Wasm_of_ocaml

Jérôme Vouillon

Tarides

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

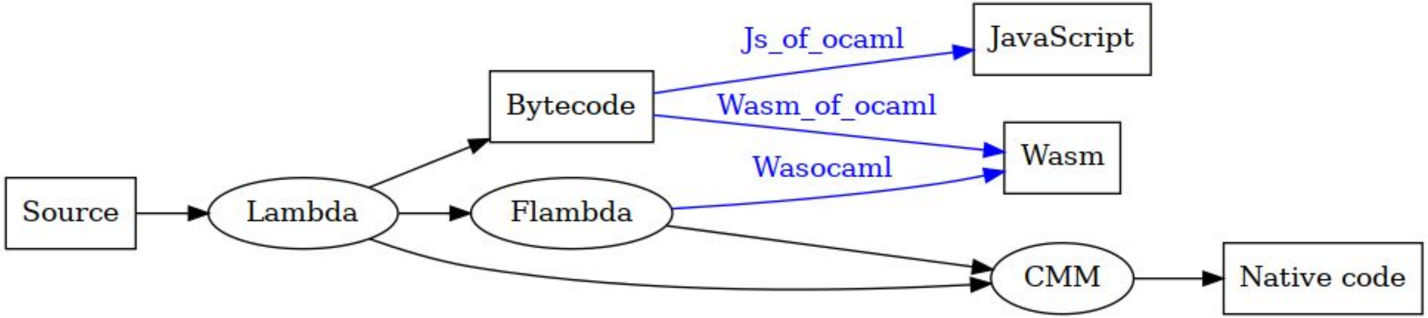
Retarget Js_of_ocaml to generate WebAssembly code

Goal: minimize user changes

Wasm GC makes it much easier to interoperate with JavaScript

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

Demos

Implementation

Compilation process

Existing 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: 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** when the function is known
- **call_ref** when the function arity is known
- 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_3
    (func (param (ref eq) (ref eq) (ref eq) (ref eq)) (result (ref eq))))

(type $closure_3
    (sub $closure (struct (field (ref $function_1)) (field (ref $function_3)))))

(type $env_3_2
    (sub final $closure_3
        (struct (field (ref $function_1)) (field (ref $function_3))
                (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 $closure))
  (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. After some reading and experiments with v8, it seems that this would be sufficient.”

- Partial CPS transformation

  Inherited from Js_of_ocaml

  Tail calls!

Interfacing with JavaScript

Js_of_ocaml

- 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
- OCaml integers and floats all mapped to JavaScript numbers

Wasm_of_ocaml

- JavaScript objects are boxed
- JavaScript integers automatically mapped to OCaml integers (ref i31)
- Primitives provided as WebAssembly functions

Eventually, should generate JavaScript code:

- Avoid string conversions for constant strings, property and method names
- More efficient code for property access / method call
- Avoid unnecessary boxing/unboxing

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))))))
```

Example: function calls

OCaml (Js_of_ocaml library)

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

Wasm

```
(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))))))

(import "bindings" "fun_call" (func $fun_call (param anyref) (param anyref) (result anyref)))
```

JavaScript

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

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

- `(import "bindings" "identity" (func $parse_float (param anyref) (result f64)))`
- Use methods toFixed / toExponential

Using maps and weak pointers

Weak arrays and ephemerons

- Weak, WeakMap

Marshalling

- Map object, to deal with sharing

Big integers

Use binaryen’s wasm-metadce + Js_of_ocaml linker

WebAssembly

