Bringing the Web up to Speed with WebAssembly

Andreas Haas Andreas Rossberg Derek L. Schuff* Ben L. Titzer
Google GmbH, Germany / *Google Inc, USA
{ahaas,rossberg,dschuff,titzer}@google.com

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

Dan Gohman Luke Wagner Alon Zakai
Mozilla Inc, USA
{sunfishcode,luke,azakai}@mozilla.com

JF Bastien
Apple Inc, USA
jfbastien@apple.com

Abstract

The maturation of the Web platform has given rise to sophisticated and demanding Web applications such as interactive 3D visualization, audio and video software, and games. With that, efficiency and security of code on the Web has become more important than ever. Yet JavaScript as the only built-in language of the Web is not well-equipped to meet these requirements, especially as a compilation target.

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable low-level bytecode called WebAssembly. It offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web. WebAssembly has been designed with a formal semantics from the start. We describe the motivation, design and formal semantics of WebAssembly and provide some preliminary experience with implementations.

CCS Concepts • **Software and its engineering** → **Virtual machines**; **Assembly languages**; *Runtime environments*; *Just-in-time compilers*

Keywords Virtual machines, programming languages, assembly languages, just-in-time compilers, type systems

1. Introduction

The Web began as a simple document exchange network but has now become the most ubiquitous application platform ever, accessible across a vast array of operating systems and device types. By historical accident, JavaScript is the only natively supported programming language on the Web, its widespread usage unmatched by other technologies available only via plugins like ActiveX, Java or Flash. Because of JavaScript's ubiquity, rapid performance improvements in modern VMs, and perhaps through sheer necessity, it has become a compilation target for other languages. Through Emscripten [43], even C and C++ programs can be compiled to a stylized low-level subset of JavaScript called asm.js [4]. Yet JavaScript has inconsistent performance and a number of other pitfalls, especially as a compilation target.

WebAssembly addresses the problem of safe, fast, portable low-level code on the Web. Previous attempts at solving it, from ActiveX to Native Client to asm.js, have fallen short of properties 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 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 such as the browser's JavaScript VM or a language plugin. Managed languages enforce *memory safety*, preventing programs from compromising user data or system state. However, managed language runtimes have traditionally not offered much for portable low-level code, such as memory-unsafe compiled C/C++ applications that do not need garbage collection but are inherently fast.

PLDI '17 June 19-24, 2017, Barcelona, Spain

ACM ISBN 978-1-4503-4988-8/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3062341.3062363

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

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 [42, 11] 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 constraints of the Chrome browser 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) [18] builds upon the sandboxing techniques of NaCl and uses a stable subset of LLVM bitcode [24] 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 exclusively in Chrome, inherently limiting their applications' portability.

Emscripten [43] is a framework for compiling mostly unmodified C/C++ applications to JavaScript and linking them with an execution environment implemented in JavaScript. Emscripten compiles to a specialized subset of JavaScript that later evolved into asm.js [4], an embedded domain specific language that serves as a statically-typed assembly-like 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, benefiting from sophisticated JIT compilers, but 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.

While Java and Flash [2] came early to the Web and offered managed runtime plugins, neither supported high-performance low-level code, and today usage 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. In this paper we focus on

- 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 production implementations in 4 major browsers, including novel techniques such as for memory protection.

To our knowledge, WebAssembly is the first industrial-strength language or VM that has been designed with a formal semantics from the start. This not only demonstrates the "real world" feasibility of such an approach, but also that it leads to a notably clean design.

While the Web is the primary motivation for WebAssembly, nothing in its design depends on the Web or a JavaScript environment. It is an open standard specifically designed for embedding in multiple contexts, and we expect that stand-alone implementations will become available in the future. The initial version primarily focuses on supporting low-level languages, 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.[1] For brevity we omit a few minor features having to do with module initialization.

2.1 Basics

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

[1] WebAssembly has an S-expression text representation that closely resembles this syntax and tools for assembling binaries from it, e.g. to write tests. A code example can be found in the Supplementary Appendix.

(value types) $t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$
(packed types) $tp ::= \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32}$
(function types) $tf ::= t^* \to t^*$
(global types) $tg ::= \mathsf{mut}^? \; t$

$unop_{iN} ::= \mathbf{clz} \mid \mathbf{ctz} \mid \mathbf{popcnt}$
$unop_{fN} ::= \mathbf{neg} \mid \mathbf{abs} \mid \mathbf{ceil} \mid \mathbf{floor} \mid \mathbf{trunc} \mid \mathbf{nearest} \mid \mathbf{sqrt}$
$binop_{iN} ::= \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div}\_sx \mid \mathbf{rem}\_sx \mid$
$\mathbf{and} \mid \mathbf{or} \mid \mathbf{xor} \mid \mathbf{shl} \mid \mathbf{shr}\_sx \mid \mathbf{rotl} \mid \mathbf{rotr}$
$binop_{fN} ::= \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div} \mid \mathbf{min} \mid \mathbf{max} \mid \mathbf{copysign}$
$testop_{iN} ::= \mathbf{eqz}$
$relop_{iN} ::= \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{lt}\_sx \mid \mathbf{gt}\_sx \mid \mathbf{le}\_sx \mid \mathbf{ge}\_sx$
$relop_{fN} ::= \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{lt} \mid \mathbf{gt} \mid \mathbf{le} \mid \mathbf{ge}$
$cvtop ::= \mathbf{convert} \mid \mathbf{reinterpret}$
$sx ::= \mathbf{s} \mid \mathbf{u}$

(instructions) $e ::= \mathbf{unreachable} \mid \mathbf{nop} \mid \mathbf{drop} \mid \mathbf{select} \mid$
$\mathbf{block}\ tf\ e^*\ \mathbf{end} \mid \mathbf{loop}\ tf\ e^*\ \mathbf{end} \mid \mathbf{if}\ tf\ e^*\ \mathbf{else}\ e^*\ \mathbf{end} \mid$
$\mathbf{br}\ i \mid \mathbf{br\_if}\ i \mid \mathbf{br\_table}\ i^+ \mid \mathbf{return} \mid \mathbf{call}\ i \mid \mathbf{call\_indirect}\ tf \mid$
$\mathbf{get\_local}\ i \mid \mathbf{set\_local}\ i \mid \mathbf{tee\_local}\ i \mid \mathbf{get\_global}\ i \mid$
$\mathbf{set\_global}\ i \mid t.\mathbf{load}\ (tp\_sx)^?\ a\ o \mid t.\mathbf{store}\ tp^?\ a\ o \mid$
$\mathbf{current\_memory} \mid \mathbf{grow\_memory} \mid t.\mathbf{const}\ c \mid$
$t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop\ t\_sx^?$

