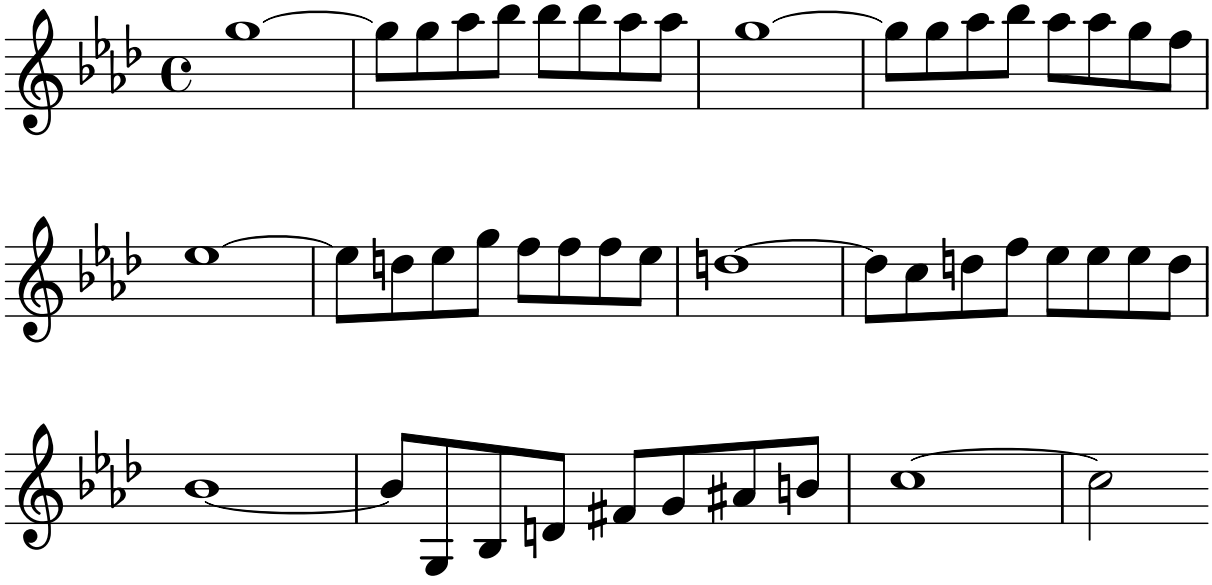
# Exécution symbolique pour tous
# ou
# Compilation d'OCaml vers WebAssembly

Léo Andrès

24 septembre 2024

Le travail présenté ici a été supervisé par mes directeurs de thèse, Pierre Chambart et Jean-Christophe Filliâtre. Il a été réalisé au Laboratoire Méthodes Formelles (Gif-sur-Yvette) et à OCamlPro (Paris). Trois ans ont été nécessaires pour le mener à bien, d'octobre 2021 à septembre 2024.



...

Yom et Wang Li, *Vegetal Love*.

**Résumé**

Les limitations de JavaScript en tant que langage par défaut du Web ont conduit au développement de WebAssembly (Wasm), un langage sûr, efficace et modulaire. Toutefois, compiler des langages à glaneur de cellules vers Wasm ne se fait pas sans peine, notamment du fait de la nécessité de réécrire le moteur d'exécution ou de la gestion des interactions avec le glaneur de cellules de l'hôte (le navigateur). Des extensions, dont WasmGC, ont été développées par les groupes de travail Wasm pour faciliter cette tâche. Nous présentons Wasocaml, le premier compilateur d'OCaml vers WasmGC. Ce projet confirme l'adéquation de la proposition WasmGC pour des langages fonctionnels et a influencé son développement. Les stratégies de compilation mises en œuvre dans Wasocaml peuvent également être appliquées à d'autres compilateurs et langages ; deux compilateurs les ont d'ailleurs déjà adoptées.

Cependant, Wasm, bien qu'initialement conçu pour les applications web, est devenu une alternative sérieuse pour les environnements serveurs et les systèmes embarqués, en raison de ses avantages en termes de performances et de sécurité. Cependant, des vulnérabilités comme les débordements de tampon et les fuites de mémoire subsistent, ce qui peut conduire à des vulnérabilités. Pour répondre à cette problématique, nous introduisons Owi, un interpréteur symbolique pour Wasm, écrit en OCaml. Owi est basé sur un interpréteur modulaire et monadique capable d'exécutions concrètes et symboliques de programmes Wasm. Grâce à cette architecture, nous avons développé un outil de détection de bogues performant qui, selon notre évaluation, est le meilleur actuellement disponible pour Wasm.

En outre, puisque Wasm est une cible de compilation pour de nombreux langages, Owi peut être utilisé pour détecter des bogues dans dans des programmes issus d'autres langages tels que C et Rust, ou dans des projets mêlant les deux. Nos expériences, basées sur les benchmarks Test-Comp 2024, montrent qu'Owi offre des performances comparables à celles des outils représentant l'état de l'art comme KLEE, avec certains avantages dans des scénarios spécifiques où les approximations de KLEE peuvent entraîner des faux négatifs. Contrairement aux autres outils d'exécution symboliques qui ont une approche ad-hoc, le notre présente l'avantage d'être facilement utilisable pour tout langage possédant un compilateur vers Wasm.

Enfin, nous montrons également comment Owi peut être utilisé à d'autres fins que la recherche de bogues, par exemple la programmation par contraintes. En particulier, Owi permet, tout comme son moteur d'exécution symbolique, d'ajouter des capacités de programmation par contraintes à n'importe quel langage possédant un compilateur vers Wasm, et ce d'une manière directe et modulaire.

# Remerciements

Cette page est intentionnellement laissée vide.

# 1 Introduction

À l'aube du XXI^e siècle, Tim Berners-Lee et Robert Cailliau inventent le Web [23], un réseau de documents interconnectés. À ses débuts, le Web ne permettait que l'affichage de texte et de liens entre pages, comme illustré à la figure 1.1. Rapidement, il évolue pour permettre l'intégration d'images, tout en s'appuyant sur le langage HTML pour structurer les informations.

L'introduction du CSS, un second langage, permet alors d'améliorer la présentation visuelle des pages, comme le montre la figure 1.2. Toutefois, à ce stade, les pages Web restent *statiques* : une fois chargées, leur contenu ne peut être modifié.

Cela va changer en 1995 avec l'apparition de JavaScript [120], un véritable langage de programmation qui permet de rendre les pages Web *dynamiques*. Grâce à lui, les pages peuvent réagir aux interactions des utilisateurs, modifier leur contenu en temps réel et interagir avec des serveurs pour actualiser des données sans rechargement complet de la page. Bien que conçu en seulement dix jours, JavaScript est devenu l'épine dorsale des interactions dynamiques sur le Web.

Depuis lors, les trois langages fondamentaux du Web — HTML, CSS et JavaScript — ont évolué et les navigateurs qui les supportent sont devenus des logiciels parmi les plus complexes au monde. Par exemple, Chromium et Firefox comptent chacun plus de 30 millions de lignes de code [113, 165, 167], ce qui les classe parmi les plus grandes bases de code jamais développées. Les moteurs JavaScript, comme V8 (Chrome) et SpiderMonkey (Firefox), comptent à eux seuls entre 500 mille et 3 millions de lignes de code [166, 104] et sont devenus des composants centraux d'une grande complexité, principalement à cause des spécificités de JavaScript.

JavaScript peut être considéré comme un *accident historique*, non adapté à l'usage qui en est fait aujourd'hui. En dépit de ses défauts, il s'est imposé comme la norme, et une fois adopté à grande échelle, il était trop tard pour revenir en arrière. Cependant, en 2015, un consensus s'est formé parmi les principaux acteurs du Web : il était temps d'introduire un nouveau langage plus performant et plus sûr. Ce langage, appelé WebAssembly (Wasm) [91], a été conçu comme une cible de compilation. Contrairement à JavaScript, Wasm n'est pas conçu pour être écrit directement par les développeurs, mais pour être généré automatiquement par des *compilateurs* depuis des programmes écrits dans d'autres langages.

Le premier standard de Wasm visait principalement les langages statiques sans gestion automatique de la mémoire, comme C, C++ et Rust. Cependant, le support efficace de langages avec glaneur de cellules, tels que Java ou OCaml, passe par l'intégration d'un glaneur de cellules

# 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.

FIGURE 1.1 – Une des premières pages Web.


FIGURE 1.2 – Une vieille page Web utilisant du CSS et des images.

dans Wasm. Sa conception soulève plusieurs questions techniques :

1. Comment permettre la compilation efficace des langages avec glaneur de cellules tout en évitant d'affecter les performances des langages ne faisant pas usage de cette fonctionnalité ?
2. Comment définir un ensemble de primitives de gestion mémoire qui soit compatible avec une large gamme de langages tout en restant efficace ?

Afin de répondre à ces questions, une extension appelée WasmGC [96] a été proposée. Au moment où cette thèse a débuté, cette extension entrait dans une phase de stabilisation après cinq années de développement intense. Nous avons activement contribué à ce processus en étant les premiers à démontrer que cette extension était adaptée à un langage fonctionnel. Notre travail a également mis en évidence l'importance de certaines fonctionnalités, initialement menacées d'être retirées, et a permis de prendre des décisions clés en vue de la stabilisation complète de WasmGC, qui a finalement été finalisée fin 2023. En outre, nous avons proposé des améliorations futures visant à optimiser les performances de cette extension.

Au début de cette thèse, aucune mise en œuvre de WasmGC n'était disponible. L'extension était trop instable pour les navigateurs et même l'interpréteur de référence ne permettait pas d'expérimenter de façon suffisante. Nous avons donc implémenté notre propre interpréteur (Owi), afin de pouvoir facilement expérimenter avec cette extension et d'autres dont nous savions qu'elles seraient nécessaires, telles que les appels terminaux.

En raison de ses performances accrues, Wasm est rapidement devenu un succès et son usage s'est étendu au-delà du Web, notamment dans les environnements de serveurs et commence même à émerger dans les systèmes embarqués. Certains de ces environnements sont soumis à des contraintes de sécurité plus fortes que le Web et il est impératif d'y garantir la correction des programmes compilés en Wasm, en particulier ceux issus de la combinaison de différents langages de programmation.

C'est alors que nous avons profité de l'existence d'Owi pour développer un outil permettant la détection de bogues dans des programmes Wasm. En effet, étendre un interpréteur existant pour effectuer de l'*exécution symbolique* nous paraissait intéressant. Rapidement, nous nous sommes aperçus que Wasm était en réalité en langage particulièrement adapté à l'exécution symbolique. Il s'agit d'une méthode permettant de vérifier la correction d'un programme. Il en existe d'autres et elles peuvent être classifiées en fonction de trois propriétés fondamentales :

**Automatisation** Cette propriété mesure si la preuve peut être entièrement automatisée par l'outil utilisé. Si ce n'est pas le cas, l'intervention du programmeur est nécessaire pour fournir des éléments complémentaires et guider le processus.

**Correction** La correction d'une méthode signifie qu'elle garantit que l'analyse couvre toutes les exécutions possibles du programme. Ainsi, si des bogues existent, ils seront tous détectés. Cette méthode évite donc les faux négatifs, mais peut produire des faux positifs.

**Complétude** La complétude indique que la méthode ne signale que des bogues réels, sans en inventer. Cependant, elle peut manquer certains bogues, produisant ainsi des faux négatifs, mais aucun faux positif.

Divers théorèmes mathématiques, comme ceux de Rice [2] et Gödel [1], démontrent qu'il est impossible de concevoir une méthode de preuve qui satisfasse simultanément ces trois

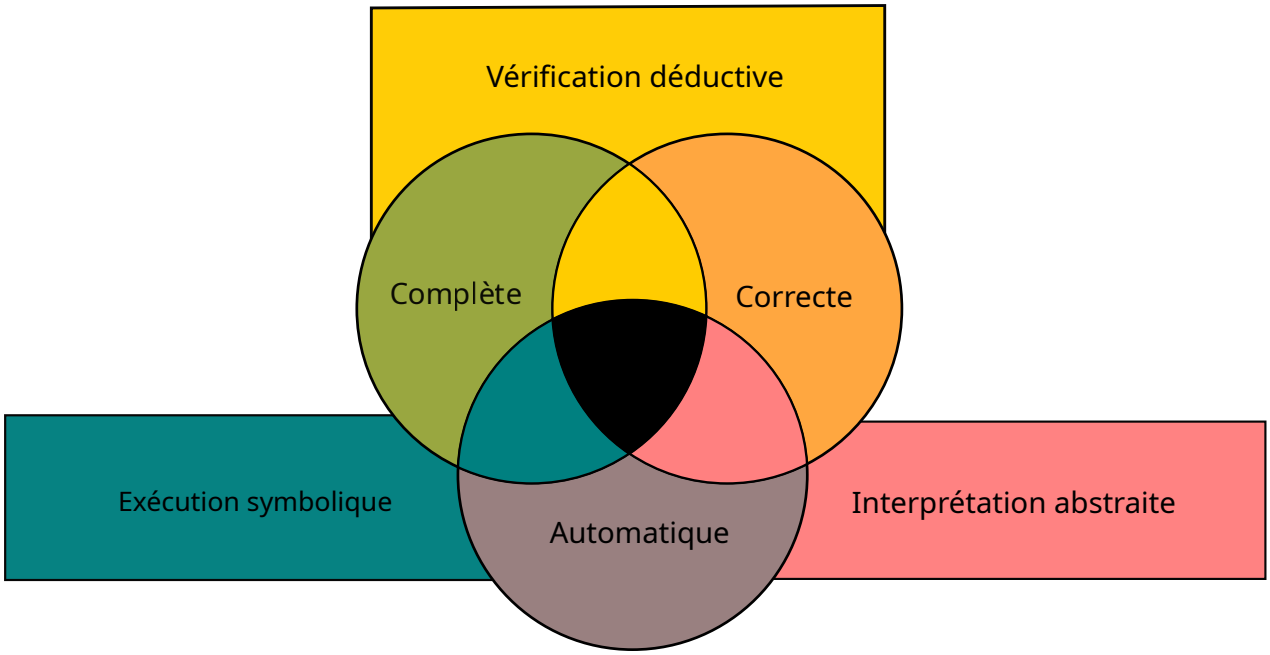



FIGURE 1.3 – Les propriétés des différentes méthodes de preuve de programme.

propriétés. Seules deux d'entre elles peuvent être garanties à la fois. Différentes méthodes existent, chacune répondant à un sous-ensemble de ces propriétés :

- l'*interprétation abstraite*, qui est automatique et correcte, mais non complète ;
- l'*exécution symbolique*, qui est automatique et complète, mais non correcte ;
- la *vérification déductive*, qui est correcte et complète, mais non automatique.

La figure 1.3 illustre la répartition de ces propriétés entre les différentes méthodes de preuve.

**L'interprétation abstraite** Proposée par Cousot et Cousot en 1977 [6], l'interprétation abstraite est une méthode automatique et correcte. Cependant, elle n'est pas complète, ce qui signifie qu'elle peut signaler des bogues qui n'existent pas réellement, issus des approximations effectuées par l'analyse. Cette méthode est donc moins adaptée à la recherche de bogues, car le tri entre les faux positifs et les véritables bogues peut s'avérer fastidieux. Néanmoins, il arrive sur certains programmes que l'approximation ne mène pas à des faux positifs. L'enjeu dans l'implémentation d'un moteur d'interprétation abstraite est de limiter le nombre de faux positifs. Des exemples notables de moteurs d'interprétation abstraite incluent ASTRÉE [51] et Mopsa [153].

**L'exécution symbolique** Introduite par King en 1976 [5], l'exécution symbolique est une méthode automatique et complète. Toutefois, elle n'est pas correcte, car elle peut passer à côté de certains bogues ou ne pas aboutir à une conclusion, surtout si l'analyse ne termine pas. Néanmoins, lorsqu'elle termine, elle offre une garantie de complétude et de correction. Cette approche est donc particulièrement efficace pour la recherche de bogues, même si elle est moins adaptée pour prouver la correction complète d'un programme. L'implémentation d'un moteur d'exécution symbolique doit chercher à minimiser les sous-approximations. Un exemple de moteur d'exécution symbolique est KLEE [60].

**La vérification déductive** La vérification déductive, bien que correcte et complète, repose en grande partie sur l'intervention humaine [71]. Cette méthode exige que le programmeur fournisse des invariants sur les boucles, les types, ou prouve la terminaison de certaines fonctions. Si elle garantit des preuves très rigoureuses de la correction des programmes, sa mise en œuvre demande des efforts considérables, car elle n'est pas entièrement automatisée. Néanmoins, il arrive que sur certains programmes, la preuve soit complètement automatique. L'objectif dans l'implémentation de cette méthode est de minimiser l'intervention manuelle du programmeur et d'automatiser le processus autant que possible. Des exemples d'outils de vérification déductive incluent Why3 [78], Boogie [55] et Viper [87].

**Choix de la méthode** Dans cette thèse, nous avons opté pour l'exécution symbolique puisque nous avions déjà développé un interpréteur pouvant servir de fondation. Ce choix s'est avéré particulièrement adapté dans le cas de Wasm pour plusieurs raisons. Tout d'abord, la majorité des programmes Wasm sont générés par des compilateurs, ce qui implique que le code change fréquemment au fil des évolutions des compilateurs. Dans ce contexte, la vérification déductive, nécessitant des ajustements manuels à chaque modification du code, serait difficilement tenable. De même, l'interprétation abstraite génère des faux positifs, ce qui impose une vérification manuelle fastidieuse, peu pratique pour du code généré automatiquement.

De plus, Wasm est un langage particulièrement propice à l'exécution symbolique. Cela est dû à sa simplicité structurelle, son typage et l'absence de comportements indéfinis. En nous appuyant entièrement sur la sémantique rigoureuse du langage, nous avons évité les approximations nécessaires dans d'autres contextes.

Enfin, un moteur d'exécution symbolique pour Wasm offre un avantage considérable : il permet, par extension, d'appliquer cette méthode à tous les langages qui compilent vers Wasm. Cela simplifie considérablement la maintenance, car un seul moteur symbolique peut analyser des programmes provenant de multiples langages. De plus, les compilateurs vers Wasm, en constante évolution pour des raisons indépendantes de la vérification symbolique (comme le déploiement d'applications sur le Web), assurent une qualité et un soutien techniques bien supérieurs à ceux que l'on pourrait attendre pour des langages dédiés uniquement à l'exécution symbolique. Grâce à cette approche, il est envisageable d'effectuer des analyses symboliques multi-langages tout en réutilisant le travail effectué par les compilateurs pour gérer les aspects complexes liés à l'édition de liens, aux FFI et aux ABI.

## Contributions

Les contributions de cette thèse, toutes liées à la compilation vers Wasm, s'inscrivent dans deux domaines principaux.

**Wasocaml** Le premier concerne l'extension de Wasm pour prendre en charge les langages avec glaneur de cellules. Les travaux que nous avons menés comprennent des aspects de conception et de mise en œuvre :

— une analyse des techniques de représentation des valeurs en présence de polymorphisme et de glaneur de cellules ;

— le développement de Wasocaml, le **premier compilateur pour un vrai**[1] **langage fonctionnel (OCaml) ciblant WasmGC** ;
— une contribution à la conception du standard WasmGC et une proposition d'extension (les valeurs gelées) ;

ainsi qu'un aspect de formalisation, à travers :

— la conception d'une sémantique alternative pour Wasm et WasmGC ;
— la définition de la première sémantique pour Flambda1 (une représentation intermédiaire utilisée dans OCaml) ;
— la formalisation et la preuve de correction d'un compilateur Flambda1 vers WasmGC.

**Owi** Le second a trait à la vérification des programmes Wasm et la détection de bogues. Dans ce cadre, nous avons développé :

— une boîte à outils OCaml pour manipuler les programmes Wasm (Owi) ;
— une FFI sûre pour un interpréteur Wasm en OCaml ;
— un outil de fuzzing pour tester les moteurs d'exécution Wasm ;
— **une méthode pour généraliser un interpréteur concret en un interpréteur monadique** ;
— **un interpréteur symbolique performant et multi-cœur pour Wasm, C et Rust (Owi)** ;
— **l'interpréteur symbolique Wasm le plus performant à ce jour (Owi)** ;
— **le premier interpréteur symbolique cross-langages (Owi)** ;
— un moteur de programmation par contraintes pour Wasm, C et Rust ;
— un langage de spécification pour Wasm (Weasel) ;
— la génération d'assertions symbolique de programmes Wasm et C à partir de spécifications.

**Collaborations** En plus de mes directeurs de thèse, Pierre Chambart et Jean-Christophe Filliâtre, plusieurs personnes ont contribué aux travaux présentés dans cette thèse. Ces collaborations ont toutes trait à Owi, qui compte aujourd'hui dix contributeurs d'après les statistiques du dépôt Git.

