Compiling FFI-heavy OCaml to WebAssembly

An Experience Report on MOPSA

Reda Boudrouss

OCaml Workshop 2026 · August 24, Paris

Sorbonne University

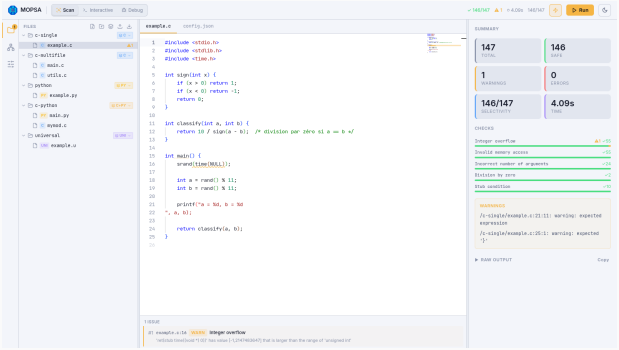
The project

MOPSA: a static analyzer by abstract interpretation, for C & Python

- written *mostly* in OCaml...
- ...on GMP, MPFR, Zarith, Apron, **LLVM/Clang**. All bound through the **FFI**

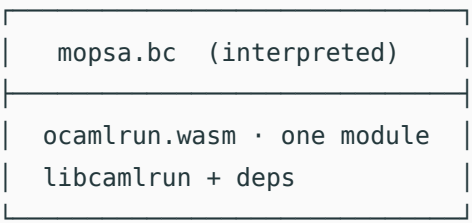
Goal: run it entirely in the browser. No server.

It works today:
https://mopsawasm.rboud.com/



Our solution

1. compile MOPSA to **bytecode**
2. compile the **runtime** + every native dep to wasm (Emscripten)
3. **link statically** into one module
4. **interpret** `mopsa.bc` on top



One toolchain (`emcc`), one memory model, no per-binding glue.

Why not x ?

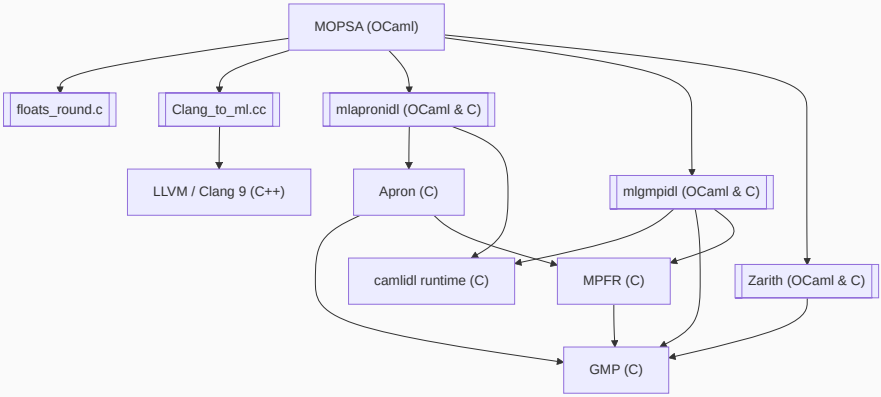
We considered using `js_of_ocaml`, `wasm_of_ocaml`, `wasocaml` at first but…
they compile the OCaml and **leave the C/C++ behind**.

- fine for pure OCaml, or with a JS reimplementation (`zarith_stubs_js`)
- here: rewrite all the parts that uses the FFI ?

A structural problem: **what do the FFI stubs link against?** `caml_alloc`, `caml_callback`, … exist in no translated module.

And exposing them wouldn't help: These backends move OCaml values *out of linear memory*, so a C stub reading a tag by pointer cannot reach them at all.

The dependency stack



Every box ships C or C++ · double-bordered boxes use the FFI to build or read OCaml values

The FFI part

`Clang_to_ml.cc` 5000 lines, both worlds in one translation unit:

```
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"

#include <caml/mlvalues.h>
#include <caml/alloc.h>
#include <caml/memory.h>
```

Each Clang AST node triggers an allocation *inside the OCaml GC heap, from C++*.

Also CamlIDL generates ~655 more stubs for Apron & GMP.

Every call site bakes in what an OCaml value is, byte for byte, in memory.