(functions) $f ::= ex^*\ \mathbf{func}\ tf\ \mathbf{local}\ t^*\ e^* \mid ex^*\ \mathbf{func}\ tf\ im$
(globals) $glob ::= ex^*\ \mathbf{global}\ tg\ e^* \mid ex^*\ \mathbf{global}\ tg\ im$
(tables) $tab ::= ex^*\ \mathbf{table}\ n\ i^* \mid ex^*\ \mathbf{table}\ n\ im$
(memories) $mem ::= ex^*\ \mathbf{memory}\ n \mid ex^*\ \mathbf{memory}\ n\ im$
(imports) $im ::= \mathbf{import}\ \text{“}name\text{”}\ \text{“}name\text{”}$
(exports) $ex ::= \mathbf{export}\ \text{“}name\text{”}$
(modules) $m ::= \mathbf{module}\ f^*\ glob^*\ tab^?\ mem^?$

Figure 1. WebAssembly abstract syntax

Modules A binary takes the form of a *module*. It contains definitions for *functions*, *globals*, *tables*, and *memories*. Each definition may 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 a function exported from the 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 WebAssembly computation is based on a *stack machine*; code for a function consists of a sequence of *instructions* that manipulate 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.The stack organization is merely a way to achieve a compact program representation, as it has been shown to be smaller than a register machine [38].[2]

Traps Some instructions may produce a *trap*, which immediately aborts the current computation. Traps cannot currently be handled by WebAssembly code, but an embedder will typically provide means to handle this condition. Embedded in JavaScript, a WebAssembly trap will throw a JavaScript exception containing a stacktrace with both JavaScript and WebAssembly stack frames. It can be caught and inspected by the surrounding JavaScript code.

Machine Types WebAssembly has 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 can be used as addresses in the linear memory (Section 2.2), and indexes into function tables (Section 2.4). Most WebAssembly instructions simply execute *operators* on values of these basic data types. The grammar in Figure 1 conveniently distinguishes several categories, such as *unary* and *binary* operators, *tests* and *comparisons*. WebAssembly provides *conversions* between all 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 f declare mutable *local variables* of types t^*. 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.

[2] 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 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.

Global Variables A module may also declare typed *global variables* 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.

So far so boring. In the following sections we turn our attention to more interesting or unusual 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 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 indicated by **grow_memory** returning -1 to be handled by the program.[3] 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 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 32 bit static offset o and the dynamic **i32** address as a 33 bit address (i.e., no wraparound), which allows specific optimizations (Section 7). All memory access is dynamically checked against the memory size; 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 value 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. Contemporary hardware seems to be converging on little-endian byte order, either being natively little-endian or having optional endian conversion included in memory access, or being architecturally neutral with both variants available. Recognizing this convergence, we chose to define WebAssembly memory to have little-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. 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 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 Web pages 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 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 program 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 con-

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

struct's block; [4] 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).

The **br_if** instruction branches if its input is non-zero, and **br_table** selects a target from a list of label immediates based on an index operand, with the last label being the target for out-of-bounds index values. These two instructions allow minimal code that avoids any jumps-to-jumps.

Block Signatures and Unwinding Every control construct is annotated with a function type $tf = t_1^* \rightarrow t_2^*$ that describes how it changes the stack.[5] 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, pops its results off the internal stack, and then pushes its results t_2^* onto the outer stack. 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, 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 them 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 still allows simple interpreters, e.g., to implement debuggers. An interpreter can have a contiguous stack and just remember the *height* upon entry to each block in a separate *control stack*. Further, it 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.[6]

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,

```
switch (x) {
  case 0: ...A...
  case 1: ...B... break;
  default: ...C...
}
```

becomes

```
block block block block
  br_table 0 1 2
  end ...A...
  end ...B... br 1
  end ...C...
end
```

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

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 (e.g. the relooper algorithm [43]), where JavaScript is also restricted to structured control. In our experience building an LLVM backend for WebAssembly, irreducible control flow is rare, and a simple restructuring algorithm is all that is necessary to translate any 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 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 simply shorthand for a branch that targets the function's block.

Direct Calls Functions can be invoked directly using the **call** instruction which takes an index immediate identifying the function to call. The call instruction pops the function arguments from the operand stack and pushes the function's return values upon return.

Indirect Calls Function pointers 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; 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

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

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

[6] That is the approach V8 takes in its debugging interpreter.

example, when WebAssembly is embedded in JavaScript, imported functions may be *host* functions that are defined in JavaScript. Values crossing the language boundary are automatically converted according to JavaScript rules.[7]

2.5 Determinism

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

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

NaN Payloads WebAssembly follows the IEEE-754 standard for floating point arithmetic. However, IEEE-754 does not specify the exact bit pattern for NaN values in all cases, and we found that CPUs differ significantly, while normalizing after every numeric operation is too expensive. However, we still want to enable compilers targeting WebAssembly to employ techniques like NaN-boxing [21]. Based on our experience with JavaScript engines, we established sufficient rules: (1) instructions only output canonical NaNs with a non-deterministic sign bit, unless (2) if an input is a non-canonical NaN, then the output NaN is non-deterministic.

Resource Exhaustion Available resources are always finite and differ wildly across devices. In particular, an engine may be *out of memory* when trying to grow the linear memory – semantically a **grow_memory** instruction non-deterministically returns -1. A **call** or **call_indirect** instruction may also experience *stack overflow*, but this is not semantically observable from within WebAssembly itself.

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

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

2.6 Omissions and Restrictions

WebAssembly as presented here is almost complete except for some minor omissions related to module initialization:

- Tables can be partially initialized, and initialization can be applied to imported tables.
- Memory segments can be pre-initialized with static data.
- A module can specify a designated *startup* function.
- Tables and memories can have an optional maximum size that limits how much they can be grown.