Tout d'abord, José Fragoso Santos a suggéré de faire d'Owi un interpréteur symbolique. Avec Filipe Marques, son doctorant, ils développaient alors WASP, un interpréteur symbolique pour Wasm basé sur l'interpréteur Wasm de référence. WASP était limité par l'implémentation de ce dernier. Ils ont décidé de ne plus maintenir WASP et ils contribuent aujourd'hui à Owi. Filipe Marques a contribué au modèle mémoire et à l'application à du code C, grâce à son expérience sur WASP. De plus, il est l'auteur principal de *Smt.ml*. Arthur Carcano est l'auteur principal de la monade de choix dans sa quatrième implémentation qui est celle décrite ici (la troisième était également multi-cœur mais n'était pas basée sur les transformateurs de monades et sa maintenance pour le moins difficile) et de l'application à du code Rust. Les contributions d'Arthur et de Filipe ne peuvent se résumer aux points cités ci-dessus, ils ont en effet tous les deux passé plusieurs mois à travailler sur Owi avec moi. Le dépôt Git contient le détail des contributions de chacun.

1. Selon la définition de Juliusz Chroboczek : "Un langage est un vrai langage s'il existe au moins un programme écrit dans ce langage ayant au moins un utilisateur ne sachant pas que ce programme est écrit dans ce langage".

De plus, j'ai eu la chance d'encadrer deux stagiaires durant ma thèse : Eric Patrizio et Zhicheng Hui. Leur contribution est indiquée au début des chapitres concernés.

# Table des matières

# Table des figures

# Table des codes sources

# Liste des tableaux

# Première partie

# WebAssembly

# 2 Introduction à Wasm

WebAssembly, plus communément appelé *Wasm*, a été conçu pour servir de remplaçant, au moins partiel, à JavaScript dans les navigateurs. Depuis, il est devenu un standard largement adopté non seulement dans les principaux navigateurs web, mais aussi dans des domaines variés tels que les systèmes embarqués ou les environnements d'exécutions sur des serveurs. Wasm a été pensé pour être une cible de compilation rapide, sûre et portable.

**Rapide** Wasm permet une analyse, une validation et une compilation efficaces, tout en générant du code performant.

**Sûr** Le langage offre une sémantique formalisée, sans comportement indéfini, et fournit des garanties solides contre certaines erreurs d'exécution grâce à son système de typage. Par ailleurs, toutes les interactions avec l'environnement sont strictement contrôlées : l'état d'un programme Wasm est, par défaut, invisible depuis l'extérieur, à moins que certaines parties ne soient explicitement marquées comme exportables.

Ce chapitre propose une introduction détaillée à Wasm, en expliquant ses concepts fondamentaux.

## 2.1 Un langage à pile

En Wasm, le code s'écrit généralement dans un format textuel avec l'extension `.wat`, tandis que le code binaire est enregistré sous l'extension `.wasm`. Un fichier `.wat` contient un *module*, qui est l'unité de base permettant d'organiser les programmes. Par exemple, un module vide s'écrit simplement :

```
(module)
```

Un module peut contenir des fonctions, qui sont définies avec un nom, une signature (paramètres et type de retour), et un corps. Voici un exemple de fonction au sein d'un module :

```
(module

  (func $square (param $n i32) (result i32)
```

```
    … ;; body of the function
  ))
```

Wasm, dans sa version initiale, ne supporte que quatre types scalaires : `i32`, `i64`, `f32` et `f64`. Le corps de la fonction est composé d'une suite d'instructions manipulant une pile. Prenons l'exemple suivant où l'état de la pile est décrit à chaque étape :

```
(module

  (func $square (param $n i32) (result i32)

    ;; ε : initially the stack is empty

    local.get $n ;; [ n ] : we added n on the stack
    local.get $n ;; [ n ; n ] : we added it again
    i32.mul      ;; [ n² ] :
      ;; - pop two values from the stack
      ;; - multiply them
      ;; - push the result on the stack

    return ;; pop values to return from the stack (there is only one here but
    ↪  there could be many)
  ))
```

Chaque instruction possède un type de la forme : $[p_0; ...; p_{n-1}] \to [r_0; ...; r_{m-1}]$, qui décrit comment elle manipule la pile. Par exemple, `i32.mul` a le type : $[\texttt{i32}; \texttt{i32}] \to [\texttt{i32}]$. Grâce à cette structure, le type de la pile à n'importe quel point du programme peut être déterminé statiquement, y compris dans des boucles.

Il existe également une syntaxe simplifiée utilisant des $S - expressions$ pour améliorer la lisibilité du code. Ainsi, le programme précédent peut se réécrire comme suit :

```
(module

  (func $square (param $n i32) (result i32)

    (i32.mul
      (local.get $n)
      (local.get $n))

    ;; the `return` instruction is actually optional at the end of the
    ↪  function's body
  ))
```

Les fonctions, comme les instructions, manipulent la pile en dépilant leurs arguments au moment de l'appel et en empilant leurs valeurs de retour. Il est important de noter que les arguments d'une fonction ne sont pas sur la pile initialement, mais sont accessibles via les paramètres. En Wasm, toutes les fonctions sont (mutuellement) récursives par défaut. Voici un

exemple illustrant la fonction de Fibonacci, combinant ces concepts :

```
(module

  (func $fib (param $n i32) (result i32)

    (i32.lt_s (local.get $n) (i32.const 2))
    ;; [ n < 2 ]
    ;; `if` will pop the condition from the stack
    (if (then
                  ;; ε
      local.get $n ;; [ n ]
      return       ;; early return
    ))

    ;; ε
    (i32.sub
      (local.get $n)
      (i32.const 2)))
    ;; [ n - 2 ]
    call $fib ;; [ fib(n - 2) ]

    (i32.sub
      (local.get $n)
      (i32.const 1))
    ;; [ n - 1; fib(n - 2) ]
    call $fib ;; [ fib(n - 1); fib(n - 2) ]

    i32.add ;; [ fib(n - 1) + fib(n - 2) ]

    ;; implicit return
  ))
```

Un programme Wasm peut échouer à l'exécution pour diverses raisons, entraînant un `trap`. Un `trap` est une exception levée par Wasm, mais qui ne peut pas être capturée par le programme lui-même (elle peut être interceptée par l'hôte, comme un programme JavaScript dans un navigateur). L'instruction `unreachable` permet de provoquer un `trap` explicite et possède un type particulier : elle peut en quelque sorte prendre n'importe quel type (comme `assert false` en OCaml qui n'a pas le type `unit`). Cela permet par exemple de typer une conditionnelle dont l'une des branches contiendrait un `unreachable` comme ayant le type de l'autre branche. Par exemple, dans la fonction Fibonacci, nous pourrions provoquer un `trap` si un entier négatif est passé en paramètre :

```
(module

  (func $fib (param $n i32) (result i32)
```

```
    (i32.lt_s (local.get $n) (i32.const 0))
    (if (then
      unreachable ;; end of execution, trap !
    ))

    ;; …
  ))
```

Pour appeler effectivement la fonction `$fib` dès le début de l'exécution du programme, il est possible d'utiliser le champ `start`, qui désigne une fonction servant de point d'entrée. Cette fonction doit avoir le type $\varepsilon \to \varepsilon$. Voici un exemple où la fonction `$fib` est appelée avec l'argument 10. Elle est suivie d'un `drop` pour correspondre au type attendu :

```
(module
  (func $fib
    ;; …
  )

  (func $start
    i32.const 10
    call $fib
    drop )

  (start $start))
```

Ce programme ne produira pas de sortie visible par défaut. Si l'on souhaite afficher une valeur, il faut utiliser une fonction d'affichage fournie par l'hôte, comme décrit plus bas.

## 2.2 Mémoire

En Wasm, un programme peut manipuler une *mémoire linéaire*, c'est-à-dire un tableau redimensionnable d'octets. Chaque programme Wasm peut disposer d'une seule mémoire. Voici comment ajouter une mémoire dans un module :

```
(module
  (memory))
```

Par défaut, la taille de la mémoire est nulle. Il est possible de définir une taille initiale minimale ainsi qu'une taille maximale. La taille de la mémoire est mesurée en pages, une page étant égale à 65 536 octets (64 Ko). Par exemple, pour spécifier une mémoire avec une taille minimale de 2 pages et une taille maximale de 10 pages, nous écrivons :

```
(module
  (memory 2 10))
```

Au démarrage, toute la mémoire est initialisée à zéro. Le type `i32` est utilisé pour repré-

senter les adresses dans cette mémoire. Les instructions `i32.load` et `i32.store` permettent respectivement de lire et d'écrire des entiers de type `i32` à une adresse donnée. D'autres instructions similaires existent pour les autres types, comme `f32.load` et `f32.store`.

Voici un exemple de la fonction de Fibonacci avec une mise en œuvre mémoïsée, utilisant la mémoire pour stocker les résultats intermédiaires :

```
(module

  (memory 1) ;; the memory is 0-initialized

  (func $fib (param $n i32) (result i32)

    (if (i32.lt_s (local.get $n) (i32.const 0))
    (then (unreachable)))

    (if (i32.lt_s (local.get $n) (i32.const 2))
    (then (return (local.get $n))))

    ;; if memory[4n] = 0, we compute fib(n) and store it
    (if
      (i32.eqz
        (i32.load (i32.mul (i32.const 4) (local.get $n))))
    (then
      (i32.mul (local.get $n) (i32.const 4)) ;; [4n]
      (call $f (i32.sub (local.get $n) (i32.const 1)))
      ;; [fib(n - 1) ; 4n]
      (call $f (i32.sub (local.get $n) (i32.const 2)))
      ;; [fib(n - 2) ; fib(n - 1) ; 4n]
      i32.add ;; [fib(n) ; 4n]
      i32.store ;; store fib(n) at offset 4n
    ))

    ;; we return the value at memory.(4n)
    (i32.mul (i32.const 4) (local.get $n))
    i32.load))
```

Dans cet exemple, la fonction calcule et stocke les résultats de Fibonacci dans la mémoire, ce qui évite de recalculer plusieurs fois la même valeur.

## 2.3 Contrôle du flot d'exécution

Wasm fournit plusieurs constructions permettant de gérer le flot d'exécution, dont les blocs et les boucles. La première construction est le bloc `block $label ...`. Il est possible de sortir d'un bloc avec l'instruction `br $label`.

```
(func $fib (param $n i32) (result i32)

  (block $base_case

    (i32.lt_s (local.get $n) (i32.const 2))
    (if (then
      br $base_case ;; jump to A
    )

    (i32.sub (local.get $n) (i32.const 2)))
    call $fib

    (i32.sub (local.get $n) (i32.const 1))
    call $fib

    i32.add
    return
  )

  ;; A is here
  ;; n < 2 so we return n

  local.get $n )
```

L'instruction `br_if` permet de faire un saut conditionnel vers un bloc. Voici le même exemple, cette fois avec l'utilisation de `br_if` :

```
(func $fib (param $n i32) (result i32)

  (block $base_case
    (i32.lt_s (local.get $n) (i32.const 2))
    br_if $base_case ;; jump to A

    (i32.sub (local.get $n) (i32.const 2)))
    call $fib

    (i32.sub (local.get $n) (i32.const 1))
    call $fib

    i32.add
    return
  )

  ;; A is here

  local.get $n )
```

Les blocs peuvent être imbriqués, mais l'instruction `br` ne peut effectuer un saut que vers un bloc englobant. Un bloc possède sa propre pile, et il est possible d'en spécifier la signature : elle détermine combien de valeurs sont retirées de la pile à l'entrée dans le bloc et combien de valeurs sont ajoutées sur la pile à la sortie du bloc. Le bloc peut donc être vu comme une fonction locale, implicitement appelée lors de sa déclaration.

Voici un exemple de bloc avec des paramètres et des résultats :

```
(func (result i32 f32)

  i32.const                          ;; [ 1 ]
  i32.const 2                        ;; [ 2; 1 ]

  (block $b (param i32) (result f32)
                                     ;; [ 2 ]   [ 1 ]
    drop                             ;; ε       [ 1 ]
    f32.const 3.4                    ;; [ 3.4 ] [ 1 ]
  )
                                     ;; [ 3.4; 1 ]
  return )
```

La construction `loop $label ...` fonctionne de manière similaire à `block`, à la différence près qu'un saut vers l'étiquette d'une `loop` entraîne un redémarrage de la boucle au lieu de sortir du bloc. Voici comment mettre en œuvre la fonction Fibonacci en utilisant une boucle (et des variables locales que nous introduisons maintenant) :

```
(func $fib (param $n i32) (result i32)
  (local $old i32) (local $old_old i32)

  (local.set $old_old (i32.const 1))

  (block $stop
    (loop $loop

      (i32.le_s (local.get $n) (i32.const 0))
      br_if $stop

      (i32.sub (local.get $n) (i32.const 1))
      local.set $n

      (i32.add (local.get $old) (local.get $old_old))
      local.set $old

      (i32.sub (local.get $old) (local.get $old_old))
      local.set $old_old

      br $loop
```

```
    ))

  local.get $old )
```

Dans cet exemple, la boucle continue de s'exécuter tant que la condition n'est pas remplie. La variable `$n` est décrémentée à chaque itération, et les variables locales `$a` et `$b` sont mises à jour pour stocker les termes de la suite de Fibonacci.

## 2.4 Références en Wasm

En plus des types scalaires, Wasm prend également en charge les références. Actuellement [1], seuls deux types de référence sont disponibles :

— `funcref` : représente des références vers des fonctions ;
— `externref` : désigne des références opaques fournies par l'hôte.

Une référence peut être nulle. Nous créons une référence nulle à l'aide de l'instruction `ref.null`, et les variables locales de type référence sont initialisées à cette valeur par défaut. Pour vérifier si une référence est nulle, nous utilisons l'instruction `ref.is_null`.

Contrairement aux types scalaires, les références ne peuvent ni être stockées ni lues directement depuis la mémoire linéaire. Pour les stocker, il faut utiliser des champs de type `table`, qui permettent de gérer les références.

Pour appeler une fonction stockée dans une table, nous utilisons l'instruction `call_ref`, qui effectue une vérification dynamique du type. Si le type de la fonction appelée ne correspond pas au type attendu, un `trap` est levé.

L'utilisation des références de fonction est particulièrement utile lorsqu'on compile des objets ou des fermetures (closures), comme cela sera exploré plus loin dans cette thèse.

Voici un exemple d'utilisation de références de fonctions dans une table :

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

  (func $square-float (param $x f32) (result f32)
    (f32.mul (local.get $x) (local.get $x)))

  (table $square-tbl funcref
    (elem $square-int $square-float))

  (func $main
      i32.const 4
      i32.const 0
      ;; call function at offset 0 in square-tbl (square-int)
      ;; with argument 4
      call_indirect $square-tbl (param i32) (result i32)
```

1. En réalité, les références de fonctions sont introduites par le *Typed Function References Proposal*, mais nous les présentons ici par souci de concision.

```
    ;; stack: [ 16 ]
    i32.const 1 ;; [ 1; 16 ]

    ;; call function at offset 1 in square-tbl (square-float)
    ;; with argument 16
    call_indirect $square-tbl (param i32)
    ;; runtime trap: indirect call type mismatch
  )

  (start $main))
```

Dans cet exemple, deux fonctions sont stockées dans une table, mais l'appel à `square-float` échoue en raison d'une incompatibilité de type lors de l'appel indirect.

## 2.5 Interactions avec l'hôte

Wasm ne dispose pas d'instructions natives permettant d'interagir directement avec l'environnement hôte. Toutes les interactions se font via des imports, où l'hôte fournit les fonctions que le module Wasm peut appeler. Inversement, pour permettre à l'hôte d'invoquer des fonctions Wasm après l'initialisation du module (c'est-à-dire après l'exécution de la fonction `start`), il est nécessaire de déclarer des exports explicites.

Voici comment définir des fonctions importées et exportées dans un module Wasm :

```
(module
  ;; import a function `print-i32` from the module `stdlib`
  (func $print-i32 (import "stdlib" "print-i32") (param i32))

  (func $fib (param $n i32) (result i32)
    …
  )

