WebAssembly: high speed at low cost for everyone

Andreas Rossberg
Google
rossberg@mpi-sws.org

Abstract

WebAssembly is a new language- and platform-independent binary code format bringing native-code performance to the web. We present its design and report our experience with specifying its semantics via a reference interpreter written in OCaml, that currently serves as a proxy for a future formal specification.

1. Introduction

Like it or not, the Web has become one of the most important platforms for application development. Yet there is only one programming language that the Web understands: JavaScript. Literally hundreds of compilers have been written that translate almost every imaginable language into JavaScript in order to run on the Web [1]. Unfortunately, JavaScript is not the most delightful compilation target: it's brittle and often slow in unpredictable ways, all static knowledge a compiler might have is lost (only to be reconstructed expensively by the JavaScript engine), and there is little to no control over aspects like memory or data layout.

Several proposals have been made to bring more well-behaved code formats to the Web, in particular (P)NaCl [2] and asm.js [3]. Unfortunately, they each have a number of shortcomings, such as lack of portability, slow client-side compilation, large distributives, or difficult forward evolution.

WebAssembly [4] is a concerted, open source effort by all major browser vendors (and including the prior asm.js and NaCl teams), to overcome these disadvantages. In a nutshell, it standardises a simple low-level binary code format that is

- compact,
- language-independent,
- platform-independent,
- easy to compile efficiently on all modern hardware,
- safe to execute.

A first version of WebAssembly is currently being implemented in several major browsers and expected to ship by the end of this year. Moreover, WebAssembly will be equipped with a formal specification. As a proxy to this specification, it is currently defined by a reference interpreter written in OCaml [5], closely mirroring formal notation, while also being executable efficiently.

We present the design and roadmap of WebAssembly and report on our use of ML/OCaml as a means for defining the language.

2. WebAssembly

In its current state, WebAssembly effectively is a very simple programming language with the following characteristics:

Machine Types and Operators. There are only four basic data types: 32/64 bit integers and 32/64 bit IEEE floating point numbers. The set of built-in operators for these types mirrors the machine instructions that are widely available on modern CPUs. They map to machine code in a predictable manner and with predictable performance on all relevant platforms.

High-level Data Flow. WebAssembly is an expression language: it is defined as an AST in which machine operators return values and can be used as operands to other operators. The binary format is a post-order serialisation of this AST. This design provides for more compact binaries than 3-address code or SSA form would.

Low-level Control Flow. Control flow is available mainly in the form of sequential blocks and branch instructions, plus a structured conditional. However, branches are still structured: they can only break *out* of an expression, not jump *into* one. This prevents irreducible control flow and associated complications (it's the producers' responsibility to transform irreducible control flow into structured form [6]). Being an expression language, WebAssemblys branch operators can also take a value that is passed on to the continuation of the exited expression.

Linear Memory. A WebAssembly program can use one big byte array (of configurable size) as its linear memory. It is not aliasing any VM-internal memory. Integer values hence can represent pointers into that memory. All memory access is bounds-checked. There is no built-in support for garbage collection (yet!), applications have to manage their own memory.

Functions. A WebAssembly module is structured into (mutually recursive) functions. Each function can declare an arbitrary number of local variables that act as virtual registers. A function can be called either directly or indirectly through an indexed table, emulating the equivalent of function pointers.

Typed and Deterministic. WebAssembly has a well-defined static and dynamic semantics. Code is fully typed, and checked prior to compilation. Failure conditions that cannot be detected statically lead to traps that abort the execution. There is no undefined, underspecified or non-deterministic behaviour.[1]

Figure 1 gives the grammar of types, functions and expressions in WebAssembly, in its current incarnation. It features the four basic machine types, and an extended set for memory access. Every expression and function evaluates to either a numeric value type, or to the empty value (i.e., has type unit).

Operators can be grouped into three main classes: control operators, arithmetic operators, and memory operators. Most of these are straighforward. Blocks and loops bind local labels (using de Bruijn indexing) for "forward" and "backward" branches, respectively: targeting a block breaks out of the block, targeting a loop continues its next iteration. A branch that targets a block of non-empty type is required to have an additional argument value to be passed to the block's continuation.

[1] With the sole exception of the exact NaN bit patterns generated by certain floating point operators, because CPUs are too diverge to fix one.

$$
\begin{array}{llll}
\text{(value types)} & t & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\text{(memory types)} & tm & ::= & t \mid \mathsf{i8} \mid \mathsf{i16} \\
\text{(expression types)} & te & ::= & t^? \\
\text{(function types)} & tf & ::= & t^* \to te
\end{array}
$$