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

The initial release of WebAssembly also imposes a few restrictions, likely lifted in future releases:

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

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.

3.1 Stores and Instances

Execution operates relative to a global *store* s. The upper part of Figure 2 defines syntax for representations of stores and other runtime objects. A store is 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 2 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. The representation of instantiated tables is simply a list of *closures* cl and that of 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 used to access stateful objects such as the globals, tables, and memories, since functions can be imported from a different instance. An implementation can 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 reduction rules.

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 indexing to stores with the following short-hands:

$$s_{\mathsf{func}}(i,j) = s_{\mathsf{inst}}(i)_{\mathsf{func}}(j) \qquad s_{\mathsf{tab}}(i,j) = s_{\mathsf{tab}}(s_{\mathsf{inst}}(i)_{\mathsf{tab}})(j)$$
$$s_{\mathsf{glob}}(i,j) = s_{\mathsf{inst}}(i)_{\mathsf{glob}}(j) \qquad s_{\mathsf{mem}}(i,j) = s_{\mathsf{mem}}(s_{\mathsf{inst}}(i)_{\mathsf{mem}})(j)$$

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

3.2 Instantiation

Instantiating a module $m = (\mathbf{module}\ f^*\ glob^*\ tab^?\ mem^?)$ in a store s requires providing external function closures

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

Reduction

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v^*; L^k[e^*] \hookrightarrow_i s'; v'^*; L^k[e'^*]} \qquad \frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v_0^*; \mathbf{local}_n\{i; v^*\}\ e^*\ \mathbf{end} \hookrightarrow_j s'; v_0^*; \mathbf{local}_n\{i; v'^*\}\ e'^*\ \mathbf{end}} \qquad \boxed{s; v^*; e^* \hookrightarrow_i s; v^*; e^*}$$

$$
\begin{array}{rcll}
L^0[\mathbf{trap}] & \hookrightarrow & \mathbf{trap} & \text{if } L^0 \neq [\_] \\
(t.\mathbf{const}\ c)\ t.unop & \hookrightarrow & t.\mathbf{const}\ unop_t(c) & \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.binop & \hookrightarrow & t.\mathbf{const}\ c & \text{if } c = binop_t(c_1, c_2) \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.binop & \hookrightarrow & \mathbf{trap} & \text{otherwise} \\
(t.\mathbf{const}\ c)\ t.testop & \hookrightarrow & \mathbf{i32.const}\ testop_t(c) & \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.relop & \hookrightarrow & \mathbf{i32.const}\ relop_t(c_1, c_2) & \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{convert}\ t_1\_sx^? & \hookrightarrow & t_2.\mathbf{const}\ c' & \text{if } c' = \mathrm{cvt}^{sx^?}_{t_1,t_2}(c) \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{convert}\ t_1\_sx^? & \hookrightarrow & \mathbf{trap} & \text{otherwise} \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{reinterpret}\ t_1 & \hookrightarrow & t_2.\mathbf{const}\ \mathrm{const}_{t_2}(\mathrm{bits}_{t_1}(c)) & \\
\mathbf{unreachable} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{nop} & \hookrightarrow & \epsilon & \\
v\ \mathbf{drop} & \hookrightarrow & \epsilon & \\
v_1\ v_2\ (\mathbf{i32.const}\ 0)\ \mathbf{select} & \hookrightarrow & v_2 & \\
v_1\ v_2\ (\mathbf{i32.const}\ k+1)\ \mathbf{select} & \hookrightarrow & v_1 & \\
v^n\ \mathbf{block}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end} & \hookrightarrow & \mathbf{label}_m\{\epsilon\}\ v^n\ e^*\ \mathbf{end} & \\
v^n\ \mathbf{loop}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end} & \hookrightarrow & \mathbf{label}_n\{\mathbf{loop}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end}\}\ v^n\ e^*\ \mathbf{end} & \\
(\mathbf{i32.const}\ 0)\ \mathbf{if}\ tf\ e_1^*\ \mathbf{else}\ e_2^*\ \mathbf{end} & \hookrightarrow & \mathbf{block}\ tf\ e_2^*\ \mathbf{end} & \\
(\mathbf{i32.const}\ k+1)\ \mathbf{if}\ tf\ e_1^*\ \mathbf{else}\ e_2^*\ \mathbf{end} & \hookrightarrow & \mathbf{block}\ tf\ e_1^*\ \mathbf{end} & \\
\mathbf{label}_n\{e^*\}\ v^*\ \mathbf{end} & \hookrightarrow & v^* & \\
\mathbf{label}_n\{e^*\}\ \mathbf{trap}\ \mathbf{end} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{label}_n\{e^*\}\ L^j[v^n\ (\mathbf{br}\ j)]\ \mathbf{end} & \hookrightarrow & v^n\ e^* & \\
(\mathbf{i32.const}\ 0)\ (\mathbf{br\_if}\ j) & \hookrightarrow & \epsilon & \\
(\mathbf{i32.const}\ k+1)\ (\mathbf{br\_if}\ j) & \hookrightarrow & \mathbf{br}\ j & \\
(\mathbf{i32.const}\ k)\ (\mathbf{br\_table}\ j_1^k\ j\ j_2^*) & \hookrightarrow & \mathbf{br}\ j & \\
(\mathbf{i32.const}\ k+n)\ (\mathbf{br\_table}\ j_1^k\ j) & \hookrightarrow & \mathbf{br}\ j & \\
s; \mathbf{call}\ j & \hookrightarrow_i & \mathbf{call}\ s_{\mathsf{func}}(i, j) & \\
s; (\mathbf{i32.const}\ j)\ \mathbf{call\_indirect}\ tf & \hookrightarrow_i & \mathbf{call}\ s_{\mathsf{tab}}(i, j) & \text{if } s_{\mathsf{tab}}(i, j)_{\mathsf{code}} = (\mathbf{func}\ tf\ \mathbf{local}\ t^*\ e^*) \\
s; (\mathbf{i32.const}\ j)\ \mathbf{call\_indirect}\ tf & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
v^n\ (\mathbf{call}\ cl) & \hookrightarrow & \mathbf{local}_m\{cl_{\mathsf{inst}}; v^n\ (t.\mathbf{const}\ 0)^k\}\ \mathbf{block}\ (\epsilon \to t_2^m)\ e^*\ \mathbf{end}\ \mathbf{end} & \ldots \\
\mathbf{local}_n\{i; v_l^*\}\ v^n\ \mathbf{end} & \hookrightarrow & v^n & \mid\ \ldots \text{if } cl_{\mathsf{code}} = (\mathbf{func}\ (t_1^n \to t_2^m)\ \mathbf{local}\ t^k\ e^*) \\
\mathbf{local}_n\{i; v_l^*\}\ \mathbf{trap}\ \mathbf{end} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{local}_n\{i; v_l^*\}\ L^k[v^n\ \mathbf{return}]\ \mathbf{end} & \hookrightarrow & v^n & \\
v_1^j\ v\ v_2^k; \mathbf{get\_local}\ j & \hookrightarrow & v & \\
v_1^j\ v\ v_2^k; v'\ (\mathbf{set\_local}\ j) & \hookrightarrow & v_1^j\ v'\ v_2^k; \epsilon & \\
v\ (\mathbf{tee\_local}\ j) & \hookrightarrow & v\ v\ (\mathbf{set\_local}\ j) & \\
s; \mathbf{get\_global}\ j & \hookrightarrow_i & s_{\mathsf{glob}}(i, j) & \\
s; v\ (\mathbf{set\_global}\ j) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \mathsf{glob}(i, j) = v \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ a\ o) & \hookrightarrow_i & t.\mathbf{const}\ \mathrm{const}_t(b^*) & \text{if } s_{\mathsf{mem}}(i, k+o, |t|) = b^* \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ tp\_sx\ a\ o) & \hookrightarrow_i & t.\mathbf{const}\ \mathrm{const}^{sx}_t(b^*) & \text{if } s_{\mathsf{mem}}(i, k+o, |tp|) = b^* \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ tp\_sx^?\ a\ o) & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ a\ o) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \mathsf{mem}(i, k+o, |t|) = \mathrm{bits}^{|t|}_t(c) \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ tp\ a\ o) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \mathsf{mem}(i, k+o, |tp|) = \mathrm{bits}^{|tp|}_t(c) \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ tp^?\ a\ o) & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
s; \mathbf{current\_memory}] & \hookrightarrow_i & \mathbf{i32.const}\ |s_{\mathsf{mem}}(i, *)|/64\,\mathrm{Ki} & \\
s; (\mathbf{i32.const}\ k)\ \mathbf{grow\_memory} & \hookrightarrow_i & s'; \mathbf{i32.const}\ |s_{\mathsf{mem}}(i, *)|/64\,\mathrm{Ki} & \text{if } s' = s \text{ with } \mathsf{mem}(i, *) = s_{\mathsf{mem}}(i, *)\ (0)^{k \cdot 64\,\mathrm{Ki}} \\
s; (\mathbf{i32.const}\ k)\ \mathbf{grow\_memory} & \hookrightarrow_i & \mathbf{i32.const}\ (-1) &
\end{array}
$$

