Symbolic Annotation Checking

Zhicheng Hui & Léo Andrès

OCamlPro

31 janvier 2025

Plan de l'exposé

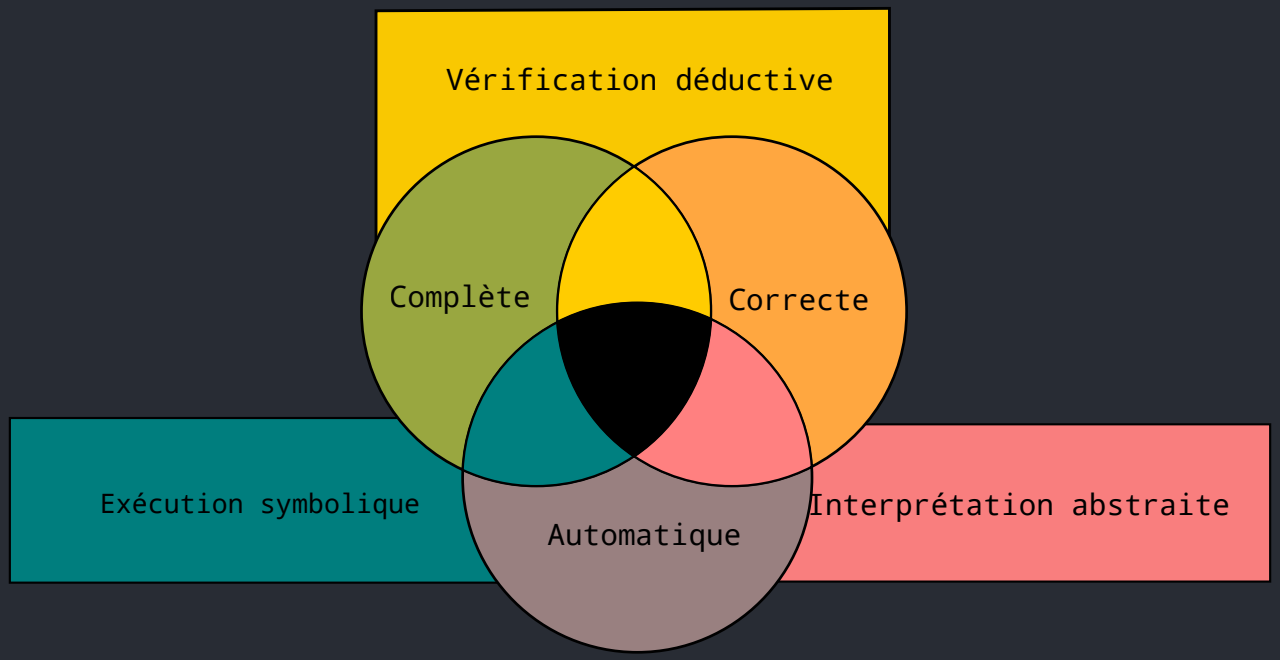
1. introduction à l'exécution symbolique
2. introduction à Wasm
3. Owi : exécution symbolique de Wasm
4. ACSL dans l'exécution symbolique
5. Weasel

1. Introduction à l'exécution symbolique

Exécution symbolique

Une méthode pour :

- trouver des bogues dans des programmes ;
- implémenter le « solver-aided programming » ;
- la génération de tests.



Principes de l'exécution symbolique

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

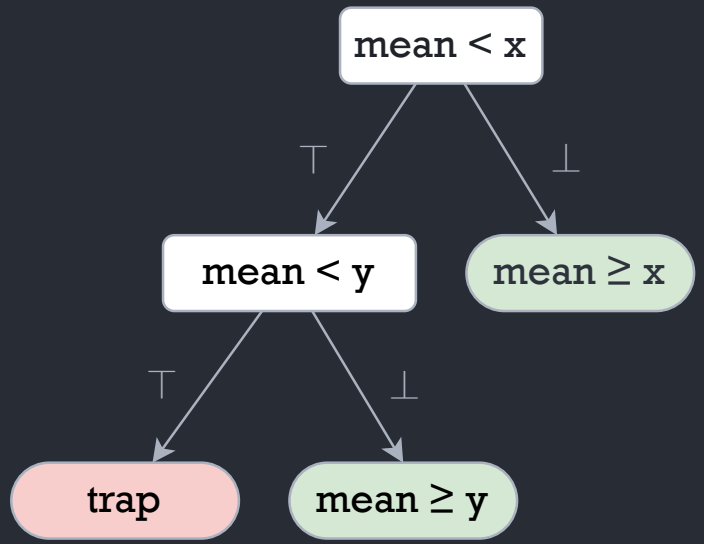
- les valeurs d'entrée sont représentées par des symboles
- on exécute le programme en conservant ces symboles
- à chaque branchement :
  - on explore les deux branches
  - on collecte des informations (condition de chemin)
- si on trouve un bogue, on génère un modèle pour la condition de chemin grâce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