  ;; export this function under the name `print-fibo`
  (func $print-fibo (export "print-fibo") (param $n i32)
    local.get $n
    call $fib
    call $print-i32
  ))
```

Dans cet exemple, l'hôte fournit la fonction `$print-i32` via un import, tandis que la fonction `$print-fibo` est exportée pour être utilisée par l'hôte ou d'autres modules Wasm. C'est à l'hôte de gérer les noms des modules et des fonctions lors de l'étape de l'édition de liens.

Les fonctions importées peuvent provenir d'un autre module Wasm ou être écrites dans le langage hôte (JavaScript dans le cas d'un environnement de navigateur, par exemple). De la même manière, les fonctions exportées peuvent être appelées par l'hôte ou utilisées dans

d'autres modules Wasm.

Il est important de noter que si le type attendu côté Wasm ne correspond pas au type de la fonction importée côté JavaScript (ou autre langage hôte), cela déclenchera une erreur d'exécution dans la plupart des moteurs d'exécution. Cette vérification se produit généralement à l'exécution, plutôt qu'au moment de l'édition de liens.

# 3 Formalisation de Wasm

Dans ce chapitre, nous proposons une sémantique alternative à celle actuellement utilisée pour Wasm [91]. La sémantique officielle repose sur le format binaire de Wasm, ce qui la rend plus difficile à manipuler. Par exemple, les blocs ne sont plus identifiés par des noms, mais par des indices numériques représentant la position relative du bloc par rapport à son point de référence. Contrairement à celle-ci, notre sémantique opère sur le format textuel plutôt que binaire, utilisant donc des identifiants au lieu d'indices numériques. Nous pensons que notre sémantique simplifie la gestion des constructions liées au flot de contrôle, telles que `block` et `loop`.

Nous proposons donc une version alternative que nous jugeons plus lisible et plus adaptée à notre contexte. Toutefois, nous reconnaissons que le choix de la sémantique binaire pour Wasm est judicieux dans son cadre d'application. Ce format est en effet celui utilisé par la majorité des outils, et il évite d'avoir à donner une sémantique distincte pour la traduction du format textuel vers le format binaire, ce qui aurait alourdi la tâche.

**Notations** Nous utilisons à plusieurs endroits certaines notations qui peuvent paraître inhabituelles. En particulier, nous notons $kind^*$ une suite finie potentiellement vide d'éléments de type $kind$ dans nos règles de typage et de sémantique, comme cela se fait généralement dans la description des grammaires. De plus, nous utilisons la notation $kind^*$ et $kind'^*$, ou encore $kind_1^*$ et $kind_2^*$, pour différencier deux listes distinctes d'éléments de type $kind$ lorsque nous avons besoin de pouvoir distinguer ces deux listes.

## 3.1 Syntaxe abstraite

**Types** Un type $t$ peut être un type scalaire ou un type de référence.

$$\begin{aligned} t ::= \ & st && \text{(type scalaire)} \\ \mid \ & rt && \text{(type référence)} \end{aligned}$$

**Types scalaires** Un type scalaire $st$ désigne un entier ou un nombre flottant, et peut être de 32 ou 64 bits.

$$
\begin{aligned}
st ::= \ & \texttt{i32} && \text{(entier 32 bits)} \\
| \ & \texttt{i64} && \text{(entier 64 bits)} \\
| \ & \texttt{f32} && \text{(flottant 32 bits)} \\
| \ & \texttt{f64} && \text{(flottant 64 bits)}
\end{aligned}
$$

**Types tas abstraits** Un type de tas abstrait *aht* est forcément externe.

$$
aht ::= \texttt{extern} \qquad \text{(externe)}
$$

**Types tas** Un type de tas est soit un identifiant de type, soit un type tas abstrait.

$$
\begin{aligned}
ht ::= \ & aht && \text{(type tas abstrait)} \\
| \ & id && \text{(identifiant de type)}
\end{aligned}
$$

**Types références** Un type de référence est un type tas qui peut être `null`.

$$
rt ::= \texttt{ref null}^? \ ht \qquad \text{(type de référence)}
$$

**Types des blocs** Un type de bloc *bt* est défini par deux suites finies : une pour les paramètres et une pour les résultats.

$$
bt ::= \{\texttt{param} : t^*, \texttt{result} : t^*\} \qquad \text{(type de bloc)}
$$

**Paramètres de fonctions** Un paramètre de fonction *fp* est un enregistrement contenant un identifiant et un type.

$$
fp ::= \{\texttt{id} : id, \texttt{t} : t\} \qquad \text{(paramètre de fonction)}
$$

**Types des fonctions** Un type de fonction *ft* est un enregistrement composé de deux suites finies : la première contient les paramètres (chacun ayant un identifiant et un type), la seconde les types de retour.

$$
ft ::= \{\texttt{param} : fp^*, \texttt{result} : t^*\} \qquad \text{(type de fonction)}
$$

La fonction $bt\_of\_ft : ft \to bt$ transforme un type de fonction en un type de bloc en retirant les identifiants des paramètres.

**Instructions** Une instruction $i$ peut consister à créer une constante, appliquer un opérateur, effectuer des opérations sur la pile, manipuler des variables locales, appeler des fonctions ou gérer le flot de contrôle.

$$
\begin{array}{rll}
i ::= & \texttt{i32.const}\ n & \text{(constante i32)} \\
| & \texttt{i64.const}\ n & \text{(constante i64)} \\
| & \texttt{f32.const}\ x & \text{(constante f32)} \\
| & \texttt{f64.const}\ x & \text{(constante f64)} \\
| & \texttt{i32.eqz} & \text{(test de nullité entiers)} \\
| & \texttt{drop} & \text{(abandon)} \\
| & \texttt{local.get}\ id & \text{(accès à une locale)} \\
| & \texttt{local.set}\ id & \text{(assignement à une locale)} \\
| & \texttt{call}\ id & \text{(appel de fonction)} \\
| & \texttt{br}\ id & \text{(branchement)} \\
| & \texttt{br\_if}\ id & \text{(branchement conditionnel)} \\
| & \texttt{block}\ id\ bt\ i^* & \text{(bloc)} \\
| & \texttt{loop}\ id\ bt\ i^* & \text{(boucle)} \\
| & \texttt{if}\ id\ bt\ \texttt{then}\ i^*\ \texttt{else}\ i^* & \text{(conditionnelle)} \\
| & \texttt{ref.null}\ ht & \text{(référence nulle)} \\
| & \texttt{ref.as\_non\_null} & \text{(test de nullité références)} \\
| & \texttt{unreachable} & \text{(code inatteignable)}
\end{array}
$$

**Fonctions** Une fonction $f$ est un enregistrement contenant un identifiant, un type, des variables locales (sous forme de paramètres supplémentaires) et le corps de la fonction.

$$
f ::= \{\texttt{id} : id, \texttt{type} : ft, \texttt{locals} : fp^*, \texttt{code} : i^*\} \qquad \text{(fonction)}
$$

**Modules** Un module $m$ est un enregistrement contenant un identifiant de fonction de démarrage et une liste de fonctions.

$$
m ::= \{\texttt{start} : id, \texttt{funcs} : f^*\} \qquad \text{(module)}
$$

## 3.2 Validité

### 3.2.1 Bonne formation

Un module $m$ est considéré comme valide si :
- chaque fonction possède un identifiant unique ;
- l'une des fonctions est associée à l'identifiant de démarrage du module ;
- les paramètres et variables locales de chaque fonction ont des identifiants distincts ;
- la fonction désignée par le champ `start` a le type $\varepsilon \to \varepsilon$ ;
- tous les identifiants utilisés dans le module font bien référence à des entités existantes.

### 3.2.2 Typage

Pour qu'un module soit valide, il est également nécessaire que toutes les expressions qu'il contient soient bien typées. Une expression est une suite d'instructions. Il s'agit par exemple du corps de fonctions ou bien du code servant à initialiser les variables globales. Nous décrivons ci-dessous les règles de typage des expressions.

**Contexte** Un contexte de typage $\Gamma$ est défini comme suit :

$$\Gamma ::= \{\texttt{locals} : id \to t, \texttt{funcs} : id \to ft, \texttt{labels} : id \to t^*\} \qquad \text{(contexte)}$$

**Jugement** Le jugement de typage a la forme $\Gamma \vdash i : bt$, ce qui signifie que dans un contexte $\Gamma$, l'instruction $i$ a le type $bt$.

**Expressions** Une expression est bien typée si elle est vide :

$$\text{liste vide}\ \frac{}{\Gamma \vdash \varepsilon : \varepsilon \to \varepsilon}$$

Une expression non vide est bien typée si sa première instruction est bien typée et si son type permet de typer correctement la suite de l'expression :

$$\text{liste non vide}\ \frac{\Gamma \vdash i : t^* \to t'^* \qquad \Gamma \vdash i^* : t'^* \to t''^*}{\Gamma \vdash i\ i^* : t^* \to t''^*}$$

**Instructions** Voici les règles de typage pour les différentes instructions.

$$\text{constante i32}\ \frac{}{\Gamma \vdash \texttt{i32.const}\ n : \varepsilon \to \texttt{i32}}$$

Cette instruction place une constante de type `i32` sur la pile.

$$\text{constante i64}\ \frac{}{\Gamma \vdash \texttt{i64.const}\ n : \varepsilon \to \texttt{i64}}$$

Cette instruction place une constante de type `i64` sur la pile.

$$\text{constante f32}\ \frac{}{\Gamma \vdash \texttt{f32.const}\ x : \varepsilon \to \texttt{f32}}$$

Cette instruction place une constante de type `f32` sur la pile.

$$\text{constante f64}\ \frac{}{\Gamma \vdash \texttt{f64.const}\ x : \varepsilon \to \texttt{f64}}$$

Cette instruction place une constante de type `f64` sur la pile.

$$\texttt{i32.eqz}\ \frac{}{\Gamma \vdash \texttt{i32.eqz}\ n : \texttt{i32} \to \texttt{i32}}$$

Cette instruction consomme une valeur de type `i32` et en produit une autre du même type sur la pile.

$$\texttt{drop}\ \frac{}{\Gamma \vdash \texttt{drop} : t \to \varepsilon}$$

L'instruction `drop` retire une valeur de n'importe quel type de la pile, sans produire de résultat. Elle est dite *stack-polymorphic*.

$$\texttt{local.get}\ \frac{\Gamma.\texttt{locals}(id) = t}{\Gamma \vdash \texttt{local.get}\ id : \varepsilon \to t}$$

Cette instruction place sur la pile une valeur dont le type $t$ est défini statiquement par l'environnement $\Gamma$.

$$\texttt{local.set}\ \frac{\Gamma.\texttt{locals}(id) = t}{\Gamma \vdash \texttt{local.set}\ id : t \to \varepsilon}$$

L'instruction `local.set` consomme une valeur de type $t$ sur la pile, où ce type est défini statiquement par l'environnement.

$$\texttt{call}\ \frac{\Gamma.\texttt{funcs}(id) = ft}{\Gamma \vdash \texttt{call}\ id : bt\_of\_ft(bt)}$$

L'instruction `call` consomme et produit un certain nombre de valeurs dont les types sont spécifiés statiquement par l'environnement.

$$\texttt{br}\ \frac{\Gamma.\texttt{labels}(id) = t^*}{\Gamma \vdash \texttt{br}\ id : t^* t'^* \to t''^*}$$

L'instruction `br` est valide si le type des éléments sur la pile correspond à celui attendu par le bloc cible du branchement.

$$\texttt{br\_if}\ \frac{\Gamma.\texttt{labels}(id) = t^*}{\Gamma \vdash \texttt{br\_if}\ id : \texttt{i32}\ t^* \to t^*}$$

Cette instruction consomme un `i32` au sommet de la pile. Elle est valide si les autres types sur la pile correspondent à ceux attendus par le bloc.

$$\texttt{block}\ \frac{\Gamma[\texttt{labels} \leftarrow \Gamma.\texttt{labels}[id \leftarrow bt.\texttt{result}]] \vdash i^* : bt}{\Gamma \vdash \texttt{block}\ id\ bt\ i^* : bt}$$

L'instruction `block` est valide si l'on peut typer le corps du bloc en ajoutant l'étiquette correspondante à l'environnement. Le type final de la pile est celui du bloc.

$$\texttt{loop}\ \frac{\Gamma[\texttt{labels} \leftarrow \Gamma.\texttt{labels}[id \leftarrow bt.\texttt{param}]] \vdash i^* : bt}{\Gamma \vdash \texttt{loop}\ id\ bt\ i^* : bt}$$

La règle pour `loop` est similaire à celle de `block`, à la différence près que l'étiquette utilise les paramètres plutôt que les résultats, puisque revenir dans la boucle implique de fournir de nouveaux arguments.

$$\texttt{if}\ \frac{bt = t^* \to t'^* \qquad \Gamma' = \Gamma[\texttt{labels} \leftarrow \Gamma.\texttt{labels}[id \leftarrow t'^*]] \qquad \Gamma' \vdash i_1^* : bt \qquad \Gamma' \vdash i_2^* : bt}{\Gamma \vdash \texttt{if}\ id\ bt\ \texttt{then}\ i_1^*\ \texttt{else}\ i_2^* : \texttt{i32}\ t^* \to t'^*}$$

L'instruction `if` est valide si un `i32` est présent sur la pile et que les deux branches (`then` et `else`) peuvent être typées de manière identique après l'ajout d'une étiquette. Le type final de la pile est celui des branches.

$$\texttt{ref.null}\ \frac{\Gamma \vdash \mathsf{ok}\ ht}{\Gamma \vdash \texttt{ref.null}\ ht : \varepsilon \to \texttt{ref null}\ ht}$$

L'instruction `ref.null` dépose une valeur de type `ref null` *ht* sur la pile, où *ht* est déterminé statiquement. Nous vérifions que ce type respecte certaines contraintes à l'aide du prédicat ok. Il s'agit par exemple de vérifier que le type correspond bien à un identifiant existant. Cette vérification est aussi nécessaire pour des raisons plus complexes que nous ne détaillerons pas ici. Le lecteur curieux est invité à consulter la discussion à ce sujet[1]. Il n'est pas clair que l'omission de cette règle dans ces cas complexes permette d'accepter des programmes dont l'exécution serait erronée, les seuls exemples existants faisant usage de code mort.

$$\texttt{ref.as\_non\_null}\ \frac{\Gamma \vdash \mathsf{ok}\ ht}{\Gamma \vdash \texttt{ref.as\_non\_null} : \texttt{ref null}\ ht \to \texttt{ref}\ ht}$$

Cette instruction consomme une valeur de type `ref null` *ht* et dépose une valeur de type `ref` *ht*. Le type de *ht* est connu statiquement.

$$\texttt{unreachable}\ \frac{\Gamma \vdash \mathsf{ok}\ (t_1^* \to t_2^*)}{\Gamma \vdash \texttt{unreachable} : t_1^* \to t_2^*}$$

L'instruction `unreachable` peut consommer et produire autant de valeurs qu'elle le souhaite sur la pile.

## 3.3 Syntaxe administrative

Cette section présente la *syntaxe administrative*, qui décrit les instructions nécessaires à la sémantique interne du système, mais qui ne sont pas directement exposées à l'utilisateur final.

**Valeurs** Une valeur $v$ peut être soit un scalaire, soit une référence.

$$\begin{aligned} v ::= {} & \mathit{scalar} && \text{(scalaire)} \\ \mid {} & \mathit{reference} && \text{(référence)} \end{aligned}$$

1. https://github.com/WebAssembly/gc/issues/512

**Scalaires** Un scalaire *scalar* est soit un nombre entier machine, soit un nombre à virgule flottante conforme à la norme IEEE 754 [108][2]. Les entiers et flottants peuvent être de taille 32 bits ou 64 bits.

$$
\begin{array}{rll}
\mathit{scalar} ::= & \texttt{i32}\ n & \text{(entier 32 bits)} \\
\mid & \texttt{i64}\ n & \text{(entier 64 bits)} \\
\mid & \texttt{f32}\ x & \text{(flottant 32 bits)} \\
\mid & \texttt{f64}\ x & \text{(flottant 64 bits)}
\end{array}
$$

**Références** Une référence *reference* est soit une référence externe (fournie par l'environnement hôte et considérée comme opaque), soit une référence nulle d'un type de référence particulier.

$$
\begin{array}{rll}
\mathit{reference} ::= & \texttt{extern}\ r & \text{(référence externe)} \\
\mid & \texttt{null}\ \mathit{ht} & \text{(référence nulle)}
\end{array}
$$

**Formes de fenêtres** Une forme de fenêtre *ff* peut être un bloc, une boucle ou une fonction.

$$
\begin{array}{rll}
\mathit{ff} ::= & \texttt{block} & \text{(bloc)} \\
\mid & \texttt{loop} & \text{(boucle)} \\
\mid & \texttt{func} & \text{(fonction)}
\end{array}
$$

**Environnements** Un environnement *env* est une application de *id* dans $v$.

$$
\mathit{env} ::= \mathit{id} \to v \qquad \text{(environnement)}
$$

**Étiquettes** Une étiquette $L$ est une structure composée de plusieurs éléments : un identifiant, une forme de fenêtre, un type, une continuation (une suite d'instructions) et un environnement.

$$
L ::= \{\texttt{id} : \mathit{id}, \texttt{form} : \mathit{ff}, \texttt{type} : \mathit{bt}, \texttt{code} : i^*, \texttt{env} : \mathit{env}\} \qquad \text{(étiquette)}
$$

**Fenêtres** Une fenêtre $F$ contient une étiquette, une pile de valeurs et une suite d'instructions.

$$
F ::= L,\ v^*,\ i^* \qquad \text{(fenêtre)}
$$

**Mémoires linéaires** Une mémoire linéaire $M$ est une application de **N** dans **N**.

$$
M ::= n \to n \qquad \text{(mémoire linéaire)}
$$

2. Le standard officiel Wasm diffère légèrement de cette norme ; nous négligeons ces différences ici.

**États** Un état $E$ est composé d'une mémoire linéaire et d'un ensemble de fonctions.

$$E ::= \{\texttt{mem} : M, \texttt{funcs} : f\} \qquad \text{(état)}$$

**Configurations** Une configuration $C$ est formée d'un état et d'une suite finie de fenêtres, dont la longueur est toujours d'au moins 1.

$$C ::= E,\ F^+ \qquad \text{(configuration)}$$

## 3.4 Sémantique

**Fonction de démarrage** Soit $m$ un module valide. Nous appelons *fonction de démarrage* de $m$ le terme $f_{start}$ tel que $f_{start} \in m.\texttt{funcs}$ et $f_{start}.\texttt{id} = m.\texttt{start}$.

**Étiquette d'une fonction** Pour une fonction $f$, son étiquette $L$ est définie par :

$$\{\texttt{id} = f.\texttt{id}, \texttt{form} = \texttt{func}, \texttt{type} = \mathit{bt\_of\_ft}(f.\texttt{type}),\ \texttt{code} = (), \texttt{env} = \ldots\}$$

Ici, *env* est initialisé avec les variables locales à leurs valeurs par défaut (c'est-à-dire `0` pour les constantes et `null` pour les références).

**Configuration initiale** Soient $m$ un module valide et $f_{start}$ sa fonction de démarrage. Nous définissons l'étiquette $L$ associé à $f_{start}$ et la pile d'exécution initiale $F = (L, (), f_{start}.\texttt{code})$. De plus, soient $M$ la fonction constante $M : n \to 0$, et $E = \{\texttt{mem} = M, \texttt{funcs} = m.\texttt{funcs}\}$. La configuration initiale de $m$ est alors $(E, F)$.

**Sémantique opérationnelle à petits pas** Notre sémantique suit un style à petits pas inspiré de Plotkin [10], avec des règles de réduction opérant sur une configuration $C$. Par souci de lisibilité, nous omettons parfois l'état de la configuration lorsqu'il n'est ni lu ni modifié, ou seulement lu sans être modifié. Nous simplifions également les notations pour les fenêtres et étiquettes inutiles, en ne conservant que les piles de valeurs et d'instructions.

**Règles de réduction** Une règle de réduction est de la forme $C \to R$, où :

$$R ::= \texttt{trap} \mid C$$

Voici les différentes règles de réduction.

$$\text{constante i32}\ \frac{}{v^*, (\texttt{i32.const}\ n, i^*) \to (\texttt{i32}\ n, v^*), i^*}$$

Cette instruction dépose une constante de type `i32` au sommet de la pile.

$$\text{constante i64}\ \frac{}{v^*, (\texttt{i64.const}\ n, i^*) \to (\texttt{i64}\ n, v^*), i^*}$$

Cette instruction dépose une constante de type `i64` au sommet de la pile.

$$\text{constante f32}\ \frac{}{v^*, (\texttt{f32.const}\ x, i^*) \to (\texttt{f32}\ x, v^*), i^*}$$

Cette instruction dépose une constante de type `f32` au sommet de la pile.

$$\text{constante f64}\ \frac{}{v^*, (\texttt{f64.const}\ x, i^*) \to (\texttt{f64}\ x, v^*), i^*}$$

Cette instruction dépose une constante de type `f64` au sommet de la pile.

$$\texttt{i32.eqz}\text{-oui}\ \frac{v_0 = \texttt{i32 0}}{(v_0, v^*), (\texttt{i32.eqz}, i^*) \to (\texttt{i32 1}, v^*), i^*}$$

Si la valeur au sommet de la pile est `i32 0`, alors `i32.eqz` dépose un entier représentant le booléen vrai.

$$\texttt{i32.eqz}\text{-non}\ \frac{v_0 \neq \texttt{i32 0}}{(v_0, v^*), (\texttt{i32.eqz}, i^*) \to (\texttt{i32 0}, v^*), i^*}$$

Si la valeur n'est pas zéro, `i32.eqz` dépose un entier représentant faux.

$$\text{abandon}\ \frac{}{(v_0, v^*), (\texttt{drop}, i^*) \to v^*, i^*}$$

L'instruction `drop` enlève la valeur au sommet de la pile.

$$\text{accès à une locale}\ \frac{L.\mathsf{env}(id) = v_{id}}{L, v^*, (\texttt{local.get}\ id, i^*) \to (v_{id}, v^*), i^*}$$

L'instruction `local.get` place la valeur de la variable locale *id* au sommet de la pile.

$$\text{assignement à une locale}\ \frac{env' = L.\mathsf{env}[id \leftarrow v_0] \qquad L' = L[\mathsf{env} \leftarrow env']}{L, (v_0, v^*), (\texttt{local.set}\ id, i^*) \to L', v^*, i^*}$$

L'instruction `local.set` met à jour la valeur de la variable locale *id* avec la valeur au sommet de la pile.

$$\text{appel}\ \frac{\begin{array}{c} f_{id} \in E.\texttt{funcs} \qquad f_{id}.\texttt{id} = id \\ f_{id}.\texttt{type.param} = p_0, ..., p_{n-1} \qquad f_{id}.\texttt{locals} = p_n, ..., p_m \\ env = [p_0.\texttt{id} \leftarrow v_0, ..., p_{n-1}.\texttt{id} \leftarrow v_{n-1}; \\ p_n.\texttt{id} \leftarrow 0, ..., p_m.\texttt{id} \leftarrow 0] \\ code = f_{id}.\texttt{code} \qquad type = bt\_of\_ft(f_{id}.\texttt{type}) \\ L' = \{\texttt{id} = id; \texttt{form} = \texttt{func}; \texttt{type} = type; \texttt{code} = (); \texttt{env} = env\} \end{array}}{E, (L, (v_0, ..., v_{n-1}, v^*), (\texttt{call}\ id, i^*)) \rightarrow (L', (), code)(L, v^*, i^*)}$$

L'instruction `call` crée une nouvelle fenêtre destinée à exécuter la fonction *id*. Cette fenêtre comporte une pile vide, une liste d'instructions correspondant au code de la fonction, et une étiquette dans lequel les valeurs de l'environnement correspondant aux paramètres ont été correctement initialisées par les valeurs se trouvant sur la pile avant l'appel de la fonction. Quant aux variables locales, elles sont initialisées à l'élément par défaut de leur type (noté 0 pour des raisons de lisibilité).

$$\text{bloc}\ \frac{\begin{array}{c} bt.\texttt{param} = t_0, ..., t_{n-1} \\ L' = \{\texttt{id} = id; \texttt{form} = \texttt{block}; \texttt{type} = bt; \texttt{code} = (); \texttt{env} = L.\texttt{env}\} \end{array}}{L, (v_0, ..., v_{n-1}, v^*), (\texttt{block}\ id\ bt\ i^*, i'^*) \rightarrow (L', (v_0, ..., v_{n-1}), i^*)(L, v^*, i'^*)}$$

La construction `block` crée une nouvelle fenêtre avec le code et l'étiquette du bloc. Les piles de valeurs de l'ancienne et de la nouvelle fenêtre sont mises à jour en fonction du type du bloc.

$$\text{boucle}\ \frac{\begin{array}{c} bt.\texttt{param} = t_0, ..., t_{n-1} \\ L' = \{\texttt{id} = id; \texttt{form} = \texttt{loop}; \texttt{type} = bt; \texttt{code} = i^*; \texttt{env} = L.\texttt{env}\} \end{array}}{L, (v_0, ..., v_{n-1}, v^*), (\texttt{loop}\ id\ bt\ i^*, i'^*) \rightarrow (L', (v_0, ..., v_{n-1}), i^*)(L, v^*, i'^*)}$$

De manière similaire, la construction `loop` génère une nouvelle fenêtre, mais cette fois, le code de l'étiquette contient les instructions à l'intérieur de la boucle.

$$\text{cond-then}\ \frac{v_0 \neq \texttt{i32 0}}{(v_0, v^*), ((\texttt{if}\ id\ bt\ \texttt{then}\ i^*\ \texttt{else}\ i'^*), i''^*) \rightarrow v^*, ((\texttt{block}\ id\ bt\ i^*), i''^*)}$$

Une conditionnelle est réduite en un bloc avec le code de la branche `then` lorsque la valeur au sommet de la pile correspond à un booléen vrai.

$$\text{cond-else}\ \frac{v_0 = \texttt{i32 0}}{(v_0, v^*), ((\texttt{if}\ id\ bt\ \texttt{then}\ i^*\ \texttt{else}\ i'^*), i''^*) \rightarrow v^*, ((\texttt{block}\ id\ bt\ i'^*), i''^*)}$$

Si la valeur au sommet de la pile est un booléen faux, la conditionnelle se réduit en un bloc avec le code de la branche `else`.

$$\text{br-bloc}\ \frac{L.\texttt{id} = id \qquad L.\texttt{form} = \texttt{block} \qquad L.\texttt{type.result} = t_0, ..., t_{n-1}}{(L, (v_0, ..., v_{n-1}, v^*), (\texttt{br}\ id, i^*))(L', v'^*, i'^*) \rightarrow L', (v_0, ..., v_{n-1}, v'^*), i'^*}$$

Lors d'un branchement vers l'identifiant *id*, si l'étiquette courante est un bloc avec cet

identifiant, nous sortons de la fenêtre courante, nous dépilons les valeurs nécessaires et nous les plaçons au sommet de la pile de la fenêtre suivante.

$$\text{br-boucle}\frac{L.\texttt{id} = id \qquad L.\texttt{form} = \texttt{loop} \qquad L.\texttt{type.param} = t_0, ..., t_{n-1}}{L, (v_0, ..., v_{n-1}, v^*), (\texttt{br}\ id, i^*) \to L, (v_0, ..., v_{n-1}), L.\texttt{code}}$$

Lors d'un branchement vers une boucle avec l'identifiant *id*, nous tronquons la pile de valeurs et nous remplaçons les instructions actuelles par le code de la boucle, réexécutant ainsi son corps.

$$\text{br-fonction}\frac{L.\texttt{id} = id \qquad L.\texttt{form} = \texttt{func} \qquad L.\texttt{type.result} = t_0, ..., t_{n-1}}{(L, (v_0, ..., v_{n-1}, v^*), (\texttt{br}\ id, i^*)), (L', v'^*, i'^*) \to L', (v_0, ..., v_{n-1}, v'^*), i'^*}$$

Cette règle fonctionne de manière similaire à la précédent mais pour les fonctions, où un branchement entraîne la sortie de la fonction.

$$\text{br-échec}\frac{L.\texttt{id} \neq id \qquad L.\texttt{form} \neq \texttt{func}}{(L, v^*, (\texttt{br}\ id, i^*))(L', v'^*, i'^*) \to L', (v^*, v'^*), \texttt{br}\ id}$$

Lorsque l'on exécute `br` avec un identifiant *id* différent de celui de l'étiquette courante, ce dernier est ignoré et nous gardons le suivant. La pile de valeurs des deux étiquettes est concaténée et le branchement est réexécuté. Cela correspond au fait de pouvoir effectuer un branchement dans des blocs imbriqués. Cette règle ne s'applique pas lorsque l'étiquette est celle d'une fonction, un branchement étant toujours local à une fonction.

$$\text{exit-func}\frac{L.\texttt{form} = \texttt{func}}{(L, v^*, ())(L', v'^*, i'^*) \to L', (v^*, v'^*), i'^*}$$

Lorsque la liste des instructions d'une fonction est vide, nous passons à la fenêtre précédente, qui avait été sauvegardée au moment du `call`, en concaténant les piles de valeurs.

$$\texttt{br\_if}\text{-oui}\frac{v_0 \neq \texttt{i32.0}}{(v_0, v^*), (\texttt{br\_if}\ id, i^*) \to v^*, \texttt{br}\ id}$$

Si la valeur au sommet de la pile est vraie, un branchement conditionnel se transforme en branchement inconditionnel.

$$\texttt{br\_if}\text{-non}\frac{v_0 = \texttt{i32.0}}{(v_0, v^*), (\texttt{br\_if}\ id, i^*) \to v^*, i^*}$$

Si la valeur est fausse, l'instruction de branchement conditionnel est éliminée.

$$\text{exit-bloc}\frac{L.\texttt{form} \neq \texttt{func} \qquad L'' = L'[\texttt{env} \leftarrow L.\texttt{env}]}{(L, v^*, ())(L', v'^*, i'^*) \to L'', (v^*, v'^*), i'^*}$$

Une fois toutes les instructions d'un bloc ou d'une boucle exécutées, nous passons à la

fenêtre précédente, qui avait été sauvegardée au moment d'entrer dans le `block` ou dans la `loop`, en concaténant les piles et en récupérant l'environnement précédent.

$$\texttt{ref.null}\ \frac{}{v^*, (\texttt{ref.null}\ ht, i^*) \to (\texttt{null}\ ht, v^*), i^*}$$

Cette instruction place une référence nulle de type *ht* au sommet de la pile.

$$\texttt{ref.as\_non\_null}\ 1\ \frac{v_0 = \texttt{null}\ ht}{(v_0, v^*), (\texttt{ref.as\_non\_null}, i^*) \to \texttt{trap}}$$

Si la valeur au sommet de la pile est une référence nulle, l'instruction `ref.as_non_null` déclenche une erreur.

$$\texttt{ref.as\_non\_null}\ 2\ \frac{v_0 \neq \texttt{null}\ ht}{(v_0, v^*), (\texttt{ref.as\_non\_null}, i^*) \to (v_0, v^*), i^*}$$

Si la valeur n'est pas nulle, l'instruction n'affecte pas la pile.

$$\texttt{unreachable}\ \frac{}{v^*, (\texttt{unreachable}, i^*) \to \texttt{trap}}$$

L'instruction `unreachable` met fin à l'exécution.

Cette sémantique alternative va nous servir de fondation pour définir une sémantique pour WasmGC mais également pour effectuer la preuve de correction de notre compilateur OCaml vers WasmGC.

# 4 Compilation vers Wasm

Bien qu'il soit possible d'écrire des programmes Wasm à la main, cette tâche reste fastidieuse. Le langage Wasm ne propose qu'un nombre restreint de constructions, ce qui rend difficile le développement de programmes complexes. Cela est conforme à son objectif initial : être une cible de compilation plutôt qu'un langage source. Ce chapitre explore les différentes manières dont plusieurs catégories de langages peuvent être compilées vers Wasm. Nous proposons de classer ces langages en trois catégories :

- langages sans glaneur de cellules (§4.1) ;
- langages avec glaneur de cellules (§4.2) ;
- langages dynamiques (§4.5).

## 4.1 Langages sans glaneur de cellules

La première catégorie de langages qui a pu être compilée efficacement vers Wasm regroupe ceux sans environnement d'exécution, donc sans glaneur de cellules. Il s'agit par exemple de langages tels que C, C++ ou Rust. Ces langages se distinguent par leur gestion manuelle de la mémoire. Nous allons ici détailler les différents mécanismes de gestion de la mémoire dans ce contexte.

Lors de l'écriture d'un programme, il est souvent nécessaire d'allouer de la mémoire. L'allocation de mémoire peut se faire de manière statique, quand la quantité de mémoire nécessaire est connue à l'avance, ou de manière dynamique, quand cette quantité dépend de l'exécution (environnement, interactions avec l'utilisateur, etc.).

### 4.1.1 Allocation statique

Prenons l'exemple du langage C, où il est possible d'allouer de la mémoire statiquement à deux endroits :

- dans le segment `.data` lors d'une initialisation explicite,
- dans le segment `.bss` lors d'une initialisation par zéro.

Par exemple, dans le programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
char *s = "this should appear in .data";
static int i;

void main(void) {
  printf("s = %s\n", s);
  printf("i = %d\n", i);
}
```