Figure 2. Small-step reduction rules

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

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

where cl^* is the list of closures such that each $cl = \{\mathsf{inst}\ i,\ \mathsf{code}\ f\}$ 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 $j = |s_{\mathsf{tab}}|$ and $k = |s_{\mathsf{mem}}|$ in case the table or memory are defined in m itself as ($\mathbf{table}\ n_{\mathrm{t}}\ i_{\mathrm{t}}^{n_{\mathrm{t}}}$) or ($\mathbf{memory}\ n_{\mathrm{m}}$). New table or memory instances are created from such definitions as:

$$tabinst = (inst_{\mathsf{func}}(i_{\mathrm{t}}))^{n_{\mathrm{t}}} \qquad meminst = (0)^{n_{\mathrm{m}} \cdot 64\,\mathrm{Ki}}$$

Instantiation results in a new store s' which is s with $inst$ and possibly $tabinst$ and $meminst$ appended.

3.3 Reduction

The lower part of Figure 2 specifies WebAssembly execution in terms of a small-step reduction relation [36], 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 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 labels we 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 write x^* where n does not matter; ϵ denotes the empty sequence.

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

$$\begin{array}{rcl@{\qquad}rcll}
unop_t & : & t \to t & \mathrm{cvt}_{t_1,t_2}^{sx^?} & : & t_1 \rightharpoonup t_2 & \\
testop_t & : & t \to \mathsf{i32} & \mathrm{bits}_t^n & : & t \to b^n & (n \le |t|) \\
binop_t & : & t \times t \rightharpoonup t & \mathrm{const}_t & : & b^n \to t & (n = |t|) \\
relop_t & : & t \times t \to \mathsf{i32} & \mathrm{const}_t^{sx} & : & b^n \to t & (n < |t|)
\end{array}$$

The left group abstracts the actual numerics of those operators with the "obvious" semantics. The others implement conversions and reinterpretation from and to sequences of raw bytes b^n with optional sign extension. Some of the binary operators and conversions may trap for some inputs and thus are partial (written $\rightharpoonup$). Notably, $\mathsf{div\_}sx_{in}$, $\mathsf{rem\_}sx_{in}$, and float-to-int conversions trap for unrepresentable results.

Control Control constructs reduce into the auxiliary **label** construct. It represents a block along with its label's arity 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 it.

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 it has only values remaining, it is exited, leaving those values as results. Similarly in case 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 label's arity 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.

Calls Performing calls is a bit more involved. Both **call** and **call_indirect** reduce to the auxiliary form ($\mathbf{call}\ cl$) with 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 a closure looks at its type $t_1^n \to t_2^m$ to determine how many arguments n to pop off the stack. It creates a new **local** block, another auxiliary form similar to **label** but representing a call frame. It is indexed by the closure's return arity and holds a reference i to the closure's instance and a list of values as the state of the function's local variables, which is initialized with the arguments v^n followed by the zero-initialized locals of types t^k.

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

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

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's value.

