Exécution symbolique pour tous

Compilation d'OCaml vers WebAssembly

Léo Andrès

Sous la supervision de :

- Pierre Chambart @ OCamlPro
- Jean-Christophe Filliâtre @ Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles

16 décembre 2024

Le Web côté client : HTML

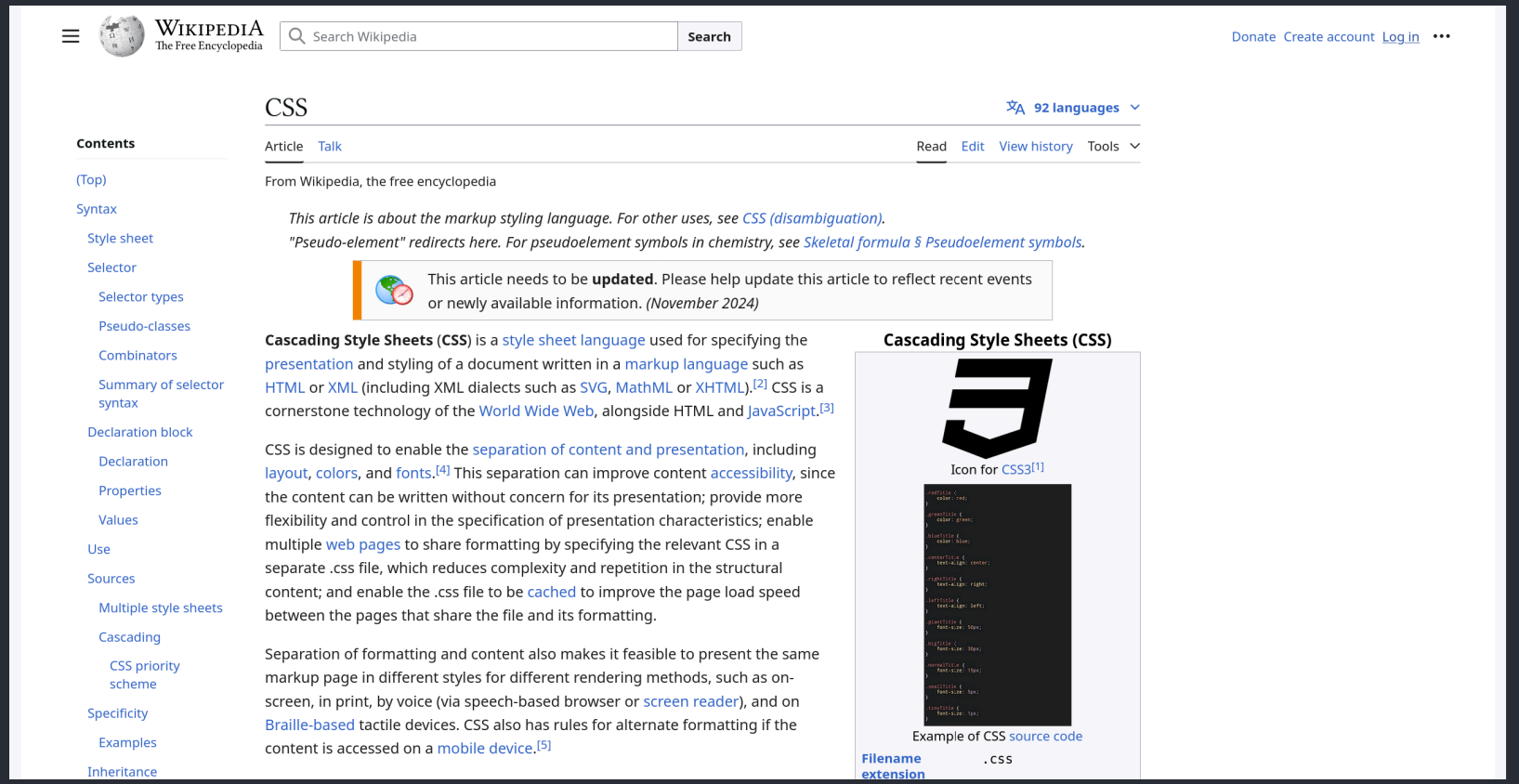
World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
Pointers to the world's online information, subjects , W3 servers, etc.
Help
on the browser you are using
Software Products
A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)
Technical
Details of protocols, formats, program internals etc
Bibliography
Paper documentation on W3 and references.
People
A list of some people involved in the project.
History
A summary of the history of the project.
How can I help ?
If you would like to support the web..
Getting code
Getting the code by anonymous FTP , etc.

Le Web côté client : CSS



WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Donate | Create account | Log in

CSS

92 languages

Article | Talk | Read | Edit | View history | Tools

From Wikipedia, the free encyclopedia

This article is about the markup styling language. For other uses, see CSS (disambiguation).
"Pseudo-element" redirects here. For pseudoelement symbols in chemistry, see Skeletal formula § Pseudoelement symbols.

This article needs to be **updated**. Please help update this article to reflect recent events or newly available information. *(November 2024)*

Cascading Style Sheets (**CSS**) is a style sheet language used for specifying the presentation and styling of a document written in a markup language such as HTML or XML (including XML dialects such as SVG, MathML or XHTML).[2] CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.[3]

CSS is designed to enable the separation of content and presentation, including layout, colors, and fonts.[4] This separation can improve content accessibility, since the content can be written without concern for its presentation; provide more flexibility and control in the specification of presentation characteristics; enable multiple web pages to share formatting by specifying the relevant CSS in a separate .css file, which reduces complexity and repetition in the structural content; and enable the .css file to be cached to improve the page load speed between the pages that share the file and its formatting.

Separation of formatting and content also makes it feasible to present the same markup page in different styles for different rendering methods, such as on-screen, in print, by voice (via speech-based browser or screen reader), and on Braille-based tactile devices. CSS also has rules for alternate formatting if the content is accessed on a mobile device.[5]

Cascading Style Sheets (CSS)

Icon for CSS3[1]