La variable `s` sera placée dans la section `.data` de l'exécutable généré, tandis que `i` sera dans la section `.bss`. La section `.bss` a l'avantage de ne presque pas occuper d'espace sur le disque, étant matérialisée seulement lors du chargement du programme en mémoire. Par exemple, un grand tableau initialisé à zéro occupera très peu d'espace disque, comparé à sa version effectivement allouée en mémoire lors de l'exécution du programme. Nous pouvons inspecter la taille des différentes sections comme suit :

```
$  gcc static_alloc.c && ./a.out
s = this should appear in .data
i = 0
$ size ./a.out
   text    data     bss     dec     hex filename
   1415     592       8    2015     7df ./a.out
```

En Wasm, l'allocation statique dans la section `.data` possède un équivalent via les champs `data` d'un module. Ces champs sont des séquences de bytes, pouvant être qualifiées d'actifs ou bien de passifs. Lorsqu'un champ est actif, il est écrit dans la mémoire linéaire lors de l'initialisation du programme. Par défaut, ce champ fait référence à la dernière mémoire déclarée et est écrit au début de celle-ci. Il est possible de spécifier un indice où commencer l'écriture :

```
(module
  (memory $m 1)
  (data (memory $m) (i32.const 42) "I'm an active data")
  (func $start
    i32.const 42
    i32.load8_u  ;; the character at offset 42
    i32.const 73 ;; ascii code for 'I'
    ;; assert the correct char was in memory
    i32.eq
    (if (then return))
    unreachable
  )
  (start $start)
)
```

Dans le cas où il est passif, il peut être chargé dans la mémoire à l'exécution avec l'instruction `memory.init`. Un champ est passif s'il ne fait référence à aucune mémoire et n'a pas d'indice

spécifié.

Wasm ne dispose pas d'un mécanisme équivalent à la section `.bss`. Toutefois, il est possible d'émuler ce comportement, par exemple, en plaçant une boucle d'initialisaiton au début de la fonction `$start`.

### 4.1.2 Allocation dynamique

L'allocation statique, bien que simple, ne suffit pas pour la majorité des programmes. L'allocation dynamique de mémoire en C était initialement réalisée uniquement via la fonction `sbrk`[1], dont la signature est la suivante :

```c
#include <unistd.h>

void *sbrk(intptr_t increment);
```

Cette fonction manipule une variable globale représentant l'adresse de fin du tas. Appeler `sbrk` avec la valeur $0$ renvoie cette adresse. Lorsqu'on souhaite allouer de la mémoire, nous appelons `sbrk` avec la quantité souhaitée. Si la mémoire disponible est insuffisante, la fonction renvoie $-1$, sinon elle renvoie l'ancienne valeur de la limite du tas, correspondant à l'emplacement où écrire la mémoire demandée. En appelant `sbrk` avec une valeur négative, il est possible de libérer l'espace des dernières allocations.

Le programme suivant illustre l'utilisation de `sbrk` en allouant un tableau de $n$ entiers, où $n$ est passé comme argument au programme. Il n'utilise pas `printf` pour afficher les valeurs des adresses afin d'éviter des allocations implicites. À la place, l'affichage est réalisé au moyen de `puts`, `snprintf` et d'un tampon `tmp`. Ainsi, il est possible de voir que les appels à `sbrk` modifient la valeur correspondant à la limite du tas exactement de la quantité de mémoire allouée.

```c
#define _DEFAULT_SOURCE // needed for sbrk

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// the little dance with tmp is to avoid hidden allocations which would break
↪  the example
char tmp[64];

int main(int argc, char *argv[]) {

  // Getting a number from the user
  const int n = atoi(argv[1]);

  // Allocating the space for n integers
```

1. De même que la fonction `brk`, que nous omettons ici pour des raisons de concision, car elle agit de manière similaire.

```
  int *old_address = sbrk(n * sizeof(int));

  // Checking the limit has been updated
  void *current_address = sbrk(0);
  snprintf(tmp, sizeof(tmp), "%p", old_address);
  puts(tmp);
  snprintf(tmp, sizeof(tmp), "%p", current_address);
  puts(tmp);
  snprintf(tmp, sizeof(tmp), "%p", old_address + n - 1);
  puts(tmp);

  // We can now use the memory as we want
  for (int i = 0; i < n; i++) {
    old_address[i] = 2*i;
  }

  printf("%d", old_address[n - 1]);;

  return 0;
}
```

Le programme ci-dessus produit :

```
$ gcc sbrk.c && ./a.out 22
0x5560f575d000 # the old limit
0x5560f575d058 # the new limit
0x5560f575d054 # the last address we are allowed to write to
42
```

L'allocation via `sbrk` agit comme un *bump allocator*. Un tel allocateur peut être écrit facilement en Wasm à l'aide d'une variable globale :

```
(module

  (memory 1)

  (global $limit (mut i32) (i32.const 0))

  (func $sbrk (param $increment i32) (result i32)
    global.get $limit
    global.get $limit
    local.get $increment
    i32.add
    global.set $limit
  )

  (func $start
```

```
    i32.const 42
    call $sbrk
    ;; we can now write 42 bytes
    ;; starting from the result of sbrk
    ;; …
    drop
  )

  (start $start)
)
```

Cet exemple est simplifié. En pratique, il faudrait également utiliser `memory.grow` dans la fonction `$sbrk` pour augmenter la taille de la mémoire lorsque celle-ci est pleine, et renvoyer $-1$ lorsque la taille maximale est atteinte.

Cependant, cette approche est limitée car elle ne permet pas de libérer facilement la mémoire dans un ordre arbitraire. Pour pallier ce problème, des allocateurs plus sophistiqués, accessibles via des fonctions comme `malloc` et `free`, sont souvent fournis. `malloc` alloue une quantité de mémoire spécifiée et renvoie un pointeur vers cette mémoire, tandis que `free` libère la mémoire associée à un pointeur précédemment renvoyé par `malloc` :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char tmp[64];

int main(int argc, char *argv[]) {

  const int n = atoi(argv[1]);
  // Allocating the space for n integers, two times
  int *first = malloc(n * sizeof(int));
  int *second = malloc(n * sizeof(int));

  for (int i = 0; i < n; i++) {
    first[i] = 2*i;
  }
  for (int i = 0; i < n; i++) {
    second[i] = first[n - 1 - i];
  }

  // We can free first
  free(first);
  // And still use second!
  printf("%d", second[0]);

  free(second);
```

```
  return 0;
}
```

Le programme produit :

```
$ gcc malloc.c && ./a.out 22
42
```

En C, ces allocateurs sont souvent construits sur `sbrk`. Il est tout à fait possible de les écrire en Wasm, voire de compiler ceux existants en C vers Wasm, avec quelques ajustements spécifiques, ce qui permet de réutiliser les allocateurs C et de ne pas avoir à en coder un nouveau pour Wasm.

## 4.2 Langages avec glaneur de cellules

De nombreux langages de programmation modernes, tels qu'OCaml, Java, Scheme, Python, C# et Go, utilisent un glaneur de cellule[2] pour gérer automatiquement la mémoire. Le but d'un glaneur de cellules est de libérer automatiquement la mémoire allouée par le programme lorsqu'elle n'est plus utilisée, évitant ainsi au programmeur de devoir gérer manuellement la libération des ressources. Cette gestion automatique permet de prévenir des erreurs courantes, telles que les fuites de mémoire (mémoire allouée mais jamais libérée), les libération prématurées (mémoire libérée alors qu'elle est encore nécessaire) ou les doubles libérations (libération de mémoire déjà libérée).

Récemment, des techniques d'analyse statique, comme celles employées dans le langage Rust, ont été développées pour garantir l'absence de telles erreurs. Cependant, ces techniques ajoutent une certaine complexité pour le programmeur. Elles ne remplacent donc pas totalement les autres méthodes de gestion de la mémoire.

### 4.2.1 Les différents types de glaneur de cellules

Pour gérer automatiquement la mémoire, un glaneur de cellule doit être capable de distinguer les zones de mémoire encore accessibles (vivantes) de celles qui ne le sont plus (mortes) et peuvent donc être libérées. Deux techniques principales permettent d'accomplir cette tâche.

La première, appelée *tracing*, consiste à partir des *racines* — les valeurs dont nous savons qu'elles sont atteignables, comme les variables globales, les valeurs sur la pile et dans les registres. Ensuite, le glaneur parcourt la mémoire en suivant les pointeurs accessibles à partir de ces racines, construisant ainsi un graphe de toutes les valeurs vivantes. Les valeurs non atteignables, étant mortes, peuvent alors être libérées.

La seconde technique, nommée *reference counting*, associe à chaque bloc de mémoire un compteur indiquant le nombre de pointeurs vivants qui y font référence. Chaque fois qu'un nouveau pointeur est créé pour un bloc de mémoire, le compteur est décrémenté. Si ce compteur atteint zéro, la mémoire n'est plus accessible et peut être libérée, ce qui peut potentiellement entraîner la libération récursive d'autres blocs de mémoire.

2. De l'anglais *Garbage Collector*. Cette traduction a pour avantage de préserver l'acronyme *GC*.

En résumé, la technique du *tracing* part des racines pour identifier les objets vivants, tandis que le *reference counting* commence par les objets morts et en libère d'autres récursivement. Bien que ces deux méthodes soient différentes dans leur approche, elles partagent des similarités comme décrit par Bacon [48].

Les langages comme Java, OCaml ou Haskell utilisent la technique du *tracing* dans leur glaneur de cellule, tandis que Python utilise du *reference counting*[3].

### 4.2.2 Compilation des langages avec glaneur de cellules vers Wasm

Nous pouvons tout d'abord envisager de compiler les langages avec glaneur de cellules sans apporter aucune modification à Wasm. Cela pourrait par exemple être fait en programmant un glaneur de cellules à la main en Wasm ou bien en compilant les environnements d'exécution (généralement écrits en C) de ces langages vers Wasm. Dès lors, un premier problème se pose. Il s'agit de celui de l'inspection de la pile par le glaneur de cellules. En effet, pour des raisons de sûreté et d'efficacité, Wasm limite fortement les capacités d'introspection des programmes et il n'est en aucun cas possible d'inspecter le contenu de la pile lors de l'exécution. Dès lors, il faudrait se passer de la pile d'appel Wasm et à la place mettre en œuvre une *shadow stack*. C'est par exemple ce qui a été fait pour les langages Go et Haskell. Cependant, cela un impact très fort sur les performances et n'est pas souhaitable en pratique. Une autre solution serait alors d'étendre Wasm pour qu'il intègre lui-même un mécanisme de glaneur de cellule. Pour cela, il faut alors réussir à concevoir une *interface de glaneur de cellules* qui conviendrait à la plupart des langages. Pour comprendre ce que cela implique d'ajouter un glaneur de cellules dans Wasm lui-même, il faut se pencher un peu plus sur les mises en œuvre de glaneur de cellules existantes.

### 4.2.3 La question des pointeurs

Dans les deux techniques, *tracing* et *reference counting*, un problème commun se pose lors du parcours récursif : comment distinguer les valeurs représentant des pointeurs de celles qui sont des scalaires ? Cette distinction est nécessaire pour effectuer le parcours correct de la mémoire. En effet, nous nous attendons à ce que tous les glaneur de cellules soient *corrects*, *i.e.* qu'ils ne libèrent pas de la mémoire alors qu'elle est encore atteignable par le programme, sans quoi l'exécution pourrait être fausse. De même, nous préférons généralement qu'ils soient *complets*, *i.e.* que toute mémoire qui n'est plus atteignable soit libérée après un certain temps, sans quoi nous aurions des fuites mémoires. Deux approches principales sont utilisées pour résoudre ce problème.

La première approche, dite *conservative*, est parfois appelée GC de Boehm, du nom de sa première mise en œuvre [15]. Elle consiste à traiter une valeur comme un pointeur si elle *ressemble* à un pointeur. Plusieurs critères peuvent être utilisés pour minimiser les erreurs, comme l'alignement des pointeurs ou la vérification de l'appartenance à une liste d'adresses allouées. Le glaneur est alors correct, mais pas complet et peut provoquer des fuites mémoires. Cette méthode, bien qu'imparfaite, est souvent suffisante en pratique. Des langages comme Crystal ou Guile, ainsi que certains logiciels écrits en C tels qu'Inkscape, utilisent cette technique. En effet, un GC de Boehm peut facilement être intégré sous forme de bibliothèque dans un langage à gestion manuelle de la mémoire.

3. ainsi qu'un second glaneur de cellules utilisant la technique du *tracing* pour collecter les cycles.

La seconde approche, dite *précise*, consiste à stocker explicitement l'information sur les pointeurs de manière à ce qu'elle soit accessible par le glaneur de cellules lors de l'exécution. Bien que cette méthode entraîne un coût supplémentaire, elle est largement utilisée dans les langages modernes comme Java et OCaml. En effet, elle permet d'obtenir un glaneur de cellules qui soit à la fois correct et complet. Les détails de mise en œuvre de ces techniques sont souvent considérés comme secondaires et il est difficile de trouver de la littérature à leur sujet. Cependant, des parallèles peuvent être établis avec les techniques utilisées pour mettre en œuvre le polymorphisme, sujet bien plus documenté. La section suivante explore ces techniques sous l'angle du polymorphisme, tout en mettant en lumière leur pertinence pour la discrimination des pointeurs des scalaires par les glaneurs de cellules.

## 4.3 Techniques de représentation des valeurs en mémoire

Une fonction est dite *polymorphe* lorsqu'elle peut être utilisée avec des valeurs de types différents. Ce type de fonction doit pouvoir gérer des valeurs qui peuvent avoir des représentations en mémoire différentes, utiliser des conventions d'appel distinctes, ou nécessiter des traitements spécifiques par le glaneur de cellules.

### 4.3.1 Méthodes manuelles : la méta-programmation

Pour commencer, nous examinons les techniques utilisées lorsqu'un langage ne prend pas en charge directement le polymorphisme.

**Duplication manuelle du code source** Dans un langage sans mécanisme de polymorphisme, la solution la plus simple consiste à dupliquer le code source en écrivant une version de la fonction pour chaque type souhaité. Bien que cette méthode soit directe, elle pose des problèmes de maintenance, de lisibilité, d'augmentation de la taille du code, ainsi que de temps de développement.

**Génération de code source** Une alternative à la duplication manuelle est d'automatiser la création de ces différentes versions à l'aide d'un programme générateur de code. Cette approche permet de réduire les erreurs humaines et d'améliorer la maintenabilité du code.

**Transformation de code source** Mieux encore, il est souvent possible de générer ce code via un *préprocesseur*, comme décrit par Solntseff [4] sous le nom de *syntax-macro extension*. Cette technique est largement utilisée [42] dans le langage C via son préprocesseur [14], et est documentée par Rebelsky [94]. Par exemple, à partir du programme C suivant :

