Wasm_of_ocaml

Jérôme Vouillon

Tarides
WebAssembly
WebAssembly (Wasm)

Widely implemented in web browsers

Low level language

- Compact binary format
- Only scalar values: \(i32\), \(i64\), \(f32\), \(f64\)
- Linear memory

+ Good target for C/C++/Rust
  - Not so suitable for a GC-based language
  - Hard to use Web APIs from Wasm
Wasm example

(module
  (func $fibonacci (param $n i32) (result i32)
    (if (i32.lt_u (local.get $n) (i32.const 2))
      (then
        (return (local.get $n)))
      (else
        (return
          (i32.add
            (call $fibonacci (i32.sub (local.get $n) (i32.const 1))))
            (call $fibonacci (i32.sub (local.get $n) (i32.const 2)))))
    (export "fibonacci" (func $fibonacci)))
)
Wasm GC

Extension of Wasm with reference types

- No need to reimplement a GC
- Can manipulate JavaScript values
Wasm_of_ocaml
Js_of_ocaml

Industrial-strength compiler

Compile OCaml bytecode to JavaScript

- Easy to maintain (fairly stable API)
- Easy to use (no need to recompile libraries)
Wasm_of_ocaml

Retarget Js_of_ocaml to generate WebAssembly code

Hope: better and more consistent performances

Goal: minimize user changes
Comparison with Wasocaml

Wasocaml (Léo Andrès, Pierre Chambard): direct modification of the OCaml compiler

- Better generated code, but probably harder to use and maintain
- Expect to share a common runtime environment
Implementation
Compilation process

Existing Js_of_ocaml code

- Bytecode parsing
- Optimization passes on SSA intermediate code

New

- Closure conversion
- Generate structured code (reimplemented)
  - *Beyond Relooper*, Norman Ramsey
- Generate Wasm instructions
Binaryen

Really useful tools

- **wasm-opt**: generate binary format + code optimizations
- **wasm-merge**: linker
- **wasm-metadce**: inter-language linking / deadcode elimination
Value representation: basic types

Uniform representation of values: (ref eq)

Integers: (ref i31)

Blocks: arrays (first field is an integer tag)
  (type $block (array (mut (ref eq))))

Other types:
  (type $string (array (mut i8)))
  (type $float (struct (field f64)))
Function calls

Need to deal with currying (functions can be overapplied or underapplied)

Most of the time, the number of parameters and arguments match

- **call** (a given function) when the function is known
- **call_ref** when the function arity is known
- use intermediate function otherwise
Value representation: closures

(type $function_1 (func (param (ref eq) (ref eq)) (result (ref eq))))

(type $closure (sub (struct (field (ref $function_1)))))

(type $env_1_2
  (sub final $closure
    (struct (field (ref $function_1))
      (field (ref eq)) (field (ref eq))))))

- Cast at the beginning of the function to recover the closure's type
- Need to experiment with more precise environment fields
Value representation: closures

(type $function_1 (func (param (ref eq) (ref eq)) (result (ref eq))))
(type $closure (sub (struct (field (ref $function_1)))))

(type $function_2
  (func (param (ref eq) (ref eq) (ref eq)) (result (ref eq))))

(type $closure_2
  (sub $closure (struct (field (ref $function_1)) (field (ref $function_2))))))

(type $env_2_2
  (sub final $closure_2
    (struct (field (ref $function_1)) (field (ref $function_2))
      (field (ref eq)) (field (ref eq)))))
Function application

(func $apply_2 (param $x (ref eq)) (param $y (ref eq)) (param $f (ref eq)) (result (ref eq))
 (local $g (ref eq))
 (drop
 (block $not_exact (result (ref eq))
 (return_call_ref $function_2
 (local.get $x) (local.get $y) (local.get $f)
 (struct.get $closure_2 1
 (br_on_cast_fail $not_exact (ref eq) (ref $closure_2) (local.get $f))))))
 (local.set $g
 (call_ref $function_1 (local.get $x) (local.get $f)
 (struct.get $closure 0
 (ref.cast (ref $closure) (local.get $f))))))
 (return_call_ref $function_1 (local.get $y) (local.get $g)
 (struct.get $closure 0 (ref.cast (ref $closure) (local.get $g))))))
Effect handlers

- **JS Promise API**

  Pierre Chambard:
  
  “I was asked [...] whether promise-integration would allow implementing OCaml effects handler. [...] it seems that this would be sufficient.”

