual Technical Conference, ATEC '02*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.
- [21] C. Lattner and V. Adve. LLVM: A compilation framework for life-long program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO '04*, Palo Alto, California, Mar.
- [22] X. Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason.*, 30(3-4):235–269, Aug. 2003.
- [23] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system*. INRIA, 2016.
- [24] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java Virtual Machine Specification (Java SE 8 Edition). Technical report, Oracle, 2015.
- [25] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, P. Cheng, P. Lee, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *In Workshop on Compiler Support for Systems Software*, 1996.
- [26] G. Morrisett, D. Walker, K. Cray, and N. Glew. From System F to Typed Assembly Language. *ACM TOPLAS*, 21(3):527–568, May 1999.
- [27] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. WatchdogLite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14*, pages 175:175–175:184, New York, NY, USA, 2014. ACM.
- [28] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. *SIGPLAN Not.*, 44(6):245–258, June 2009.
- [29] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97*, pages 106–119, New York, NY, USA, 1997. ACM.
- [30] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction, CC '02*, pages 213–228, London, UK, UK, 2002.
- [31] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. *SIGPLAN Not.*, 37(1):128–139, Jan. 2002.
- [32] B. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, USA, 2002.
- [33] G. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
- [34] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.
- [35] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimizations*, 4(4):2:1–2:36, Jan. 2008.
- [36] R. F. Strk and J. Schmid. Java bytecode verification is not possible (extended abstract). In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 232–234, 2001.
- [37] K. Wang, Y. Lin, S. M. Blackburn, M. Norrish, and A. L. Hosking. Draining the Swamp: Micro virtual machines as a solid foundation for language development. In T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, editors, *1st Summit on Advances in Programming Languages*, volume 32 of *SNAPL '15*, pages 321–336, Dagstuhl, Germany, 2015.
- [38] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115, 1994.
- [39] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy, Oakland'09*, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009.
- [40] A. Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 301–312, New York, NY, USA, 2011. ACM.