```c
// asking for an int version
#define TYPE int
#define TYPED(X) int_##X

// generic code
struct TYPED(list) {
  TYPE head;
```

```
  struct TYPED(list) *tail;
};

TYPE hd(struct TYPED(list) *l) {
  if (l) { return l->head; }
  _Exit(42);
}
```

En utilisant la commande `l@tex:~$ cpp -P list.c`, nous obtenons :

```
struct int_list {
  int head;
  struct int_list *tail;
};

int hd(struct int_list *l) {
  if (l) { return l->head; }
  _Exit(42);
}
```

En pratique, le code générique serait dans un fichier séparé, et les macros `TYPE` et `TYPED(X)` seraient définis avant son inclusion pour générer une version spécialisée quand cela est nécessaire.

**String mixins** Les *String mixins* [136] permettent de générer du code source en utilisant toutes les constructions du langage source qui peuvent être évaluées à la compilation. La chaîne de caractères obtenues doit correspondre à un code source valide et sera alors compilée statiquement. Cela revient à utiliser le langage source comme un langage de macros. Par exemple, en partant du fichier suivant :

```
template GenList(string Type) {
  const char[] GenList =
    "struct List_" ~ Type ~ " {\n" ~
    " " ~ Type ~ " head;\n" ~
    " " ~ "List_" ~ Type ~ "* tail;\n" ~
    "}\n" ~
    Type ~ " hd(List_" ~ Type ~ "* l) {\n" ~
    " if (l) { return l.head; }\n" ~
    " assert(0);\n" ~
    "}";
}

mixin(GenList!("int"));
```

Nous obtenons, avec la commande `l@tex:~$ gdc -c list.d -fsave-mixins=/dev/stdout`, le résultat suivant :

```
// expansion at list.d:13:1
struct List_int {
  int head;
  List_int* tail;
}
int hd(List_int* l) {
  if (l) { return l.head; }
  assert(0);
}
```

**Autres techniques** De nombreuses autres techniques existent, telles que la transformation du flux de tokens en Rust [133] ou encore la transformation de l'AST en OCaml [130]. Pour une liste plus exhaustive, le lecteur est invité à consulter l'article de Lilis [109].

### 4.3.2 Monomorphisation

La *monomorphisation* est une technique utilisée dans les langages de programmation pour gérer le polymorphisme en créant des versions spécialisées de fonctions génériques pour chaque combinaison de types avec laquelle elles sont utilisées. Cette méthode permet de produire un ensemble de fonctions spécifiques (monomorphes) à partir d'une fonction polymorphe, d'où le terme *monomorphisation*.

#### Processus de monomorphisation

Le processus de monomorphisation commence par une analyse statique du programme pour identifier toutes les combinaisons de types utilisées dans les appels aux fonctions polymorphes. Cela implique de parcourir le graphe d'appel du programme et d'ajouter chaque combinaison de types rencontrée à un ensemble dédié pour chaque fonction polymorphe.

Une fois cet ensemble complet, la fonction polymorphe originale est supprimée du programme. À sa place, des versions spécialisées (monomorphes) sont générées et insérées dans le programme. Enfin, chaque site d'appel est modifié pour appeler la version monomorphe appropriée.

La monomorphisation est utilisée pour mettre en œuvre les génériques en Rust [101], les *templates* en C++ [17], les fonctions polymorphes dans l'implémentation MLton de SML [57], les paquets génériques et les sous-programmes en Ada [8, 9] ainsi que l'extraction de code en Coq [105] ou plus récemment Go [135] qui implémente une méthode hybride (l'article cité propose une méthode alternative à celle effectivement implémentée dans Go mais décrit tout de même cette dernière).

Bien que cette technique soit parfois confondue avec les méthodes de méta-programmation, elle se distingue par son objectif spécifique de gestion du polymorphisme. Alors que la méta-programmation peut permettre un polymorphisme de manière incidente, la monomorphisation est explicitement conçue pour cet usage.

La mise en œuvre d'un glaneur de cellules pour du code monomorphisable est décrite parGoldberg [20].

### Avantages de la monomorphisation

**Spécialisation du code** L'un des principaux avantages de la monomorphisation est l'efficacité du code produit. Comme les types sont entièrement connus à la compilation, le compilateur peut générer du code machine spécialisé, optimisé pour chaque type spécifique. Cela inclut l'utilisation d'instructions assembleur ou de conventions d'appel optimisées pour le type traité, ce qui améliore les performances d'exécution.

**Gestion optimale de la mémoire** La monomorphisation permet une utilisation optimale de la mémoire, car seules les valeurs nécessaires sont stockées, sans avoir besoin de méta-données supplémentaires pour gérer différents types. Cela peut réduire la surcharge mémoire associée à la gestion du polymorphisme. De plus, elle permet d'adopter des représentations optimisées des valeurs en mémoire, par exemple, un tableau de booléens pourra être représenté par un tableau de bits.

**Simplicité de mise en œuvre** La mise en œuvre de la monomorphisation est relativement directe. Comme décrit précédemment, elle repose sur trois étapes : la collecte des informations sur les types, la génération des fonctions spécialisées, et la mise à jour des sites d'appel.

### Inconvénients de la monomorphisation

**Coût de compilation** L'un des principaux inconvénients de la monomorphisation est l'augmentation significative du temps de compilation et de la consommation mémoire, proportionnelle au nombre de versions spécialisées générées pour chaque fonction polymorphe. Ce phénomène est bien connu des utilisateurs de langages comme C++ et Rust, où les temps de compilation peuvent devenir prohibitifs.

**Taille du binaire** La génération de multiples versions spécialisées d'une fonction polymorphe peut également entraîner une augmentation substantielle de la taille du binaire. Ce problème est particulièrement aigu dans les programmes comportant de nombreuses fonctions polymorphes avec diverses combinaisons de types.

**Langages dynamiques** Dans les langages dynamiques, la monomorphisation statique est impraticable, car les types ne sont pas connus à l'avance. Par exemple, dans un langage comme Python, où les types des variables peuvent changer à l'exécution, la monomorphisation ne peut pas être appliquée.

**Modularité** La monomorphisation pose également des défis en termes de modularité. Pour être efficace, elle nécessite l'accès à l'ensemble du code source du programme, ce qui complique la compilation modulaire. Alternativement, une représentation polymorphe doit être maintenue jusqu'à l'étape de l'édition de lien pour permettre une compilation séparée.

**Récursion polymorphe** Un autre inconvénient majeur de la monomorphisation est sa limitation au polymorphisme de deuxième classe. En particulier elle ne peut pas gérer les cas de récursion polymorphe. Ce type de récursion se produit lorsqu'une fonction polymorphe s'appelle elle-même avec un type différent de celui de l'appel initial. Cela peut conduire à des

problèmes d'infinité dans le processus de monomorphisation où le compilateur peut entrer dans une boucle infinie en tentant de générer un nombre infini de versions spécialisées. Ce problème est illustré par l'exemple de types récursifs non-uniformes en Rust dans cette mise en œuvre des *Binary Random-Access Lists* [37] :

```
pub enum Enum<T> {
  Nil,
  Zero(Box<Enum<(T, T)»),
  One { head : T, tail : Box<Enum<(T, T)» },
}
impl<T> Enum<T> {
  pub fn len(&self) -> usize {
    match self {
      Self::Nil => 0,
      Self::Zero(sub) => 2 * sub.len(),
      Self::One { tail, .. } => 1 + 2 * tail.len()
    }
  }
}
fn main() {
  let val: Enum<usize> =
    Enum::One { head : 42, tail: Box::new(Enum::Nil)};
  println!("len: {}", val.len());
}
```

Le compilateur Rust boucle indéfiniment sur ce fichier puisqu'il essaie de générer un nombre infini de version spécialisées de la fonction `len`. Il faut noter que sans fonction `main`, le compilateur ne boucle pas. Cela est dû au fait que la monomorphisation ne peut se produire qu'au moment de l'édition de lien. Sans fonction `main`, la fonction `len` ne peut être spécialisée puisque ses sites d'appels ne sont pas encore complètement connus.

Ce problème se manifeste également dans d'autres langages comme C++, où il existe une limite au nombre d'instanciations de templates, au-delà de laquelle une erreur est levée. Bien qu'il soit envisageable de détecter statiquement l'impossibilité de monomorphiser une fonction, ce problème reste ouvert à ce jour. La question est abordée dans un article de Griesmer [115], mais les auteurs proposent une analyse conservative qui rejette plus de programmes que nécessaire.

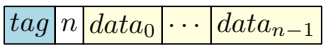
### Optimisations possibles

**Éviter les spécialisations inutiles** Il est possible d'optimiser la monomorphisation en évitant de générer des versions spécialisées inutiles. Par exemple, si un paramètre de type n'est pas utilisé dans une fonction, il n'est pas nécessaire de spécialiser cette fonction pour ce paramètre. Le compilateur Rust effectue cette optimisation, bien que dans la pratique, les cas où un paramètre de type est complètement inutilisé soient rares (il s'agit généralement de types fantômes).

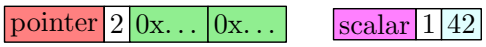
**Polymorphisation** La polymorphisation est une autre optimisation qui concerne les fermetures de fonctions. Les fermetures héritent des paramètres de type de la fonction à laquelle elles appartiennent. Dans ce cas, il est courant que des paramètres de types soient inutilisés. L'optimisation qui évite aux fermetures d'être spécialisées pour les paramètres de types inutilisés est appelée la *polymorphisation*. Sa mise en œuvre initiale dans le compilateur Rust est décrite par Wood [121].

### 4.3.3 Emballage complet

La technique de l'emballage complet utilise une *représentation uniforme* des valeurs, où toutes les valeurs, qu'elles soient scalaires (comme les entiers, booléens ou caractères) ou non, sont représentées par des pointeurs. Les valeurs scalaires sont stockées dans des *blocs* alloués sur le tas, souvent appelés *boîtes*, et sont représentées par des pointeurs vers ces blocs. Chaque bloc contient des méta-données qui décrivent sa taille ainsi que le type des données qu'il contient. Les valeurs qui sont déjà des pointeurs (celles qui le seraient même dans un code monomorphisé par exemple) restent des pointeurs, mais elles pointent désormais vers un bloc contenant des méta-données et les données effectives.



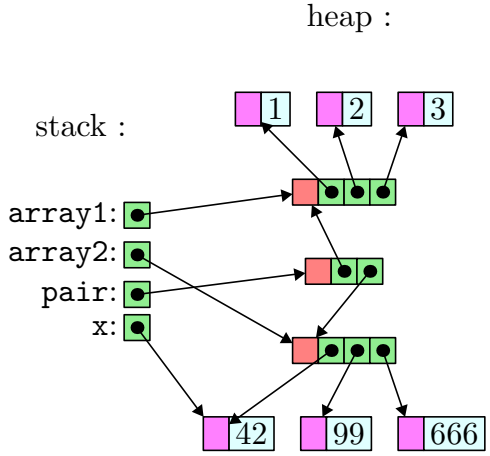

Grâce à cette représentation uniforme, les fonctions polymorphes peuvent manipuler des paramètres de types variés de manière unique. À l'exécution, il est souvent nécessaire d'emballer et de déballer les valeurs pour correspondre à cette représentation. Lorsqu'il faut différencier les scalaires et les pointeurs (par exemple, pour un glaneur de cellules), cette information est disponible dans les méta-données des blocs. Par exemple :




Cette technique est utilisée dans la mise en œuvre CPython de Python, où les blocs sont appelés des `PyObject` [131]. Considérons le code suivant :

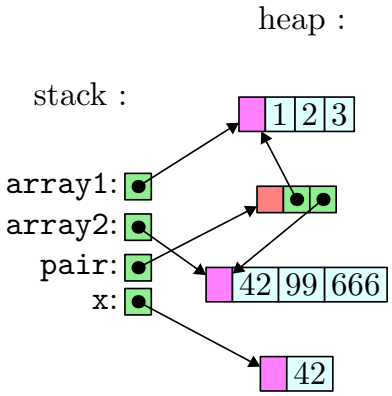
```
array1 = [1, 2, 3]
array2 = [42, 99, 666]
pair = (array1, array2)
x = 42
```

La représentation en mémoire de ce code serait la suivante :



Il est intéressant de noter que les variables `x` et `array2[0]` pointent vers le même bloc. Cela s'explique par le fait que Python pré-alloue les petits entiers pour des raisons d'optimisation.

Nous pourrions nous demander pourquoi les tableaux d'entiers ne sont pas simplement représentés par un bloc de scalaires plutôt que par un bloc de pointeurs vers des blocs individuels, ce qui donnerait une représentation comme celle-ci :




La raison en est que les entiers en Python ont une taille arbitraire. Un entier peut donc occuper une quantité d'espace mémoire variable, rendant impossible l'utilisation d'un bloc unique composé de scalaires avec une taille fixe. C'est pourquoi chaque entier est encapsulé dans son propre bloc, avec sa taille stockée dans les méta-données. Il serait envisageable d'utiliser un encodage à taille variable pour résoudre ce problème, mais à notre connaissance, aucune expérimentation sérieuse n'a été menée dans ce sens.

### Avantages de l'emballage complet

**Facilité de mise en œuvre** L'emballage est la technique la plus simple à mettre en œuvre. Elle ne nécessite que l'insertion de code pour emballer, déballer et lire les méta-données des blocs à l'exécution, ce qui en fait une solution accessible et directe.

**Coût de la compilation** Le coût de compilation lié à l'emballage est extrêmement faible. Chaque fonction polymorphe n'est compilée qu'une seule fois, quelle que soit la diversité des types de ses paramètres, réduisant ainsi le temps de compilation et la complexité.

**Taille du binaire** La taille du binaire produit par l'emballage est également très réduite. Étant donné que chaque fonction polymorphe n'est traduite qu'en un seul segment de code assembleur, la quantité de code générée reste minimale.

**Compatibilité avec les langages dynamiques** L'emballage est parfaitement adapté aux langages dynamiques, où les types des variables peuvent changer au cours de l'exécution. La représentation uniforme permet de gérer cette flexibilité sans difficulté.

**Modularité** L'emballage est une technique modulaire par excellence. Chaque fonction peut être compilée indépendamment, sans connaissance préalable de ses sites d'appel. Cette modularité facilite la compilation séparée.

**Support de la récursion polymorphe** L'emballage gère la récursion polymorphe efficacement. Lorsqu'une fonction s'appelle elle-même avec un nombre non borné de types, ceux-ci partagent la même représentation en mémoire, permettant ainsi leur traitement par le même code assembleur, sans nécessiter de versions spécialisées.

#### Désavantages

**Performance d'exécution** Un des inconvénients majeurs de l'emballage est sa lenteur d'exécution. Cette technique impose des coûts additionnels liés à l'initialisation, entraîne des accès indirects en mémoire, réduit la localité des données, et alourdit la charge de travail du glaneur de cellules en raison de l'augmentation de la consommation mémoire.

**Utilisation de la mémoire** L'emballage est également inefficace en termes d'utilisation de la mémoire. Par exemple, un petit entier en Python utilise 28 octets de mémoire :

```
»> import sys
»> a = 42
»> sys.getsizeof(a)
28
```

Cette surconsommation de mémoire est due à l'infrastructure de blocs et de méta-données nécessaire pour représenter chaque valeur, même les plus simples.

### 4.3.4 Emballage sélectif

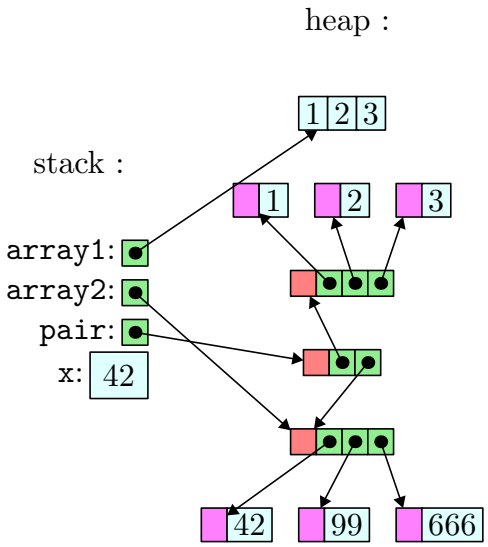
Cette technique consiste à n'emballer que les valeurs qui seront utilisées de manière polymorphe, tout en conservant les valeurs non emballées lorsqu'il est possible de déterminer statiquement qu'elles seront toujours utilisées dans un contexte monomorphe. C'est la méthode employée par Java [35]. En Java, les valeurs de type `int` sont des entiers de 32 bits non emballés, tandis que les valeurs de type `Integer` sont des entiers de 32 bits emballés. Seules les valeurs de type `Integer` peuvent être utilisées dans des génériques, tels que `ArrayList`. Cependant, Java permet la création de tableaux qui ne peuvent contenir que des valeurs de type `int`, notés `int[]`.

Prenons l'exemple du code suivant :

```
import java.util.ArrayList;
import java.util.AbstractMap.SimpleEntry;

public class layout {
  public static void main(String[] args) {
    int[] array1 = { 1, 2, 3 };
    ArrayList<Integer> array2 = new ArrayList<>();
    array2.add(42);
    array2.add(99);
    array2.add(666);
    SimpleEntry<int[], ArrayList<Integer» pair =
      new SimpleEntry<>(array1, array2);
```

```
    int x = 42;
  }
}
```

Ce code aura la représentation mémoire suivante :




En revanche, le code suivant sera rejeté par le compilateur Java :

```
import java.util.ArrayList;

class Forbidden {
  ArrayList<int> a; // unexpected type
}
```

Java ne permet pas d'utiliser des types scalaires comme `int` directement dans des collections génériques comme `ArrayList`, car ces collections ne fonctionnent qu'avec des objets, nécessitant ainsi l'emballage des types scalaires.

### Avantages

**Spécialisation de code** L'emballage sélectif permet d'éviter l'emballage des valeurs non polymorphes, ce qui se traduit par un code plus performant et mieux spécialisé lorsque les types monomorphes sont utilisés. Le code est plus efficace, notamment en termes de vitesse d'exécution, car il n'impose pas de coûts inutiles liés à l'emballage.

**Usage de la mémoire** Cette technique optimise l'utilisation de la mémoire en évitant l'emballage des types primitifs non polymorphes. Ainsi, les valeurs comme `int` peuvent être stockées directement dans des tableaux sans nécessiter l'overhead des objets. Cela réduit la consommation de mémoire par rapport à une approche d'emballage complet.

**Mise en œuvre** Bien que la mise en œuvre de cette technique soit légèrement plus complexe que celle de l'emballage complet, elle reste relativement simple. Les ajustements nécessaires pour différencier les valeurs polymorphes des valeurs non polymorphes sont minimes et permettent d'améliorer la performance globale du programme.

**Coût de la compilation** Le coût de la compilation est faible, similaire à celui de l'emballage complet, puisque chaque fonction polymorphe n'est compilée qu'une seule fois.

**Taille du binaire** La taille du binaire produit reste faible, car chaque fonction polymorphe ne produit qu'un seul morceau de code assembleur.

**Langages dynamiques** Comme pour l'emballage complet, cette technique est compatible avec les langages dynamiques, car elle permet de gérer des types polymorphes sans nécessiter de compilation anticipée de toutes les formes possibles.

**Modularité** Chaque fonction peut être compilée indépendamment des autres, ce qui permet de maintenir la modularité du code et de faciliter la compilation séparée.

**Récursion polymorphe** Cette technique est également compatible avec la récursion polymorphe. Les fonctions qui s'appellent elles-mêmes avec un nombre non borné de types peuvent être correctement gérées, car les valeurs polymorphes seront toujours emballées de manière uniforme.

#### Désavantages

**Vitesse d'exécution** Bien que cette technique améliore les performances par rapport à l'emballage complet, elle reste moins efficace que d'autres techniques comme la monomorphisation. L'emballage des valeurs impose toujours un coût supplémentaire par rapport à l'utilisation directe de valeurs non emballées.

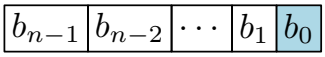
**Utilisation de la mémoire** Même si l'usage de la mémoire est optimisé par rapport à l'emballage complet, cette technique entraîne toujours une consommation de mémoire plus élevée que d'autres techniques.

**Mise en œuvre** La mise en œuvre de cette technique demande plus d'efforts que l'emballage complet, car elle nécessite de distinguer statiquement entre les valeurs qui doivent être emballées et celles qui peuvent rester non emballées. Cependant, la mise en œuvre n'en est que peu complexifiée et reste simple.

**Simplicité du langage** La séparation entre les types primitifs et leurs équivalents emballés peut rendre le langage plus difficile à maîtriser. Par exemple, il n'est pas intuitif pour un développeur débutant de comprendre pourquoi il ne peut pas utiliser des types primitifs comme éléments d'une `ArrayList`, ce qui complique l'apprentissage et l'utilisation du langage.

### 4.3.5 Marquage complet

La technique du marquage complet utilise une représentation uniforme des valeurs en utilisant un mot machine, où un bit spécifique est utilisé pour indiquer s'il s'agit d'un *petit scalaire* ou bien d'un pointeur vers un bloc alloué sur le tas. Dans un système avec des mots de $n$ bits, il faut d'abord choisir un bit qui servira de *bit* de marquage. Par exemple, si l'on choisit le bit de poids faible, nous commençons par tester sa valeur comme suit :




Si $b_0 = 0$, alors toute la valeur est un pointeur :

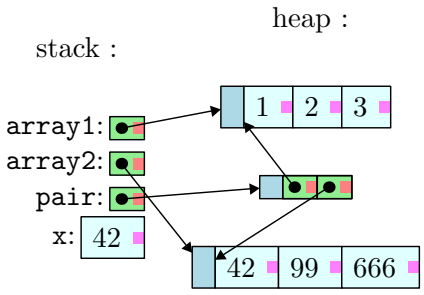

Si $b_0 = 1$, les $n - 1$ bits de poids fort représentent un petit scalaire, et $b_0$ est ignoré :


Dans le second cas, nous parlons de *petit scalaire* plutôt que de *scalaire* car nous ne pouvons représenter que $2^{n-1}$ valeurs, au lieu des $2^n$ représentables lorsque tous les bits sont disponibles. Pour les pointeurs, nous ne perdons rien, car ils doivent être *alignés*, ce qui implique que le bit de poids faible est toujours à zéro.

Cette technique est utilisée par SML/NJ [26], OCaml [33, 149] et JavaScript (dans le moteur V8 [117]). Regardons le code OCaml suivant :