Memory Load instructions look up their instance's memory and extract a slice of raw bytes of the appropriate length. We write $|t|$ or $|tp|$ for the byte width of 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** pushes the size of the memory in page units (i.e., dividing by $64\,\mathrm{Ki}$); **grow_memory** grows 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 fetched from untrusted sources. Before it 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, interleaved with binary decoding and compilation.

4.1 Typing Rules

Figure 3 defines the WebAssembly type system declaratively via the usual system of deduction rules [35]. 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. As in Section 3, we write C_{func} to access the func component of C and $C_{\mathsf{func}}(i)$ for indexing it; similarly for other components. We make a special case for $C_{\mathsf{label}}(i)$ 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 [14]). For extension, $C, \mathsf{local}\, t^*$ denotes the same context as C, but with the sequence t^* appended to the list of locals. In the case of $C, \mathsf{label}\,(t^*)$ the parentheses indicate that C's labels are extended with a single entry that is itself a list.

Instructions The function type $tf = t_1^* \to t_2^*$ 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 $C_{\mathsf{table}} = n$ in the rule for **call_indirect** ensures that it can only be used when a table is present. Similar conditions exist for all memory-related instructions, ensuring that a memory has been declared. The other side conditions for loads and stores make sure that alignment is no larger than the type's width and that packed access happens only with a packed type narrower than the computation type.

When typing sequences of instructions, the rule requires 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 effects to the top portion of the stack. 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 ($\mathbf{i32.const}$ 1) ($\mathbf{i32.const}$ 2) **i32.add**, this rule is invoked to weaken the type of the middle instruction from $\epsilon \to \mathsf{i32}$ to $\mathsf{i32} \to \mathsf{i32}\ \mathsf{i32}$, as needed to compose with the other two.

Polymorphism Some instructions are *polymorphic*. 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 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 or manually adjusting the stack before a control transfer. For example, a simple compiler can compositionally compile an expression like "$a + b$" into "$\mathrm{compile}(a)\ \mathrm{compile}(b)$ **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 only contain references to previous globals, and no 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 contains 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.

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

Typing Instructions $\boxed{C \vdash e^* : tf}$

$$\frac{}{C \vdash t.\mathbf{const}\ c : \epsilon \to t} \qquad \frac{}{C \vdash t.unop : t \to t} \qquad \frac{}{C \vdash t.binop : t\ t \to t} \qquad \frac{}{C \vdash t.testop : t \to \mathsf{i32}} \qquad \frac{}{C \vdash t.relop : t\ t \to \mathsf{i32}}$$

$$\frac{t_1 \neq t_2 \qquad sx^? = \epsilon \Leftrightarrow (t_1 = \mathbf{i}n \wedge t_2 = \mathbf{i}n' \wedge |t_1| < |t_2|) \vee (t_1 = \mathbf{f}n \wedge t_2 = \mathbf{f}n')}{C \vdash t_1.\mathbf{convert}\ t_2\_sx^? : t_2 \to t_1} \qquad \frac{t_1 \neq t_2 \qquad |t_1| = |t_2|}{C \vdash t_1.\mathbf{reinterpret}\ t_2 : t_2 \to t_1}$$

$$\frac{}{C \vdash \mathbf{unreachable} : t_1^* \to t_2^*} \qquad \frac{}{C \vdash \mathbf{nop} : \epsilon \to \epsilon} \qquad \frac{}{C \vdash \mathbf{drop} : t \to \epsilon} \qquad \frac{}{C \vdash \mathbf{select} : t\ t\ \mathsf{i32} \to t}$$

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

$$\frac{tf = t_1^n \to t_2^m \qquad C, \mathsf{label}\ (t_2^m) \vdash e_1^* : tf \qquad C, \mathsf{label}\ (t_2^m) \vdash e_2^* : tf}{C \vdash \mathbf{if}\ tf\ e_1^*\ \mathbf{else}\ e_2^*\ \mathbf{end} : t_1^n\ \mathsf{i32} \to t_2^m}$$

$$\frac{C_{\mathsf{label}}(i) = t^*}{C \vdash \mathbf{br}\ i : t_1^*\ t^* \to t_2^*} \qquad \frac{C_{\mathsf{label}}(i) = t^*}{C \vdash \mathbf{br\_if}\ i : t^*\ \mathsf{i32} \to t^*} \qquad \frac{(C_{\mathsf{label}}(i) = t^*)^+}{C \vdash \mathbf{br\_table}\ i^+ : t_1^*\ t^*\ \mathsf{i32} \to t_2^*}$$

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

$$\frac{C_{\mathsf{local}}(i) = t}{C \vdash \mathbf{get\_local}\ i : \epsilon \to t} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \mathbf{set\_local}\ i : t \to \epsilon} \qquad \frac{C_{\mathsf{local}}(i) = t}{C \vdash \mathbf{tee\_local}\ i : t \to t} \qquad \frac{C_{\mathsf{global}}(i) = \mathsf{mut}^?\ t}{C \vdash \mathbf{get\_global}\ i : \epsilon \to t} \qquad \frac{C_{\mathsf{global}}(i) = \mathsf{mut}\ t}{C \vdash \mathbf{set\_global}\ i : t \to \epsilon}$$

$$\frac{C_{\mathsf{memory}} = n \qquad 2^a \leq (|tp| <)^?|t| \qquad (tp\_sz)^? = \epsilon \vee t = \mathbf{i}m}{C \vdash t.\mathbf{load}\ (tp\_sz)^?\ a\ o : \mathsf{i32} \to t} \qquad \frac{C_{\mathsf{memory}} = n \qquad 2^a \leq (|tp| <)^?|t| \qquad tp^? = \epsilon \vee t = \mathbf{i}m}{C \vdash t.\mathbf{store}\ tp^?\ a\ o : \mathsf{i32}\ t \to \epsilon}$$

$$\frac{C_{\mathsf{memory}} = n}{C \vdash \mathbf{current\_memory} : \epsilon \to \mathsf{i32}} \qquad \frac{C_{\mathsf{memory}} = n}{C \vdash \mathbf{grow\_memory} : \mathsf{i32} \to \mathsf{i32}}$$

$$\frac{}{C \vdash \epsilon : \epsilon \to \epsilon} \qquad \frac{C \vdash e_1^* : t_1^* \to t_2^* \qquad C \vdash e_2 : t_2^* \to t_3^*}{C \vdash e_1^*\ e_2 : t_1^* \to t_3^*} \qquad \frac{C \vdash e^* : t_1^* \to t_2^*}{C \vdash e^* : t^*\ t_1^* \to t^*\ t_2^*}$$