  No cost when not performing effects, slow otherwise

- **Partial CPS transformation**

  Inherited from Js_of_ocaml

  Tail calls!
Interfacing with JavaScript
How it works

- Enough to provide just a rather small number of primitives
  - Property access: `x[y]`
  - Function call: `x.apply(null, args)`
  - Conversions between JavaScript and OCaml strings

- The compiler actually generates inline JavaScript code
  - Avoid string conversions for constant strings, property and method names
  - More efficient code for property access / method call
Example: function calls

**JavaScript**

```javascript
fun_call:(f,args)=>f.apply(null,args)
```

**Wasm**

```wasm
(import "bindings" "fun_call" (func $fun_call (param anyref) (param anyref) (result anyref)))
(func (export "caml_js_fun_call") (param $f (ref eq)) (param $args (ref eq)) (result (ref eq))
   (return_call $wrap (call $fun_call (call $unwrap (local.get $f))
      (call $unwrap (call $caml_js_from_array (local.get $args))))))
```

**OCaml** *(Js_of_ocaml library)*

```ocaml
external fun_call : 'f -> any array -> 'res = "caml_js_fun_call"
```
Differences between 

Js_of_ocaml and Wasm_of_ocaml

Js_of_ocaml

- JavaScript objects manipulated directly
- OCaml integers and floats all mapped to JavaScript numbers

Wasm_of_ocaml

- JavaScript objects (including floats) are boxed (do not belong to `(ref eq)`)
- JavaScript integers still mapped to OCaml integers `(ref i31)`
JavaScript object wrapping

(type $js (struct (field anyref)))

(func $wrap (param $v anyref) (result (ref eq))
  (block $is_eq (result (ref eq))
    (return (struct.new $js (br_on_cast $is_eq anyref (ref eq) (local.get $v))))))