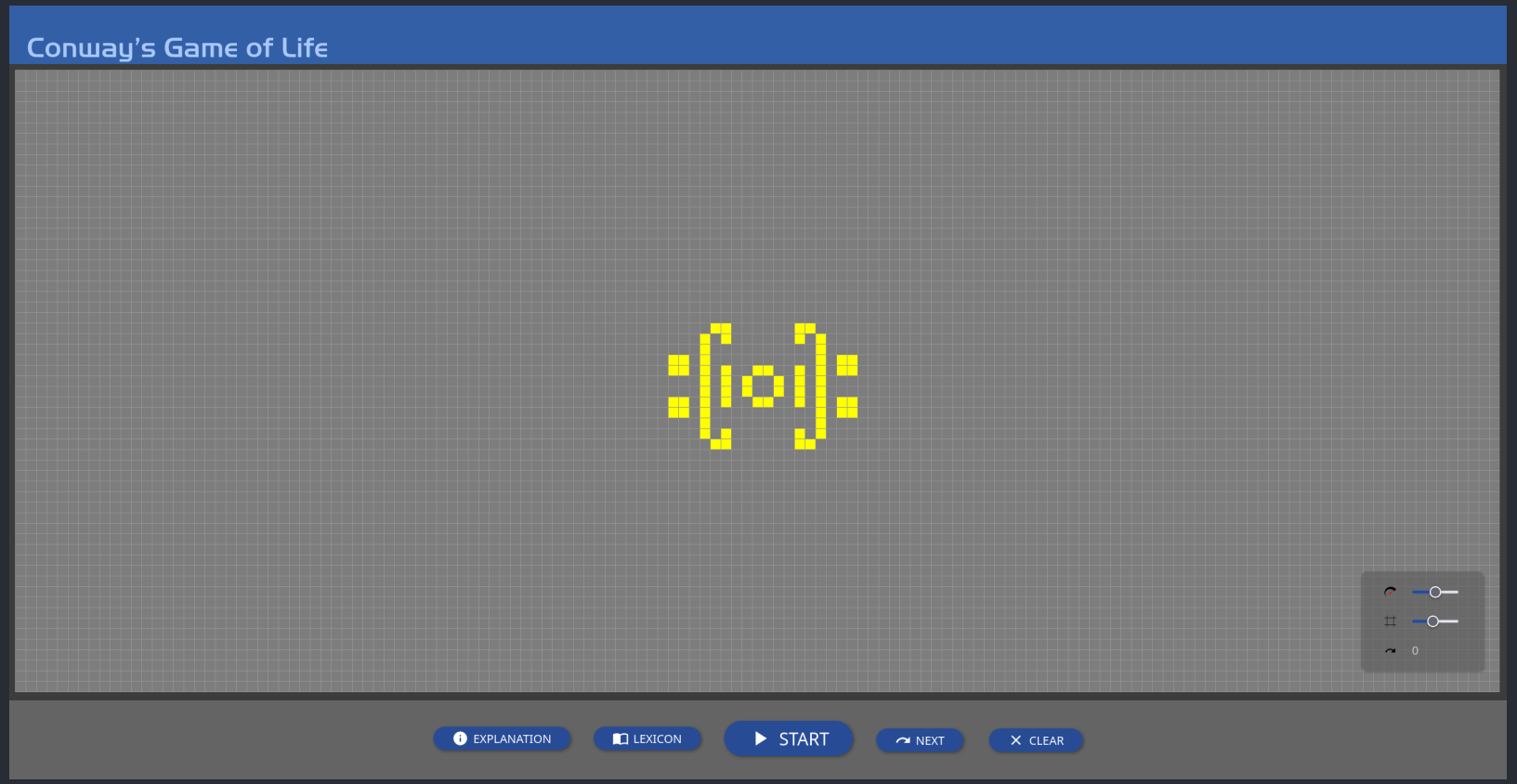
Example of CSS source code

Filename extension .css

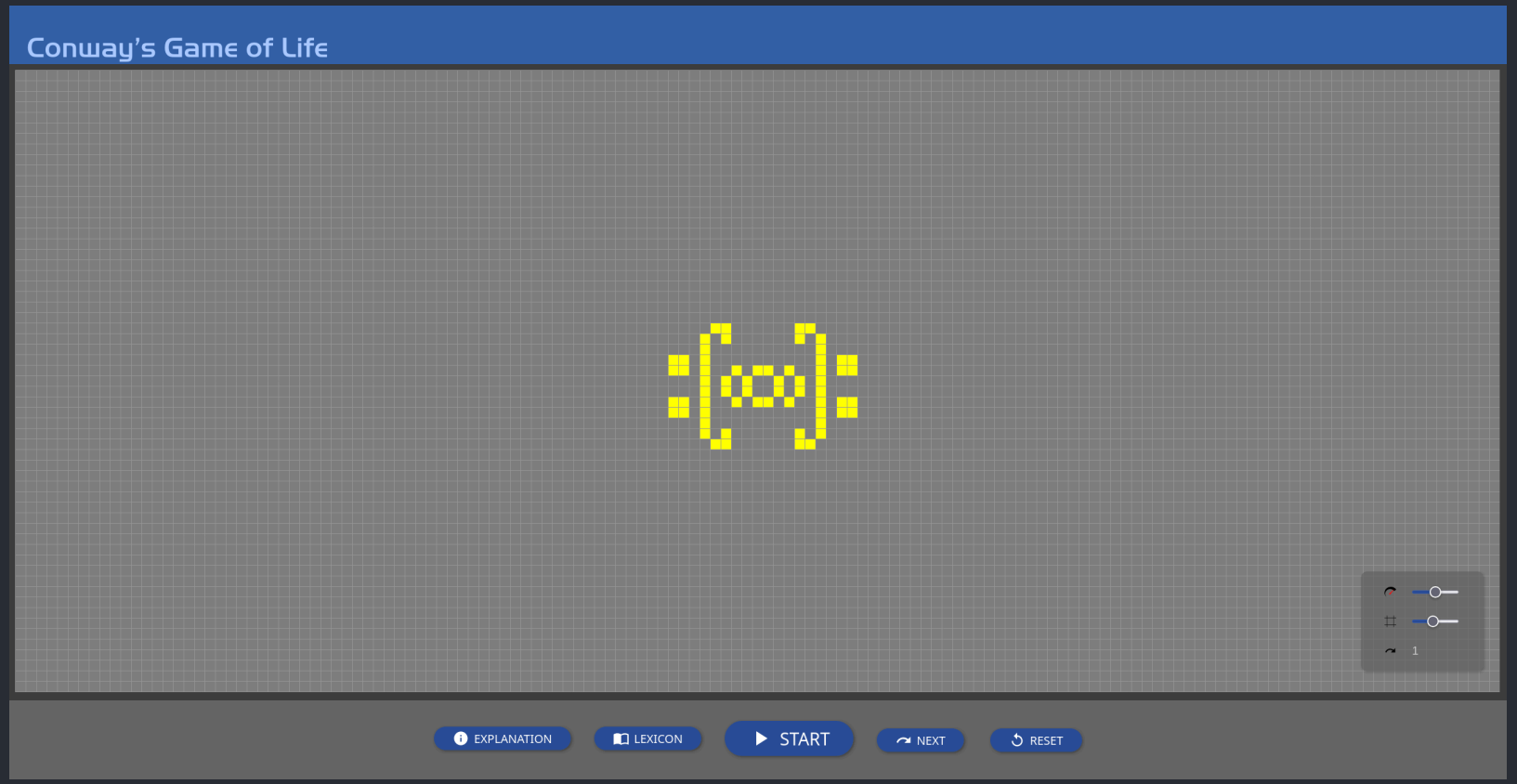
Contents

(Top)
Syntax
Style sheet
Selector
Selector types
Pseudo-classes
Combinators
Summary of selector syntax
Declaration block
Declaration
Properties
Values
Use
Sources
Multiple style sheets
Cascading
CSS priority scheme
Specificity
Examples
Inheritance

