Wasocaml: compiling OCaml to WebAssembly

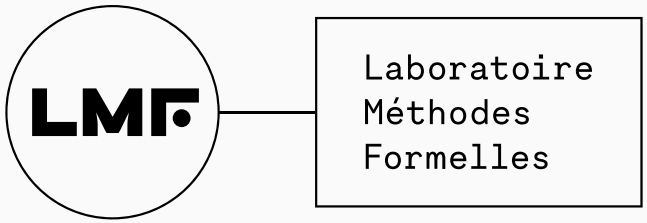
Léo Andrès <l@ndrs.fr>[1, 2]

Pierre Chambart <pierre.chambart@ocamlpro.com>[1]

Jean-Christophe Filliâtre <jean-christophe.filliatre@cnrs.fr>[2]

August 2023 – IFL'23 – Braga

1. OCamlPro
2. Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles



JavaScript:

- bad/unpredictable performances
- unsafe

WebAssembly (Wasm) is safe and has good/predictable performances, used:

- on the Web: V8, SpiderMonkey
- on the Cloud: Fastly, CloudFlare
- as a portable binary format
- to interface with C from other languages

Wasm1:

- compact binary format (Wasm) and text format (Wat)
- functions
- stack (can not be inspected)
- static verification and typecheck (few dynamic tests)
- one `memory` per module
- only scalar types: `i32`, `i64`, `f32`, `f64`
- only exported items can be used by other modules
- can be seen as a simplified C

Wat with S-expressions:

```
(func $fact (param $x i32) (result i32)
  (if (i32.eq (local.get $x) (i32.const 0))
    (then (i32.const 1))
  (else
    (i32.mul
      (local.get $x)
      (call $fact
        (i32.sub
          (local.get $x)
          (i32.const 1)))))))
```

Wat with ASM syntax:

```
(func $fact (param $x i32) (result i32)
  i32.const 0
  local.get $x
  i32.eq
  (if
    (then i32.const 1)
    (else
      local.get $x
      i32.const 1
      i32.sub
      call $fact
      local.get $x
      i32.mul)))
```

Compiling runtime-free languages such as C/C++/Rust to Wasm is straightforward.

Some primitives such as `malloc` need to be rewritten in Wasm and provided by the compiler.

How does a GC work?

- *Tracing*: we start from roots and we find live objects recursively (OCaml, Java)
- *Reference counting*: we start from dying objects and we kill objects recursively (Python[1])

[1] plus a tracing one for cycles...

We need to discriminate pointers:

- *Conservative (Boehm)*: we make a guess and accept memory leaks (Crystal, Guile, Inkscape).
- *Precise*: we have the right information (almost everybody).

Need the information somewhere at runtime: similar to polymorphism

Compiling GC languages to Wasm1 is more involved:

- runtime must be rewritten, or compiled from C to Wasm: difficult because of Wasm safety properties
- GC need to inspect the stack to find roots, not possible in Wasm, requires a shallow stack
- interactions with the GC of the embedder are difficult (cycles can't be collected)

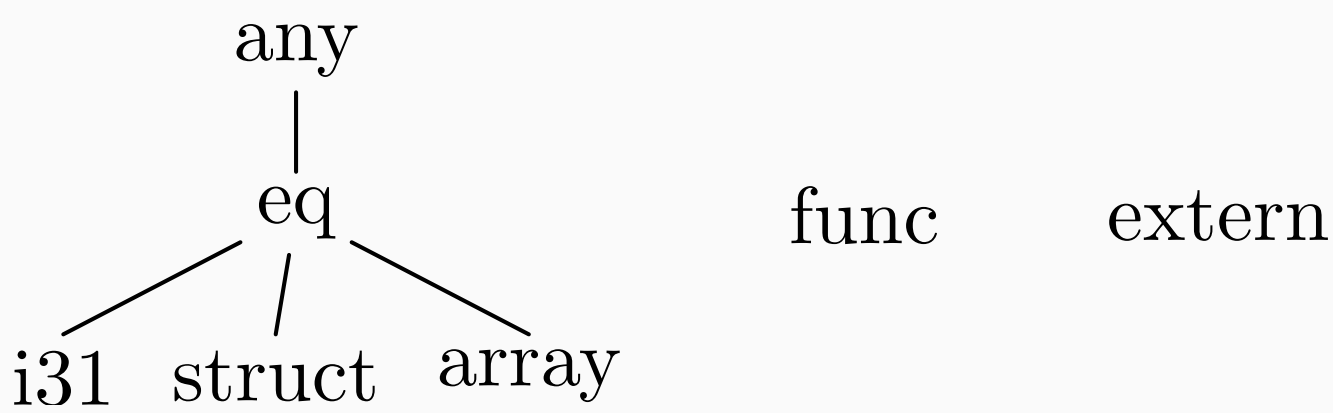
Need for a proper GC in Wasm.