```
let array1 = [| 1; 2; 3 |]
let array2 = [| 42; 99; 666 |]
let pair = (array1, array2)
let x = 42
```

Sa représentation en mémoire sera la suivante :




#### Avantages

**Facilité de mise en œuvre** Cette technique a l'avantage d'être simple à mettre en œuvre.

**Performance accrue** En évitant les emballages inutiles, cette technique permet d'améliorer significativement les performances, en particulier en ce qui concerne les accès mémoire et la gestion des petits scalaires. Les accès indirects sont réduits, ce qui permet une meilleure localité des données et, par conséquent, une exécution plus rapide.

**Usage efficace de la mémoire** Le marquage complet optimise l'utilisation de la mémoire, surtout pour les petites valeurs scalaires qui sont représentées directement dans le mot machine, sans avoir besoin d'allocation supplémentaire sur le tas. Cela permet de réduire la fragmentation de la mémoire et de diminuer la charge sur le glaneur de cellules.

**Coût de la compilation** Cette technique présente également un coût de compilation faible puisque chaque fonction n'est compilée que vers un unique code assembleur.

**Taille du binaire** La taille du binaire généré est réduite, car la technique permet une représentation uniforme sans nécessite de code supplémentaire pour gérer différents types de données. Cela se traduit par un binaire plus compact.

**Modularité** Le marquage complet conserve la modularité du code. Chaque fonction peut être compilée de manière indépendante, ce qui permet une compilation séparée efficace.

**Récursion polymorphe** De par la représentation uniforme utilisée dans cette technique, elle est compatible avec la récursion polymorphe. Les appels récursifs impliquant un nombre non borné de types utiliseront tous les même code.

#### Désavantages

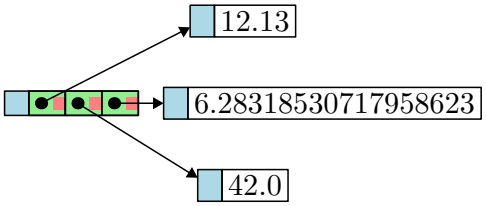
**Limitation des scalaires complets** Cette technique ne permet de manipuler que $2^{n-1}$ scalaires, ce qui est inférieur à l'intervalle complet de $2^n$ scalaires disponibles avec toutes les autres méthodes. Par exemple, en OCaml, les valeurs de type `int` sont des entiers de 63 bits sur des plateformes 64 bits. Bien que cela ne pose généralement pas de problème, car les grands entiers peuvent être manipulés via des modules comme `Int64`, il s'agit néanmoins d'une limitation inhérente à cette approche. Nous pouvons cependant argumenter que si le programmeur a besoin d'entiers qui ne tiennent par sur 63 bits, il est probable qu'ils ne tiennent pas non plus sur 64 bits et qu'une représentation emballée pour des grands entiers soit nécessaire dans tous les cas. Cela passe par l'utilisation d'une bibliothèque telle que Zarith [75] en OCaml. Même lorsque c'est le cas, il est possible d'utiliser des optimisations permettant dans certains cas de conserver une version déballée pour les petits scalaires [163]

**Nombres flottants emballés** Contrairement aux entiers, les flottants ne peuvent pas être facilement représentés en utilisant des mots de 63 bits, car il n'existe pas de standard pour les flottants de cette taille. Les processeurs n'ont donc pas d'instructions pour opérer sur des flottants 63 bits. De plus, du fait de leur représentation, il n'est pas non plus possible de simplement tronquer et décaler un flottant 64 bits comme avec les entiers. Par conséquent, les flottants doivent être représentés comme des valeurs 64 bits emballées, ce qui augmente la complexité et le surcoût lié à leur manipulation.

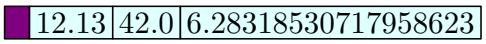
#### Optimisation

**Tableaux de flottants** Bien que les flottants soient toujours emballés, une optimisation existe pour les tableaux de flottants. Au lieu de stocker chaque flottant dans une boîte distincte, OCaml utilise une étiquette spéciale dans les méta-données du bloc du tableau pour indiquer

que les données sont des flottants déballés. Cela permet de stocker tous les flottants dans une seule boîte. Sans cette optimisation, nous nous attendrions donc à obtenir la représentation mémoire suivante[4] :




À la place, nous n'avons qu'une seule boîte pour le tableau et chaque flottant est déballé :




Cette optimisation permet de conserver les avantages en termes de performance, tout en réduisant les coûts associés à l'emballage des flottants.

### 4.3.6 Marquage sélectif

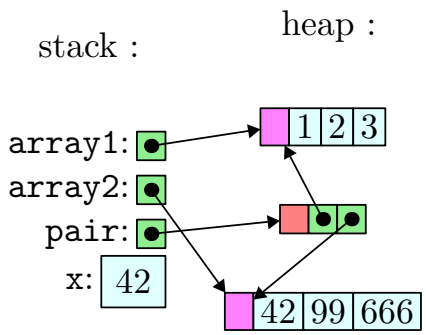
Cette technique, employée par WasmGC, combine les avantages des deux méthodes décrites précedemment : l'emballage sélectif et le marquage complet. Plutôt que de s'appuyer exclusivement sur l'une ou l'autre de ces techniques, cette approche hybride permet d'optimiser la représentation mémoire en tirant parti des points forts des deux méthodes.

Un exemple de code en Wasm est donné ci-dessous. Nous ne le détaillons pas ici, puisque WasmGC et la compilation vers celui-ci seront abordés en profondeur dans une section ultérieure. Il est néanmoins intéressant de regarder dès à présent la représentation mémoire que cette approche permet d'obtenir, offrant une solution qui combine les bénéfices des deux techniques tout en minimisant leurs inconvénients respectifs.

```
(module

  ;; the type for array of unboxed 32-bits integers
  (type $t1 (array (mut i32)))
  ;; the type for array of unboxed 31-bits integers
  ;; (they are expected to be implemented as 32-bits tagged integers)
  (type $t2 (array (mut i31ref)))
  ;; the type for a pair of t1 and t2, where each element is boxed
  (type $tpair (struct
    (field $left (ref $t1))
    (field $right (ref $t2))))

  (func $f
    (local $array1 (ref $t1))
    (local $array2 (ref $t2))
```

4. En réalité, la représentation serait un peu plus compliquée. Il faut une étiquette spéciale indiquant que la valeur est un flottant afin qu'elle ne soit pas interprétée comme une valeur potentiellement marquée.

```
    (local $pair (ref $tpair)) (local $x i32)

    ;; building an array of type t1
    (array.new $t1 (i32.const 0) (i32.const 3))
    (local.set $array1)
    (array.set $t1 (local.get $array1)
               (i32.const 0) (i32.const 1))
    (array.set $t1 (local.get $array1)
               (i32.const 1) (i32.const 2))
    (array.set $t1 (local.get $array1)
               (i32.const 2) (i32.const 3))

    ;; building an array of type t2
    (array.new $t2 (ref.i31 (i32.const 0)) (i32.const 3))
    (local.set $array2)
    (array.set $t2 (local.get $array2) (i32.const 0)
               (ref.i31 (i32.const 42)))
    (array.set $t2 (local.get $array2) (i32.const 1)
               (ref.i31 (i32.const 99)))
    (array.set $t2 (local.get $array2) (i32.const 2)
               (ref.i31 (i32.const 666)))

    ;; building a pair of t1 and t2 with the two previous arrays
    (struct.new $tpair
                (local.get $array1) (local.get $array2))
    (local.set $pair)

    ;; building an unboxed 32-bits integer
    (i32.const 42)
    (local.set $x))

  (start $f))
```




### 4.3.7 Union marquée

L'union marquée est une technique utilisée dans l'interpréteur Lua [53] et par PHP [76]. Cette méthode repose sur l'encapsulation des valeurs dans une structure C où un champ

spécifique (l'étiquette) est utilisé pour identifier le type de la valeur contenue. La structure est définie comme suit :

```
typedef struct {
  int t;   // tag to identify the type
  Value v; // actual value
} TObject;

typedef union {
  // Nil is not represented here because the tag is enough to encode such
  ↪  values
  GCObject *gc; // strings, tables, functions, heavy userdata, and thread
  ↪  which are subject to GC
  void *p; // light user data
  lua_Number n; // numbers
  int b; // booleans
} Value;
```

Cette représentation, bien que flexible, présente un inconvénient notable : elle est coûteuse en termes de mémoire, chaque instance de `TObject` occupant jusqu'à 16 octets. Cette approche est également inefficace pour les copies fréquentes de données, en raison de la taille des structures à manipuler.

Une optimisation potentielle consisterait à encoder le type directement dans les bits inutilisés des pointeurs, comme cela a été fait dans la mise en œuvre originale de SmallTalk80 [11]. Cependant, dans le contexte d'un interpréteur écrit en C, cette technique n'est ni portable ni compatible avec le standard C ANSI, rendant son utilisation impossible pour Lua.

Cette représentation est assez similaire à la représentation des `PyObject` en Python. Cependant, elle est plus efficace, comme expliqué par Ierusalimschy [53] :

> Another option to reduce the size of a value would be to keep the explicit tag, but to avoid putting a double in the union. For instance, all numbers could be represented as heap-allocated objects, just like strings. (Python uses this technique, except that it preallocates some small integer values.) However, that representation would make the language quite slow.

### 4.3.8 Passage de type

Décrite dans plusieurs travaux [21, 29], la technique du passage de type consiste à transmettre explicitement le type de chaque valeur inconnue à la compilation comme un paramètre supplémentaire des fonctions, soit au moment de l'édition de liens, soit lors de l'exécution. Cette approche permet aux fonctions d'effecteur une analyse de cas basée sur le type reçu et de sélectionner dynamiquement le code à exécuter.

Prenons, par exemple, un langage ML fictif où les tableaux sont représentés différemment selon le type de leurs éléments pour des raisons d'optimisation. Un tableau de booléens pourrait être représenté comme un tableau de bits pour économiser de la mémoire, tandis qu'un tableau d'entiers serait encodé sans emballage. Tous les autres types seraient emballés.

La fonction `array_get` pourrait être compilée de manière à s'adapter dynamiquement au type des éléments du tableau, comme suit :

```
let array_get_bool       t i = …
let array_get_int        t i = …
let array_get_boxed type t i = …

let array_get type =
  match type with
  | Bool -> fun t i -> array_get_bool t i
  | Int  -> fun t i -> array_get_int t i
  | Boxed type -> fun t i -> array_get_boxed type t i
```

Supposons maintenant l'expression suivante :

```
if (array_get t1 24)
  then 1313 + (array_get t2 42)
  else (array_get t3 13) 12
```

Ici, la valeur renvoyée par le premier appel à `array_get` doit être un booléen (puisqu'elle est utilisée comme condition), tandis que la valeur renvoyée par le second appel doit être un entier, car elle est utilisée dans une opération arithmétique. Enfin, le troisième appel renvoie une fonction, qui est donc emballée.

Après compilation, cette expression pourrait ressembler à ceci :

```
if (array_get Bool t1 24)
  then 1313 + (array_get Int t2 42)
  else (array_get (Boxed (Fun (([Int], [Int])))) t3 13) 12
```

Si les types sont connus à la compilation et qu'une passe d'inlining est effectuée, le code pourrait être simplifié comme suit :

```
if (array_get_bool t1 24)
  then 1313 + (array_get_int t2 42)
  else (array_get_boxed (Fun (([Int], [Int]))) t3 13) 12
```

Cela permet de réduire les coûts associés à ce mécanisme dans certains cas, où cela revient en fait à effectuer une monomorphisation.

Cette technique a été mise en œuvre dans le compilateur de recherche TIL [31], dans AliceML au niveau du système de modules [50, 56, 59] et plus récemment dans Swift [90].

### 4.3.9 Monomorphisation à l'exécution

La monomorphisation à l'exécution est une technique sophistiquée où chaque objet transporte son type exact lors de l'exécution. Lorsqu'une fonction polymorphe est appelée, elle inspecte le type de tous ses arguments. Si la fonction n'a jamais été appelée avec cette combinaison de types, un compilateur JIT (Just-in-Time) est invoqué pour monomorphiser la fonction pendant l'exécution. Cette version monomorphisée est ensuite mise en cache pour les

appels futurs. Sinon, la version déjà monomorphisée est récupérée depuis le cache et utilisée immédiatement.

Cette technique est mise en œuvre par .NET [40] et offre des performances proches de celles de la monomorphisation statique, mais avec une complexité de mise en œuvre plus élevée. Les performances passent par une représentation extrêmement efficace et légère des types à l'exécution (*runtime types*).

### Récursion polymorphe

L'un des avantages majeurs de la monomorphisation à l'exécution, comparée à sa version statique, est sa capacité à gérer la récursion polymorphe. Même si une fonction peut être appelée avec un nombre statiquement non borné de types, ce nombre est fini à l'exécution tant que le programme termine[5]. Ainsi, à chaque nouvel appel avec un type différent, le JIT monomorphise la fonction pour ce type spécifique.

Prenons l'exemple des *Revisited Random Access Lists*, cette fois en F# :

```
type ral<'a> =
  | Nil
  | Zero of ral<'a * 'a>
  | One of 'a * ral<'a * 'a>

let rec length<'a> (l : ral<'a>) : int =
  match l with
  | Nil -> 0
  | Zero s -> 2 * length s
  | One (_v, s) -> 1 + 2 * length s

let rec cons<'a> (x : 'a) (l : ral<'a>) : ral<'a> =
  match l with
  | Nil -> One (x, Nil)
  | Zero s -> One (x, s)
  | One (y, s) -> Zero (cons (x, y) s)

let rec loop l =
  let len = length l
  if len < 3000000 then
    let l = cons 42 l
    printfn "len = %d" (length l)
    loop l

loop Nil
loop Nil
```

Pour exécuter ce code, nous pouvons utiliser les commandes suivantes :

5. Il est possible d'écrire un programme qui ne termine pas et dont le nombre de types augmente indéfiniment au cours de l'exécution.

```
$ fsharpc ral.fsx
$ chmod +x ral.exe
$ ./ral.exe
```

À l'exécution, l'effet du JIT est notable. À mesure que la taille de la liste double, le type des valeurs dans les feuilles grandit, ce qui déclenche le JIT et interrompt brièvement l'exécution. Bien que ce phénomène soit imperceptible au début, il devient de plus en plus évident à mesure que la liste grandit. Nous pouvons confirmer que ces interruptions sont causées par le JIT, grâce à l'absence de telles pauses lors du second appel `loop Nil`, où le JIT n'a plus de travail à faire, ayant déjà monomorphisé les fonctions pour les types rencontrés.

### 4.3.10 Comparaison

Le tableau 4.1 expose un récapitulatif des différentes méthodes. Elles sont classées selon certaines propriétés telles que l'efficacité du code produit ou l'usage mémoire. Un nombre bas représente un avantage tandis qu'un plus grand nombre représente un désavantage. Ces nombres sont donnés pour aider le lecteur à comparer les différentes méthodes mais ceux-ci ne sont bien évidemment qu'approximatifs. Il est difficile de comparer ces différentes méthodes dans l'absolu. En effet, elles sont généralement appliquées à des langages différents et de nombreux autres facteurs entrent en jeu. De plus, il n'est a priori pas évident qu'une méthode soit meilleure qu'une autre pour une propriété donnée, ainsi, les nombres donnés ne forment pas un ordre total et sont parfois les mêmes pour deux langages et une propriété donnés.

## 4.4 Compilation vers WasmGC des langages avec glaneur de cellule

Maintenant que l'on a vu les différentes techniques de représentation du polymorphisme et comment détecter les pointeurs lors de l'exécution, se pose la question de comment compiler les langages utilisant ces différentes techniques vers Wasm. Nous avons plusieurs contraintes, nous voulons notamment être une cible de compilation simple pour les langages existants, sans qu'ils n'aient à réécrire tout leur compilateur, et ne pas impacter négativement les programmes sans glaneur de cellule.

Pour la monomorphisation, il n'est pas nécessaire de fournir une interface sophistiquée. Toutes les informations nécessaires sont disponibles à la compilation. La technique utilisant une représentation uniforme doit être supportée, puisqu'elles se retrouvent dans un nombre importants de langages différents. Les techniques d'union marquée et de passage de type peuvent être encodées facilement vers une représentation uniforme (on utilise une valeur représentée uniformément pour représenter le type, un type union pour la valeur elle-même et nous générons un code qui effectue le dispatch à l'exécution). La technique du JIT semble plus compliquée à gérer. Nous décrivons dans la section suivante le design retenu pour WasmGC et comment il répond à ces objectifs.

| Technique | Efficacité du code | Usage mémoire | Mise en œuvre | Temps de compilation | Taille du binaire | Langages dynamiques | Modularité | Récursion polymorphe | Langages |
|---|---|---|---|---|---|---|---|---|---|
| Monomorphisation | 1 | 1 | 6 | 7 | 7 | ❌ | ❌ | ❌ | Rust, C++, D, Ada, Go, SML (MLton) |
| Emballage complet | 6 | 6 | 1 | 1 | 1 | ✅ | ✅ | ✅ | Python |
| Emballage sélectif | 5 | 5 | 4 | 4 | 4 | ✅ | ✅ | ✅ | Java |
| Marquage complet | 4 | 4 | 3 | 3 | 3 | ✅ | ✅ | ✅ | OCaml, JavaScript (V8) |
| Marquage sélectif | 3 | 3 | 5 | 5 | 5 | ✅ | ✅ | ✅ | WasmGC (V8) |
| Union marquée | 6 | 6 | 2 | 2 | 2 | ✅ | ✅ | ✅ | Lua, PHP (Zend) |
| Passage de type | 6 | 6 | 7 | 6 | 6 | ✅ | ✅ | ✅ | TIL, AliceML, Swift |
| Monomorphisation à l'exécution | 2 | 2 | 8 | 6 | 6 | ✅ | ✅ | ✅ | C#, F# |

TABLE 4.1 – Comparaison des différentes techniques de mise en œuvre du polymorphisme.

## 4.5 Langages dynamiques

Les langages dynamiques sont plus compliqués à gérer que les langages avec glaneur de cellule, puisqu'ils présentent généralement des fonctions telles que `eval` qui permettent la compilation à l'exécution. Ces constructions sont pour l'instant laissées de côté.

Une solution pourrait cependant être de compiler les langages dynamiques de façon similaire aux langages statiques avec glaneur de cellules, d'écrire un interpréteur pour ces langages (par exemple en OCaml), de compiler cet interpréteur vers WasmGC et de faire appel à l'interpréteur compilé pour évaluer le code dynamiquement lors d'un appel à une fonction `eval`.

# 5 WasmGC

WasmGC étend le langage avec de nouveaux types de références et des instructions associées. Ces extensions permettent de compiler divers langages utilisant un glaneur de cellules vers Wasm. Ce chapitre présente ces nouveautés à travers des exemples, puis en propose une formalisation.

## 5.1 Introduction à WasmGC

WasmGC introduit une nouvelle hiérarchie de types de références, comme illustré à la figure 5.1. Certains types, tels que `ref i31`, `ref eq`, `ref any`, `ref func` et `ref extern`, sont prédéfinis. D'autres types, comme `struct` et `array`, peuvent être définis par l'utilisateur.

Ces types forment une hiérarchie de sous-typage. Par exemple, `ref i31` est un sous-type de `ref eq`. Les conversions descendantes dans cette hiérarchie sont explicites grâce à l'instruction `ref.cast`, tandis que les conversions ascendantes sont implicites.

La notation `ref i31` ou `ref eq` fait référence à des types de référence non nulle. Lorsqu'une référence peut être nulle, nous utilisons `ref null i31` ou `ref null eq`. Pour plus de concision, `i31ref` ou `eqref` désignent respectivement les versions potentiellement nulles de ces types. Ces notations abrégées sont souvent employées lorsque le fait que la référence puisse être nulle n'est pas le point central.

### 5.1.1 Les petits scalaires `ref i31`

Le type `ref i31` représente des entiers sur 31 bits, mais, contrairement aux entiers 32 ou 64 bits, ils font partie de la hiérarchie des types de références. En tant que tels, ils ne peuvent pas être stockés dans la mémoire linéaire. Cependant, en raison de la relation de sous-typage, les `ref i31` peuvent être utilisés comme des `eqref`, offrant une représentation uniforme pour certaines références, comme nous le verrons plus loin.

Les `ref i31` peuvent également être utiles pour représenter de petits scalaires dans un contexte polymorphe compatible avec un ramasse-miettes. Cela évite ainsi d'avoir à utiliser un `i32` qui devrait donc être emballé et alloué sur le tas. En effet, bien que les `ref i31` soient des références, ils ne sont pas alloués sur le tas. Il est attendu des moteurs d'exécution de Wasm qu'ils les représentent comme des pointeurs marqués, bien que cela ne soit pas spécifié explicitement.

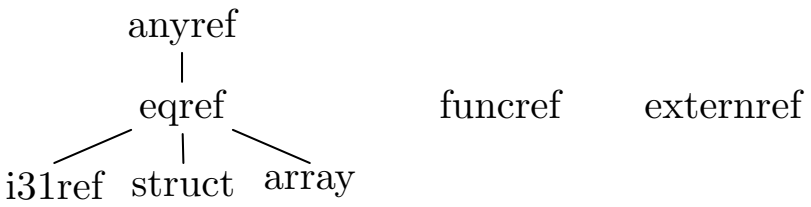



FIGURE 5.1 – Hiérarchie de sous-typage de WasmGC.

Pour créer un `ref i31` à partir d'un entier `i32`, nous utilisons l'instruction `ref.i31` (dans notre formalisation, nous l'appellerons `i31.of_i32` pour lever certaines ambiguïtés). Inversement, nous pouvons convertir un `ref i31` en `i32` avec `i31.get_s` ou `i31.get_u` selon l'interprétation souhaitée (dans notre formalisation, seule la version signée sera proposée sous le nom `i32.of_i31`).

### 5.1.2 Les tableaux

WasmGC permet également la définition de types de tableaux, qui sont homogènes et dynamiquement indexés. Ces tableaux sont gérés par le glaneur de cellules, ce qui dispense le programmeur de gérer explicitement la libération de la mémoire.

Un type de tableau est défini ainsi :

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

Ici, `$my_new_type` est le nom du type de tableau, qui est muable et dont les éléments sont de type `ref i31`. Nous pouvons créer un tableau avec une taille fixe en utilisant `array.new_fixed`, dont les opérandes sont le type du tableau et sa taille. Les valeurs sont extraites de la pile.

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

Pour accéder à un élément d'un tableau, nous utilisons `array.get`, et pour modifier un élément, `array.set`.