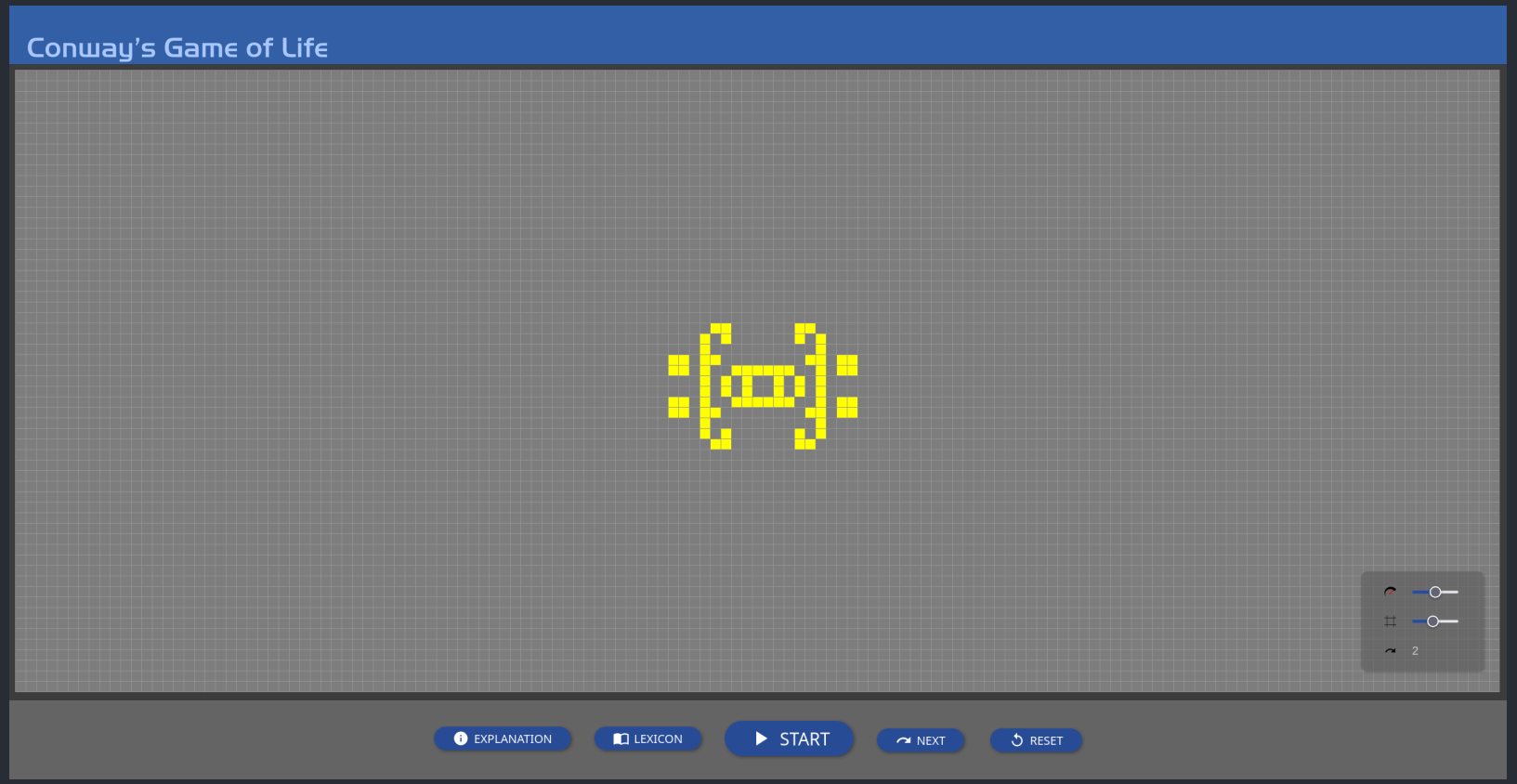
Le Web côté client : JavaScript



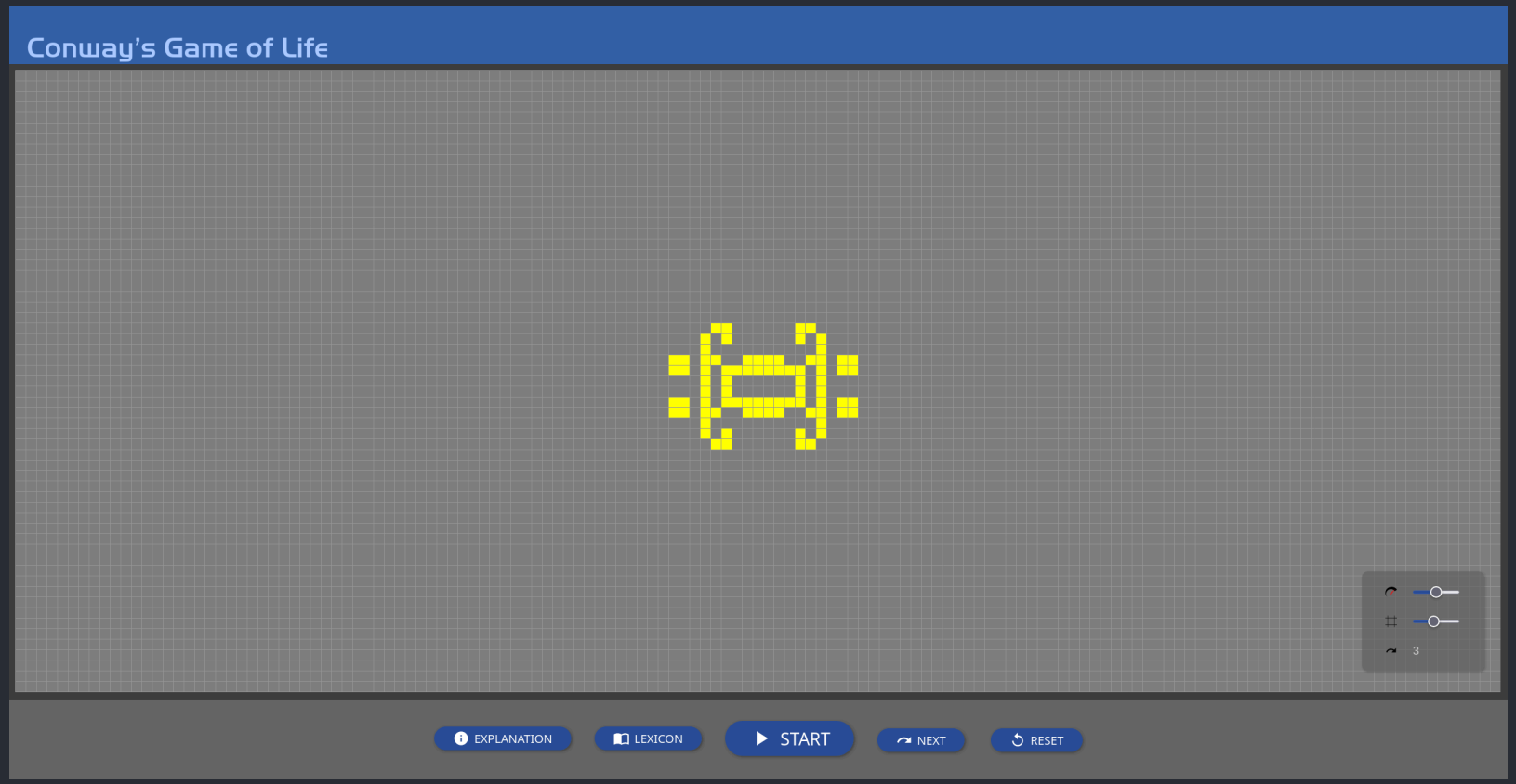
Le Web côté client : JavaScript



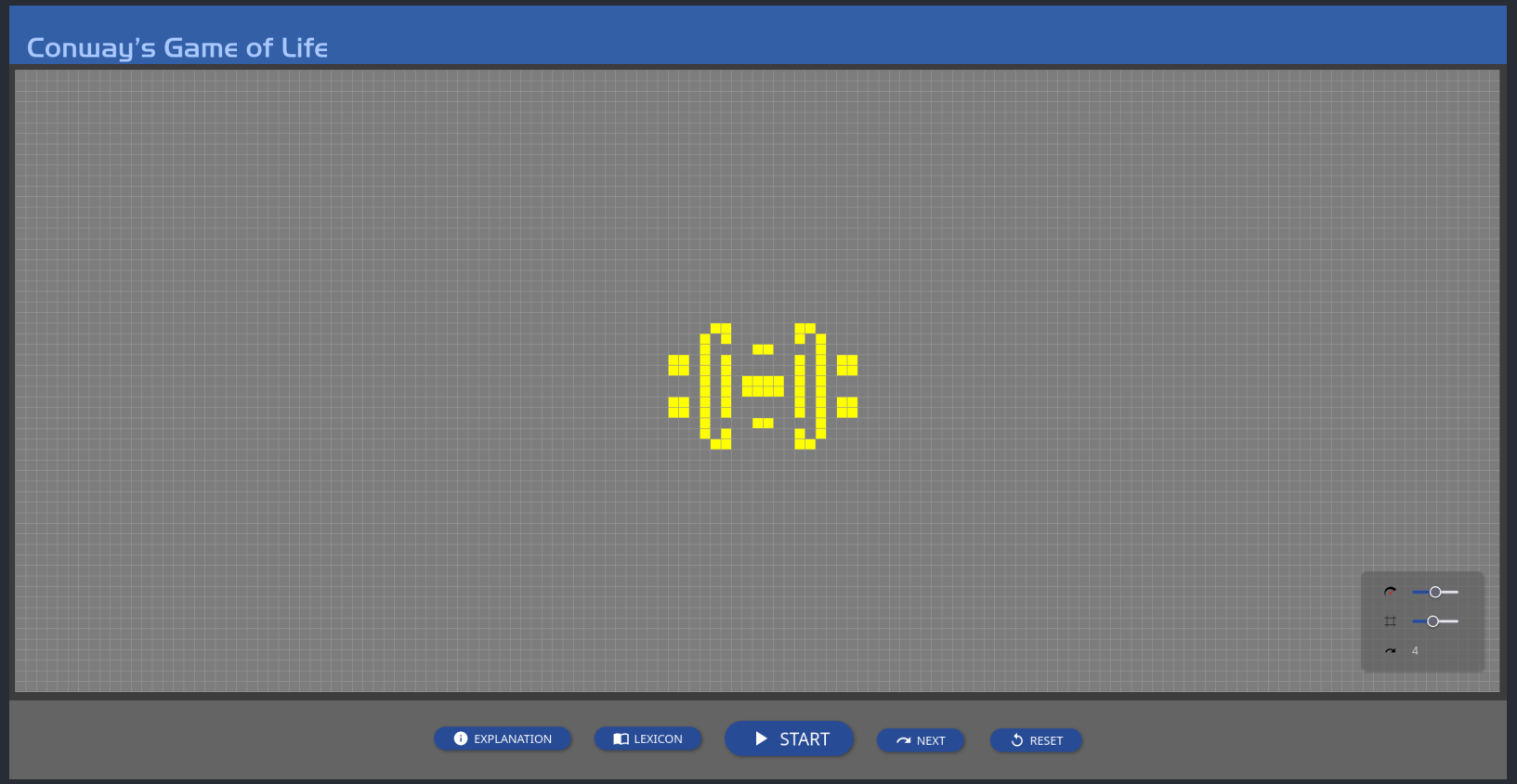
Le Web côté client : JavaScript



Le Web côté client : JavaScript



Le Web côté client : JavaScript



JavaScript comme langage de programmation

Difficile sur plusieurs aspects :

- correction
- sécurité

Volonté de pouvoir utiliser d'autres langages.