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

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

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

$$= 4$$

2. Introduction à Wasm

WebAssembly (Wasm)

- annoncé en 2015
- disponible depuis 2017 dans les navigateurs

- un complément de JavaScript
- une cible de compilation
- rapide, sûr et portable
- le standard = sémantique à petits pas
- pas de comportement indéfini
- utilisé pour des serveurs ou de l'embarqué

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,

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

  ))
```

3. Owi: exécution symbolique de Wasm

Owi concret

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`

Owi symbolique

On a paramétré l'interpréteur concret par une monade de choix, qui permet d'explorer toutes les branches, (et en parallèle grâce à OCaml 5).

On peut faire de l'exécution symbolique de code Wasm !

Compilation vers Wasm

Exécution symbolique de code C

```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!
```

4. ACSL dans l'exécution symbolique

Langage de spécification

L'approche précédente présente certains inconvénients :

- il mélange la logique de la vérification avec la logique du programme
- il intègre la logique de la vérification dans la logique du programme
- il nous demande à écrire des codes de harnais à la main
- peu modulaire

Nous utilisons des langages de spécification.

- ACSL pour le code C
- Weasel pour le code Wasm

Le langage de spécification ACSL

Le langage de spécification ANSI/ISO C (ACSL) est un langage de spécification pour les programmes C.

La notion primordiale dans ACSL est le contrat de fonction:

```
/*@ requries precondition
    ensures  postcondition
*/
```

d'autres notions incluent l'assertion, l'invariant de boucle, l'invariant de donnée, etc.

Le langage de spécification E-ACSL

Sous-ensemble exécutable du ACSL, aussi un plug-in du Frama-C

```
/*@ 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(...); }
}
```

Intégration du E-ACSL dans Owi

Nous avons:

- réutilisé le générateur du code de E-ACSL
- réimplémenté les fonctions de la bibliothèque E-ACSL en utilisant les primitives d'Owi

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

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.

Optimisation

Le générateur de code d'E-ACSL utilise parfois des boucles :

```
/*@ assert \forall integer i;
    0 <= i < 100 ==> num[i] == 0; */
```

```
int __gen_e_acsl_forall = ...;
while (1) {
  if (__gen_e_acsl_i < 100) ; else break; {
    __e_acsl_assert(__gen_e_acsl_i < 100);
    __e_acsl_assert(0 <= __gen_e_acsl_i);
    if (num[__gen_e_acsl_i] == 0) ;
    else { __gen_e_acsl_forall = 0;
      goto e_acsl_end_loop1; } }
  __gen_e_acsl_i ++; }
e_acsl_end_loop1: ;
__e_acsl_assert(__gen_e_acsl_forall);
```

Grâce aux symboles on pourrait faire mieux :

```
int i = owi_i32();
owi_assume(0 <= i && i < 100);
owi_assert(num[i] == 0);
```

Permet de minimiser les branchements (important pour l'exécution symbolique)

La contrainte de quantification bornée

E-ACSL impose des contraintes sur les quantifications. Par exemple, le quantificateur universel doit être borné :

```
/*@ assert \forall integer i;
    0 <= i < 100 ==> p(i); */
```

Ce qui va générer du code avec une boucle.

Grâce à l'utilisation de symboles, cette contrainte pourrait être relâchée :

```
/*@ assert \forall integer i;
    p(i); */

// ...

int i = owi_symbol_i32();
owi_assert(p(i));
```

5. Weasel

Le langage de spécification Weasel

Nous avons fait la même chose en 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
  ))
```

Conception de Weasel (WEbAssembly SpEcification Language), le premier langage de spécification conforme au standard.

A nécessité l'implémentation de l'extension « custom annotation syntax » dans Owi.