Typing Modules

$$\frac{tf = t_1^* \to t_2^* \qquad C, \mathsf{local}\ t_1^*\ t^*, \mathsf{label}\ (t_2^*), \mathsf{return}\ (t_2^*) \vdash e^* : \epsilon \to t_2^*}{C \vdash ex^*\ \mathbf{func}\ tf\ \mathbf{local}\ t^*\ e^* : ex^*\ tf} \qquad \frac{tg = \mathsf{mut}^?\ t \qquad C \vdash e^* : \epsilon \to t \qquad ex^* = \epsilon \vee tg = t}{C \vdash ex^*\ \mathbf{global}\ tg\ e^* : ex^*\ tg}$$

$$\frac{(C_{\mathsf{func}}(i) = tf)^n}{C \vdash ex^*\ \mathbf{table}\ n\ i^n : ex^*\ n} \qquad \frac{}{C \vdash ex^*\ \mathbf{memory}\ n : ex^*\ n}$$

$$\frac{}{C \vdash ex^*\ \mathbf{func}\ tf\ im : ex^*\ tf} \qquad \frac{tg = t}{C \vdash ex^*\ \mathbf{global}\ tg\ im : ex^*\ tg} \qquad \frac{}{C \vdash ex^*\ \mathbf{table}\ n\ im : ex^*\ n} \qquad \frac{}{C \vdash ex^*\ \mathbf{memory}\ n\ im : ex^*\ n}$$

$$\frac{\begin{array}{c}(C \vdash f : ex_{\mathrm{f}}^*\ tf)^* \qquad (C_i \vdash glob_i : ex_{\mathrm{g}}^*\ tg_i)_i^* \qquad (C \vdash tab : ex_{\mathrm{t}}^*\ n)^? \qquad (C \vdash mem : ex_{\mathrm{m}}^*\ n)^? \\ (C_i = \{\mathsf{global}\ tg^{i-1}\})_i^* \qquad C = \{\mathsf{func}\ tf^*, \mathsf{global}\ tg^*, \mathsf{table}\ n^?, \mathsf{memory}\ n^?\} \qquad ex_{\mathrm{f}}^{**}\ ex_{\mathrm{g}}^{**}\ ex_{\mathrm{t}}^{*?}\ ex_{\mathrm{m}}^{*?}\ \text{distinct}\end{array}}{\vdash \mathbf{module}\ f^*\ glob^*\ tab^?\ mem^?}$$

Figure 3. Typing rules

4.2 Soundness

The WebAssembly type system enjoys standard *soundness* properties [41]. 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 the absence of undefined behavior in the execution semantics (assuming the auxiliary numeric primitives are well-defined).

In particular, this implies the absence of *type safety* violations such as invalid calls or illegal accesses to locals, it guarantees *memory safety*, and it ensures the inaccessibility of code addresses or the call stack. It also implies that the use of the operand stack is structured and its layout determined statically at all program points, which is crucial for efficient compilation on a register machine. Furthermore, it establishes memory and state *encapsulation* – i.e., abstraction properties on the module and function boundaries, which cannot leak information unless explicitly exported/returned – necessary conditions for user-defined security measures.

Store Typing Before we can state the soundness theorems concretely, we must extend the typing rules to stores and configurations as defined in Figure 2. These rules, shown in Figure 4, 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 3 is extended to $S; C \vdash e^* : tf$ by implicitly adding S to all rules – S is never modified or used by those rules, but is accessed by the new rules for **call** cl and **local**.

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

$$\frac{S = \{\mathsf{inst}\ C^*,\ \mathsf{tab}\ n^*,\ \mathsf{mem}\ m^*\} \qquad (S \vdash inst : C)^* \qquad ((S \vdash cl : tf)^*)^* \qquad (n \leq |cl^*|)^* \qquad (m \leq |b^*|)^*}{\vdash \{\mathsf{inst}\ inst^*,\ \mathsf{tab}\ (cl^*)^*,\ \mathsf{mem}\ (b^*)^*\} : S}$$

$$\frac{S_{\mathsf{inst}}(i) \vdash f : tf}{S \vdash \{\mathsf{inst}\ i,\ \mathsf{code}\ f\} : tf} \qquad \frac{(S \vdash cl : tf)^* \qquad (\vdash v : t)^* \qquad (S_{\mathsf{tab}}(i) = n)^? \qquad (S_{\mathsf{mem}}(j) = m)^?}{S \vdash \{\mathsf{func}\ cl^*,\ \mathsf{glob}\ v^*,\ \mathsf{tab}\ i^?,\ \mathsf{mem}\ j^?\} : \{\mathsf{func}\ tf^*,\ \mathsf{global}\ (\mathsf{mut}^?\ t)^*,\ \mathsf{table}\ n^?,\ \mathsf{memory}\ m^?\}}$$

$$\frac{\vdash s : S \qquad S; \epsilon \vdash_i v^*; e^* : t^*}{\vdash_i s; v^*; e^* : t^*} \qquad \frac{(\vdash v : t_v)^* \qquad S; S_{\mathsf{inst}}(i), \mathsf{local}\ t_v^*, \mathsf{return}\ (t^*)^? \vdash e^* : \epsilon \to t^*}{S; (t^*)^? \vdash_i v^*; e^* : t^*} \qquad \frac{}{\vdash t.\mathsf{const}\ c : t}$$

$$\frac{}{C \vdash \mathsf{trap} : tf} \qquad \frac{C \vdash e_0^* : t_1^n \to t_2^* \qquad C, \mathsf{label}\ (t_1^n) \vdash e^* : \epsilon \to t_2^*}{C \vdash \mathsf{label}_n\{e_0^*\}\ e^*\ \mathsf{end} : \epsilon \to t_2^*} \qquad \frac{S \vdash cl : tf}{S; C \vdash \mathsf{call}\ cl : tf} \qquad \frac{S; (t^n) \vdash_i v^*; e^* : t^n}{S; C \vdash \mathsf{local}_n\{i; v^*\}\ e^*\ \mathsf{end} : \epsilon \to t^n}$$

Figure 4. Store and configuration typing and rules for administrative instructions

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* $e^* = \mathsf{trap}$, *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.

