Introduction to WebAssembly

Harold Mills

2 January 2018

Overview

- Motivation
- WebAssembly
- Status
- Examples
- Future

Motivation

- By historical accident, JavaScript is *the* programming language supported directly by all of the major browsers.
- Other languages are supported indirectly via compilation to JavaScript or a subset thereof (e.g. asm.js).
- But JavaScript was not designed as a compilation target, and one could do better.
- This hinders the use of a lot of existing code (e.g. C/C++) in web browsers.

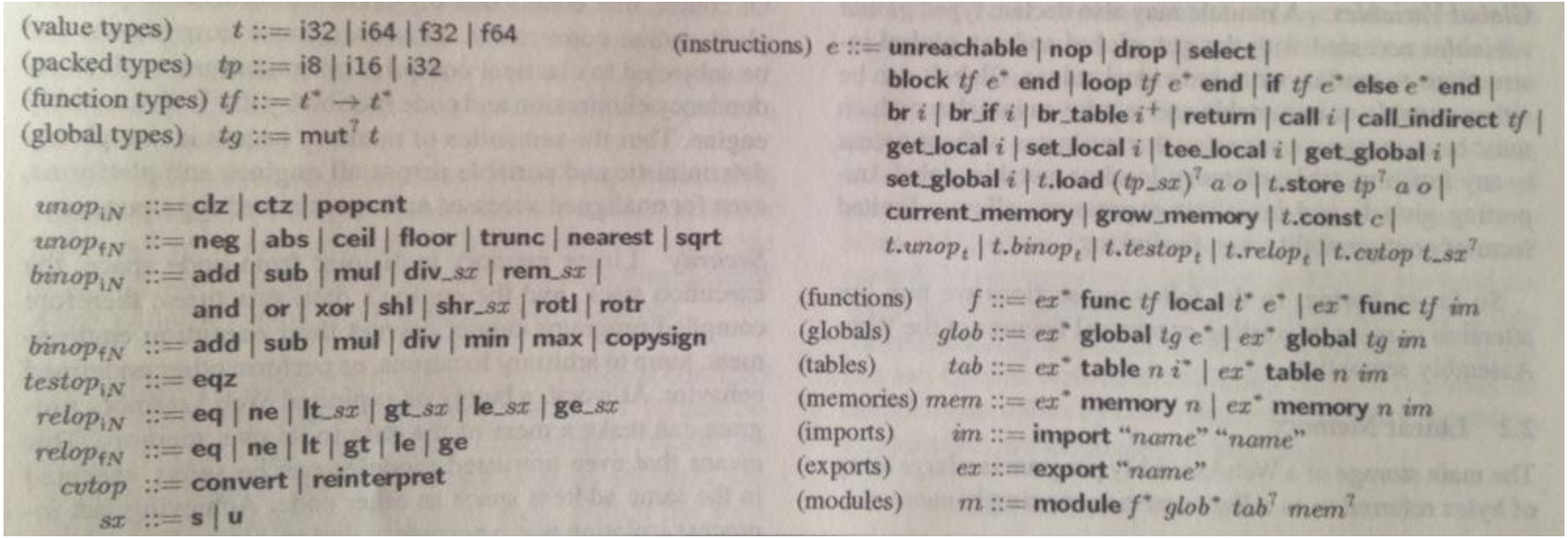
WebAssembly (a.k.a. wasm)

- Safe, portable, compact, and fast low-level bytecode for the web.
- Designed from the ground up as a compilation target for higher-level languages.
- Will complement, not replace, JavaScript.

Stack Machine

- WebAssembly bytecode is for a stack machine that is an abstraction over modern general-purpose computing hardware.
- The bytecode and machine were designed with a formal semantics from the start to help ensure safety.
- Implementations have a lot of latitude internally as long as they interpret the bytecode with the specified semantics.