Compilation vers JavaScript

→

JavaScript comme cible de compilation intermédiaire

Pour être efficaces, les navigateurs doivent compiler le JavaScript à la volée.

Cela est difficile :

- sémantique complexe
- lent à compiler
- code produit peu efficace

Consensus quant à la nécessité de fournir un nouveau langage :

- rapide
- sûr
- portable
- adapté comme cible de compilation

WebAssembly (Wasm) 1.0

- annoncé en 2015
- disponible depuis 2017
- dans les navigateurs

« The initial version primarily focuses on supporting low-level languages, but we intend to grow it further in the future. »

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,

Compilation vers Wasm 1.0

Dans cette thèse

- Comment compiler OCaml vers Wasm ?
  - Wasocaml
  - Owi : un interpréteur Wasm pour expérimenter avec des extensions non disponibles à l'époque

- Peut-on ré-utiliser cet interpréteur pour analyser du code Wasm ?
  - Owi : un interpréteur Wasm pour ~~expérimenter~~ l'exécution symbolique

Plan de l'exposé

1. introduction à Wasm
2. Wasocaml : compilation d'OCaml vers Wasm
3. introduction à l'exécution symbolique
4. Owi : exécution symbolique de Wasm

1. Introduction à Wasm

Wasm 1.0 (2017)

- langage à pile ;
- types de base simples (`i32`, `i64`, `f32`, `f64`) ;
- typé statiquement (`[ i32 ; f32 ] -> [ i32 ]`);
- fonctions ;
- une mémoire linéaire (un tableau d'octets) ;
- possibilité d'importer et d'exporter des fonctions ;
- une sémantique formelle, sans comportements non définis.

Wasm 2.0 (2022) Non-trapping float-to-int conversion, Sign-extension operators, Multi-value, Bulk memory operations, Fixed-width SIMD…

Wasm 3.0 (2024/2025) Typed Function References, Tail Call, Garbage Collection, Exception handling…

Exemple de programme Wasm

```
(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

  ))
```

Mémoire linéaire

```
(module
  (memory 1) ;; initial size of 1 page
  (func $f (param $addr i32) (result f32)
                    ;; []
    local.get $addr ;; [ addr ]
    i32.const 4     ;; [ 4; addr ]
    i32.mul         ;; [ 4 * addr ]
    f32.load        ;; [ float(memory[4 * addr]) ]
  )
)
```

Interactions avec l'environnement

```
(module
  (import "stdlib" "print_i32" (func $print_i32 (param i32)))
  (func $f
                    ;; []
    i32.const 42    ;; [ 42 ]
    call $print_i32 ;; [] ; 42 is printed
  )
)
```

Gestion du GC avec Wasm 1.0

Pas de solution complètement satisfaisante pour compiler les langages avec GC.

Consensus pour fournir un GC à travers Wasm mais :

- ne pas dégrader les performances du code sans GC
- adapté à une grande diversité de langages sources

L'extension WasmGC :

- 7 ans de travail
- instable durant ma thèse, qui a eu une influence sur l'extension
- maintenant stable et disponible dans les navigateurs

WasmGC : ref i31

- ajout d'un nouveau type `ref i31`
- peut être construit depuis un `i32` et converti en `i32`
- ne peut pas être stocké dans la mémoire linéaire
- pas une référence au sens d'OCaml
- c'est un entier non emballé, similaire au type `int` d'OCaml

```
              ;; []
i32.const 21  ;; [ 21 : i32 ]
ref.i31       ;; [ 21 : ref i31 ]
i31.get_s     ;; [ 21 : i32 ]
```

WasmGC : array & struct

On peut définir des types de tableaux et de structures gérés par le GC.

Ils ne peuvent pas être stockés dans la mémoire linéaire.

```
(type $my_new_type (array (mut (ref i31))))


i32.const 21
ref.i31
i32.const 20
ref.i31
array.new_fixed $my_new_type 2

;; ...
array.get $my_new_type
array.set $my_new_type
```

```
(type $pair (struct
  (field $fst (ref i31))
  (field $snd (mut i64))))

i32.const 21
ref.i31
i64.const 42
struct.new $pair

;; ...
struct.get $pair $fst
struct.set $pair $snd
```

WasmGC : ref eq

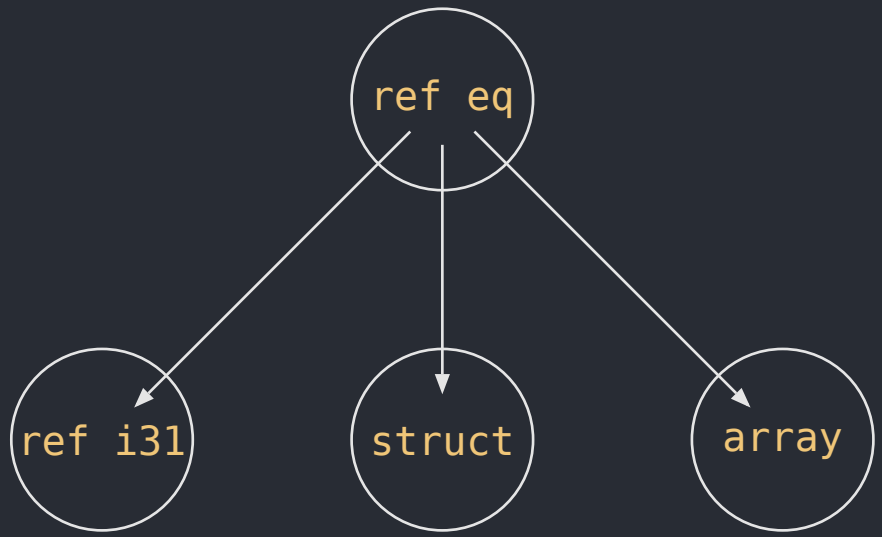
Un type qui peut contenir n'importe lequel des trois précédents :

```
(type $my_array (array i32))

(func $f (param $r (ref eq)) (result i32)
                            ;; []
  local.get $r              ;; [ r : ref eq ]
  ref.cast (ref $my_array)  ;; [ r : my_array ]
  i32.const 0               ;; [ 0 ; r : my_array ]
  array.get $my_array       ;; [ r.(0) : i32 ]
)
```

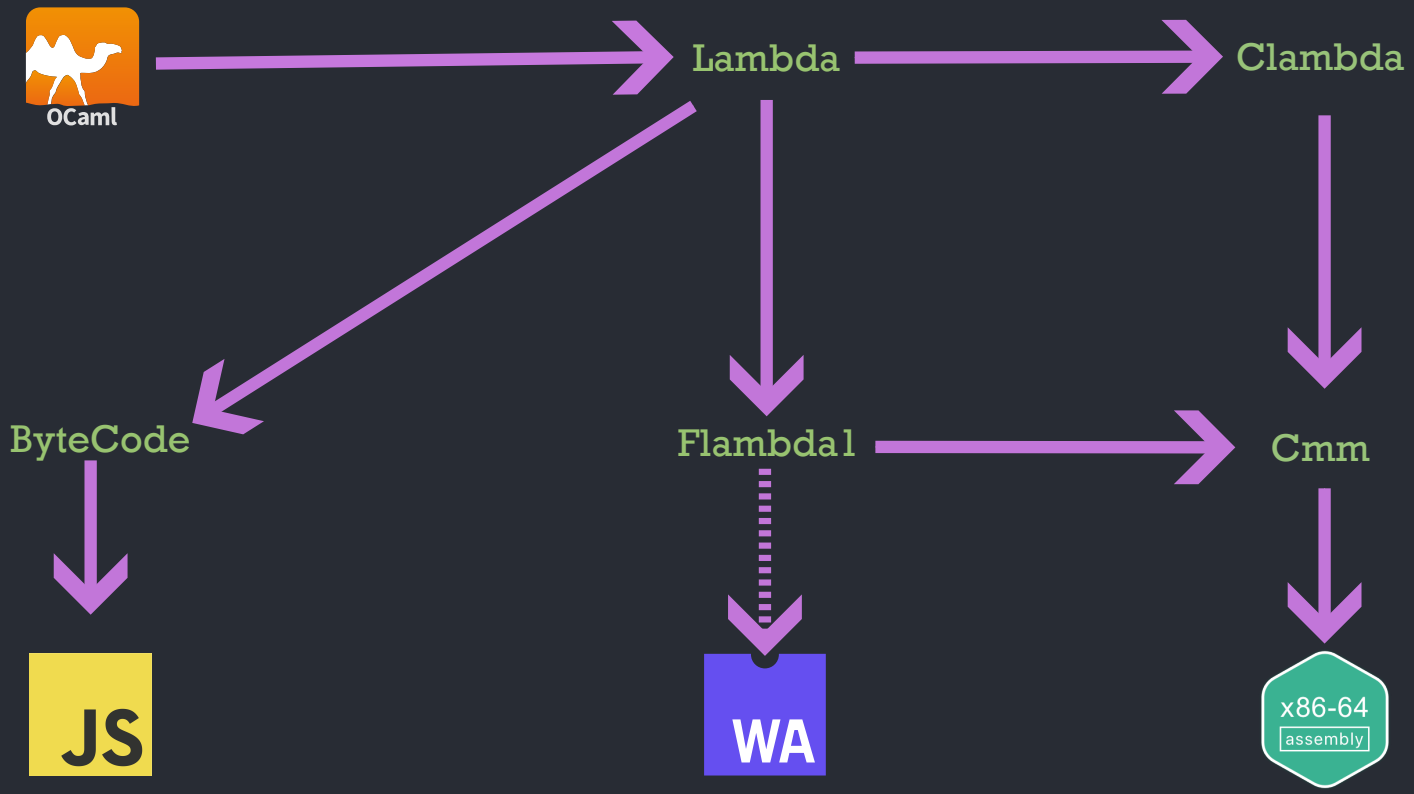
- les conversions vers `ref eq` sont implicites
- les conversions depuis `ref eq` sont explicites et peuvent échouer
- instructions pour tester le résultat d'une conversion