5. Binary Format

WebAssembly is transmitted over the wire as a binary encoding of the abstract syntax presented in Figure 1. 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 are collected in their own section to allow sharing. Code for function bodies is deferred to a separate section placed after all declarations. This way, a browser engine can minimize page-load latency by starting *streaming compilation* as soon as function bodies arrive 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 one-byte opcodes (future opcodes may be multi-byte). All integral numbers, including opcode immediates, are encoded in LEB128 format [6]. The binary format strives for overall simplicity and is regular enough to be expressible by a simple grammar.

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 I/O. This intentional design separation is captured in the notion of *embedding* a WebAssembly implementation into an execution environment. The embedder defines how modules are loaded, how imports and exports between modules are resolved, provides foreign functions to accomplish I/O and timers, and specifies how WebAssembly traps are handled. In our work the primary use case has been the Web and JavaScript embedding, so these mechanisms are implemented in terms of JavaScript and Web APIs.

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.

Linking An embedder can instantiate multiple modules and use exports from one as imports to the other. This 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. It is up to producers to map their data types to numbers or the memory. This design provides maximum flexibility to producers, and unlike previous VMs, does not privilege any specific programming or object model while penalizing others. Though WebAssembly has a programming language shape, it is an abstraction over hardware, not over a programming language.

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

7. Implementation

A major design goal of WebAssembly has been high performance without sacrificing safety or portability. Throughout

its design process, we have developed independent implementations of WebAssembly in all major browsers to validate and inform the design decisions. This section describes some points of interest of those implementations.

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 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 individual functions to an interpreted internal bytecode format upon first execution, and later JIT-compiling the hottest functions. The advantage of this approach is faster startup and potentially lower memory consumption. We expect more strategies to evolve over time.

Validation A key design goal of WebAssembly has been fast validation of code. In the four aforementioned implementations, the same basic strategy of an abstract control stack, an abstract operand stack with types, and a forward program counter is used. Validation proceeds by on-the-fly checking of the incoming bytecodes, with no intermediate representation being constructed.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 Mozilla's SpiderMonkey engine includes two WebAssembly 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 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 details 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 flow[8]. In the case of V8, decoding targets the TurboFan compiler's sea of nodes [12] 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 directly 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.

Reference Interpreter In addition to production-quality implementations for four major browsers, we implemented a reference interpreter for the entire WebAssembly language. For this we used OCaml [26] 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 as well as 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 $t.\textbf{load}\ a\ o\ k$ with type t, alignment a, static offset o and dynamic address k represents a read of the memory at $s_{\mathsf{mem}}(i, k+o, |t|)$. 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 $k + o + |t| \leq memsize$. 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 $base[k+n]$.

Code Specialization While $base$ can be stored in a dedicated machine register for quick access, a JIT compiler 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 $k+o+|t| \leq memsize$ to $k \leq memsize - o - |t|$ and then constant-fold the right hand side.[9] Although $memsize$ is not necessarily constant (since memory can be grown dynamically) it changes so infrequently that the JIT can embed it in generated machine code, later *patching* the machine code if the memory size changes. Unlike other speculation techniques, the change of a constant value is controlled enough that deoptimization [22] of the code is not necessary.

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

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

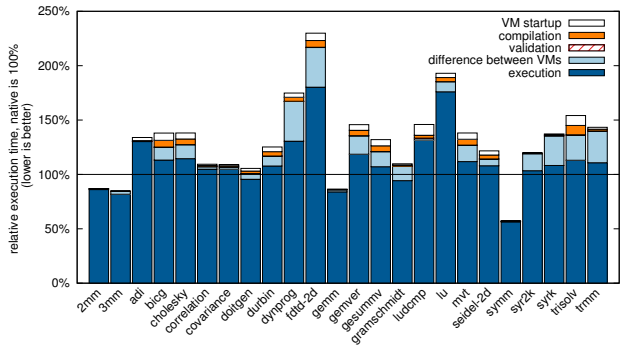

Figure 5. Relative execution time of the PolyBenchC benchmarks on WebAssembly normalized to native code

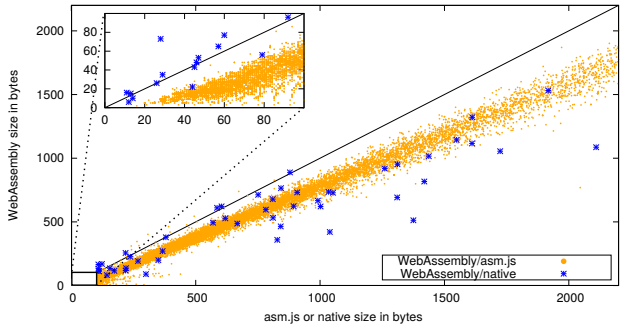

Figure 6. Binary size of WebAssembly in comparison to asm.js and native code

Virtual Memory Techniques On 64 bit platforms, the WebAssembly engine can make use of virtual memory techniques to eliminate 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 and offsets 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 and base memory address does not change, no code patching is necessary to grow memory.

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× improvement in compilation speed with 8 compilation threads.

Code Caching 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 Web pages 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 an order of magnitude startup time improvement.

7.3 Measurements

Figure 5 shows the execution time of the PolyBenchC [7] benchmark suite running on WebAssembly on both V8 and SpiderMonkey normalized to native execution.[10] Times for both engines are shown as stacked bars, and the results show that there are still some execution time differences between them because of different code generators[11]. We measured a fixed VM startup time of 18 ms for V8 and 30 ms for SpiderMonkey. These times are included along with compilation times as bars stacked on top of the execution time of each benchmark. Note that the fixed cost of VM startup as a stacked bar also gives a clue to which benchmarks are short-running (startup and compilation are significant). Overall, the results show that WebAssembly is very competitive with native code, with 7 benchmarks within 10 % of native and nearly all of them within 2× of native.

