Owi: cross-language, multi-core, multi-solver symbolic execution

Léo …, Pierre Chambart, Arthur Carcano @ OCamlPro

Filipe Marques, José Fragoso Santos @ University of Lisbon

… Andrès, Jean-Christophe Filliâtre @ Université Paris-Saclay

With contributions from Dario Pinto, Eric Patrizio, Frederico Ramos, Olivier Pierre, Zhicheng Hui, Vasu Singh, Simon Ser, Neha Chriss, Hichem Rami Ait El Hara, Basile Clément, Saïd Zuhair, Émilien Lemaire, Félix Loyau-Kahn, Nathanaëlle Courant, Gabriel Scherer

4th of June – <Programming> 2025 @ Prague

Outline

1. The story
2. Technical stuff
3. Fun stuff

The Story

2015 - 2017

- a fast, safe, portable compilation target
- available since 2017 in browsers
- used in cloud, edge, IoT, embedded systems…
- C, C++ and Rust have a Wasm backend

Bringing the Web up to Speed with WebAssembly

Andreas Haas Andreas Rossberg Derek L. Schuff* Ben L. Titzer
Google GmbH, Germany / *Google Inc, USA
{ahaas,rossberg,dschuff,titzer}@google.com

Michael Holman
Microsoft Inc, USA
michael.holman@microsoft.com

Dan Gohman Luke Wagner Alon Zakai
Mozilla Inc, USA
{sunfishcode,luke,azakai}@mozilla.com

JF Bastien
Apple Inc, USA
jfbastien@apple.com

Abstract

The maturation of the Web platform has given rise to sophisticated and demanding Web applications such as interactive 3D visualization, audio and video software, and games. With that, efficiency and security of code on the Web has become more important than ever. Yet JavaScript as the only built-in language of the Web is not well-equipped to meet these requirements, especially as a compilation target.

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable low-level bytecode called WebAssembly. It offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web. WebAssembly has been designed with a formal semantics from the start. We describe the motivation, design and formal semantics of WebAssembly and provide some preliminary experience with implementations.

CCS Concepts • **Software and its engineering** → **Virtual machines**; **Assembly languages**; *Runtime environments*; *Just-in-time compilers*

Keywords Virtual machines, programming languages, assembly languages, just-in-time compilers, type systems

1. Introduction

The Web began as a simple document exchange network but has now become the most ubiquitous application platform device types. By historical accident, JavaScript is the only natively supported programming language on the Web, its widespread usage unmatched by other technologies available only via plugins like ActiveX, Java or Flash. Because of JavaScript's ubiquity, rapid performance improvements in modern VMs, and perhaps through sheer necessity, it has become a compilation target for other languages. Through Emscripten [43], even C and C++ programs can be compiled to a stylized low-level subset of JavaScript called asm.js [4]. Yet JavaScript has inconsistent performance and a number of other pitfalls, especially as a compilation target.

WebAssembly addresses the problem of safe, fast, portable low-level code on the Web. Previous attempts at solving it, from ActiveX to Native Client to asm.js, have fallen short of properties that a low-level compilation target should have:

- Safe, fast, and portable *semantics*:
  - safe to execute
  - fast to execute
  - language-, hardware-, and platform-independent
  - deterministic and easy to reason about
  - simple interoperability with the Web platform
- Safe and efficient *representation*:
  - compact and easy to decode
  - easy to validate and compile
  - easy to generate for producers
  - streamable and parallelizable

Why are these goals important? Why are they hard?

