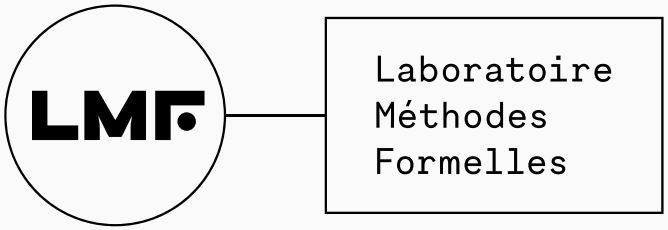
OCaml WebAssembly Interpreter

Léo Andrès <l@ndrs.fr>[1, 3]
Pierre Chambart <pierre.chambart@ocamlpro.com>[1]
Filipe Marques
<filipe.s.marques@tecnico.ulisboa.pt>[2]
José Fragoso Santos
<jose.fragoso@tecnico.ulisboa.pt>[2]

1. OCamlPro
2. INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal
3. Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles

October 2023 – Wasm Research Day – Munich



A WebAssembly interpreter in OCaml:

- started to learn and experiment with Wasm
- to produce an OCaml to Wasm compiler (Wasocaml) which required many proposals
- passing the full test suite

Can we test it more?

With Eric Patrizio, we wrote a grey-box fuzzer with Crowbar/AFL:

- write generators
- modified compiler to produce instrumentable binaries
- guide the generators towards unseen path
- differential testing: reference interpreter/optimisations

```
let fields env =
  let* memory = option (memory env) in
  let* datas = list (data env) in
  let* types = list (typ env) in
  let* tables = list (table env) in
  let* elems = list (elem env) in
  let* globals = list (global env) in
  let* funcs = list (func env) in
  ...
```

expression generated by choosing one instruction amongst all instructions matching the type of the stack and going on recursively (+ a fuel mechanism to avoid too big expressions)

Found bugs in various places: pretty printing, typechecker, overflow.
Can we test it more?

symbolic execution no tool working on OCaml

abstract interpretation no tool working on OCaml

rewrite in Coq tedious

I'll come back to this at the end.

```
(module
  (func $square (param $x i32) (result i32)
    (i32.mul (local.get $x) (local.get $x))))
```

```
$ owi module.wast
<- nothing
```

By default nothing useful can happen: Wasm has no IO capabilities.

```
(module
  (func $print_i32
    (import "io" "print_i32") (param i32))
  (func $start
    (call $print_i32 (i32.const 42)))
  (start $start))
```

```
let imports = [
  "print_i32",
  Extern_func (Func (Arg (I32, Res), R0), print_i32) ]
let link_state = Link.module empty_state "io" imports
let modul = Parse.Module.from_file "test.wasm_module"
Run.until_interpret link_state modul
```

Nothing really special except linking

- Syntax
- Name resolution
- Typechecking
- Optional optimisation
- Linking + Constants evaluation
- Evaluation

Adopted extensions:

- **OK** Import/Export of Mutable Globals
- **OK** Non-trapping float-to-int conversions
- **OK** Sign-extension operators
- **OK** Multi-value
- **OK** Reference Types
- **OK** Bulk memory operations
- **KO** Fixed-width SIMD (tedious)

Proposed extensions:

- **OK** Tail calls
- **OK** Typed Function References
- **OO** GC (ongoing)
- **KO** Exception handling
- **KO** Reference-Typed Strings
- **KO** Stack-switching

Met José Fragoso Santos in Dagstuhl

What's symbolic execution ?

- Static analysis
- White box fuzzing
- Use SMT formulas to test reachability

Our goal: finding input values leading to a trap.

Generate a path condition for each reachable branch.

```
;; PC
  (local.set $cond
    (i32.gt_s (local.get $x) (i32.const 0)))
;; PC && cond = (x > 0)
  (if (local.get $cond)
    (then ...
;; PC && cond = (x > 0) && cond = true
    ) (else ...
;; PC && cond = (x > 0) && cond = false
    ))
```

Proper bindings to create symbolic values:

```
(func $start
  (local $x i32)
  (local.set $x (call $gen_i32 (i32.const 42)))
  (if (i32.lt_s (i32.const 5) (local.get $x))
    (then unreachable)))
```

If we reach a trap, we ask the SMT solver a model for the PC.