Abstract Syntax



(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) $tg ::= \mathsf{mut}^? \; t$

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

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

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

Figure from Haas et al. - Bringing the Web up to Speed with WebAssembly (2017)

WebAssembly Modules

- WebAssembly code lives in *modules* that define entities of several types, including *functions* and *globals*.
- A WebAssembly module can export definitions for import into other WebAssembly or JavaScript modules, and can likewise import definitions from other modules.

WebAssembly Text Format

```
(module
  (func $add (param $lhs i32) (param $rhs i32) (result i32)
    get_local $lhs
    get_local $rhs
    i32.add)
  (export "add" (func $add))
)
```

Safety

- All WebAssembly code is *verified* before execution to check its integrity.
- Because WebAssembly was developed in conjunction with a formal semantics, its verification is *much* simpler than that of, say, JVM bytecode.
- At runtime memory is partitioned so that WebAssembly code cannot corrupt its execution environment. For example, all memory references are bounds checked and WebAssembly code cannot modify itself.

Portability

- Teams from all four major browser vendors are participating in WebAssembly development.
- Recent versions of all four major browsers support WebAssembly out of the box.
- Additional WebAssembly implementations will make it available elsewhere.

Compactness

- Stack machine architecture was chosen over other contenders (e.g. register machine) because resulting code is more compact.
- WebAssembly code is usually more compact than minified JavaScript.

Speed

- Because of its compactness, WebAssembly downloads quickly.
- Compilation and some optimization happen before downloading, not after.
- Verification is performed in one very quick pass.
- Code execution is fast (see later slides).

Status

- Still somewhat early in WebAssembly development.
- But as of late 2017, WebAssembly Minimum Viable Product (MVP) supported out of the box in recent versions of Chrome, Firefox, Safari, and Edge.
- Future WebAssembly versions will be backward compatible with MVP.

Emscripten

- Compiles C and C++ code to WebAssembly (or asm.js).
- Can also generate “glue” JavaScript and (optionally) HTML.