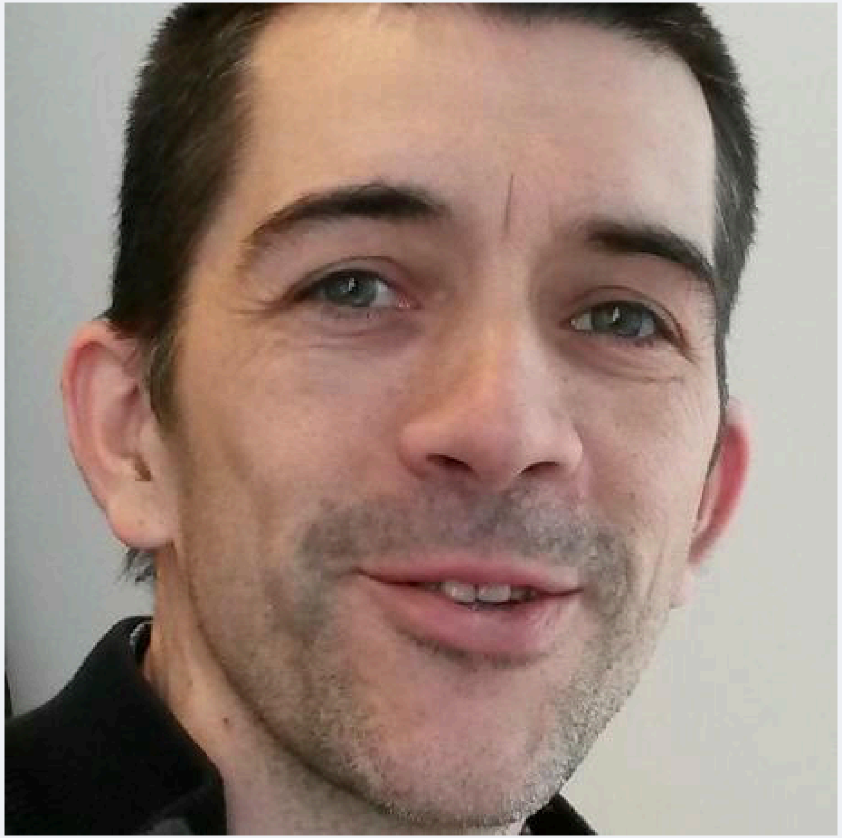
Safe Safety for mobile code is paramount on the Web,

August 2020

Do you want to do a PhD thesis at OCamlPro? What about compiling OCaml to WebAssembly?

— Pierre Chambart

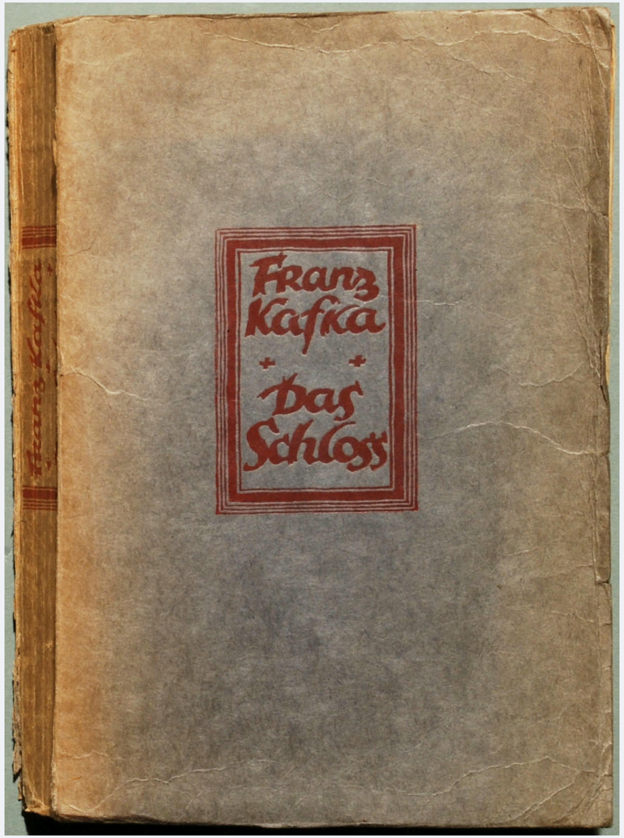
September 2020



A PhD about compiling garbage-collected languages to Wasm? I don't know what is Wasm but yes, sure.

— Jean-Christophe Filliâtre

October 2020 – September 2021



Waiting for french administration to validate the PhD project. Took 1 year!

October 2021 – December 2022

PhD finally started, in the first year:

- made Owi, a Wasm interpreter to learn and experiment
- made Wasocaml, an OCaml to Wasm compiler

Dagstuhl Mars 2023

Presented Wasocaml, an OCaml to Wasm compiler, and Owi was briefly mentionned.