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

Figure 6 compares code sizes between WebAssembly (generated from asm.js inside V8), minified asm.js, and x86-64 native code. For the WebAssembly to asm.js comparison we use the Unity benchmarks [9], for the WebAssembly to native code comparison the PolyBenchC [7] and SciMark [8] benchmarks. For each function in these benchmarks, a yellow point is plotted at $\langle size_{\mathsf{asmjs}}, size_{\mathsf{wasm}} \rangle$ and a blue point at $\langle size_{\mathsf{x86}}, size_{\mathsf{wasm}} \rangle$. Any point below the diagonal represents a function for which WebAssembly is smaller than the corresponding other representation. On average, WebAssembly code is 62.5 % the size of asm.js (median 68.6 %), and 85.3 % of native x86-64 code size (median 78 %). The few cases where WebAssembly is larger than native code are due to C's pointers to stack locals requiring a shadow stack.

[10] Experiments were performed on a Linux 3.13.0-100 workstation with two 12-core 2.60 GHz Intel Xeon processors (2 hyperthreads per core), 30 MiB shared L3-cache, and 64 GiB RAM. We use Clang 3.8.0-2 to generate native code and Emscripten 1.37.3 to generate both WebAssembly and asm.js code. All results are averaged over 15 runs.

[11] V8 is faster on some benchmarks and SpiderMonkey on others. Neither engine is universally faster than the other, but both achieve good results. The difference is most pronounced for short-running programs.

8. Related Work

The most direct precursors of WebAssembly are (P)NaCl [42, 11, 18] 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 [34] and Cyclone [23] have imposed safety at the C language level, which generally requires program changes. Other attempts have enforced it at the C abstract machine level with combinations of static and runtime checks [10, 20, 31], sometimes assisted by hardware [16, 30]. For example, the Secure Virtual Architecture [13] defines an abstract machine based on LLVM bitcode that enforces the SAFECode [17] properties.

Typed intermediate languages carry type information throughout the compilation process. For example, TIL [28] and FLINT [37] pioneered typed ILs for functional languages, allowing higher confidence in compiler correctness and more type-based optimizations. However, typed ILs have a different purpose than a compilation target. They are typically compiler-specific data structures that exist only as an intermediate stage of compilation, not as a storage or execution format.

Typed Assembly languages [29] do serve as a compilation target, usually 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 [32]) code by preserving types throughout compilation. The modelling of complex types imposes a severe burden on compilers, requiring them to preserve and transform quantified types throughout compilation and avoid optimizations that break the type system.

We investigated reusing another compiler IR which has a binary format. In fact, LLVM bitcode is the binary format used by PNaCl. Disadvantages with LLVM bitcode in particular are that (1) it is not entirely stable, (2) it has undefined behavior which had to be corrected in PNaCl, (3) it was found to be less compact than a stack machine, (4) it essentially requires every consumer to either include LLVM or reimplement a fairly complex LLVM IR decoder/verifier, and (5) the LLVM backend is notoriously slow. Other compiler IRs have similar, sometimes worse, properties. In general, compiler IRs are better suited to optimization and transformation, and not as compact, verifiable code formats.

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

Mu [40] 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, 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 with bytecode as their 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 [27] and CIL [33] and their validation algorithms. By designing 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 in the current JVM specification, while for WebAssembly it fits on one page (Figure 3). It took a decade of research to hash out the details of correct JVM verification [25], including the discovery of inherent vulnerabilities [15, 19] – 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 [39] that had to be fixed with the addition of stack maps.

Both the JVM and the CIL, as well as Android Dalvik [3], allow bytecode to create irreducible loops and unbalanced locking structures, features which usually cause optimizing JITs to give up and relegate methods containing those constructs to 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 from C/C++. A few important features are still missing for fully comprehensive support of this domain and will be added in future versions, such as zero-cost *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 an attractive target for *high-level* languages by including relevant primitives like *tail calls*, *stack switching*, or *coroutines*. A highly important goal is to provide access to the advanced and highly tuned *garbage collectors* that are built into all Web browsers, thus eliminating the main shortcoming relative to JavaScript when compiling to the Web.

Finally, we anticipate that WebAssembly will find a wide range of use cases off the Web, and expect that it will potentially grow additional feature necessary to support these.

Acknowledgements

We thank the members of the W3C WebAssembly community group for their many contributions to the design and implementation of WebAssembly, and Conrad Watt for valuable feedback on the formalization.

References

[1] Activex controls. 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] PolyBenchC: the polyhedral benchmark suite. http://web.cs.ucla.edu/~pouchet/software/polybench/. Accessed: 2017-03-14.

[8] Scimark 2.0. http://math.nist.gov/scimark2/. Accessed: 2017-03-15.

[9] Unity benchmarks. http://beta.unity3d.com/jonas/benchmark2015/. Accessed: 2017-03-15.

[10] 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.

[11] 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.

[12] C. Click and M. Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, Mar. 1995.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Not.*, 41(6):144–157, June 2006.

[18] A. Donovan, R. Muth, B. Chen, and D. Sehr. PNaCl: Portable native client executables. Technical report, 2010.

[19] 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.

[20] 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.

[21] D. Gudeman. Representing type information in dynamically typed languages. Technical Report 93-27, Department of Computer Science, University of Arizona, Phoenix, Arizona, October 1993.

[22] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Not.*, 27(7):32–43, July 1992.

[23] 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.

[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, Palo Alto, California, Mar 2004.

[25] X. Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason.*, 30(3-4):235–269, Aug. 2003.

[26] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system*. INRIA, 2016.

[27] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java Virtual Machine Specification (Java SE 8 Edition). Technical report, Oracle, 2015.

[28] 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.

[29] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM TOPLAS*, 21(3):527–568, May 1999.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe

retrofitting of legacy code. *SIGPLAN Not.*, 37(1):128–139, Jan. 2002.

[35] B. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, USA, 2002.

[36] G. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[37] 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.

[38] 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.

[39] 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.

[40] 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 *1st Summit on Advances in Programming Languages*, volume 32 of *SNAPL '15*, pages 321–336, Dagstuhl, Germany, 2015.

[41] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115, 1994.

[42] 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.

[43] 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.