wasocaml: compiling OCaml to Wasm

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

Léo Andrès `<l@ndrs.fr>`[1, 2]

January 10, 2023

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

OCamlPro

- we're programming languages consultants
- specialized in OCaml, Rust and formal methods
- we develop flambda/flambda2 (ocaml optimizer IR)

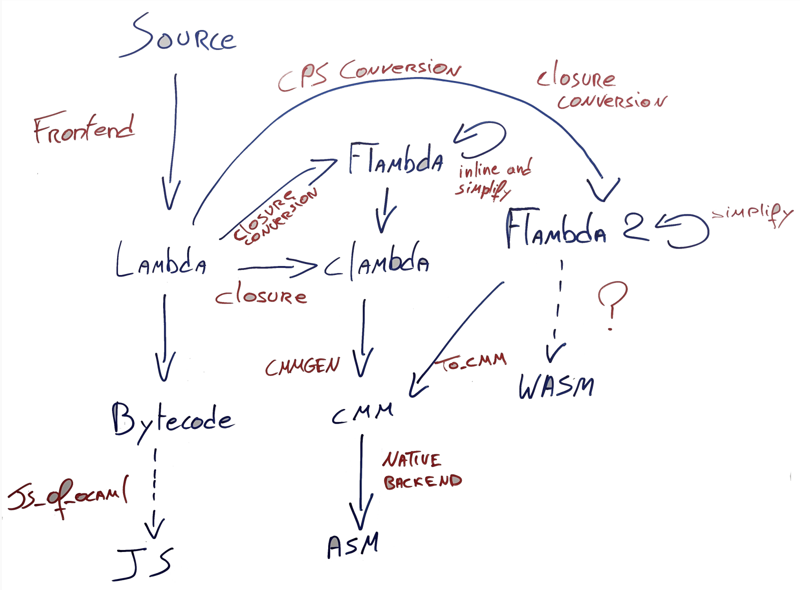
OCaml and Wasm

- `js_of_ocaml`: OCaml bytecode to JavaScript
- `wasicaml`: OCaml bytecode to Wasm

OCaml

- functions can be: mutually recursive, nested, polymorphically recursive, first class, partially applied
- algebraic datatypes, GADTs, polymorphic variants, objects
- exceptions
- modules, functors, first class modules
- allow low-level manipulations through the `Obj` module
- GC with good performances (minor/major heaps)

OCaml compilation chain



IR choice

- we chose to go from `Flambda` to `Wasm` (we'll use `Flambda2` in the future)
- why? we don't want to rewrite the compiler (OCaml typechecking is hard)

Values representation

- uniform representation
- 1 bit to distinguish between scalars and pointers to heap-allocated blocks
- a given type can have values of both kinds:
    - `Obj.is_int (Obj.repr []) = true`
    - `Obj.is_block (Obj.repr [1; 2]) = true`

Flambda…

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

Block compilation

Two strategies:

- before: `struct { i16 size; i8 tag; refeq data }`
- didn't work because of too long subtyping chains
- now: `refeq array` (`tag` followed by `data`) (size is not needed)
- we kept both and can switch easily
- we could use more precise types with the `struct` strategy

...to Wasm

```
(type $Gen_block (struct (field (mut i8)) (field (mut i16))))
 (type $Block_0
   (struct_subtype (field (mut i8)) (field (mut i16))
   ↪   $Gen_block))
 (type $Block_1
   (struct_subtype (field (mut i8)) (field (mut i16)) (field
   ↪   (mut (ref eq)))
     $Gen_block))
 (type $Block_2
   (struct_subtype (field (mut i8)) (field (mut i16)) (field
   ↪   (mut (ref eq)))
     (field (mut (ref eq))) $Block_1))
```

…to Wasm

```
(type $Float (struct (field (mut f64))))
(type $Int32 (struct (field (mut i32))))
(type $Int64 (struct (field (mut i64))))
(type $String (array (mut i8)))
(type $Array (array (mut (ref eq))))
(type $FloatArray (array (mut f64)))
(rec
  (type $Func_1
    (func (param (ref eq)) (param (ref $Env)) (result (ref
    ↪   eq))))
  (type $Env (struct (field (mut i8)) (field (mut (ref
  ↪   $Func_1))))))
(type $Func_2
  (func (param (ref eq)) (param (ref eq)) (param (ref $Env))
    (result (ref eq))))
(type $Gen_block (array (mut (ref eq))))
```

…to Wasm

```
(type $Gen_closure_3
   (struct_subtype (field (mut i8)) (field (mut (ref $Func_1)))
     (field (mut (ref $Func_3))) $Env))
 (type $Closure_3_1
   (struct_subtype (field (mut i8)) (field (mut (ref $Func_1)))
     (field (mut (ref $Func_3))) (field (mut (ref eq))) $Gen_closure_3))
```

Why do we need i31

- two cases: `int` and *small scalars*
- GADTs are complicated:

```
type _ tag =
  | I : int tag
  | F : float tag
  | Box : (int * int) tag

let f : type t. t -> t tag -> int =
  fun v tag ->
    match tag with
    | I -> v
    | F -> int_of_float v
    | Box -> fst v + snd v
```

Difficulties

- FFI probably won't work with emscriptem (in `js_of_ocaml` one needs to write bindings by hand)
- partial application
- too long subtyping chain
- objects (possible but not done yet)
- exceptions: we can't use wasm exceptions as OCaml allows dynamic exception creation (through functors), so we only have one wasm exception and then we use our own identifiers for everything else

Things that went well

- GC proposal well designed
- i31 really helped
- people were very reactive (e.g. closed world assumption by binaryen)
- OCaml compilation is fast, so having one Wasm module per OCaml module makes separate compilation of the two match quite well (but we have to export a lot of globals)

Benchmarks

compiler	fib 36	fib 37	ocamlish test
ocaml native	0.14	0.21	1.83
ocaml bytecode	0.72	1.21	3.21
js_of_ocaml	0.29	0.40	4.98
wasicaml	0.55	0.85	3.65
ocamlrun wasm	4.09	6.50	8.85
wasocaml	0.31	0.42	3.54

- fib: `js_of_ocaml` is OK
- ocamlish test: `js_of_ocaml` is not good enough, but we are
- js_of_ocaml is sometimes up to $x40$ times slower than native in real examples

OCaml 5

- OCaml is now multicore
- `js_of_ocaml` is going to use CPS+trampoline
- it's too costly for us
- need for stack switching (effects handlers)

Future

- `let` will probably be useful
- we won't need wasm closures
- most of our instructions are casts, binaryen removes some, flambda2 will remove much more; for others we'll have to propagate types further in the compiler or generate better code
- we probably won't need rtt but they may lead to better performances (need to think more about it)
- experiment with whole-program linking (binaryen-merge would be useful), we won't use `ld` from llvm as we don't have `.o` files