Dagstuhl Mars 2023

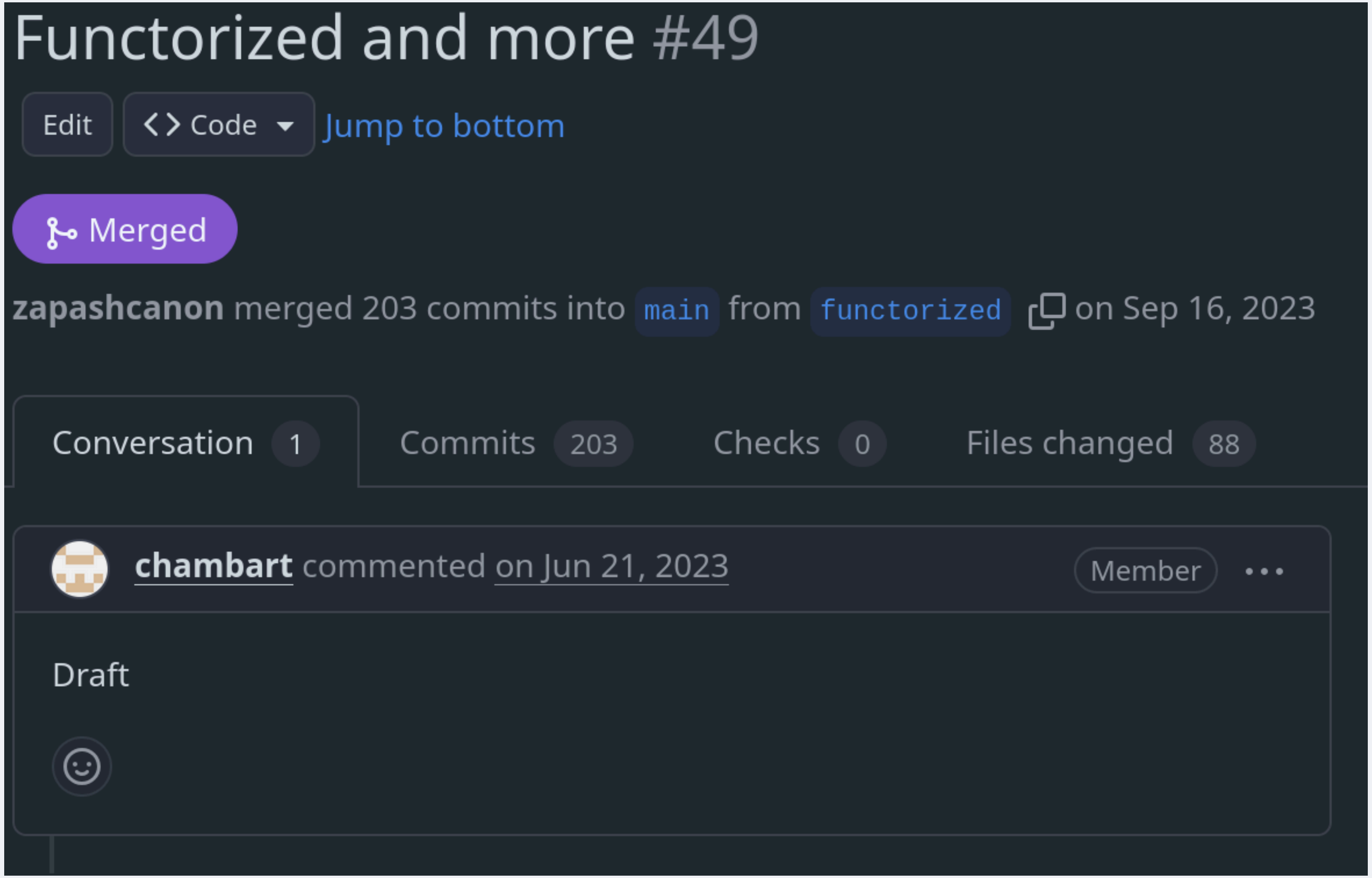
What about making a symbolic interpreter with Owi?

— José Fragoso Santos (Assistant Professor in Lisbon)

Paris June 2023

- Filipe Marques is the PhD student of José
- they made WASP, a Wasm concolic interpreter based on the reference interpreter
- published at ECOOP
- both came one week in Paris to tell us about it

June – September 2023



Owi is now symbolic !

December 2024

- I defended!

What happened in between?

Answer after the technical part!

2. Technical stuff

Outline

1. Wasm 101
2. Symbolic Execution 101
3. From concrete to symbolic

Wasm 101

- stack-based language;
- simple types (`i32`, `i64`, `f32`, `f64`) ;
- statically typed (`[ i32 ; f32 ] -> [ i32 ]`);
- functions ;
- a formal semantics, with no undefined behabiour.