Génération d'assertions à partir de Weasel

On a ajouté un générateur d'assertions exécutables qui travaille à partir des annotations :

```
(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)
)
```

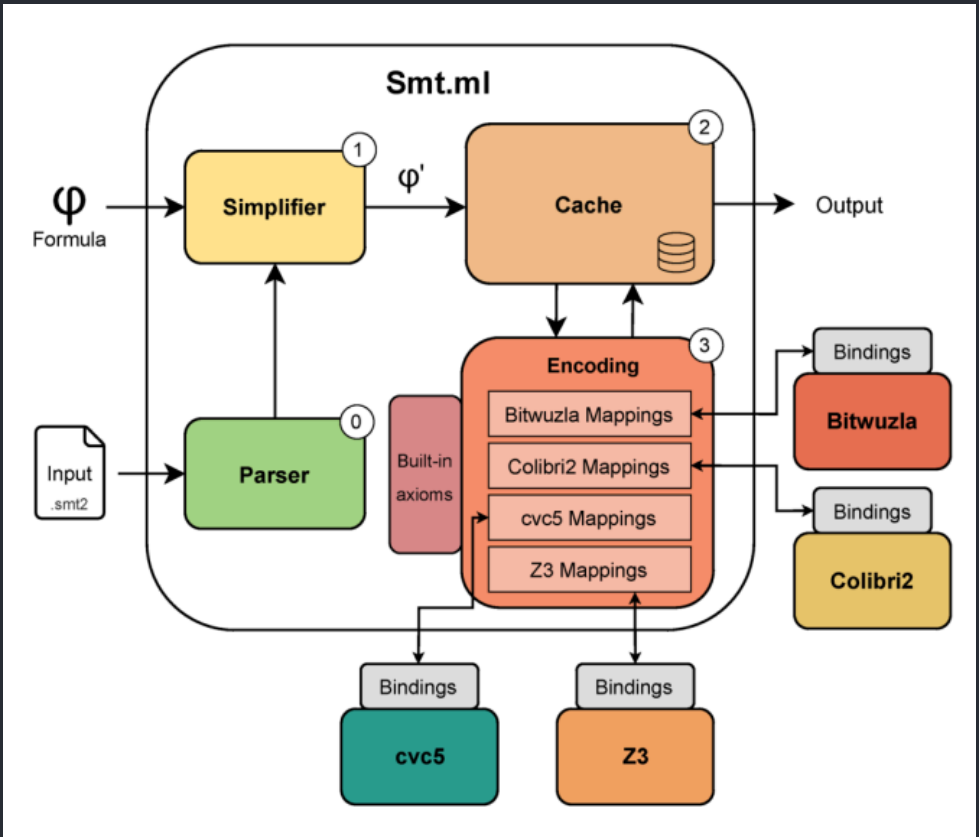
```
(import "symbolic" "assert"
  (func $assert (param i32)))
(func $__weasel_plus_three (param $n i32)
(result i32) (local $__weasel_temp i32)
(local $__weasel_res_0 i32)
  (call $plus_three (local.get 0))
  local.set 2
  (i32.eq (local.get 2) (i32.add (local.get
0) (i32.const 3)))
  call $assert
 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

Travaux futurs

- Étendre le langage Weasel, notamment à une gamme plus large de constructions Wasm
- Vérifier le programme combinant plusieurs langages, par exemple une bibliothèque optimisée écrite en Wasm, embarquée dans un algorithme C
- Appliquer cette approche à d'autres langages, par example Rust (en utilisant Pearlite/Creusot)
- Une fois qu'on aura implémenté le GC dans Owi et qu'on pourra faire de l'exécution symbolique d'OCaml, on pourra réutiliser notre approche avec Gospel et Ortac !

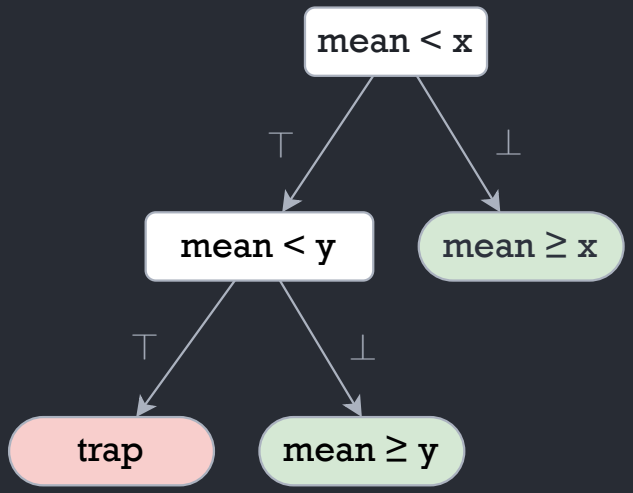
Bonus

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

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;
}
```