$$
\begin{array}{rlll}
\text{(var's)} & x & ::= & n \\
\text{(const's)} & c & ::= & \dots \\
\text{(expr's)} & e & ::= & \textbf{unreachable} \mid \textbf{nop} \mid \\
 & & & \textbf{block}\ e^* \mid \textbf{loop}\ e^* \mid \\
 & & & \textbf{br}\ x\ e^? \mid \textbf{br\_if}\ x\ e^?\ e \mid \textbf{br\_table}\ x^*\ x\ e^?\ e \mid \\
 & & & \textbf{if}\ e\ e^*\ e^* \mid \textbf{select}\ e\ e\ e \mid \\
 & & & \textbf{call}\ x\ e^* \mid \textbf{call\_indirect}\ tf\ e\ e^* \mid \\
 & & & \textbf{get\_local}\ x \mid \textbf{set\_local}\ x\ e \mid \\
 & & & \textbf{load}\ tm\ n\ e \mid \textbf{store}\ tm\ n\ e\ e \mid \\
 & & & t.\textbf{const}\ c \mid t.unop\ e \mid t.binop\ e\ e \mid \\
 & & & t.cmpop\ e\ e \mid t.cvtop\ e \mid \\
 & & & \textbf{current\_memory} \mid \textbf{grow\_memory}\ e \\
 & unop & ::= & \textbf{neg} \mid \textbf{abs} \mid \dots \\
 & binop & ::= & \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div\_s} \mid \textbf{div\_u} \mid \dots \\
 & cmpop & ::= & \textbf{eq} \mid \textbf{ne} \mid \textbf{lt} \mid \textbf{gt} \mid \dots \\
 & cvtop & ::= & \textbf{convert}/t \mid \textbf{reinterpret}/t \\
\text{(func's)} & f & ::= & \textbf{func}\ tf\ t^*\ e
\end{array}
$$

Figure 1. WebAssembly core syntax

A family of load and store operators is provided for accessing the linear memory. Memory is not typed, any address can be accessed as any of the four base types or as 8/16 bit integers. Each WebAssembly module has to specify an initial size for its linear memory, but can later grow it using the grow_memory operator.

3. Reference Interpreter

We intend to equip WebAssembly with a complete formal specification, consisting of grammar, typing rules, structured operational semantics, and a corresponding soundness proof. However, while still iterating on the design it is beneficial to have a specification that is both executable and easy to change. We hence started out "specifying" the language by means of a reference interpreter, as a proxy for the real specification. Much care has been taken to structure the implementation in a way that comes as close to a formal formulation as possible.

We chose OCaml for a various reasons:

- As a functional language, it is high-level and abstract enough for quick turn-around; the initial interpreter was completed in less than 3 days. Algebraic types and pattern matching are of course crucial for adequate formulation of AST and semantics.
- The type system provides a lot of high-level guidance and directly expresses most of the invariants we care about. In 9 months less than a handful of bugs surfaced, all of them minor.
- Modules and functors avoid code duplication, e.g., for implementing the semantics of arithmetic operators of different sizes.
- The execution model is a good match for WebAssembly; e.g., we use exceptions to implement structured branching.
- OCaml can efficiently run sample programs, even when working in a fairly high-level and declarative style.
- All necessary machine types are readily available – except 32 bit floats, which require emulation through 64 bit and rounding; similarly, unsigned arithmetics.
- Big arrays can emulate WebAssembly's untyped linear memory efficiently (with a tiny bit of Obj.magic).
- OCaml is portable and easily installed on most relevant platforms, without external dependencies. Flexible versions of lex and yacc are part of the distribution.

Most other typed functional languages would probably have fit the bill on most of these points, but not all.

However, not all is golden. We ran into a number of problems, too. These can broadly be grouped into the following categories:

- Language. Mostly minor issues, but some create pitfalls (e.g., unspecified evaluation order), or harm readability (e.g., the lack of convenient operators for non-default integer types), or increase the barrier to entry (e.g., syntax quirks, type errors).
- Compiler. One problem is that OCaml uses x87 instructions and registers on x86 platforms, and provides no way to disable that. That causes a number of violations of IEEE semantics, not all of which can be worked around. The lack of single precision floats or unsigned arithmetics is unfortunate for us as well. Likewise, big arrays are limited to `int` size, i.e., 1 GiB on 32 bit machines.
- Eco system. To minimise the barrier to entry, a "single-click" solution was desired for building the interpreter. Unfortunately, OCaml binaries aren't available or up-to-date on all relevant platforms. Moreover, the tool chain has little love for Windows.
- Familiarity. The biggest practical problem. Most project members did not know OCaml (or functional programming in general), and the barrier to entry seemed too high for them. There was significant pressure to "rewrite it in a production language like C++" – which of course would lose most benefits. A few members came up to speed quickly and contribute significantly (though still feel uncomfortable), but most just ignore it. Consequently, the "spec" is not as accessible as we hoped, despite being written for clarity. Inexperience with formal semantics contributed to that.

We will go into much more detail in the presentation.

4. Outlook

As is obvious from Figure 1, WebAssembly is a really primitive language at the moment. That is intentional: version 1.0 of WebAssembly focusses on being a better replacement for asm.js and compiling low-level languages like C.

For the future, however, various extensions are planned, in order to support more high-level languages. These include garbage-collected data (tuples and arrays), closures, tail calls, multiple return values, exceptions, threads, and SIMD. Moreover, despite the name, we want to make WebAssembly useful outside the Web, essentially growing it into a proper "micro VM" [7]. We expect the reference interpreter to continue to serve as a test bed and proxy "spec" for the further evolution of WebAssembly.

References

[1] List of languages that compile to JS. `github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js`

[2] Native Client. `developer.chrome.com/native-client`

[3] asm.js – an extraordinarily optimizable, low-level subset of JavaScript. `asmjs.org`

[4] WebAssembly. `webassembly.github.io`

[5] OCaml. `ocaml.org`

[6] A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. OOPSLA 2011

[7] K. Wang, Y. Lin, S. M. Blackburn, M. Norrish, and A. L. Hosking, Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development. SNAPL 2015