Wasm 101

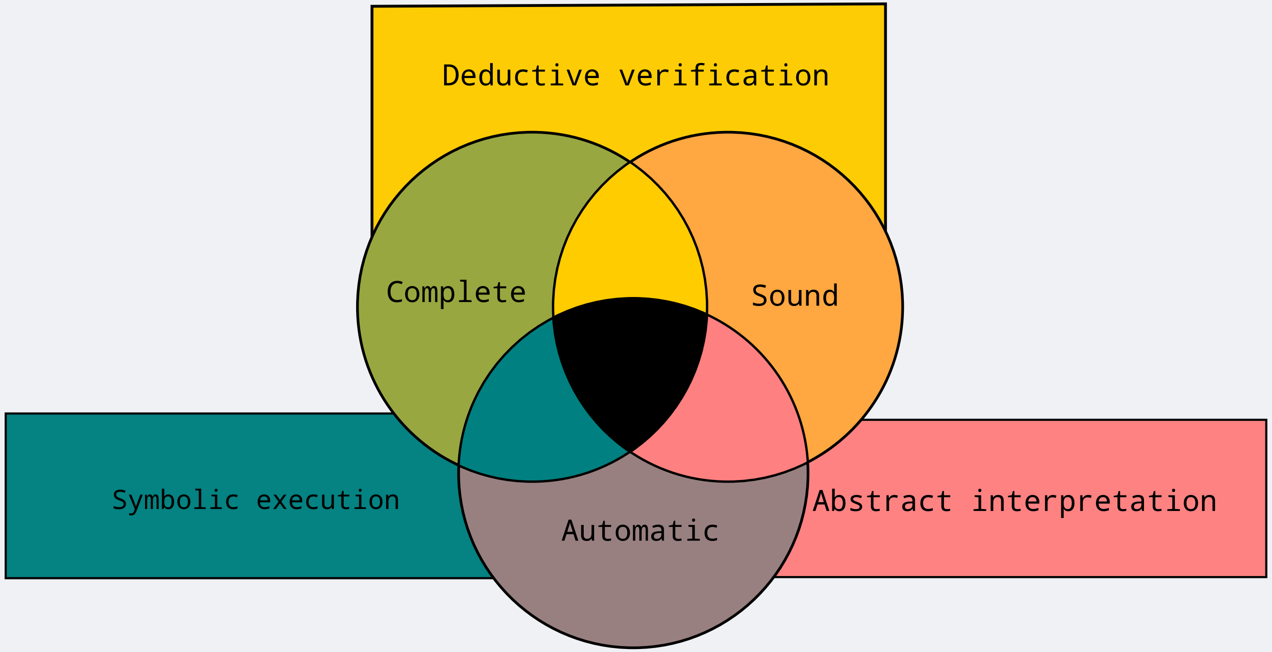
```
(module

  (func $f (param $n i32) (result i32)
                                                ;; []
    (i32.lt_s (local.get $n) (i32.const 2)) ;; [ n < 2 ]
    (if (then      ;; [ ]
      local.get $n ;; [ n ]
      return ))    ;; early return
                                               ;; [ ]
    (i32.sub (local.get $n) (i32.const 2)) ;; [ n-2 ]
    call $f                                ;; [ f(n-2) ]
    (i32.sub (local.get $n) (i32.const 1)) ;; [ n-1; f(n-2) ]
    call $f                                ;; [ f(n-1); f(n-2) ]
    i32.add                                ;; [ f(n-1) + f(n-2)]
                                           ;; implicit return
  ))
```

Symbolic Execution 101

A technique for:

- finding bugs in programs (and proving properties);
- implementing solver-aided programming;
- test-case generation.



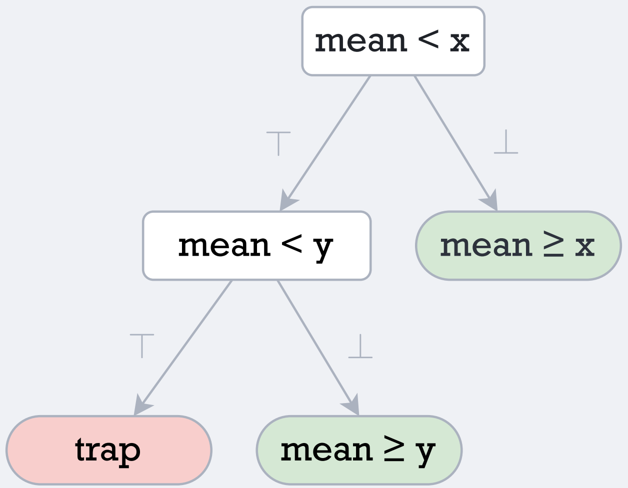
Symbolic Execution 101

```
(func $mean (param $x i32) (param $y i32)
  (local $mean i32)

  i32.const 2                          ;; [ 2 ]
  (i32.add
    (local.get $x) (local.get $y)) ;; [ (x+y) ; 2 ]
  i32.div_u                            ;; [ (x+y) / 2 ]
  local.set $mean                      ;; [ ]

  (i32.lt_u (local.get $mean) (local.get $x))
  (if (then
    (i32.lt_u (local.get $mean) (local.get $y))
    (if (then unreachable ))))

  local.get $mean)
```



Symbolic Execution 101

```
(func $mean (param $x i32) (param $y i32)
  (local $mean i32)

  i32.const 2                         ;; [ 2 ]
  (i32.add
    (local.get $x) (local.get $y)) ;; [ (x+y) ; 2 ]
  i32.div_u                           ;; [ (x+y) / 2 ]
  local.set $mean                     ;; [ ]

  (i32.lt_u (local.get $mean) (local.get $x))
  (if (then
    (i32.lt_u (local.get $mean) (local.get $y))
    (if (then unreachable ))))

  local.get $mean)
```

```
Unreachable
model {
  symbol x 2147483650
  symbol y 2147483655
}
```

Indeed:

$$\frac{(x \oplus y)}{2}$$
$$= \frac{2147483650 \oplus 2147483655}{2}$$
$$= \frac{9}{2}$$
$$= 4$$

Symbolic Execution 101

We want to find input values leading to a state S.