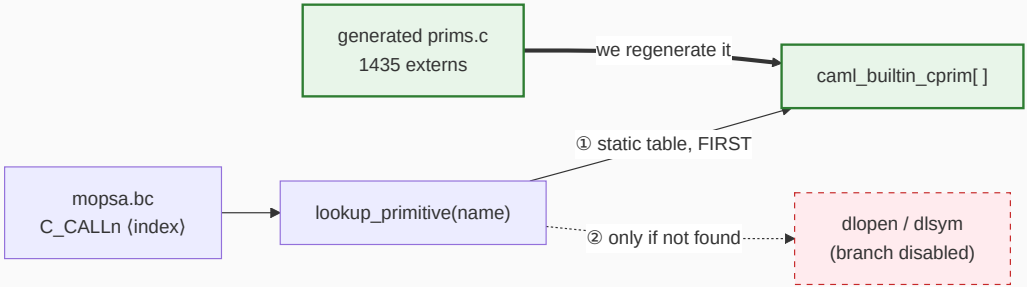
The build montage

library	version	manual trick
OCaml runtime	4.14.2	sak is a *host* tool, rebuild with real cc
GMP	6.1.2	n/a
MPFR	4.2.2	n/a
CamlIDL runtime	latest	3 files, compiles as-is
Apron	lastest	n/a
LLVM/Clang	9	Crosscompilation & adapting the flags
`Clang_to_ml.cc`	n/a	n/a (appart from linking the needed headers)
Camlidl stubs	n/a	Generated C files & compiled them

OCaml 4.14, not 5: the target is wasm32, and OCaml 5 dropped 32-bit.
Several dependencies compiled with **no patch at all**.

Binding 1435 externals without dlopen

Bytecode never calls C directly: `C_CALLn <index>` is resolved *by name* at startup, but the interpreter checks its own table **before** `dlsym`:



Scan every C/C++ source for stubs, emit a **superset** table (runtime core, `unix` (131), `str`, bigarray, **655 camlidl** Apron/GMP stubs) so every primitive resolves **inside the module**, no `dlopen` ever.

The final link

```
emcc ... -o ocamlrun.js \
  --preload-file mopsa.bc@/build/mopsa.bc \
  libs/*.a \
  -s ERROR_ON_UNDEFINED_SYMBOLS=1 \
  prims.o libcamlrun.a
```

- **everything statically linked**: one self-contained 15 MB `.wasm`
- `ERROR_ON_UNDEFINED_SYMBOLS=1`: a missing primitive fails the *link*, loudly, not the browser at runtime
- `mopsa.bc` is *preloaded*, not linked: it's data, interpreted

The final link

```
emcc ... -o ocamlrun.js \
  --preload-file mopsa.bc@/build/mopsa.bc \
  libs/*.a \
  -s ERROR_ON_UNDEFINED_SYMBOLS=1 \
  prims.o libcamlrun.a
```

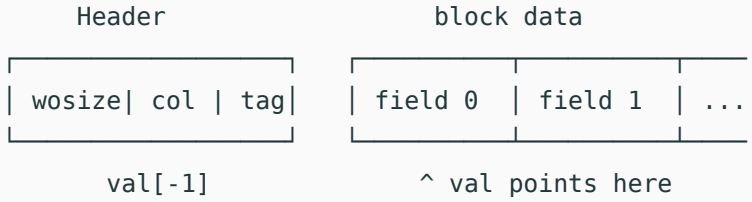
- **everything statically linked**: one self-contained 15 MB `.wasm`
- `ERROR_ON_UNDEFINED_SYMBOLS=1`: a missing primitive fails the *link*, loudly, not the browser at runtime
- `mopsa.bc` is *preloaded*, not linked: it's data, interpreted

Every symbol resolved. It links. It's self-contained.

and at runtime, in the browser

```
Uncaught RuntimeError:
  index out of bounds
```

Exhibit A: what an OCaml value is



The trap traces to reading a block's *tag* (OCaml 4.14, little-endian):

```
#define Tag_val(val) (((unsigned char *) (val)) [-sizeof(value)])
```

Easy hypotheses, eliminated:

- `sizeof(value)` miscompiled? No: a compile-time constant, always **4**
- a 64/32-bit mismatch? No: the ILP32 config is perfectly consistent

Exhibit A: `-sizeof(value)` is not -4

`sizeof` has type `size_t`, which is **unsigned**:

```
-sizeof(value) = -(size_t)4 = 0xFFFFFFFC          (not -4!)
```

Native 32-bit: `p + 0xFFFFFFFC` is a wrapping 32-bit add $\rightarrow$ exactly `p - 4`.
Upstream OCaml works everywhere, *by wrap-around*.

wasm32: two ways to add an offset:

- an explicit `i32.add` wraps modulo 2^{32}: fine
- the **static `offset=N`** immediate of a load is an *unsigned* u32, bounds-checked on the full untruncated value: **no wraparound**

LLVM may fold any *non-negative* constant into `offset=N`. `0xFFFFFFFC` qualifies $\rightarrow$ effective address $\approx$ 4 GiB $\rightarrow$ the bounds check traps.