Requirements:

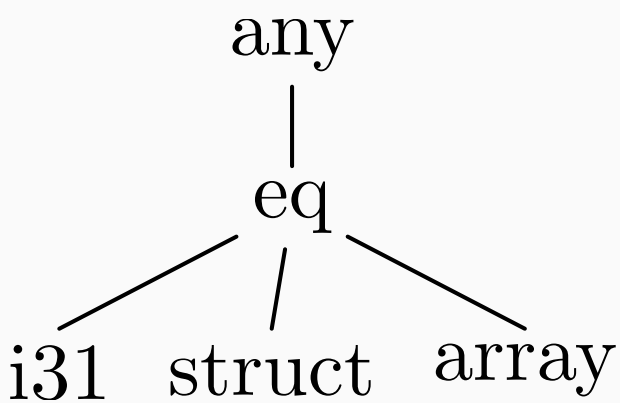
- safe and fast
- do not make Wasm1 code slower
- can represent values from any language

A type system to represent values from any kind of source language is too complex.

Instead, WasmGC introduces reference types and a subtyping hierarchy:



The hierarchy tells which casts are allowed.



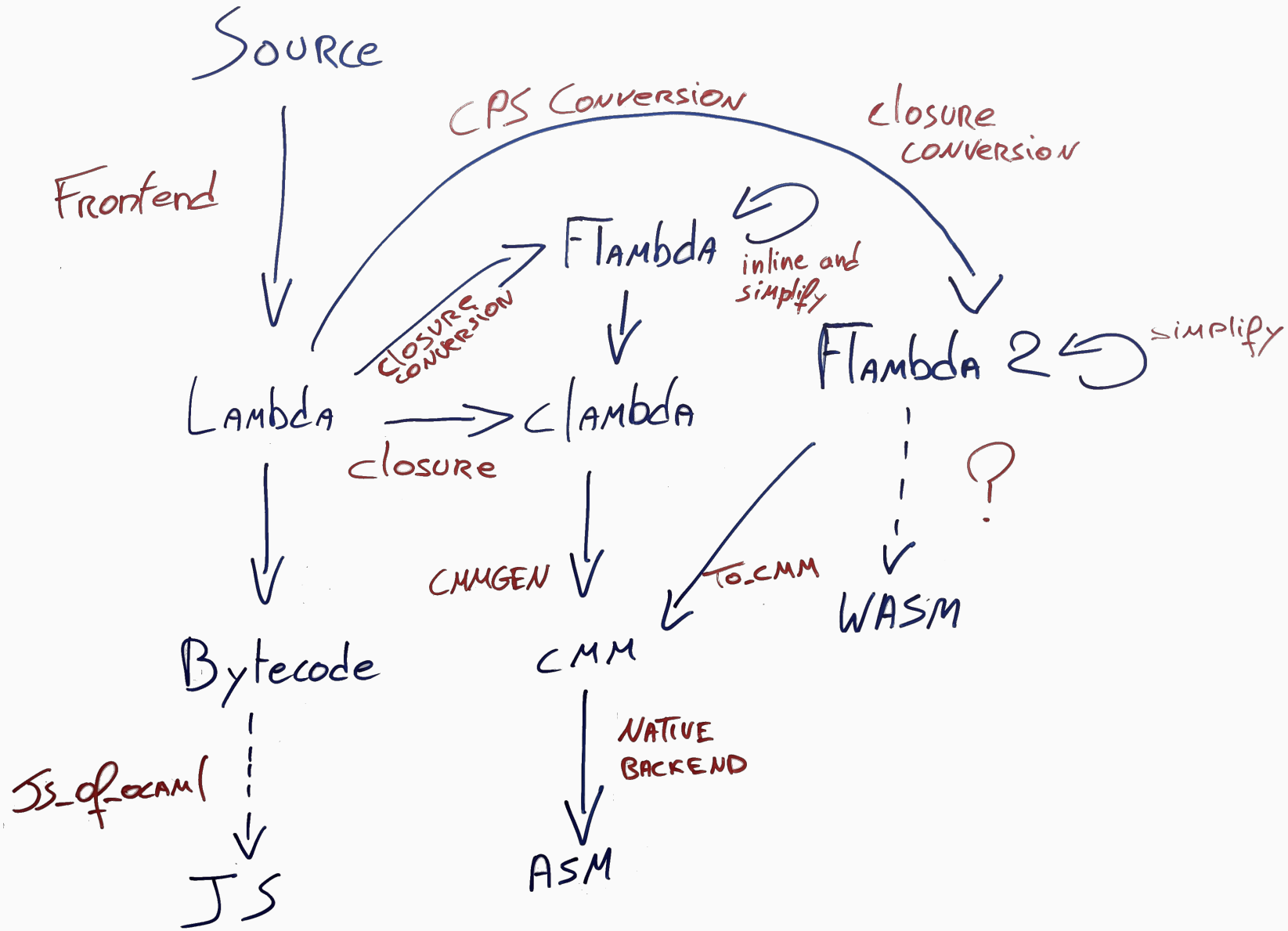
func extern

Upcasts are implicit.

Downcasts are explicit and lead to runtime errors if incorrect.

Casts are cheap.

Possible to dynamically test for compatibility.


Source
Frontend
CPS Conversion
closure conversion
Flambda
inline and simplify
closure conversion
Flambda 2
simplify
Lambda
Clambda
closure
?
Cmmgen
To_cmm
WASM
Bytecode
Cmm
Native backend
js_of_ocaml
JS
ASM

Lambda	closures are still implicit, not optimised
Bytecode	not enough optimised
Clambda	code pointers and values mixed in closures
Cmm	even more low-level (pointer arithmetic)
Flambda	our choice for the first prototype
Flambda2	our choice for the future

Flambda:

- ANF
- explicit closures
- high-level: works on abstract values and not directly on the actual memory layout (this is done by `Cmm`)

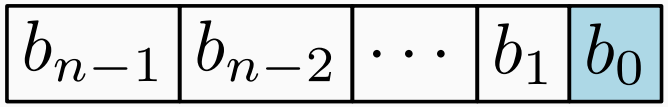
```
let rec iter f l =
  match l with
  | [] -> ()
  | hd :: tl ->
    f hd;
    iter f tl

let () =
  let iter_print = iter print_int in
  iter_print [2; 1]
```

```
let iter =
  let set = make_closures
    | cl_iter { f } env ->
      let otherset = make_closures
        | cl_iter_f { l } envtwo ->
          switch l
          with int | 0 -> const 0
          with tag | 0 ->
            let x = get_field 0 l
            let f = project_var f from envtwo
            let dummy = f x
            let tl = get_field 1 l
            envtwo tl
        with vars | f -> f end
      project_closure cl_iter_f from otherset
  project_closure cl_iter from set

let lempty = const 0
let one = const 1
let lone = make_block 0 one lempty
let two = const 2
let ltwo = make_block 0 two lone
let iter_print = iter print_int
iter_print ltwo
```