- input values are represented by symbols
- the program executes with expressions made of concrete values + symbols
- when branching both branches are explored
- information about previous branches is kept in the path condition (PC)
- when S is reached, a model is generated by an SMT solver from the PC

This model corresponds to the input values leading to the state S.

From Concrete To Symbolic

The initial Owi concrete interpreter:

```
match instr, stack with
| Binop Add,       (x :: y :: stack) ->
  (add_i32 x y) :: stack
| If_else (t, f), (b :: stack)       ->
  let b = bool_of_i32 b in
  if b then eval t stack
       else eval f stack
```

How to get a symbolic interpreter from this?

Step 1/2 : Abstract Over the Type of Values

We use an abstract `Value` module:

```
match instr, stack with
| Binop Add,       (x :: y :: stack) ->
  (Value.add_i32 x y) :: stack
| If_else (t, f), (b :: stack)       ->
  let b = Value.bool_of_i32 b in
  if b then eval t stack
       else eval f stack
```

```
module type Value = sig
  type t
  val add_i32 : t -> t -> t
  type bool
  val bool_of_i32 : t -> bool
end
```

Step 2/2 : Abstract Over the Execution Strategy

We use an abstract `Choice` module:

```
match instr, stack with
| Binop Add,        (x :: y :: stack) ->
  (Value.add_i32 x y) :: stack
| If_else (t, f), (b :: stack)       ->
  let b = Value.bool_of_i32 b in
  (* the single new line: *)
  let* b = Choice.select cond in
  if b then eval t stack
       else eval f stack
```

```
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
```

- most of the code unchanged
- we must insert `Choice.select` and `Choice.bind` at branching point

How is implemented the symbolic Choice monad?

It was hard. We rewrote it four times. Then…

Arthur Carcano implemented it nicely with:

- an error monad;
- a state monad;
- a coroutine monad.

Combined using three layered monad transformers.

The exploration is done in parallel thanks to OCaml 5!

Described in length in the paper.

Fun stuff

Multi-Language

C Symbolic Execution

```c
#include <owi.h>

unsigned int mean1(unsigned int x,
                   unsigned int y) {
  return (x & y) + ((x ^ y) >> 1); }

unsigned int mean2(unsigned int x,
                   unsigned int y) {
  return (x + y) / 2; }

void check(unsigned int x, unsigned int y) {
  owi_assert(mean1(x, y) == mean2(x, y)); }

void main(void) {
  unsigned int x = owi_i32();
  unsigned int y = owi_i32();
  check(x, y); }
```

The subcommand `owi c` takes care of compiling and linking:

```
$ owi c ./function_equiv.c
Assert failure
model {
  symbol_0 i32 -922221680
  symbol_1 i32 1834730321
}
Reached problem!
```

Standard library based on `dietlibc`. Special handling of `malloc` and `free` to detect use-after-free or double-free.

C++ Symbolic Execution

```cpp
#include <owi.h>

struct IntPair {
  int x, y;
  int mean1() const {
    return (x & y) + ((x ^ y) >> 1);
  }
  int mean2() const {
    return (x + y) / 2;
  }
};

int main() {
  IntPair p{owi_i32(), owi_i32()};
  owi_assert(p.mean1() == p.mean2());
}
```

The subcommand `owi c++` takes care of compiling and linking:

```
$ owi c++ ./poly.cpp
Assert failure
model {
  symbol symbol_0 i32 -2147483648
  symbol symbol_1 i32 -2147483646
}
Reached problem!
```

Re-using the symbolic libc.

Rust Symbolic Execution

```rust
fn mean1(x: i32, y: i32) -> i32 {
  (x + y) / 2
}

fn mean2(x: i32, y: i32) -> i32 {
  (x & y) + ((x ^ y) >> 1)
}

fn main() {
  let x = owi_sym::u32_symbol() as i32;
  let y = owi_sym::u32_symbol() as i32;
  owi_sym::assert(mean1(x, y) == mean2(x, y))
}
```

The subcommand `owi rust` takes care of compiling and linking:

```
$ owi rust ./main.rs
Assert failure
model {
  symbol symbol_0 i32 1073741835
  symbol symbol_1 i32 -2147483642
}
Reached problem!
```

Re-using the symbolic libc.