WasmGC : hiérarchie de sous-typage



2. Compilation d'OCaml vers Wasm

Choix du langage source



Flambda1

J'ai conçu la première sémantique formelle pour Flambda1 :

$$\texttt{set\_field}\ \frac{\begin{array}{c} V_0(id_1) = a \qquad M(a) = n', v^* \qquad v^* = v_0, \ldots, v_{m-1} \\ m > n \geq 0 \qquad\qquad V_0(id_2) = v \\ a' = n', v_0, \ldots, v_{n-1}, v, v_{n+1}, \ldots, v_{m-1} \end{array}}{V_0 V^*, M, \texttt{set\_field}\ n\ id_1\ id_2 \rightarrow M[a \leftarrow a'], \texttt{0}}$$

- termes en forme A-normale
- fermetures explicites
- deux types de valeurs
  1. scalaires
  2. blocs de mémoire alloués sur le tas

Représentation des valeurs

- représentation uniforme à travers le type `ref eq`
- petits scalaires (`unit`, `bool`, `char`, `int`) représentés par le type `ref i31`
- comment représenter les blocs de mémoire ?
  1. ~~des structures~~
  2. des tableaux

Type des blocs à la compilation

En Flambda1, le type des blocs n'est pas toujours connu à la compilation.

Lorsque l'on compile `get_field n b`, on sait seulement que `b` est de taille au moins $n + 1$.

On peut faire cette supposition parce que l'on fait confiance au compilateur. Ce n'est pas le cas en Wasm.

Représentation des blocs par des structures

Sous-typage sur les structures :

```
(type $block1 (struct
  (field $tag i8)
  (field $field0 (ref eq))))


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

;; ...
```

Conversion et accès au champ :

```
(func $snd (param $x (ref eq)) (result (ref eq))
                                ;; []
  local.get $x                  ;; [ x: ref eq ]
  ref.cast (ref $block2)        ;; [ x: block2 ]
  struct.get $block2 $field1    ;; [ x.field1: ref eq]
)
```

Compilation d'ocaml-emoji

```
(** ↗ (U+2650): Sagittarius *)
let sagittarius = "\xe2\x99\x90"
(** ⛵ (U+26F5): sailboat *)
let sailboat = "\xe2\x9b\xb5"
(** 🍶 (U+1F376): sake *)
let sake = "\xf0\x9f\x8d\xb6"
(** 🧂 (U+1F9C2): salt *)
let salt = "\xf0\x9f\xa7\x82"
(** 🫡 (U+1FAE1): saluting face *)
let saluting_face = "\xf0\x9f\xab\xa1"
(** 🥪 (U+1F96A): sandwich *)
let sandwich = "\xf0\x9f\xa5\xaa"
(** 🎅 (U+1F385): Santa Claus *)
let santa_claus = "\xf0\x9f\x8e\x85"
```

- les modules sont des blocs
- chaque valeur globale est un champ
- 3782 champs
- type `$block3783`
- « error: too long subtyping chain »
- non spécifié
- maintenant limité à 64

Représentation des blocs par des tableaux

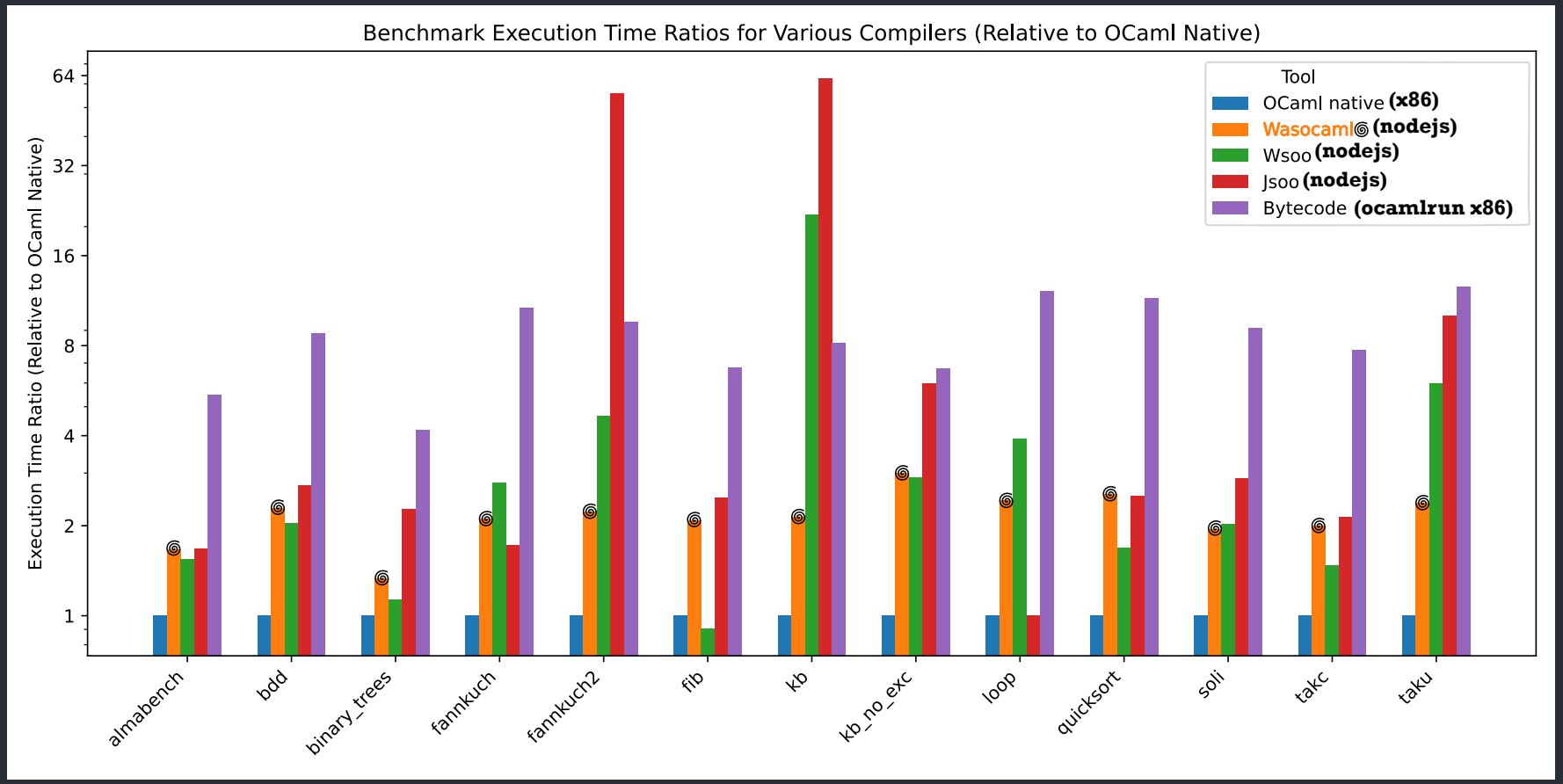
Tout est un tableau de `ref eq` :

- l'étiquette est placée au début du tableau
- vérification des bornes à chaque accès
- impossibilité d'utiliser des types plus précis pour les éléments
- évite le coût du test de sous-typage

```
(type $block (array (ref eq)))

(func $snd (param $x ref eq) (result ref eq)
                          ;; []
  local.get $x            ;; [ x : ref eq ]
  ref.cast (ref $block)   ;; [ x : block ]
  i32.const 2             ;; [ 2 : i32 ; x : block ]
  array.get $block        ;; [ block.(2) : ref eq ]
)
```

Résultats


Benchmark Execution Time Ratios for Various Compilers (Relative to OCaml Native)
Execution Time Ratio (Relative to OCaml Native)
64
32
16
8
4
2
1
Tool
OCaml native (x86)
Wasocaml (nodejs)
Wsoo (nodejs)
Jsoo (nodejs)
Bytecode (ocamlrun x86)
almabench
bdd
binary_trees
fannkuch
fannkuch2
fib
kb
kb_no_exc
loop
quicksort
soli
takc
taku

Résultats

Ratio des temps d'exécutions par rapport au code produit par le compilateur natif.