```
(import "js" "wasm_z_add" (func $add (param (ref any)) (param (ref any)) (result (ref any))))
```

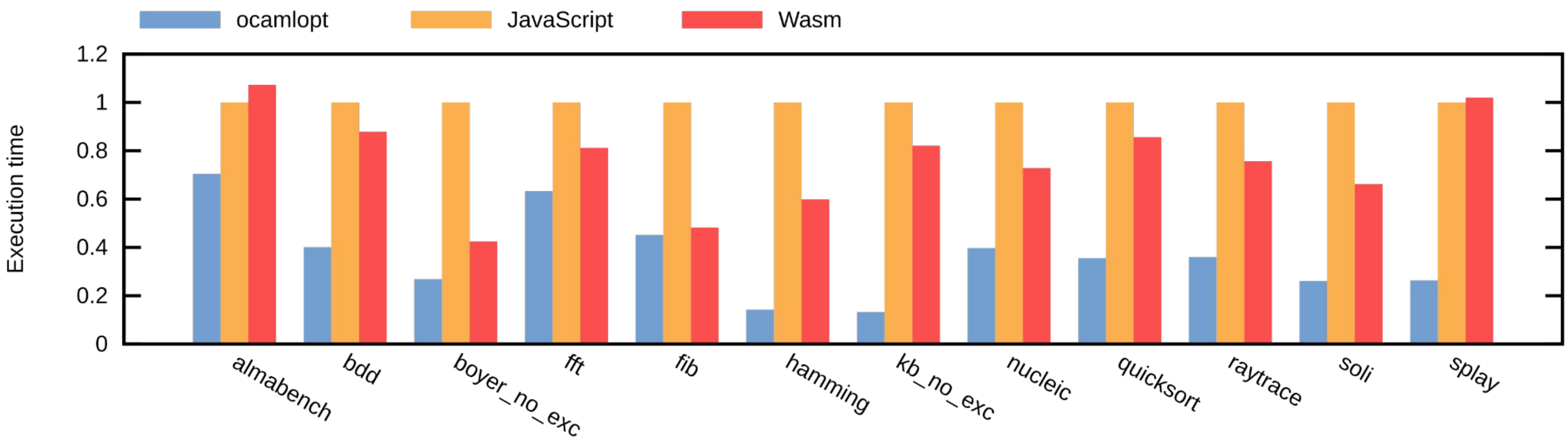
```
(func (export "ml_z_add")
   (param $z1 (ref eq)) (param $z2 (ref eq)) (result (ref eq))
   (return_call $wrap_bigint
      (call $add (call $unwrap_bigint (local.get $z1)) (call $unwrap_bigint (local.get $z2)))))
```

JavaScript

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

Performance results

Microbenchmarks



- ~30% faster than JavaScript
- Twice slower than native code

Larger benchmarks

ocamlc

About 30% faster than JavaScript
But about as fast as bytecode interpreter, 7 times slower than native code

22,71% v8::internal::wasm::WasmCompilationUnit::ExecuteFunctionCompilation
18,17% v8::internal::Runtime_UnwindAndFindExceptionHandler
17,92% v8::internal::Runtime_WasmThrow

CAMLBOY

Headless benchmarking mode: from 1180 fps to 1850 fps (30% faster)

The framebuffer (typed array) is the bottleneck

Bonsai

Library for building interactive browser-based UI

Table benchmark: 100 small benchmarks

Arithmetic mean:

- Javascript: 1.98ms
- Wasm (stringref): 1.66ms (~16% faster)
- Wasm (buffer): 2.24ms

But should convert constant strings only once

Cost of casts/bound checks

V8 makes it possible to skip checks

ocamlc

- 5% cast/null checks
- 2% bound checks

File size

ocamlc

	JavaScript	WebAssembly
uncompressed	1937055	2441862 (+26%)
bzip2	466632	516703 (+10%)

Be Sport Web app

	JavaScript	WebAssembly
uncompressed	3827108	6846836 (+80%)
bzip2	989089	1251620 (+25%)

Wish list

Type imports

- Do not duplicate some abstract types
- Check coherence

Also, binaryen’s wasm-merge should check types

Example: 64-bit integers