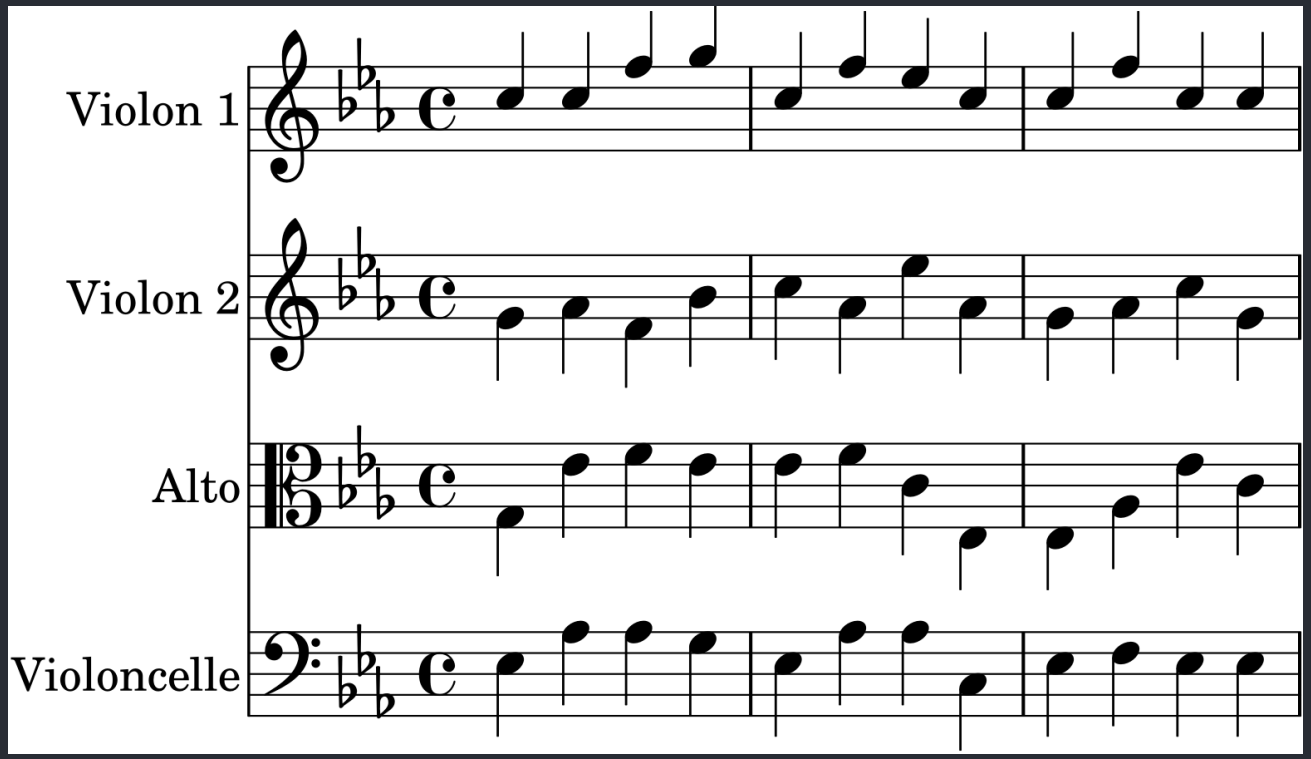
Similaire à Rosette pour Racket (« solver-aided programming ») mais :

- 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

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

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.

Autre optimisation du code

Le générateur du code de E-ACSL calcule une expression en la décomposant en plusieurs étapes respectant sa structure.

```
int __gen_e_acsl_implies;                    // Assertion:
if (! (P != 0)) __gen_e_acsl_implies = 1;    /*@ assert P ==> (Q && R); */
else {
  int __gen_e_acsl_and;
  if (Q != 0) __gen_e_acsl_and = R != 0;     __e_acsl_assert(!P || (Q && R));
  else __gen_e_acsl_and = 0;
  __gen_e_acsl_implies = __gen_e_acsl_and;
}
__e_acsl_assert(__gen_e_acsl_implies);
```

La seconde version peut rendre l'exécution symbolique plus efficace, car elle admet moins de variables et branchements.