Outil	Minimum	Maximum	Moyenne arith.	Médiane	Écart-type	Moyenne géom.
Wasocaml	1.34	3.00	2.17	2.13	0.39	2.12
Wsoo	0.91	21.99	4.08	2.04	5.36	2.65
Jsoo	1.00	62.48	11.84	2.51	20.36	4.24
Bytecode	4.17	12.61	8.75	8.82	2.48	8.36

- Wasocaml est un prototype avec de bonnes performances prédictibles
- a convaincu le comité de garder les `ref i31`
- démontre l'adéquation de WasmGC pour les langages fonctionnels
- pas encore de déballage

Extensions nécessaires

Plusieurs extensions nécessaires :

- Typed Function References + WasmGC ;
- Tail Call ;
- Exceptions (optionnel).

Ces extensions sont toutes disponibles dans le navigateur depuis quelques mois. À l'époque, aucun moteur n'implémentait le GC et les appels terminaux à la fois.

Développement d'Owi, un interpréteur Wasm en OCaml, pour expérimenter avec ces extensions.

Owi

Extension	Statut
Import/Export of Mutable Globals	✓
Non-trapping float-to-int conversions	✓
Sign-extension operators	✓
Multi-value	✓
Reference Types	✓
Bulk memory operations	✓
Fixed-width SIMD	❌
Tail calls	✓
Typed Function References	✓
GC	❌
Custom Annotation Syntax in the Text Format	✓
Extended Constant Expressions	✓
Exception handling	❌

- `owi fmt`
- `owi opt`
- `owi run`
- `owi script`
- `owi validate`
- `owi wasm2wat`
- `owi wat2wam`

Dagstuhl Mars 2023

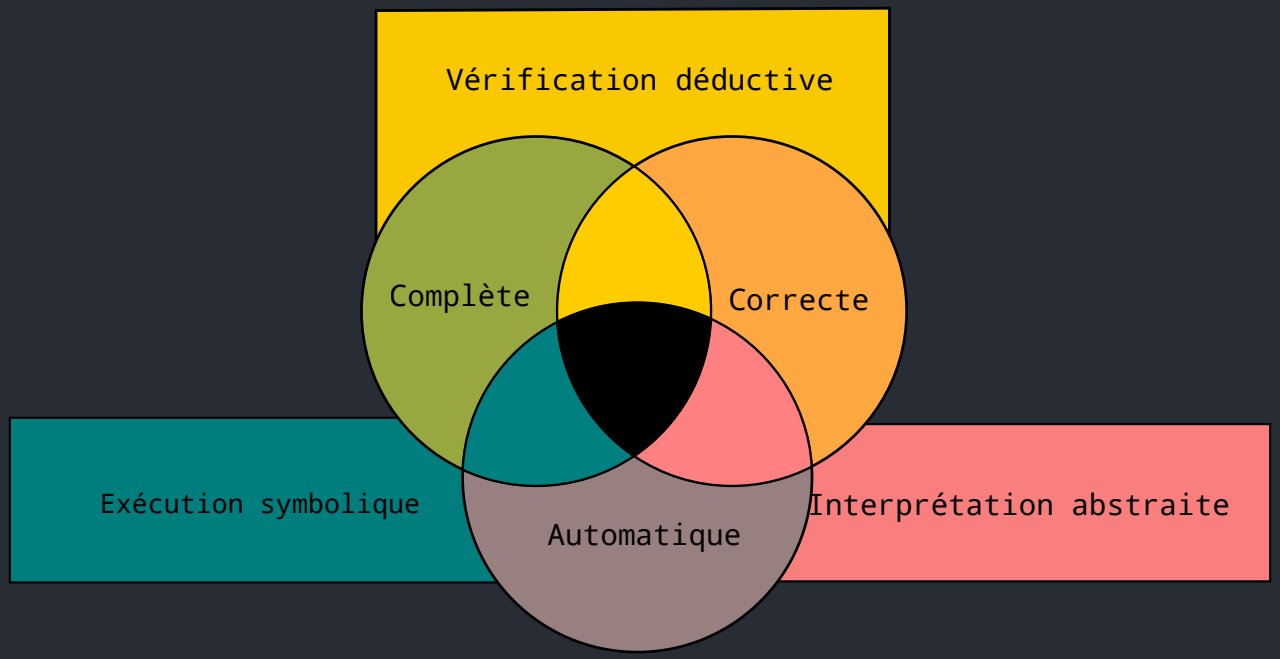
En mars 2023, présentation de Wasocaml à Dagstuhl.

> What about making a symbolic interpreter with Owi?
>
> — José Fragoso Santos (Assistant Professor à Lisbonne)

3. Introduction à l'exécution symbolique

Exécution symbolique

Une méthode pour trouver des bogues dans des programmes.



Principes de l'exécution symbolique

On veut trouver les valeurs d'entrée du programme menant à un bogue.

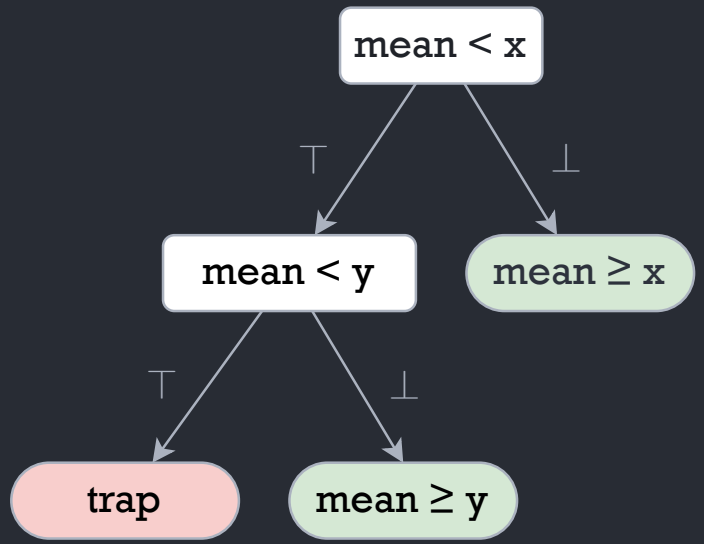
- les valeurs d'entrée sont représentées par des symboles
- on exécute le programme en conservant ces symboles
- à chaque branchement :
  - on explore les deux branches
  - on collecte des informations (condition de chemin)
- si on trouve un bogue, on génère un modèle pour la condition de chemin grâce à un solveur SMT

Ce modèle correspond à des valeurs d'entrées provoquant un bogue.

Arbre d'exécution

```
unsigned int mean(unsigned int x,
                  unsigned int y) {

  unsigned int mean = (x + y) / 2;
  if (mean < x) {
    if (mean < y) { assert(false); }
  }
  return mean;
}
```



Symboles, harnais et modèle

```
unsigned int mean(unsigned int x,
                  unsigned int y) {

  unsigned int mean = (x + y) / 2;
  if (mean < x) {
    if (mean < y) { assert(false); }
  }
  return mean;
}

void harness(void) {
  unsigned int x = symbol_int(),
               y = symbol_int();
  mean(x, y);
}
```

```
Assert failure
model {
  symbol x 2147483650
  symbol y 2147483655
}
```

En effet :

$$\frac{x \oplus y}{2}$$

$$= \frac{2147483650 \oplus 2147483655}{2}$$

$$= \frac{9}{2}$$

$$= 4$$

4. Exécution symbolique de Wasm

Implémentation concrète

Initialement :

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

Comment implémenter une version symbolique en gardant la version concrète ?