```
(type $string (array (mut i8)))
(type $compare (func (param (ref eq)) (param (ref eq)) (param i32) (result i32)))
(type $hash  (func (param (ref eq)) (result i32)))
(type $fixed_length (struct (field $bsize_32 i32) (field $bsize_64 i32)))
(type $serialize  (func (param (ref eq)) (param (ref eq)) (result i32) (result i32)))
(type $deserialize (func (param (ref eq)) (result (ref eq)) (result i32)))
(type $custom_operations
   (struct
         (field $id (ref $string))
         (field $compare (ref null $compare))
         (field $compare_ext (ref null $compare))
         (field $hash (ref null $hash))
         (field $fixed_length (ref null $fixed_length))
         (field $serialize (ref null $serialize))
         (field $deserialize (ref null $deserialize))))
(type $custom (sub (struct (field (ref $custom_operations)))))
(type $int64 (sub final $custom (struct (field (ref $custom_operations)) (field i64))))
```

Efficient conversion between JS and OCaml strings

- Ocaml strings are array of bytes (UTF-8)
- Js_of_ocaml: insert U+FFFD on error (following best practices)
- Initial implementation based on the stringref proposal
- Now going through the Wasm linear memory
- JS String Builtins: does not provide the right functions yet

String conversions using stringref

```
(type $string (array (mut i8)))

(func (export "jsstring_of_string") (param $s (ref $string)) (result anyref)
  (string.new_lossy_utf8_array (local.get $s)
    (i32.const 0) (array.len (local.get $s))))

(func (export "string_of_jsstring") (param $s (ref string)) (result (ref $string))
  (local $s' (ref $string))
  (local.set $s'
    (array.new $string (i32.const 0) (string.measure_wtf8 (local.get $s))))
  (drop (string.encode_lossy_utf8_array (local.get $s) (local.get $s') (i32.const 0)))
  (local.get $s'))
```

String conversion through a buffer

Fixed 64 kB buffer (linear memory)

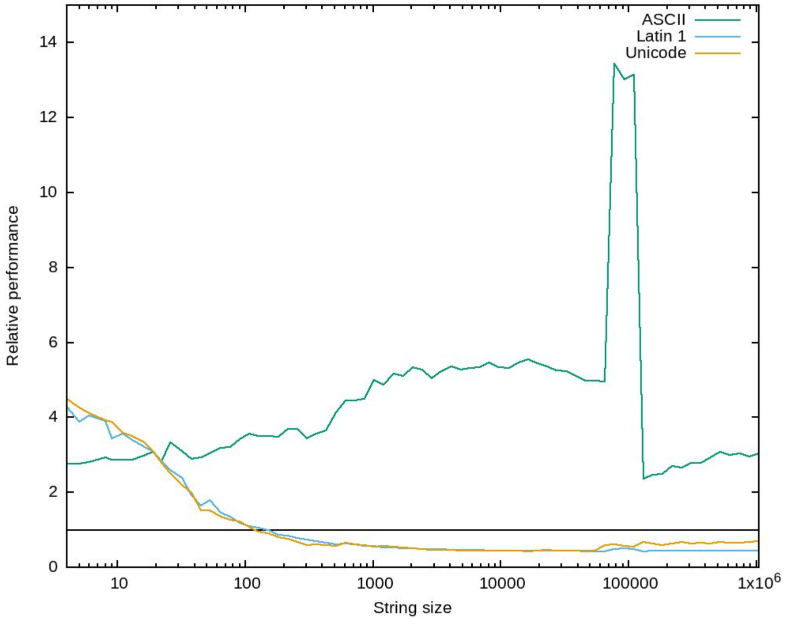
Conversion to JavaScript

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

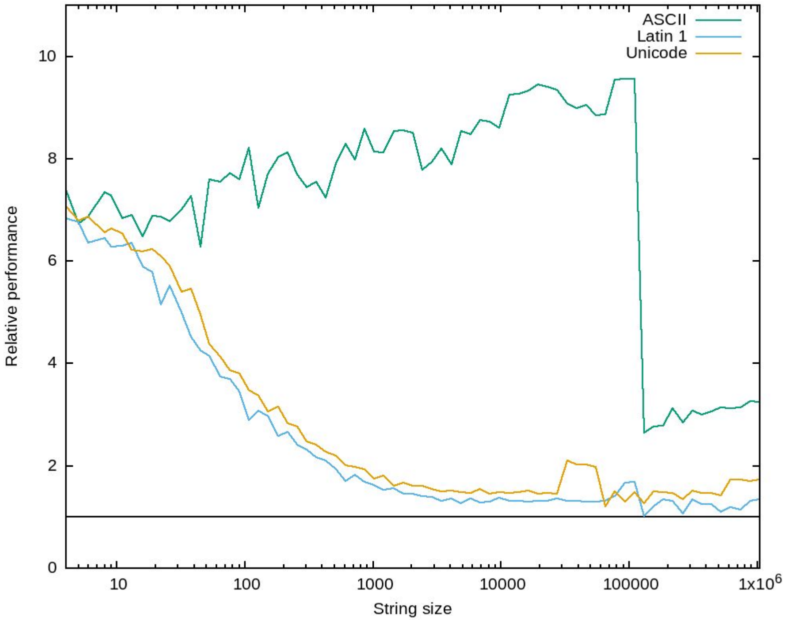
Conversion to WebAssembly