Exhibit A: the only difference is signedness

```
;; ((unsigned char *) v)
;;      [-sizeof(value)]
(func $tag_old (param i32)
               (result i32)
  local.get 0
  i32.load8_u
    offset=4294967292)
```

folded into the load → traps

```
;; ((unsigned char *) v)
;;      [-(int)sizeof(value)]
(func $tag_fixed (param i32)
                 (result i32)
  local.get 0
  i32.const -4
  i32.add
  i32.load8_u)
```

wrapping add → `p - 4`, fine

The fix: cast to *signed* before negating, since a negative displacement can't be folded into `offset=N`. (re-verified with the project's `emcc` 4.0.22)

and at runtime, in the browser

```
unhandled type:
lvalue_ref(__builtin_va_list=void*)
```

Exhibit B: `va_list` changes shape

Porting to wasm32 also retargeted the *embedded* Clang to 32-bit: the sources MOPSA parses are now typed for a 32-bit ABI.

- **x86-64**: `va_list` is an array → decays to a pointer, handled
- **32-bit**: a scalar `void *`, passed *by reference* to `__builtin_va_start` → an `LValueReferenceType`

The fix :

```
| C.LValueReferenceType tq -> T_pointer (type_qual range tq), no_qual
```

and at runtime, in the browser

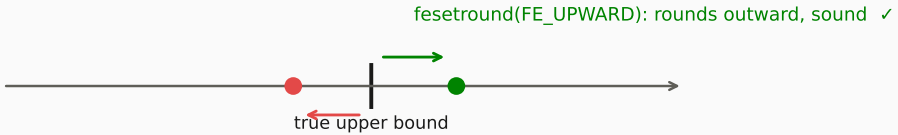
I works finaly !

But float interval analysis are weird ?

+ some FPU warnings

Exhibit C: nobody controls the rounding

Wasm has **no FPU rounding-mode control**: everything rounds to nearest,



- **Apron**: compile every domain with `NUM_MPQ`, so bounds are computed in exact GMP rationals and `fesetround` is never needed.*Residual*: floats reaching Apron's API.
- **MOPSA's own `floats_round.c`**: *no satisfying fix yet*; widening keeps the analysis sound but coarse.

How does the web app work ?

A fresh instance per analysis

The OCaml runtime keeps its state **global** and is not re-entrant, and `mopsa.bc` ends with `exit`. There is no clean way back.

- Existing possible solution are not compatible with Ocaml's exceptions that uses `setjmp/longjmp`.
- **a fresh instance for every analysis** (`MODULARIZE=1`)

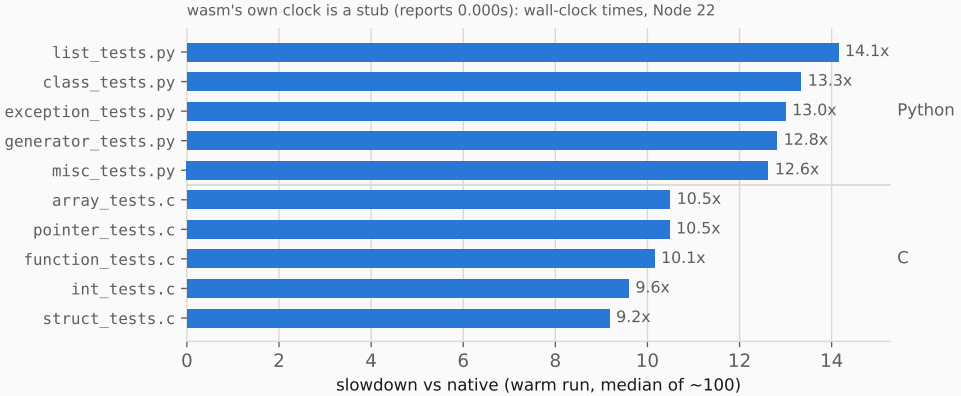
The web worker fetches + compiles the module **once**, then each analysis only *re-instantiates* the already-compiled module.

Full module setup (compile, instantiate, unpack the FS data) is **63 ms** under Node. (Browser: 641 ms for the one-shot first start, fetch included.)

Performance

How does our wasm perform ?