Étape 1/2 : abstraire le type des valeurs

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

```
| Binop Add, x :: y :: stack ->
  (Value.add_i32 x y) :: t
| If_else (if_t, if_f), cond :: stack ->
  let cond = Value.bool_of_i32 cond in
  if cond then eval if_t stack
          else eval if_f stack
```

La définition du type `t` varie selon le cas :

- cas concret : une valeur concrète (`42`)
- cas symbolique : une expression symbolique (`x < 42 && y = x || y = 22`)

Étape 2/2 : abstraire sur la stratégie d'exécution

```
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_else (if_t, if_f), cond::stack ->
  let cond = Value.bool_of_i32 cond in
  (* the single new line: *)
  let* cond = Choice.select cond in
  if cond then eval if_t stack
          else eval if_f stack
```

- majorité du code inchangée
- il faut insérer `Choice.select` et `Choice.bind` aux points de branchement

Implémentation de Choice

La définition de `Choice` varie selon les cas :

- cas concret : monade identité
- cas symbolique :
  - évalue les deux branches
  - stocke l'état du programme et la condition de chemin

Implémentation symbolique basée sur trois couches de monades :

- monade d'erreur ;
- monade d'état ;
- monade de coroutines coopératives.

Les branches sont explorées en parallèle grâce à OCaml 5.

Exécution symbolique de code C compilé vers Wasm

```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 main(void) {
  unsigned int x = owi_symbolic_int();
  unsigned int y = owi_symbolic_int();
  owi_assert(mean1(x, y) == mean2(x, y));
}
```

Exécution symbolique de code C

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

Exécution symbolique cross-language

Version C :

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

Version Rust :

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

Exécution symbolique cross-language

Étonnamment, Owi trouve un contre-exemple :

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

Exécution symbolique cross-language

Version C :

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

Version Rust :

```
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.
```

- bibliothèque standard de Rust corrigée
- l'accumulateur initial de `sum()` est maintenant `-0.`
- cela a cassé typst (l'outil utilisé pour faire ces transparents) qui dépendait de ce comportement

Rapidité à détecter des bogues dans du code Wasm

Bibliothèque Wasm d'arbres-B, avec 27 configurations différentes de symboles (peu de symboles vs. beaucoup de symboles).

Outil	Minimum	Maximum	Moyenne
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

Capacité à détecter des bogues dans du code C

1215 programmes C issus de Test-Comp.

Outil	Bogues trouvés	Délai dépassé	Bogues manqués
KLEE	782	368	65
Owi	676	539	0
Symbiotic	489	657	69

Owi :

- ne fait aucune approximation
- ne manque aucun bug si l'analyse termine (modulo les choix faits par le compilateur C quant aux comportements non-définis)
- n'implémente pas encore un modèle mémoire sophistiqué ou une heuristique d'exploration des chemins

Owi comme moteur de programmation par contrainte

```c
#include <owi.h>

int main() {
  int x = owi_symbolic_int();
  int x2 = x * x;
  int x3 = x * x * x;

  int a = 1, b = -7,
      c = 13, d = -8;
  int poly = a*x3 + b*x2 + c*x + d;

  owi_assert(poly != 0);

  return 0;
}
```

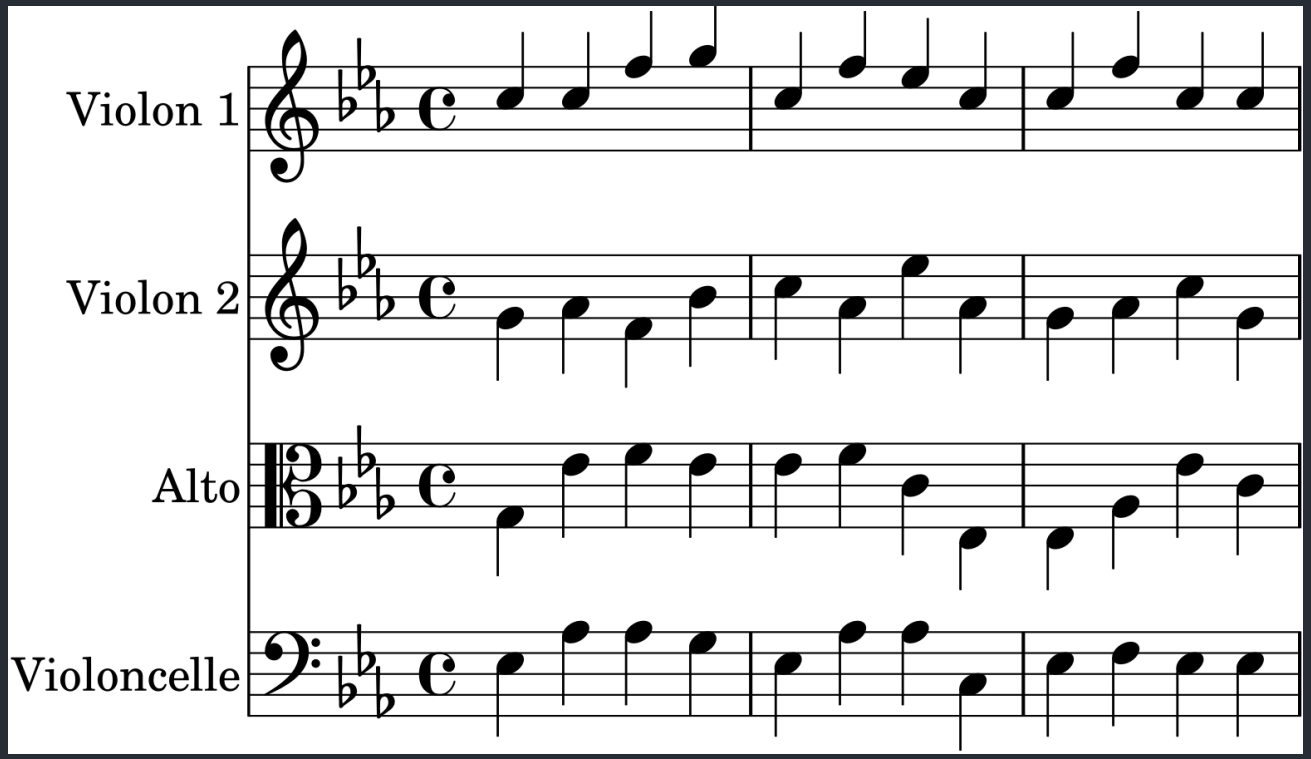
Similaire à Rosette pour Racket (« solver-aided programming ») mais :

- parallèle
- multi-langage
- cross-langage

Utilisé pour :

- résoudre un labyrinthe
- générer des cartes pour le jeu Dobble
- générer des partitions pour quatuor à cordes…

Owi comme moteur de programmation par contrainte



- limite sur l'ambitus
- pas de « croisement »
- pas de mouvement de plus d'une octave
- chaque note appartient à la tonalité
- la sensible doit se résoudre sur la tonique
- les instruments forment des accords
- pas de quinte ou d'octave parallèle

Contributions

Logiciel : Owi (18k lignes d'OCaml), Wasocaml (6k lignes d'OCaml/Wasm)

Publications :

- Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly (The Programming Journal)
- Cross-Language Symbolic Runtime Annotation Checking (JFLA)
- Smt. ml: A Multi-Backend Frontend for SMT Solvers in OCaml (soumis pour publication)
- Wasocaml: Compiling OCaml to WebAssembly (pré-publication)

Exposés : Dagstuhl, IFL'23, Wasm Research Day, OUPS, journée LVP

Encadrement : Dario Pinto (Alternance, 2 ans), Eric Patrizio (M2, 6 mois), Zhicheng Hui (L3, 3 mois)

Conclusion

Ce travail est une collaboration avec Filipe Marques, Arthur Carcano et José Fragoso Santos.

Autres contributions :

- sémantique de Flambda1
- extension Wasm : valeurs gelées
- fuzzing d'interprètes Wasm
- Smt.ml
- symbolic annotation checking
- exécution concolique cross-langage

Travaux futurs :

- heuristiques de parcours dans Owi
- support de WasmGC dans Owi
- porter Wasocaml sur Flambda2
- exécution symbolique d'OCaml
- Wasm et langages dynamiques

Merci !