Zig symbolic execution

```
fn fibonacci(n: i32) i32 {
    if (n < 0) {
        @panic("expected a positive number");
    }
    if (n <= 2) return n;
    return fibonacci(n - 1) + fibonacci(n -
2);
}

pub fn main() void {
    const n: i32 = i32_symbol();
    assume(n > 0);
    assume(n < 10);
    const result = fibonacci(n);
    assert(result != 21);
}
```

The subcommand `owi zig` takes care of compiling and linking:

```
$ owi zig ./fib.zig
owi: [ERROR] Assert failure
model {
  symbol symbol_0 i32 7
}
owi: [ERROR] Reached problem!
```

Re-using the symbolic libc.

Cross-language

Moving a Codebase from C to Rust

Original C version:

```c
float dot_product(float x[2], float y[2]) {
    return (x[0]*y[0] + x[1]*y[1]);
}
```

New Rust version:

```rust
fn dot_product_rust(x: &[f32; 2], y: &[f32; 2]) -> f32 {
    x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
}
```

Is It Correct?

Owi says no:

```
model {
  symbol_0 f32 -0.
  symbol_1 f32 -0.
  symbol_2 f32 0.
  symbol_3 f32 0.
}
```

Breaking it Down

C version:

```
x[0] * y[0] + x[1] * y[1]
-0. *   0. +  -0. *   0.
-0. + -0.
-0.
```

Rust version:

```
x.iter().zip(y).map(|(xi, yi)| xi * yi).sum()
[-0.,-0.].iter().zip([0.,0.]).map(|(xi,yi)|xi*yi).sum()
[(-0.,0.),(-0.,0.)].map(|(xi,yi)|xi*yi).sum()
[-0., -0.].sum()
+0. + -0. + -0.
+0.
```

- fixed in the Rust standard library
- initial accumulator for `sum()` is now `-0.`
- it broke typst (the tool used to make these slides) that was relying on this behaviour

Solver-aided Programming

Polynomial Example

```c
#include <owi.h>

void f(void) {
  int x = owi_i32();
  int x2 = x * x;
  int x3 = x * x * x;

  int a = 1;  int b = -7;
  int c = 14; int d = -8;

  int poly =
    a * x3 + b * x2 + c * x + d;

  owi_assert(poly != 0);
}
```

This is similar to Rosette for Racket ("solver-aided programming") but:

- parallel
- multi/cross-language

We used it for :

- solving a maze
- generate a set of cards for dobble
- generate strongly regular graph with parameters (9,4,1,2)
- generate music sheet for a string quartet

Music Generation

- limit on the instruments' range
- no crossing
- no leap of more than an octave
- notes belongs to the key
- the leading tone resolves to the tonic
- instruments form chords
- no parallel fifths or octaves

Stuff I did not talked about

- the Smt.ml library
- automatic harness generation
- a fuzzer for Wasm interpreters
- iso-behaviour checker (to test Binaryen, the Wasm optimizer)
- we have support for symbolic runtime annotation checking of ACSL
- Weasel (WEbAssembly Specification Language)
- benchmarks (we are the best for Wasm, close to KLEE, the best one for C)
- concolic execution
- optimisations we have (path-condition independence, negation shortcut)

Current work by interns

- complex coverage-critera test-case generation (MCDC) (Saïd)
- better use of multi-solver (Félix)
- add heuristics to explore interesting paths first (Julie, starting two weeks)

Future work already funded

- abstract interpretation of Wasm
- map model back to complex source structures
- proper symbolic system interface (WASI, Componen Model, Common ABI)
- support missing proposals (SIMD, multi-memories, exceptions, memory64, GC)
- new languages: Haskell, TinyGo, OCaml, Guile, (maybe Dart, Java, Kotlin)
- build system integration (`CC='owi clang'`, `cargo owi`)
- case study on real libraries (started to check Wayland stuff with emersion)
- add locations and source map support (if time permits)

If you have ideas, please tell me!

Conclusion

Owi is an efficient symbolic execution engine for Wasm, C, C++, Rust and Zig that can perform cross-language analysis but can also be used as a Wasm toolkit for developpers and researchers. We want to make it a real-world tool.