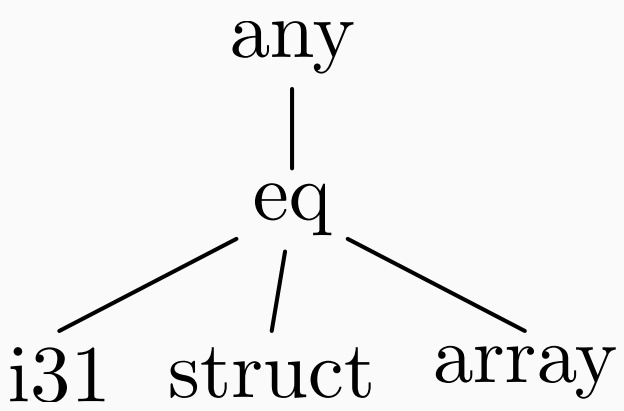
uniform representation using a tagged single machine word:



if $b_0 = 0$, the value is a pointer to a heap-allocated block:

if $b_1 = 1$, the $n - 1$ most significant bits are a small scalar:

small scalars: `bool`, `char`, `int`, constant constructors of ADTs



func extern

Uniform representation through `eqref` (== operator).

Can't be more precise because of `Obj.magic`, GADTs and types that have scalar/blocks values.

Small scalars are `i31ref`.

OCaml arrays are `array`.

Others heap-allocated blocks are `struct` or `array`.

In `get_field n x` the type of `x` is unknown.

Could propagate more types but breaks some optimisations.

Not enough because of `Obj.field`.

All we know is that `x` is a block of size $n + 1$ at least.

Blocks as structs:

```
(type $block1 (struct
  (field $tag i8)
  (field $field0 eqref)))

(type $block2 (sub $block1) (struct
  (field $tag i8)
  (field $field0 eqref)
  (field $field1 eqref)))

;; and so on...
```

We cast x to $\texttt{\$block}(n+1)$.

swrup/ocaml-emoji:

```
let fox = "🦊"
let framed_picture = "🖼"
let free_button = "🆓"
let french_fries = "🍟"
let fried_shrimp = "🍤"
let frog = "🐸"
let front_facing_baby_chick = "🐥"
```

swrup/ocaml-emoji:

```
let fox = "🦊"
let framed_picture = "🖼"
let free_button = "🆓"
let french_fries = "🍟"
let fried_shrimp = "🍤"
let frog = "🐸"
let front_facing_baby_chick = "🐥"
```

Modules represented as blocks: each toplevel value is a field.

swrup/ocaml-emoji:

```
let fox = "🦊"
let framed_picture = "🖼"
let free_button = "🆓"
let french_fries = "🍟"
let fried_shrimp = "🍤"
let frog = "🐸"
let front_facing_baby_chick = "🐥"
```

Modules represented as blocks: each toplevel value is a field.

too long subtyping chain 😵

swrup/ocaml-emoji:

```
let fox = "🦊"
let framed_picture = "🖼"
let free_button = "🆓"
let french_fries = "🍟"
let fried_shrimp = "🍤"
let frog = "🐸"
let front_facing_baby_chick = "🐥"
```

Modules represented as blocks: each toplevel value is a field.

too long subtyping chain 🤯

We have two variants to avoid this problem while still using `struct`, not implemented yet.

Blocks as arrays:

```
(type $block (array eqref))
```

An array of `eqref` with the tag stored at position 0.

Tradeoffs:

- implicit bounds check at each access
- cast to read the tag
- probably cheaper than subtyping test
- can't use more precise types if they were propagated

Closures:

```
;; a closure with two captured variables
(type $closure1 (struct
  (field funcref)
  (field $v1 eqref)
  (field $v2 eqref)))
```

Actual representation is more complex to handle mutually recursive functions and to reduce casts.
Only place where we use Wasm recursive types.
To handle currification, functions of arity one need to be supertypes of all others closures.

Exceptions:

- use the exception handling proposal
- maps quite directly
- don't have runtime generated exceptions
- a single Wasm exception, handle identifiers on the side

A proper small-step semantics for a subset of Flambda where functions all have arity one, objects primitives are removed and where exceptions are only static.

A proper small-step semantics for a subset of Flambda where functions all have arity one, objects primitives are removed and where exceptions are only static.

The first Flambda1 semantics.

A proper small-step semantics for a subset of Flambda where functions all have arity one, objects primitives are removed and where exceptions are only static.

The first Flambda1 semantics.

A formalized compilation scheme from Flambda1 to WasmGC.

A proper small-step semantics for a subset of Flambda where functions all have arity one, objects primitives are removed and where exceptions are only static.