Bonus

Shadow stack

Plutôt que WasmGC on aurait pu envisager de réécrire l'environnement d'exécution en Wasm, mais :

1. impossible pour le GC de scanner la pile
2. nécessite une shadow stack (choix fait par Haskell), complexe et lent
3. fuites mémoire lors de l'interaction avec le GC de l'hôte

Représentation des fermetures

```
(type $mlfun (sub final
  (func (param eqref) (param eqref) (result eqref))))
;; the captured variables
(type $vars (sub final (array (mut eqref))))
(rec ;; the set of recursive closures
  (type $set_of_closures (sub final
    (struct
      (field $closures (mut (ref null $closures)))
      (field $vars          (ref null $vars))))))
;; the closures
(type $closures (sub final
  (array (mut (ref $closure)))))
;; the type of a single closure
(type $closure (sub final
  (field $fun (ref $mlfun))
  (field $set (mut (ref null $set_of_closures))))))
```

Valeurs gelées 1/3

```
(module
  (rec
    (type $t (struct (field $f (mut (ref null $t))))))

  (func $cycle (result (ref $t))
    (local $l (ref $t))

    ;; l : t = { f = null }
    (local.set $l (struct.new $t (ref.null $t)))
    ;; l.f <- l
    (struct.set $t $f (local.get $l) (local.get $l))

    (local.get $l)))
```

Valeurs gelées 2/3

```
(module
  (rec
    (type $t  (struct (field $f (mut (ref null $t))))))

  (rec
    (type $t' (struct (field $f      (ref      $t')))))

  (func $cycle (result (ref $t'))
    ;; impossible to write for now
  )
)
```

Valeurs gelées 3/3

```
(module
  (rec (type $t
    freezable   (struct (field $f (mut (ref null $t))))))

  (rec (type $t'
    (freeze $t) (struct (field $f      (ref      $t')))))

  (func $cycle_init (result (ref $t)) … )

  (func $cycle (result (ref $t'))
    (ref.freeze $t' $t (call $cycle_init))
  )
)
```

- des constructions supplémentaires pour les variables globales (système de phases)

Fuzzing d'interprètes Wasm 1/2

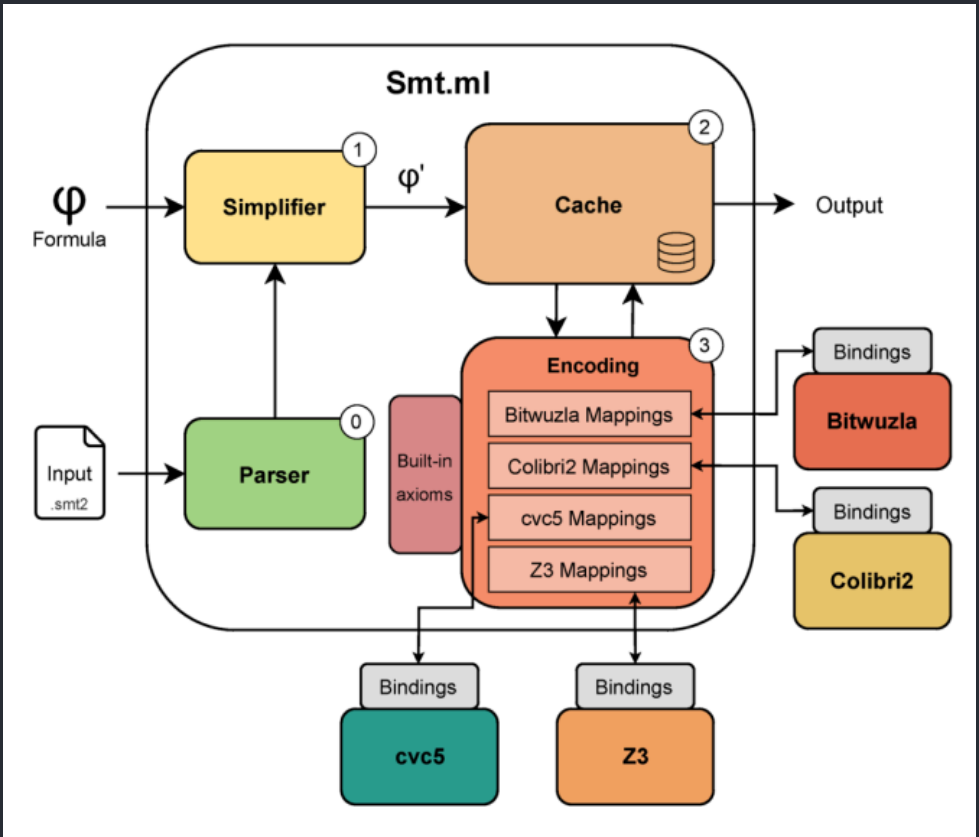
Stage d'Eric Patrizio :

```
let fields env =
  let* memories = list (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
  (* … *)
  memories @ datas @ types @ tables @ elems @ globals @ funcs
```

Fuzzing d'interprètes Wasm 2/2

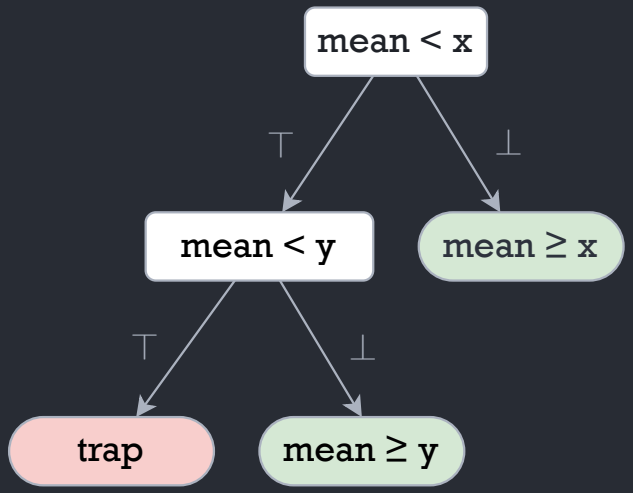
```
let rec expr ~block_type ~stack env =
  Env.use_fuel env;
  if Env.has_no_fuel env then
    (* Drop everything on the stack, then add constants on the stack *)
  else
    let instr_available =
      match stack with
      | Num_type I32 :: Num_type I32 :: Num_type I32 :: _tl ->
        (* all instructions using 0 to 3 integers *)
      | Num_type I32 :: Num_type I32 :: _tl ->
        (* all instructions using 0 to 2 integers *)
      (* … *)
    in
    let* instr, stack_ops = choose instr_available in
    let stack = S.apply_stack_ops stack stack_ops in
    let+ next = expr ~block_type ~stack env in
    instr :: next
```

Smt.ml



- indépendance aux solveurs (API commune)
- optimisations (simplification des expressions, cache avec hash-consing)
- facilitié d'utilisation (plus de typage)
- mode incrémental

Exécution concolique



- on part de valeurs aléatoires pour les symboles
- on maintient la version concrète et symbolique
- plus besoin d'appeler les solveur à chaque branche
- on collecte malgré tout les conditions de chemin
- si on trouve un bug, on s'arrête
- sinon, on recommence avec un modèle généré par le SMT dont on sait qu'il va mener à une nouvelle branche de l'arbre des exécutions

Des annotations à l'exécution symbolique 1/2

Stage de Zhicheng Hui :

```
/*@ requires 2 <= n <= MAX_SIZE;
    requires \valid(is_prime + (0 .. (n - 1)));
    ensures
      \forall integer i; 0 <= i < n ==> ... */
void primes(bool *is_prime, int n) {
  for (int i = 0; i < n; ++i) { is_prime[i] = true; }
  for (int i = 2; i * i < n; ++i) {
    if (!is_prime[i]) continue;
    for (int j = i*i; j < n; j += i) {
      is_prime[j] = false; } } }
```

Option `owi c --e-acsl` pour d'exécuter symboliquement du code annoté par des spécifications en générant des assertions via E-ACSL et notre variante du runtime E-ACSL.

Des annotations à l'exécution symbolique 2/2

Nous avons fait la même chose en Wasm :

```
(module

  (@contract $sum
    (ensures (= result (+ $p1 (+ $p2 (+ $p3 $p4))))))
  (func $sum (param $p1 i32) (param $p2 i32) ...
    ;; ...
  ))
```

Conception de Weasel (WEbAssembly SpEcification Language), le premier langage de spécification conforme au standard. On génère des assertions à partir de celui-ci, comme le fait E-ACSL. Permet par exemple de vérifier des runtimes écrits en Wasm.