The cost of interpretation

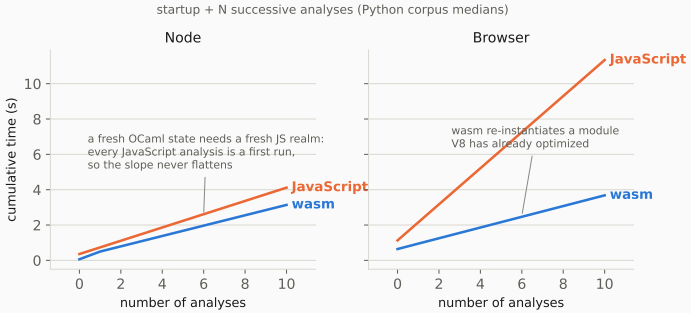


Interpreted bytecode on wasm vs native compiled code, V8 steady state. `int_tests.c` 40 ms → 384 ms

`struct_tests.c` 411 ms → 3.77 s

Comparaison with javascript

the baseline is **Try-MOPSA**, a scaled-down jsoo build: no C frontend, Apron replaced by VPL (pure OCaml).



startup 63 vs 359 ms (Node), 641 ms vs 1.13 s (browser)
repeated Python: wasm ~1.3× (Node) / ~3.4× (browser) ahead
optimistic for jsoo: its 22 MB bundle parse is charged only once

Conclusion

Conclusion

wasm32 is almost a 32-bit target

looked like	actually was	failure mode
bad index / codegen bug	unsigned negation folded into `offset=N`	loud trap
a Clang parser bug	32-bit ABI: `va_list` passed by reference	loud error
nothing at all	no rounding-mode control in wasm	silent unsoundness

The community question

None of this is specific to MOPSA, and most of it is mechanical:

step	
compile project to bytecode	mechanical
compile runtime + deps with emscripten	mostly mechanical (several deps unpatched)
generate `prims.c`, link statically	systematic
genuinely native pieces (LLVM, FFI hazards)	needs care

Worth a shared effort? A dune/opam build target: whole project + native stack → wasm, from an opam switch, at least until a WasmGC backend can interoperate with Emscripten/WASI-compiled C. (OCaml 5? `Tag_val` is fixed in 5.x, but 5 dropped 32-bit.)

Try it & thanks

Demo (works right now):
https://mopsawasm.rboud.com/
Code:
https://github.com/rboudrouss/mopsa-emcc

Antoine Miné: guidance through MOPSA and support throughout
Raphaël Monat: Try-MOPSA, the jsoo/VPL baseline
Vincent Chan: `ocaml-wasm` configure tweaks & Unix stubs
Ben Smith (binji): LLVM-to-wasm fork and notes

I'm on the job market, looking for a **CIFRE PhD** host company (French industrial PhD, on static analysis) or an engineering role.
`contact@rboud.com` - `https://rboud.com`

Backups

Full benchmark synthesis (~100 reps, medians)

target	files	inst.	cold run	hot run	Mopsa self-time	RSS MB
native	14	n/a	23 ms	n/a	23 ms	70
wasm-node	14	63 ms	476 ms	294 ms	n/a*	193
jsoo-node	9	359 ms	371 ms	n/a**	364 ms	200
wasm-browser	14	641 ms	356 ms	304 ms	n/a*	35***
jsoo-browser	9	1.13 s	1.01 s	1.00 s	298 ms	35***

* the wasm runtime clock is a stub (always 0.000 s) → wall-clock used · ** jsoo-node cannot re-enter a fresh OCaml state in-process · *** browser RSS ≈ main-thread heap only, worker not counted · inst. = compile + instantiate + unpack `.data` (one-shot first start in the browser) · browser "cold" is quasi-hot: the page is shared across files. AMD Ryzen 7 1700X,

Backup: what about JSPI?

The "existing solutions" we ruled out for suspending around I/O:

- **Asyncify** (Emscripten): rewrites the wasm to unwind/replay the stack, but conflicts with OCaml exceptions (`setjmp`/`longjmp`), they both rewrite the stack
- so interactive/DAP modes *block* on stdin in a Worker, fed via `SharedArrayBuffer` + `Atomics.wait` (needs cross-origin isolation)

JSPI (JavaScript-Promise Integration): the *engine* suspends the whole wasm stack on an async import, resumes it on resolve. No instrumentation, so the `setjmp`/`longjmp` conflict likely disappears; a suspending `read` import would replace the whole SAB channel.

We stayed with SAB for coverage: JSPI ships in Chrome/Edge (2025) and Firefox (2026) but not yet Safari. SAB works in every isolated browser.

Its underlying *stack switching* is also what OCaml 5 effects would need on wasm.