The first Flambda1 semantics.

A formalized compilation scheme from Flambda1 to WasmGC.

Proof of correctness is not done yet (would like to use WasmGC semantics which is still ongoing work).

A proper small-step semantics for a subset of Flambda where functions all have arity one, objects primitives are removed and where exceptions are only static.

The first Flambda1 semantics.

A formalized compilation scheme from Flambda1 to WasmGC.

Proof of correctness is not done yet (would like to use WasmGC semantics which is still ongoing work).

The parser, interpreter and compiler targeting WasmGC for mini-Flambda1 fit in 1300 lines of OCaml.

The real compiler is called Wasocaml and is available at github.com/ocamlpro/wasocaml.

The real compiler is called Wasocaml and is available at github.com/ocamlpro/wasocaml.

Meant as a way to demonstrate the usefulness of `i31ref` and convinced the WasmGC working group (along with the Guile implementation that came a few months later).

The real compiler is called Wasocaml and is available at github.com/ocamlpro/wasocaml.

Meant as a way to demonstrate the usefulness of `i31ref` and convinced the WasmGC working group (along with the Guile implementation that came a few months later).

Only a fraction of the stdlib externals are provided and the object fragments of the language has not yet been implemented.

The real compiler is called Wasocaml and is available at github.com/ocamlpro/wasocaml.

Meant as a way to demonstrate the usefulness of `i31ref` and convinced the WasmGC working group (along with the Guile implementation that came a few months later).

Only a fraction of the stdlib externals are provided and the object fragments of the language has not yet been implemented.

The first compiler for a real-world functional language targeting WasmGC.

Benchmarks using the experimental branch of V8:

Benchmarks using the experimental branch of V8:

No real sized programs for now.

Benchmarks using the experimental branch of V8:

No real sized programs for now.

Classical functional microbenchmarks are two times slower than native OCaml..

Benchmarks using the experimental branch of V8:

No real sized programs for now.

Classical functional microbenchmarks are two times slower than native OCaml..

Knuth-Bendix: exceptions are slow (100 times slower than native for a raise) and we need to discuss this with the V8 team (in SpiderMonkey they’re fast but other extensions are missing).

Since yesterday:

Since yesterday:

- do not use Wasm exceptions

Since yesterday:

- do not use Wasm exceptions
- return a pair with a boolean set to true when an exception was raised

Since yesterday:

- do not use Wasm exceptions
- return a pair with a boolean set to true when an exception was raised
- a hundred times faster than Wasm exceptions

Since yesterday:

- do not use Wasm exceptions
- return a pair with a boolean set to true when an exception was raised
- a hundred times faster than Wasm exceptions
- KB: 2.7 times slower than native

With casts as no-ops we have a 10% gain.

With optimisations and Flambda2 it should be much better.

Jsoo is slower in an unpredictable fashion (up to 40 times)

In the JS FFI all calls go through `Js.Unsafe.meth_call` of type `'a -> string -> any array -> 'b`

We can provide:

```
(func $meth_call
  (param $obj externref)
  (param $method stringref)
  (param $args $anyarray)
  (result externref))
```

With recent additions to Clang, it would be possible to re-use existing bindings and to compile the C code with emscriptem with almost no changes to the bindings.

We only need to provide alternative FFI headers files, replacing usual macros by hand-written Wasm functions.

The only limitation we forsee is that the `Field` macro won't be usable as an l-value anymore. We would need a new `Set_field` macro instead.

```
Field(v, n) = ...; // not anymore
Set_field(b, v, n); // OK
```

OCaml 5.0 is multicore. We're based on 4.14 so effects handlers are not supported.

We could use:

- CPS transformation
- stack-switching proposal
- JS Promise Integration

Contributions:

- OCaml Wasm backend github.com/ocamlpro/wasocaml
- the first compiler for a functional language to WasmGC
- impact on the GC proposal for Wasm
- the first Flambda1 semantics
- a formalized compilation scheme
- compilation strategies usable by others compilers
- ongoing: proof of the compilation
- ongoing: symbolic execution of WasmGC program

Thanks!