```
(module

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

  (func $add_one_to_index_two (param $a (ref $my_new_type))

    ;; put some values on the stack for later
    local.get $a           ;; [ a ]
    i32.const 1            ;; [ 1 ; a ]

    ;; read the first value of the array
```

```
    local.get $a           ;; [ a ; 1 ; a ]
    i32.const 1            ;; [ 1 ; a ; 1 ; a ]
    array.get $my_new_type ;; [ (a.(1) : ref i31) ; 1 ; a ]

    ;; convert it to an i32 and add 1 to it
    i31.get_s              ;; [ (a.(1) : i32) ; 1 ; a ]
    i32.const 1            ;; [ 1 ; a.(1) ; 1 ; a ]
    i32.add                ;; [ 1 + a.(1) ; 1 ; a ]

    ;; make it an i31 again
    ref.i31                ;; [ (1 + a.(1) : ref i31) ; 1 ; a ]

    ;; store it inside the array at offset 1
    array.set $my_new_type ;; ε

    ;; the array stored on the heap has been modified
    ;; it can be read from somewhere else and the value will appear as
    ↪  incremented
  )
)
```

Il est également possible de définir des tableaux contenant des scalaires, comme ici :

```
(type $floatarray (array (mut f64)))
```

Les tableaux de scalaires ne nécessitent pas d'indirection supplémentaire car les valeurs sont directement stockées dans le tableau.

### 5.1.3 Les structures

WasmGC introduit aussi les structures, qui sont similaires aux tableaux, mais leurs indices sont statiques et elles peuvent contenir des types hétérogènes. Voici un exemple de définition de structure :

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

Cette structure contient deux champs : `$fst`, qui est un `ref i31`, et `$snd`, un `i64`. Nous pouvons créer une instance de cette structure avec `struct.new` et accéder à ses champs avec `struct.set` et `struct.get`.

```
(module

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

```
  (func $mk_pair (param $a (ref i31)) (param $b i64) (result (ref $pair))
                    ;; ε
    local.get $a    ;; [ a ]
    local.get $b    ;; [ b ; a ]
    struct.new $pair ;; [ { fst = a ; snd = b } ]
  )

  (func $get_fst (param $p (ref $pair)) (result (ref i31))
                       ;; ε
    local.get $p       ;; [ p ]
    struct.get $pair $fst ;; [ p.fst ]
  )
)
```

### 5.1.4 Sous-typage

Les `ref i31`, `struct`, et `array` sont des sous-types de `ref eq`, ce qui fournit une représentation uniforme. Par exemple, un `ref eq` peut représenter soit un petit entier, soit un tableau. Nous pouvons convertir dynamiquement un `ref eq` en `ref i31` ou en tableau avec `ref.cast`. Cette instruction effectue un test dynamique qui peut échouer et mettre fin à l'exécution. Des variantes permettent de conditionner un branchement à ce test plutôt que d'échouer :

```
(module

  (type $my_array (array i32))

  (func $get_scalar_or_first_array_elem (param $r (ref eq)) (result i32)

    (block $array_case (result (ref i31))
                                                  ;; ε
      local.get $r                                ;; [ r : ref eq ]
      ;; if r is an i31 we branch out of array case ;;
      ;; and put this i31 on the stack
      br_on_cast $array_case (ref eq) (ref i31)   ;; ε
      ;; it is not an i31 so it must be a my_array
      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 ]
      return
    )
    ;; we exited array_case so a scalar of type ref i31 in on the stack
    ;; because of the result type of the block
                      ;; [ r : ref i31 ]
    i31.get_s         ;; [ r : i32 ]
```

```
  )
)
```

Le sous-typage s'applique également aux types définis par l'utilisateur. Un tableau immuable est un sous-type d'un autre si les types des éléments sont eux-mêmes des sous-types. Des règles de variances empêchent les comportements erronés présents par exemple dans Java. Pour déclarer un sous-typage, il faut indiquer explicitement cette relation :

```
(type $t1 (array (ref eq)))
(type $t2 (sub $t1 (array (ref i31))))
```

Ici, `$t2` est un sous-type de `$t1` parce que `ref i31` est un sous-type de `ref eq`. Cependant, les types définis par l'utilisateur ne sont pas par défaut considérés comme des sous-types lorsqu'ils peuvent l'être. Pour cela, il faut indiquer de quel type nous sommes un sous-type. Par exemple, ici, il faut indiquer que `$t2` est un sous-type de `$t1` :

```
(type $t1 (array (ref eq)))
(type $t2 (sub $t1 (array (ref i31))))
```

Cependant, cela ne suffit pas. Si l'on tente de lancer l'interpréteur de référence, nous obtenons l'erreur suivante : `invalid module: sub type $t2 has final super type $t1`. Par défaut, les types sont marqués avec l'attribut `final`, empêchant un type d'être déclaré comme sous-type d'un autre. Ainsi, notre déclaration précédente revient à :

```
(type $t1 (sub final (array (ref eq))))
(type $t2 (sub $t1 (array (ref i31))))
```

Pour permettre à `$t1` de servir de super-type à `$t2`, il faut retirer le mot-clé `final`, comme suit :

```
(type $t1 (sub (array (ref eq))))
(type $t2 (sub $t1 (array (ref i31))))
```

Une fois cela fait, nous pouvons écrire :

```
(func $f (param $t (ref $t1)) (result (ref $t2))
  local.get $t
  ref.cast (ref $t2)
)
```

La définition du sous-typage pour les structures suit une logique similaire. Un type peut être un sous-type si les types de tous ses champs sont eux-mêmes des sous-types des champs correspondants de la structure parente. De plus, une structure ayant plus de champs peut aussi être considérée comme un sous-type, tant que la condition précédente est respectée. Par exemple :

```
(module

  (type $s1 (sub (struct
```

```
    (field $x (ref eq))
    (field $y (ref eq)))))

  (type $s2 (sub $s1 (struct
    (field $x (ref i31))
    (field $y (ref i31))
    (field $z (ref i31)))))

  (func $f (param $s (ref $s2)) (result (ref $s1))
    local.get $s
    ref.cast (ref $s1))

  (func $main
    i32.const 42
    ref.i31
    i32.const 24
    ref.i31
    i32.const 12
    ref.i31
    struct.new $s2
    call $f
    drop
  )
  (start $main)
)
```

Il est important de noter que les noms donnés aux champs n'ont aucune influence sur la relation de sous-typage. Celle-ci est purement structurelle, c'est-à-dire qu'elle se base sur l'ordre et le type des champs, indépendamment de leurs noms.

### 5.1.5 Types récursifs

WasmGC permet également la définition de types récursifs. Par exemple, une liste chaînée peut être définie ainsi :

```
(type $list (struct
  (field $hd i32)
  (field $tl (ref null $list))
))
```

Il est également possible de définir des types mutuellement récursifs, comme dans cet exemple :

```
  (rec
    (type $list (struct
      (field $hd (ref $something))
      (field $tl (ref null $list))
```

```
    ))
    (type $something (struct
      (field $v (ref $list))
      )
    )
  )
```

## 5.2 Syntaxe abstraite

**Types tas abstraits** Nous étendons les types tas abstraits en y ajoutant les nouveaux types de référence proposés par WasmGC :

$$
\begin{array}{rll}
\mathit{aht} ::= & \ldots & \\
& \mid \texttt{func} & \text{(fonction)} \\
& \mid \texttt{any} & \text{(quelconque)} \\
& \mid \texttt{eq} & \text{(comparable)} \\
& \mid \texttt{i31} & \text{(i31)} \\
& \mid \texttt{struct} & \text{(structure)} \\
& \mid \texttt{array} & \text{(tableau)}
\end{array}
$$

**Types définis par l'utilisateur**

**Champs** Un type de champ spécifie si un champ est muable et indique le type qu'il contient (qui peut être un scalaire ou une référence) :

$$\mathit{fieldt} ::= \texttt{mut}?\ t$$

**Types composés** Les types composés englobent les différents types pouvant être définis par l'utilisateur, comme les fonctions, les structures ou les tableaux.

$$
\begin{array}{rll}
\mathit{compt} ::= & \texttt{func}\ \mathit{ft} & \text{(fonction)} \\
& \mid \texttt{struct}\ \mathit{fieldt}^* & \text{(structure)} \\
& \mid \texttt{array}\ \mathit{fieldt} & \text{(tableau)}
\end{array}
$$

**Sous-type** Un sous-type se définit par plusieurs identifiants (correspondant aux super-types) et un type composé.

$$\mathit{subt} ::= \texttt{sub}\ \mathit{id}^*\ \mathit{compt}$$

**Types récursifs** Un type récursif est un ensemble de sous-types nommés.

$$\mathit{rect} ::= \texttt{rec}\ (\mathit{id}\ \mathit{subt})^*$$

**Instructions** Nous étendons les instructions de Wasm de la manière suivante :

$$\begin{array}{rrr}
i ::= & \ldots & \\
& \mid \texttt{i31.of_i32} & \text{(constante i31)} \\
& \mid \texttt{i32.of_i31} & \text{(lecture i31)} \\
& \mid \texttt{ref.cast}\ \mathit{id} & \text{(cast de référence)} \\
& \mid \texttt{call_ref}\ \mathit{id} & \text{(appel via référence)} \\
& \mid \texttt{struct.new}\ \mathit{id} & \text{(nouvelle structure)} \\
& \mid \texttt{struct.get}\ \mathit{id}\ n & \text{(lecture de structure)} \\
& \mid \texttt{array.new_fixed}\ \mathit{id}\ n & \text{(nouveau tableau)} \\
& \mid \texttt{array.get}\ \mathit{id} & \text{(lecture de tableau)} \\
& \mid \texttt{array.set}\ \mathit{id} & \text{(écriture de tableau)}
\end{array}$$

WasmGC propose de nombreuses autres instructions telles que `struct.set` ou encore `array.new` (dont la taille n'est pas connue statiquement et qui initialise les éléments avec une valeur par défaut). Celles-ci n'étant pas utiles pour la suite, nous ne les présentons pas ici.

## 5.3 Validation

Nous décrivons ici le typage des nouvelles instructions. Il est nécessaire d'introduire une relation de sous-typage $\leq$, définie comme suit.

**Sous-typage** Un type `$t1` est un sous-type de `$t2`, noté $\texttt{\$t1} \leq \texttt{\$t2}$, si :

- ils sont des sous-types selon la hiérarchie de sous-typage, comme illustré dans la figure 5.1 ;
- s'il s'agit de deux tableaux, le type des éléments de `$t1` est un sous-type de celui des éléments de `$t2` ;
- s'il s'agit de deux structures, `$t2` a au moins autant de champs que `$t1`, et chaque champ de `$t1` est un super-type du champ correspondant (selon leur ordre de définition) de `$t2`.

**Typage des instructions** Nous allons à présent détailler les règles de typage pour les nouvelles instructions introduites par WasmGC.

$$\texttt{i31.of\_i32}\ \frac{}{\Gamma \vdash \texttt{i31.of\_i32} : \texttt{i32} \to \texttt{i31}}$$

Cette instruction dépile une valeur de type `i32` et dépose une valeur de type `i31` sur la pile.

$$\texttt{i32.of\_i31}\ \frac{}{\Gamma \vdash \texttt{i32.of\_i31} : \texttt{null i31} \to \texttt{i32}}$$

Cette instruction dépile une valeur de type `null i31` et dépose une valeur de type `i32` sur la pile.

$$\texttt{ref.cast}\ \frac{\Gamma \vdash \texttt{ok}\ id \qquad \Gamma \vdash \texttt{ok}\ id' \qquad id \leq id'}{\Gamma \vdash \texttt{ref.cast}\ id : id' \to id}$$

L'instruction `ref.cast` dépile une valeur de type $id'$ et dépose une valeur de type $id$ si et seulement si $id$ est un sous-type de $id'$.

$$\texttt{call.ref}\ \frac{\Gamma.\texttt{types}(id) = \texttt{func}\ t_1^* \to t_2^*}{\Gamma \vdash \texttt{call.ref}\ id : t_1^*(\texttt{null}\ id) \to t_2^*}$$

L'instruction est valide si la pile contient une référence potentiellement nulle du type de fonction spécifié, suivie d'arguments correspondant aux paramètres attendus par ce type de fonction.

$$\texttt{struct.new}\ \frac{\Gamma.\texttt{types}(id) = \texttt{struct}\ (\texttt{mut}?\ st)^*}{\Gamma \vdash \texttt{struct.new}\ id : st^* \to id}$$

L'instruction est valide si la pile contient des éléments dont les types correspondent à ceux des champs de la structure spécifiée. Elle place ensuite une référence vers une instance de cette structure sur la pile.

$$\texttt{struct.get}\ \frac{\Gamma.\texttt{types}(id) = \texttt{struct}\ fieldt^* \qquad fieldt^*[n] = \texttt{mut}?\ st}{\Gamma \vdash \texttt{struct.get}\ id\ n : \texttt{null}\ id \to st}$$

L'instruction est valide si la pile contient une référence potentiellement nulle vers une structure du type spécifié. Elle dépose une valeur du type du champ spécifié.

$$\texttt{array.new\_fixed}\ \frac{\Gamma.\texttt{types}(id) = \texttt{array}\ (\texttt{mut}?\ st)}{\Gamma \vdash \texttt{array.new\_fixed}\ id\ n : st_0 \ldots st_{n-1} \to id}$$

Cette instruction est valide si la pile contient exactement le nombre d'éléments correspondant à la taille du tableau, et si chacun de ces éléments correspond au type des éléments du tableau spécifié.

$$\texttt{array.get}\ \frac{\Gamma.\texttt{types}(id) = \texttt{array}\ (\texttt{mut}?\ st)}{\Gamma \vdash \texttt{array.get}\ id : (\ \texttt{i32}, \texttt{ref null}\ id) \to st}$$

L'instruction est valide si la pile contient une référence potentiellement nulle vers un tableau ainsi qu'un entier représentant l'indice de lecture. Elle dépose sur la pile une valeur du type des éléments du tableau.

$$\texttt{array.set}\ \frac{\Gamma.\texttt{types}(id) = \texttt{array}\ (\texttt{mut}\ st)}{\Gamma \vdash \texttt{array.set}\ id : (st, \texttt{i32}, \texttt{ref null}\ id) \to \varepsilon}$$

L'instruction est valide si la pile contient une référence potentiellement nulle vers un tableau, un entier pour l'indice d'écriture, et une valeur correspondant au type des éléments du tableau. De plus, le tableau doit être muable.

## 5.4 Syntaxe administrative

**Instances de références** Les instances de références (*i.e.* les valeurs effectivement allouées sur le tas) sont définies de la manière suivante :

$$\begin{aligned} \mathit{refinst} ::= {} & \texttt{array}\ id\ v^* && \text{(instance de tableau)} \\ | {} & \texttt{struct}\ id\ v^* && \text{(instance de structure)} \end{aligned}$$

**Tas** Un tas W est une application de **N** dans *refinst*.

$$W ::= n \to \mathit{refinst} \qquad \text{(tas)}$$

**États** L'état est étendu comme suit :

$$E ::= \{..., \mathsf{heap} : W\} \qquad \text{(état)}$$

**Références** Les références sont étendues avec les nouveaux types suivants :

$$\begin{aligned} \mathit{reference} ::= {} & ... \\ | {} & \texttt{i31}\ n && \text{(i31)} \\ | {} & \texttt{array}\ n && \text{(array)} \\ | {} & \texttt{struct}\ n && \text{(struct)} \\ | {} & \texttt{func}\ id && \text{(func)} \end{aligned}$$

## 5.5 Sémantique

Nous introduisons maintenant la sémantique des nouvelles instructions. Une fonction *getreftype* permet de récupérer le type d'une référence à l'exécution, et *gettypeof* permet de récupérer le type associé à un identifiant.

$$\texttt{i31.of\_i32}\ \frac{n' = n\ \texttt{land}\ \texttt{0x7fff\_ffff}}{(\texttt{i32}\ n, v^*), (\texttt{i31.of\_i32}, i^*) \to (\texttt{i31}\ n', v^*), i^*}$$

L'instruction convertit un `i32` en un `i31`.

$$\texttt{i32.of\_i31}\ \frac{}{(\texttt{i31}\ n, v^*), (\texttt{i32.of\_i31}, i^*) \to (\texttt{i32}\ n, v^*), i^*}$$

Cette instruction convertit un `i31` en `i32`.

$$\texttt{ref.cast}\ 1\ \frac{\mathit{getreftype}(v_0) = id_2 \qquad id_2 \leq id_1}{(v_0, v^*), (\texttt{ref.cast}\ id_1, i^*) \to (v_0, v^*), i^*}$$

Si la valeur au sommet de la pile est un sous-type de $id_1$, rien ne se passe.

$$\texttt{ref.cast}\ 2\ \frac{\mathit{getreftype}(v_0) = id_2 \qquad id_2 \not\leq id_1}{(v_0, v^*), (\texttt{ref.cast}\ id_1, i^*) \to \texttt{trap}}$$

Dans le cas contraire, si la valeur n'est pas un sous-type, l'exécution échoue avec un `trap`.

$$\texttt{call\_ref}\ 1\ \frac{}{(\texttt{null}, v^*), (\texttt{call\_ref}\ id, i^*) \to \texttt{trap}}$$

La règle spécifie que l'appel de `call_ref` sur une référence nulle interrompt immédiatement l'exécution du programme.

$$\texttt{call\_ref}\ 2\ \frac{}{(\texttt{func}\ id_f, v^*), (\texttt{call\_ref}\ id, i^*) \to v^*, (\texttt{call}\ id_f, i^*)}$$

À l'inverse, si la référence n'est pas nulle, elle pointe vers une fonction (comme le garantit le système de typage). Dans ce cas, l'exécution se poursuit par un appel à la fonction référencée.

$$\texttt{struct.new}\ \frac{\begin{array}{c} n_a \notin \mathit{dom}(E.W) \qquad E' = E[W \leftarrow W'] \\ W' = E.W[n_a \leftarrow \texttt{struct}\ id\ v_0\ ...\ v_{n-1}] \\ \mathit{gettypeof}(id) = \texttt{sub}\ id^*_{sub}\ \texttt{struct}\ \mathit{fieldt}_0\ ...\ \mathit{fieldt}_{n-1} \end{array}}{E, (v_0, ..., v_{n-1}, v^*), (\texttt{struct.new}\ id, i^*) \to E', (\texttt{struct}\ n_a, v^*), i^*}$$