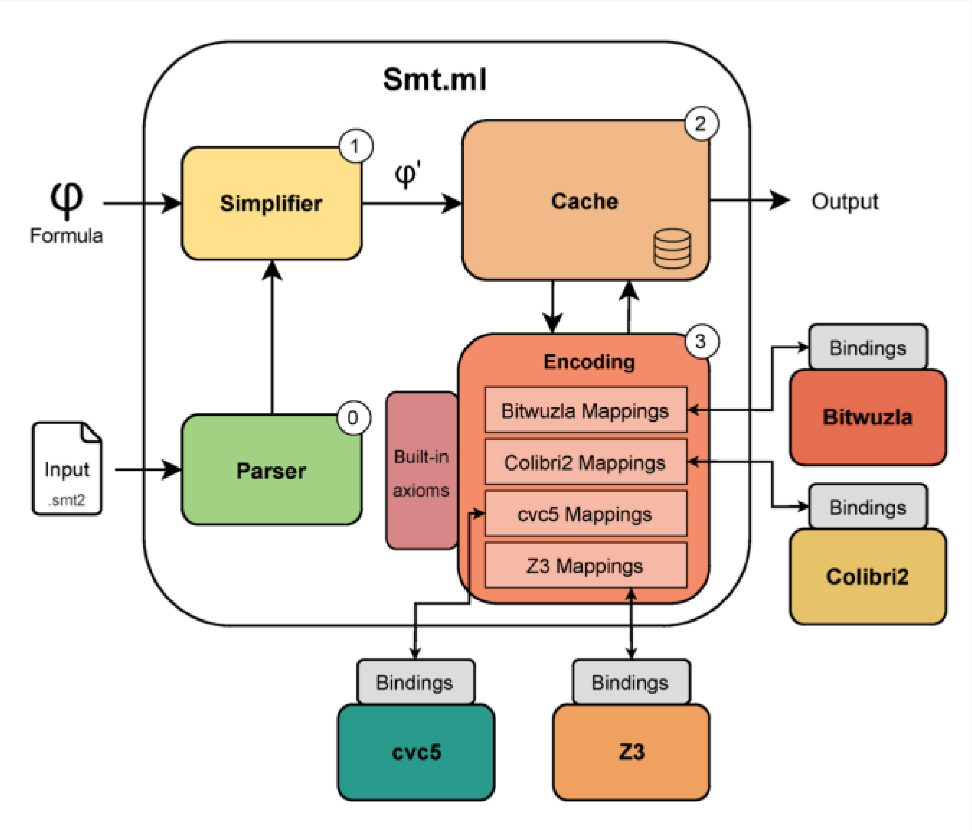
It is free software! You can try it at github.com/ocamlpro/owi . Documentation is available at ocamlpro.github.io/owi . You must built it from sources to get most of what I presented but I am preparing a new release.

OCaml PRO

We want to explore industrial applications and welcome discussions with users interested in Owi, as well as R&D on Wasm and programming languages.

Bonus

The Smt.ml library



- provides a type of symbolic expressions
- can map expressions to many SMT-solver
- provides optimisations (simplifications, cache through hash-consing)
- ease of use (more typing)
- incremental mode

Iso-behaviour checker

Binaryen takes `original.wasm` and produces `optimized.wasm`.

To test their optimizations:

- a fuzzer that generates random `original.wasm` files
- compare output of original and optimized…
- … with `0`, `1` and `MAX_INT` as input values

This is bad. I wrote `owi iso original.wasm optimized.wasm` so that all inputs are considered.

Seems to work well, but I only did half of the work so that we can ask Google to pay for the other half.

They'd also like to be able to verify optimisations with wasm-threads, we said we can do it but they have to pay.

Automatic Harness Generation

```c
void f(unsigned int x, unsigned int y) {
  // ... complicated stuff
}

void f_harness(void) {
  unsigned int x = owi_i32(); unsigned int y = owi_i32();
  f(x, y); }
```

It is annoying to write, so we have automatic harness generation:

```c
void f(unsigned int x, unsigned int y) {
  // ... complicated stuff
}
```

```
$ owi c ./function_equiv.c --entry-point=f --invoke-with-symbols
```

No need to touch source code to test the program anymore!

Towards Proofs

ACSL

The ANSI/ISO C Specification Language (ACSL).

Allows to write function contracts:

```
/*@ requires precondition
    ensures  postcondition
*/
int f(int n) { ... }
```

But also assertions, loop invariants, type invariants…

E-ACSL

The Executable subset of ACSL (E-ACSL).

```
/*@ requires n <= INT_MAX - 3;
    ensures \result == n + 3; */
int plus_three(int n) {
  return n + 3;
}
```

```
int __gen_e_acsl_plus_three(int n) {
  long __gen_e_acsl_at = (long)n;
  int __retres;
    { __e_acsl_assert_register_int(...);
    __e_acsl_assert(n <= 2147483644);
    __e_acsl_assert_clean(...); }
  __retres = plus_three(n);
    { __e_acsl_assert_register_int(...);
    __e_acsl_assert_register_long(...);
    __e_acsl_assert((long)__retres ==
__gen_e_acsl_at + 3L);
    __e_acsl_assert_clean(...); }
}
```

E-ACSL Support in Owi

We re-use the code generator from E-ACSL, but uses our own symbolic E-ACSL runtime:

```
void __e_acsl_assert(int predicate, __e_acsl_assert_data_t *data) {
    owi_assert(predicate);
}
```

Available through `owi c --e-acsl`. It allows to symbolically execute code annotated by specifications by generating executable assertions.