```
const encoder = new TextEncoder;
var out_buffer = new Uint8Array(buffer,0,buffer.length)
{read,written} = encoder.encodeInto(s.slice(start), out_buffer);
```

String conversion performance



node



Chrome

Efficient manipulation of typed arrays and array buffers

Use cases

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

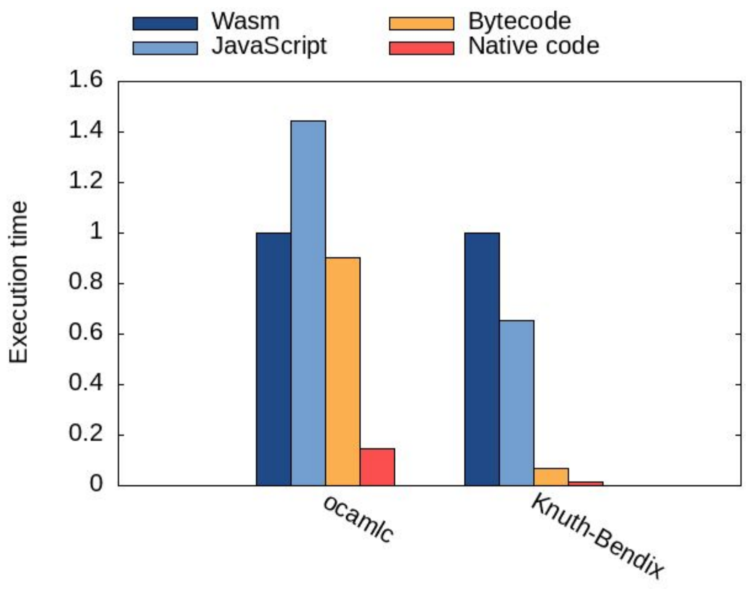
Alternatives

- One JavaScript call per access
- Reserve space on the linear memory
    - Allocation has to happen on the Wasm side
    - Endianness mismatch?

Faster exceptions

Zero-cost exceptions are slow…

Room for improvements?



Others

Tagged arrays

Store a 8-bit tag in the array's header

Non-trapping array access?

```
(br_if $bound_error (i32.ge_u (local.get $i) (i32.sub (array.len $a) (i32.const 1))
(local.set $v (array.get $block (local.get $a) (i32.add (local.get $i) (i32.const 1)))
```

Non-trapping division

```
(br_if $zero_divide (i32.eqz (local.get $y))
(local.set $z (i32.div_s (local.get $x) (local.get $y))
```

Concluding

Implementation status

- Full language supported
- Large part of the runtime support implemented
- Adapted libraries and build system (dune)

Future work

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

Conclusion

Wasm_of_ocaml source code: https://github.com/ocaml-wasm/wasm_of_ocaml

Wasm GC

- Very well designed
- Very encouraging performances