No false positive. Gives us the input values leading to a crash:

```
$ owi --symbolic file.wat
PATH CONDITION: (i32.lt_s (i32 5) x)
TRAP: unreachable
Model:
  (x i32 (i32 6))
Reached problem!
Solver time 0.001574s
```

How to implement this and keep the concrete interpreter ?

Or: how to make your interpreter symbolic easily ?

Put everything in a functor!

And use the choice monad.

How it looked like before:

```
match instr, stack with
| Binop Add, x :: y :: t ->
  (x + y) :: t
| Binop GT, x :: y :: t ->
  (x > y) :: t
| If_then_else (it_true, if_false), cond :: t->
  if cond then
    eval if_true t
  else
    eval if_false t
```

We started by functorizing on values:

```
module type Values = sig
  type t
  val add : t -> t -> t
  val gt : t -> t -> t
end
```

```
| Binop Add, x :: y :: t ->
  (Value.add x y) :: t
| Binop GT, x :: y :: t ->
  (Value.gt x y) :: t
```

concrete case: regular Wasm values

symbolic case: expressions as found in the PC

Then functorizing on the choice monad:

```
module type Choice = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
  val select : Value.bool -> bool t
end
```

```
| If_then_else (it_true, if_false), cond :: t->
  let* cond_bool = Choice.select cond in
  if cond_bool then
    eval if_true t
  else
    eval if_false t
```

Then functorizing on the choice monad:

```
module type Choice = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
  val select : Value.bool -> bool t
end
```

```
| If_then_else (it_true, if_false), cond :: t->
  let* cond_bool = Choice.select cond in
  if cond_bool then
    eval if_true t
  else
    eval if_false t
```

no abstraction runtime cost: thanks to flambda and the right `[@inlined]` annotation, the concrete performance is identical to the original version

```
let* cond_bool = Choice.select cond in
if cond_bool then
  eval if_true t
else
  eval if_false t
```

select + bind:

- in the concrete case: it is the identity
- in the symbolic case: it evaluates both branches

- almost the continuation + state monad
- the state contains the path condition
- `select` adds `condition` or `not condition` to the path condition and call an SMT solver to check if the branch is reachable
- if there are 2 valid branches the state is copied
- expression simplification to avoid calling the SMT when the condition is trivial
- we have a multicore version

To benchmark we compile C to Wasm with Clang and compare with existing C tools.

We have some simple optimisations to handle malloc and free.

The preliminary benchmarks are quite good: multicore version from 100 to 2000 times faster than the old Wasp version.

Most of the time is now spent in the solver.

Things we want to try:

- Rust, Cobol, C++
- proper join points to limit combinatorial explosion
- guiding execution with A*
- add a domain to propagate some simplifications
- more solvers (Colibri2, Alt-ergo, CVC5)
- JIT (MetaOCaml)
- Wasm to Wasm optimisations using symbolic execution
- exceptions, algebraic and full GC

Remember the question: can we test the interpreter further than our fuzzer ?

- the interpreter is written in OCaml

Remember the question: can we test the interpreter further than our fuzzer ?

- the interpreter is written in OCaml
- we now have ~~one~~ two compilers from OCaml to WasmGC

Remember the question: can we test the interpreter further than our fuzzer ?

- the interpreter is written in OCaml
- we now have ~~one~~ two compilers from OCaml to WasmGC
- with GC support we could run OCaml program

Remember the question: can we test the interpreter further than our fuzzer ?

- the interpreter is written in OCaml
- we now have ~~one~~ two compilers from OCaml to WasmGC
- with GC support we could run OCaml program
- that is symbolic execution/whitebox fuzzing of OCaml

Remember the question: can we test the interpreter further than our fuzzer ?

- the interpreter is written in OCaml
- we now have ~~one~~ two compilers from OCaml to WasmGC
- with GC support we could run OCaml program
- that is symbolic execution/whitebox fuzzing of OCaml
- could find more bugs by interpreting the interpreter

github.com/ocamlpro/owi can be used to:

- debug and experiment with Wasm
- embed C in OCaml (but slow)
- perform static analysis on Wasm
- find bugs in programs compiled to Wasm
- reusable in your interpreter (choice monads, encoding library)

Thanks!