(func $unwrap (param $v (ref eq)) (result anyref)
  (block $not_js (result anyref)
    (return
      (struct.get $js 0 (br_on_cast_fail $not_js (ref eq) (ref $js) (local.get $v))))))

```
Needed changes in user code

- Explicit float conversions
- Physical equality no longer works on JavaScript values
- Typed array (typing / performance)

Be Sport web app

- About 100,000 lines of code
- About 100 lines changed (mostly float conversions)
Taking advantage of JavaScript
Floats

Math operations

- Many function from the Math object (cos, exp, …)
- Remainder operator x % y (for floats)

Conversions between floats and strings
Using maps and weak pointers

Weak arrays and ephemerons

- Weak, WeakMap

Marshalling

- Map object, to deal with sharing
Big integers (zarith)

Use binaryen’s wasm-metadce + Js_of_ocaml linker

JavaScript

//Provides: wasm_z_add
//Requires: wasm_z_normalize
function wasm_z_add(z1, z2) { return wasm_z_normalize(BigInt(z1) + BigInt(z2)) }

WebAssembly

(import "js" "wasm_z_add" (func $add (param (ref any)) (param (ref any)) (result (ref any)))))
(func (export "ml_z_add")
 (param $z1 (ref eq)) (param $z2 (ref eq)) (result (ref eq))
 (return_call $wrap_bigint
  (call $add (call $unwrap_bigint (local.get $z1)) (call $unwrap_bigint (local.get $z2))))))
Performance results
Microbenchmarks

- Twothird of the JavaScript running time
- Twice slower than native code
Exceptions

Zero-cost exceptions are slow…
Larger benchmarks

ocamlc

CAMLBOY

Headless benchmarking mode: from 1200 fps to 1850 fps (50% faster)

The framebuffer (typed array) is the bottleneck
Bonsai

Library for building interactive browser-based UI

Table benchmark: 100 small benchmarks

Arithmetic mean:

  Javascript: 1.76ms
  Wasm (current implementation): 0.95ms
  Wasm (with stringref proposal): 0.84ms
Cost of casts and bound checks

V8 makes it possible to skip checks

ocamlc
- 8% cast and null checks
- 3.5% bound checks
- 10% total

bonsai
- 20% total
## File size

### ocamlc

<table>
<thead>
<tr>
<th>Format</th>
<th>JavaScript</th>
<th>WebAssembly</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 937 055</td>
<td>2 441 862 (+26%)</td>
</tr>
<tr>
<td>uncompressed</td>
<td>466 632</td>
<td>516 703 (+10%)</td>
</tr>
<tr>
<td>bzip2</td>
<td>989 089</td>
<td>1 251 620 (+25%)</td>
</tr>
</tbody>
</table>

### Be Sport Web app

<table>
<thead>
<tr>
<th>Format</th>
<th>JavaScript</th>
<th>WebAssembly</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 827 108</td>
<td>6 846 836 (+80%)</td>
</tr>
<tr>
<td>uncompressed</td>
<td>989 089</td>
<td>1 251 620 (+25%)</td>
</tr>
<tr>
<td>bzip2</td>
<td>989 089</td>
<td>1 251 620 (+25%)</td>
</tr>
</tbody>
</table>
## Effects: CPS impact on size

<table>
<thead>
<tr>
<th></th>
<th>Javascript</th>
<th>Wasm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Direct</td>
<td>CPS</td>
</tr>
<tr>
<td><strong>ocamlc</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uncompressed</td>
<td>1936871</td>
<td>2303918 (+19%)</td>
</tr>
<tr>
<td>bzip2</td>
<td>466637</td>
<td>472691 (+1.3%)</td>
</tr>
<tr>
<td><strong>bonsai</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uncompressed</td>
<td>1196757</td>
<td>1425899 (+19%)</td>
</tr>
<tr>
<td>bzip2</td>
<td>356138</td>
<td>363080 (+1.9%)</td>
</tr>
</tbody>
</table>

Explicit closure allocation vs rather regular transformation
## Effects: CPS performance

<table>
<thead>
<tr>
<th></th>
<th>Javascript</th>
<th></th>
<th>Wasm</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Direct</td>
<td>CPS</td>
<td>Direct</td>
<td>CPS</td>
</tr>
<tr>
<td><strong>Camlboy</strong></td>
<td>1300fps</td>
<td>750fps (-42%)</td>
<td>1750fps</td>
<td>1480fps (-15%)</td>
</tr>
<tr>
<td><strong>bonsai</strong></td>
<td>1.76s</td>
<td>12.4s (x7)</td>
<td>0.95s</td>
<td>1.63s (+70%)</td>
</tr>
</tbody>
</table>

Less overhead in Wasm
Effect benchmarks

<table>
<thead>
<tr>
<th></th>
<th>JavaScript (CPS)</th>
<th>Wasm (CPS)</th>
<th>Wasm (JSPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chameneos</td>
<td>2.6s</td>
<td>1.15s</td>
<td>6.6s</td>
</tr>
<tr>
<td>Generator</td>
<td>16s</td>
<td>6.8s</td>
<td>80s</td>
</tr>
</tbody>
</table>

JS Promise Integration API

- Not well optimized yet
- Lot of overhead going through JavaScript event loop
Rough edges
Efficient conversion between JS and OCaml strings

- Ocaml strings are array of bytes (UTF-8)
- Initial implementation based on the stringref proposal
- Now going through the Wasm linear memory
  - Copy to a shared buffer on one side
  - Read from the buffer on the other side
  - Conversions from/to UTF-8 on the JavaScript side
- JS String Builtins: does not provide the right functions yet
String conversion through a buffer

Fixed 64 kB buffer (linear memory)

Conversion to JavaScript

```javascript
const decoder = new TextDecoder('utf-8', {ignoreBOM: 1});
decoder.decode(new Uint8Array(buffer, 0, len), {stream})
```

Conversion to WebAssembly

```javascript
const encoder = new TextEncoder;
var out_buffer = new Uint8Array(buffer,0,buffer.length)
{read,written} = encoder.encodeInto(s.slice(start), out_buffer);
```
Efficient manipulation of typed arrays and array buffers

Use cases

- Camlboy: writing to a framebuffer
- I/O buffers
- WebGL

At the moment, one JavaScript call per access
Concluding
Implementation status

- Full language supported
- Large part of the runtime support implemented
- Adapted libraries (brr, gen_js_api, zarith, …) and build system (dune)

Future work

- Documentation / release
- Separate compilation / dynamic linking
- Performance optimizations: try to avoid some casts, unnecessary boxing, …
- Make it easier to debug generated code (sourcemap, keep variable names)
Conclusion


Wasm GC

- Very well designed
- Very encouraging performances
- Available now in Chrome / Firefox