Described in our paper Cross-Language Symbolic Runtime Annotation Checking. There we show how the subset of ACSL supported by E-ACSL can be extended when targetting symbolic execution.

Weasel

We started to do the same in Wasm:

```
(module
  (@contract $plus_three
    (ensures (= result (+ $n 3))))
  (func $plus_three (param $n i32)
      (result i32)
    local.get $n
    i32.const 3
    i32.add
  ))
```

Design of Weasel (WEbAssembly SpEcification Language).

It uses the custom annotation syntax proposal.

Generating Assertions from Weasel

We did something similar to E-ACSL, still experimental :

```
(module
  (@contract $plus_three
    (ensures (= result (+ $n 3))))
  (func $plus_three (param $n i32)
(result i32)
    local.get $n
    i32.const 3
    i32.add
  )
  (start $plus_three)
)
```

```
(import "symbolic" "assert"
  (func $assert (param i32)))
(func $__weasel_plus_three (param $n i32)
(result i32) (local $__weasel_temp i32)
(local $__weasel_res_0 i32)
  (call $plus_three (local.get 0))
  local.set 2
  (i32.eq (local.get 2) (i32.add (local.get
0) (i32.const 3)))
  call $assert
  local.get 2
)
(start $__weasel_plus_three)
```

What can we do with this?

Contrary to most symbolic execution engine, Owi does not perform any approximation (modulo the source language compiler approximation wrt. undefined behaviours).

When the analysis terminates, we've got a proof!

It could be used to proove programs or functions on its own.

It could also combined with:

- a deductive verification tool to automate the proof, or find counter-examples;
- an abstract interpretation engine to confirm or infirm bugs found.

Benchmarks

On Wasm Code

A hand-written Wasm B-Tree library with 27 possible configurations (number of symbols):

Tool	Min	Max	Mean
Owi-24	1.0	1.0	1.0
Owi-1	0.6	14.0	4.5
WASP	0.4	16.4	4.1
SeeWasm	2.5	101	57.1
Manticore	17.2	844	312

On C Code

1215 C programs from Test-Comp.

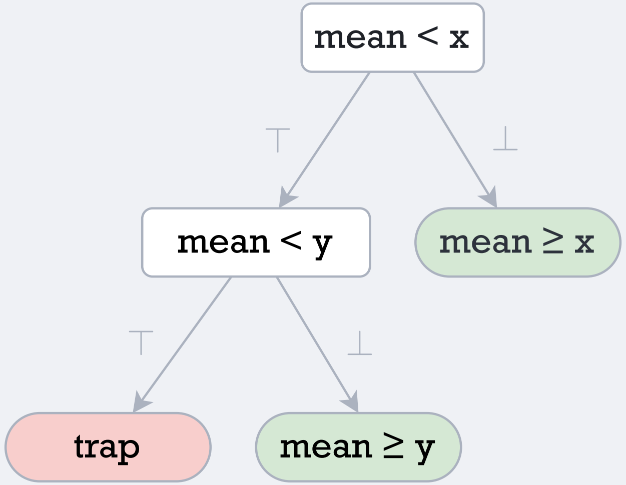
Tool	Bug found	Timeout	Bug not found
KLEE	782	368	65
Owi	676	539	0
Symbiotic	489	657	69

Good results, especially when we know that Owi:

- does no approximation;
- has no optimisation appart from the multi-core;
- these are old benchmarks…

Most of the time is spent in the solver. What can we do about it?

One new optimisation: concolic execution



- we begin with random values for symbols
- we keep the symbolic and concrete state
- no need to call the solver at each branch
- but we still keep the PC
- if we found a bug, we're done
- otherwise, we start again but with values leading to a new branch (we ask the SMT using our list of PC)

This is what most engine are doing. AKA "dynamic symbolic execution".

Another new optimisation: path-condition slicing

The PC contains many formulas unrelated to most branching conditions.

This is slowing-down the SMT-solver.

We use a union-find data-structure where keys are variables and nodes are set of (related) constraints.

When meeting a new branch, we add the condition to the PC, then slice it, and only send the slice to the solver.