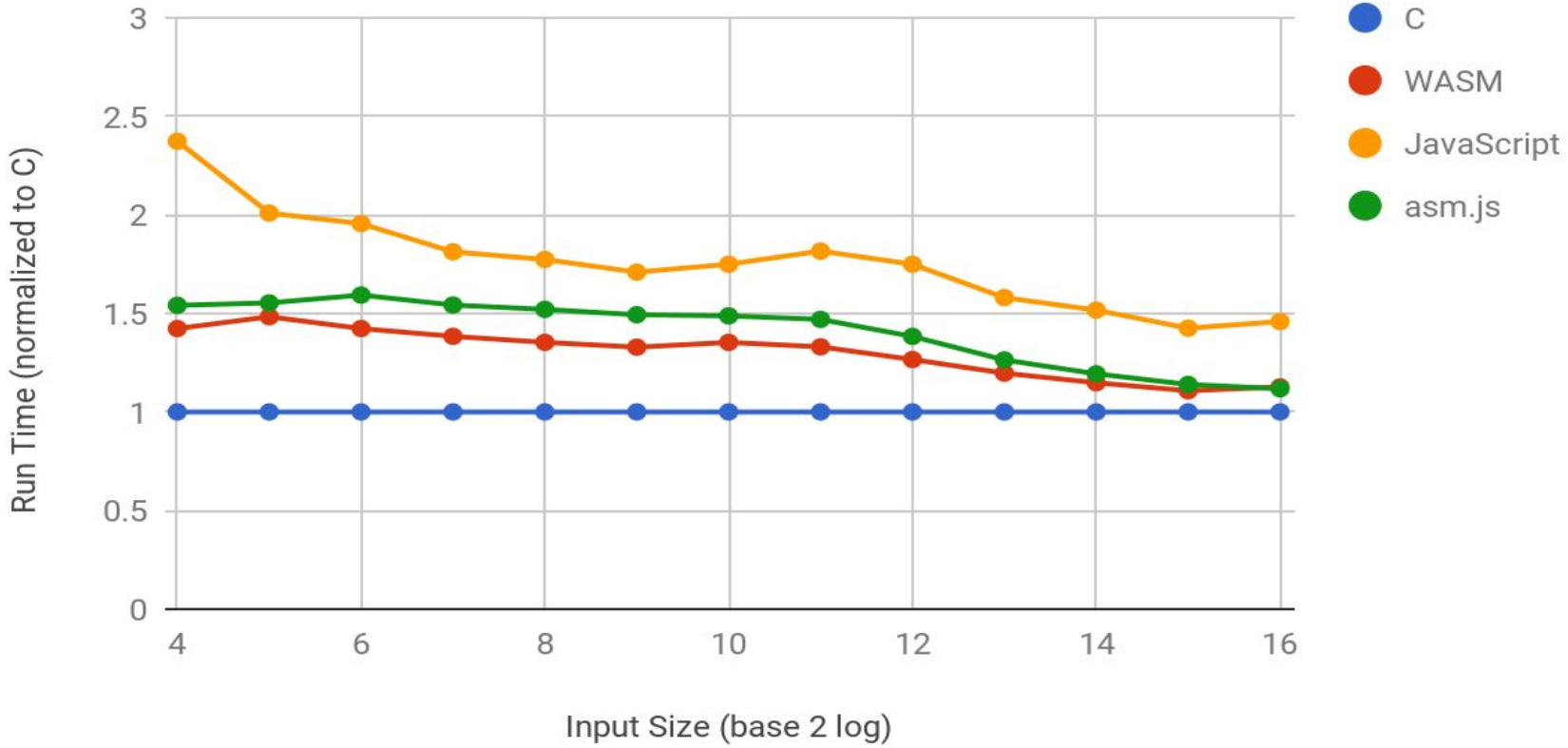
WebAssembly/JavaScript Interaction

- Unfortunately, can’t just use <script type=’module’> tag or ES6 import statement (at least not yet).
- Emscripten supports specification of WebAssembly module exports, either in C and C++ source code or via compiler command line arguments.
- A WebAssembly JavaScript API supports instantiation of and interaction with WebAssembly modules, for example to call exported functions.
- Emscripten also supports several ways to invoke JavaScript code from C and C++ code.

FFT Speed Tests

- The Fast Fourier Transform (FFT) is an important numerical algorithm that multiples a sequence of numbers (i.e. a vector) by a particular matrix.
- The FFT has multiple uses for audio signal processing, including spectrum analysis and frequency-selective filtering.
- I measured the speeds of FFTs of a range of sizes (powers of two from 2^4 to 2^{16}) coded, compiled, and run 4 ways:
  - C compiled to native code with clang, run via bash
  - C compiled to WebAssembly by Emscripten, run in Firefox
  - C compiled to asm.js by Emscripten, run in Firefox
  - JavaScript, run in Firefox

FFT Run Times



Demo

http://websightjs.com

Future

- To better support C/C++:
  - Zero-cost exceptions
  - Threads
  - SIMD instructions
- To support other, higher-level languages:
  - Garbage collection
  - Additional features
- Better integration with browser developer tools.
- Further standardization:
  - Formation of W3C WebAssembly Working Group
  - Formal W3C standard

Summary

- WebAssembly is a low-level compilation target that should help bring first-class support for languages other than JavaScript to web browsers.
- WebAssembly has arrived in the sense that it is now supported by all major browsers.
- WebAssembly will continue to develop. I look forward to:
  - More straightforward WebAssembly/JavaScript interaction.
  - Support for languages other than C and C++.

Resources

- WebAssembly home page: http://webassembly.org
- MDN WebAssembly documentation: https://developer.mozilla.org/en-US/docs/WebAssembly
- Academic paper for programming language aficionados: Bringing the Web up to Speed with WebAssembly
- Emscripten: https://github.com/kripken/emscripten
- These slides and my FFT tests are at https://github.com/HaroldMills/WebAssembly-Introduction