Cette instruction crée une nouvelle structure à partir des valeurs présentes sur la pile. Une adresse fraîche est générée pour cette structure, et l'état global est mis à jour de manière à ce que l'adresse nouvellement créée pointe vers cette structure dans le tas.

$$\texttt{struct.get}\ 1\ \frac{}{(\texttt{null}\ id_1, v^*), (\texttt{struct.get}\ id_2\ n, i^*) \to \texttt{trap}}$$

Cette règle précise qu'une tentative d'accès à un champ d'une structure via une référence nulle entraîne l'arrêt du programme.

$$\texttt{struct.get}\ 2\ \frac{W(n_a) = \texttt{struct}\ id_2\ v_0,\ ...,\ v_{m-1}}{(\texttt{struct}\ n_a, v^*), (\texttt{struct.get}\ id_1\ n, i^*) \to (v_n, v^*), i^*}$$

En revanche, si la référence est non nulle et pointe bien vers une structure, la valeur du champ demandé est extraite du tas à l'adresse donnée. Le typage assure l'existence du champ en question.

$$\texttt{array.new\_fixed}\ \frac{\begin{array}{c} n_a \notin dom(E.W) \qquad E' = E[W \leftarrow W'] \\ W' = E.W[n_a \leftarrow \texttt{array}\ id\ v_0\ ...\ v_{n-1}] \end{array}}{E, (v_0, ..., v_{n-1}, v^*), (\texttt{array.new\_fixed}\ id\ n, i^*) \rightarrow E', (\texttt{array}\ n_a, v^*), i^*}$$

L'instruction `array.new_fixed` permet de créer un tableau dont la taille est statiquement définie. Les valeurs initiales du tableau sont extraites de la pile, et une nouvelle adresse est allouée dans le tas pour ce tableau.

$$\texttt{array.get}\ 1\ \frac{}{(\texttt{i32}\ n,\ \texttt{null}\ id_1, v^*), (\texttt{array.get}\ id_2, i^*) \rightarrow \texttt{trap}}$$

Une tentative de lecture dans un tableau référencé par une valeur nulle provoque l'arrêt immédiat du programme.

$$\texttt{array.get}\ 2\ \frac{W(n_a) = \texttt{array}\ id\ v_0\ ...\ v_{m-1} \qquad n \geq m}{(\texttt{i32}\ n,\ \texttt{array}\ n_a, v^*), (\texttt{array.get}\ id_1, i^*) \rightarrow \texttt{trap}}$$

La lecture d'un élément en dehors des bornes d'un tableau provoque également l'arrêt du programme. Il n'est pas nécessaire de tester le cas où $n$ est négatif puisqu'il est ici interprété comme non-signé.

$$\texttt{array.get}\ 3\ \frac{W(n_a) = \texttt{array}\ id\ v_0\ ...\ v_{m-1} \qquad n < m}{(\texttt{i32}\ n,\ \texttt{array}\ n_a, v^*), (\texttt{array.get}\ id_1, i^*) \rightarrow (v_n, v^*), i^*}$$

Lorsque l'indice de lecture est valide, la valeur correspondant à l'indice du tableau est récupérée depuis le tas et placée sur la pile. Il n'est pas nécessaire de tester le cas où $n$ est négatif puisqu'il est ici interprété comme non-signé.

$$\texttt{array.set}\ 1\ \frac{}{(v_{new}, \texttt{i32}\ n,\ \texttt{null}\ id_1, v^*), (\texttt{array.set}\ id_2, i^*) \rightarrow \texttt{trap}}$$

De même, une tentative d'écriture dans un tableau référencé par une valeur nulle entraîne l'arrêt du programme.

$$\texttt{array.set}\ 2\ \frac{W(n_a) = \texttt{array}\ id\ v_0\ ...\ v_{m-1} \qquad n \geq m}{(v_{new}, \texttt{i32}\ n,\ \texttt{array}\ n_a, v^*), (\texttt{array.set}\ id_1, i^*) \rightarrow \texttt{trap}}$$

Une écriture en dehors des bornes d'un tableau provoque également l'arrêt du programme. Il n'est pas nécessaire de tester le cas où $n$ est négatif puisqu'il est ici interprété comme non-signé.

$$\texttt{array.set}\ 3\ \frac{W(n_a) = \texttt{array}\ \mathit{id}\ v_0\ ...\ v_{m-1} \qquad n < m \qquad E' = E[W \leftarrow W'] \qquad W' = E.W[n_a \leftarrow \texttt{array}\ \mathit{id}\ v_0\ ...\ v_{n-1}\ v_{new}\ v_{n+1}\ ...\ v_{m-1}]}{E, (v_{new}, \texttt{i32}\ n,\ \texttt{array}\ n_a, v^*), (\texttt{array.set}\ \mathit{id}_1, i^*) \rightarrow E', v^*, i^*}$$

Si l'indice est valide, l'écriture d'une nouvelle valeur dans le tableau met à jour l'état global du tableau dans le tas. Il n'est pas nécessaire de tester le cas où $n$ est négatif puisqu'il est ici interprété comme non-signé.

Grâce à l'extension WasmGC, dont nous venons d'exposer la sémantique, nous serons en mesure de développer un compilateur pour OCaml vers WasmGC dans le chapitre 7. La sémantique de WasmGC servira de base pour établir la preuve de correction de ce compilateur dans le chapitre 8.

# Deuxième partie

# Compilation d'OCaml vers Wasm

# 6 OCaml et Flambda

OCaml est un langage de programmation fonctionnel, qui prend également en charge la programmation impérative et orientée objet. Il est caractérisé par un système de typage statique et fort, associé à un mécanisme d'inférence de types, ce qui permet d'éliminer la nécessité de déclarer explicitement les types dans de nombreux cas. De plus, OCaml inclut un glaneur de cellules qui gère automatiquement la mémoire, libérant ainsi les programmeurs de cette tâche complexe.

## 6.1 Modèle d'exécution et choix du langage intermédiaire

Le modèle d'exécution d'OCaml est similaire à celui de Java. Une valeur peut être soit un scalaire (`unit`, `bool`, `int`, etc.), soit un pointeur vers un bloc de mémoire alloué sur le tas. Le passage des arguments se fait par valeur, ce qui signifie qu'aucune valeur n'est implicitement copiée en profondeur, garantissant une exécution efficace et des performances prédictibles.

Pour compiler OCaml vers WasmGC, il est essentiel de choisir la bonne représentation intermédiaire du langage à partir de laquelle la compilation sera effectuée. Le compilateur OCaml utilise plusieurs langages intermédiaires au cours du processus de compilation : *ParseTree*, *TypedTree*, *Lambda*, *Bytecode*, *Flambda*, *Flambda2*, *Clambda* et *Cmm*. La chaîne de compilation est représentée dans la figure 6.1.

**ParseTree** Le *ParseTree* est la représentation de l'AST obtenue directement après l'analyse syntaxique du programme. À ce stade, l'inférence et la vérification de types n'ont pas encore été effectuées. En raison de la complexité du système de types d'OCaml, notamment avec les GADTs, il est préférable de s'appuyer sur le compilateur pour effectuer cette étape, ce qui exclut l'utilisation du *ParseTree* comme point de départ pour la compilation vers WasmGC.

**TypedTree** Le *TypedTree* est l'AST après l'inférence et la vérification des types. L'arbre syntaxique est alors annoté avec des informations de type. Cependant, de nombreuses constructions du langage sont encore présentes à ce stade, comme le filtrage de motifs, qui n'a pas encore été compilé, ou encore les modules et les classes qui n'ont pas été éliminées. Ces opérations étant à la fois complexes et déterminante pour les performances, il est préférable de s'appuyer sur les optimisations existantes dans le compilateur OCaml, plutôt que de partir du *TypedTree*.

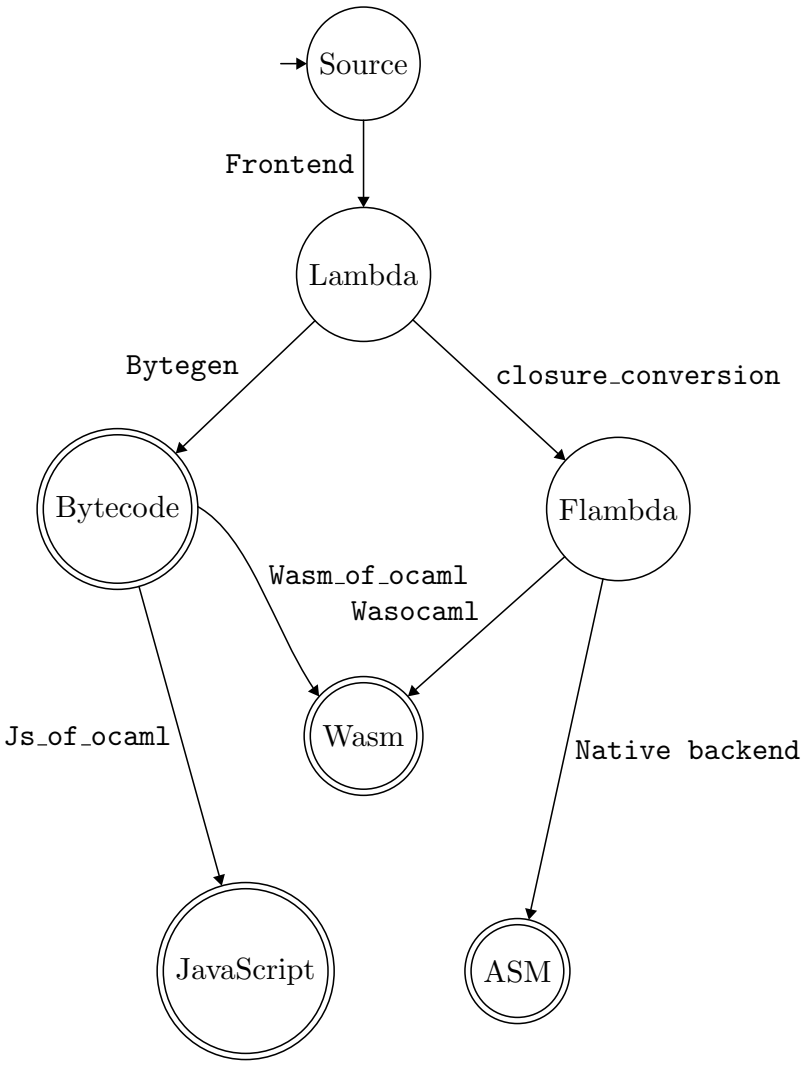



FIGURE 6.1 – Relation entre les différentes représentations intermédiaires du compilateur OCaml.

**Lambda** *Lambda* est une version simplifiée du *TypedTree* où de nombreuses constructions ont déjà été transformées ou éliminées. Les modules et les objets ont été remplacés par des enregistrements et des pointeurs de fonctions. Le filtrage a été compilé vers des automates optimisés. Les valeurs ont été départagées entre les scalaires et les blocs, comme supposé dans le modèle d'exécution. Toutefois, certaines constructions restent encore présentes, telles que les fermetures implicites, et la plupart des optimisations n'ont pas encore été appliquées (*inlining*, *uncurrying*). Pour ces raisons, il est préférable d'utiliser un stade ultérieur du processus de compilation.

**Bytecode** Le *bytecode* est dérivé de *Lambda*. Il s'agit d'une représentation de plus bas niveau où les fermetures sont explicites, rendant ce format directement exécutable par `ocamlrun`, l'interpréteur de bytecode d'OCaml. Cependant, ce stage ne bénéficie pas de la plupart des optimisations, ce qui le rend peu adapté pour une compilation efficace vers WasmGC.

**Flambda** *Flambda* est une autre représentation intermédiaire issue de *Lambda*. Elle explicite les fermetures tout en effectuant un important nombre d'optimisations, telles que l'*inlining*, ce qui en fait un candidat adapté pour la compilation vers WasmGC.

**Flambda2** *Flambda2* est également dérivée de *Lambda*. Bien qu'elle soit similaire à *Flambda*, elle est basée sur un langage CPS à double canon [43], ce qui la rend plus complexe à manipuler. Malgré cette complexité, elle reste adaptée pour une compilation vers Wasm, bien que sa manipulation soit plus délicate.

**Clambda** *Clambda* peut être issue de *Lambda* ou de *Flambda*. Cependant, il s'agit d'une représentation de trop bas niveau pour une compilation directe vers WasmGC, car elle mélange les valeurs et les pointeurs de code au sein des fermetures.

**Cmm** *Cmm*, dérivée de *Flambda2* ou de *Clambda*, est encore plus bas niveau, incluant de l'arithmétique sur les pointeurs. Cette caractéristique la rend encore moins adaptée pour une compilation vers WasmGC.

En conclusion, pour notre prototype initial, nous avons choisi de nous baser sur *Flambda*. Une version future pourrait envisager l'utilisation de *Flambda2*, bien que cela implique des difficultés supplémentaires en raison de la complexité de ce langage intermédiaire.

## 6.2 Syntaxe abstraite de Flambda

**Valeurs** Une valeur $v$ peut être un entier, une adresse, une fermeture ou un ensemble de fermetures. Les entiers représentent des valeurs pour plusieurs types OCaml tels que `int`, `bool`, `char`, `unit`, ainsi que les constructeurs constants des type somme. La sémantique d'OCaml n'impose pas de taille fixée pour les entiers, ici nous faisons le choix d'utiliser des entiers 31-bits. Une adresse pointe vers un bloc de mémoire alloué sur le tas. Une fermeture se compose d'un ensemble récursif parent et d'un identifiant. Enfin, un ensemble récursif regroupe plusieurs fermetures, représentant des fonctions mutuellement récursives.

$$
\begin{aligned}
v ::= \ & n && \text{(entier)} \\
| \ & a && \text{(adresse)} \\
| \ & S\,.\,id && \text{(fermeture)} \\
| \ & S && \text{(ensemble récursif)}
\end{aligned}
$$

**Mémoires** Une mémoire $M$ est une application qui à une adresse $a$ associe un bloc de mémoire. Un bloc mémoire est composé d'une étiquette $n$ et d'un ensemble de valeurs. Elle représente l'état global du programme en cours d'exécution.

$$M ::= a \to n, v^* \qquad \text{(mémoire)}$$

**Ensembles récursif** Un ensemble récursif $S$ représente un ensemble de fonctions mutuellement récursives. Il se compose d'un ensemble de fonctions et d'un environnement de valeurs, permettant de capturer le contexte d'exécution de ces fonctions.

$$S ::= V, F \qquad \text{(ensemble de fermetures)}$$

**Environnements de valeurs** Un environnement de valeurs $V$ est une application qui à un identifiant *id* associe une valeur $v$.

$$V ::= id \to v \qquad \text{(environnement de valeur)}$$

**Fonctions** Une fonction $f$ a deux arguments et un corps. Le premier argument est le paramètre de la fonction, et le second est l'adresse de la fermeture associée. Le corps de la fonction est un terme qui exprime le calcul effectué par la fonction.

$$f ::= \lambda\ id\ id\ t \qquad \text{(fonction)}$$

**Ensemble de fonctions** Un ensemble de fonctions $F$ est une application qui à un identifiant *id* associe une fonction $f$.

$$F ::= id \to f \qquad \text{(environnement de fonctions)}$$

**Opérateurs binaires** Un opérateur binaire *op* effectue une opération entre deux entiers. Les opérateurs disponibles sont l'égalité, l'addition et la soustraction.

$$
\begin{aligned}
op ::= \ & = && \text{(égalité)} \\
| \ & + && \text{(addition)} \\
| \ & - && \text{(soustraction)}
\end{aligned}
$$

**Branches** Une branche $B$ est une application qui à un entier $n$ associe à un terme $t$. Cela permet de représenter des cas de filtrage de motifs.

$$B ::= n \to t \qquad \text{(branche)}$$

**Termes** Un terme $t$ représente une unité de calcul dans le programme. Leur représentation est principalement en forme A-normale [22, 24]. Voici les différentes formes qu'un terme peut prendre :

$$\begin{aligned}
t ::= \ & v && \text{(valeur)}\\
\mid\ & id && \text{(identifiant)}\\
\mid\ & \texttt{let}\ id\ \texttt{=}\ t\ \texttt{in}\ t && \text{(liaison)}\\
\mid\ & id\ id && \text{(application)}\\
\mid\ & id\ \texttt{<-}\ id && \text{(mue)}\\
\mid\ & \texttt{if}\ id\ \texttt{then}\ t\ \texttt{else}\ t && \text{(conditionnelle)}\\
\mid\ & \texttt{switch}\ id\ B\ B && \text{(aiguillage)}\\
\mid\ & \texttt{sraise}\ id\ id^* && \text{(lancer statique)}\\
\mid\ & \texttt{scatch}\ t\ \texttt{with}\ id\ id^*\ t && \text{(interception statique)}\\
\mid\ & \texttt{while}_t\ t\ \texttt{do}\ t && \text{(boucle)}\\
\mid\ & \texttt{get_field}\ n\ id && \text{(lecture de champ)}\\
\mid\ & \texttt{set_field}\ n\ id\ id && \text{(écriture de champ)}\\
\mid\ & \texttt{make_block}\ n\ id^* && \text{(création de bloc)}\\
\mid\ & \texttt{project_closure}\ id\ id && \text{(projection de fermeture)}\\
\mid\ & \texttt{project_var}\ id\ id && \text{(projection de variable)}\\
\mid\ & \texttt{move_within_set}\ id\ id && \text{(déplacement de fermeture)}\\
\mid\ & \texttt{pop}\ t && \text{(abandon)}\\
\mid\ & id\ op\ id && \text{(application binaire)}\\
\mid\ & \texttt{make_set}\ F^*\ (\texttt{|}\ id\ \texttt{->}\ id)^* && \text{(ensemble récursif)}
\end{aligned}$$

Un `switch` comporte deux branches. La première contient les cas où la valeur filtrée est un scalaire et la seconde ceux où il s'agit d'un bloc alloué sur le tas. Le `while` possède un terme en indice, celui-ci permet de sauvegarder le code de la boucle pour le réutiliser lors d'une prochaine itération, une fois le terme à l'intérieur de la boucle réduit en une valeur. L'instruction `make_block` est composée d'un entier correspondant à l'étiquette du bloc et des valeurs de celui-ci. L'instruction `pop` est une instruction administrative et sert à délimiter les appels de fonction afin de mettre à jour l'environnement de façon appropriée.

## 6.3 Exemple de programme

Nous présentons un programme OCaml avant d'en montrer une version Flambda équivalente.

Ce programme définit un type de vecteurs de dimension 2 :

```
type vec2 = {
  mutable x : int;
  mutable y : int;
}
```

Ensuite, il définit deux déplacements de ce type et une valeur `max` :

```
let v_even = { x = 21; y = 32 } in
let v_odd = { x = 43; y = 54 } in
let max = 1000 in
```

Puis deux fonctions mutuellement récursives, `even` et `odd` qui vont modifier une position `pos` jusqu'à ce qu'elle atteigne la valeur `max` sur sa coordonnée `pos.x`. Le déplacement appliqué est `v_even` pour la fonction `even` et `v_odd` pour la fonction `odd` :

```
let rec even pos =
  if pos.x < max then begin
    pos.x <- pos.x + v_even.x;
    pos.y <- pos.y + v_even.y;
    odd pos
  end
and odd pos =
  if pos.x < max then begin
    pos.x <- pos.x + v_odd.x;
    pos.y <- pos.y + v_odd.y;
    even pos
  end
in
```

Enfin, une position initiale est créée et passée à la fonction `even` :

```
let initial_pos = { x = 1; y = 1 } in
even initial_pos
```

Le programme Flambda équivalent va représenter les valeurs de type `vec2` comme des blocs dont l'étiquette est 0 et contenant deux valeurs :

```
let x = 21 in let y = 32 in
let v_even = make_block 0 x y in

let x = 43 in let y = 54 in
let v_odd = make_block 0 x y in

let max_x = 1000 in
```

Puis, les deux fonctions récursives sont représentées comme un ensemble de fermetures. Cet ensemble contient d'abord les deux fonctions, puis les trois variables capturées (`v_even`, `v_odd`

et `max_x`), qu'il enregistre sous le même nom. Les fonctions ont toujours pour argument la position `pos`, mais ont en plus une référence `self` vers elle-même, ce qui leur permet d'accéder à l'ensemble auquel elles appartiennent. En particulier, elles peuvent appeler l'autre fonction au moyen de `move_within_set` et accéder à une variable capturée au moyen de `project_var`. Pour lire et écrire dans les blocs, elles utilisent les instructions `get_field` et `set_field` :

```
let set =
  make_set
  (* functions *)
  | even -> \ pos self.
    let max_x = project_var self max_x in
    let old_x = get_field 0 pos in
    let is_lower = old_x < max_x in
    if is_lower then
      let v_even = project_var self v_even in
      let even_x = get_field 0 v_even in
      let new_x = old_x + even_x in
      let _ = set_field 0 pos new_x in
      let old_y = get_field 1 pos in
      let even_y = get_field 1 v_even in
      let new_y = old_y + even_y in
      let _ = set_field 1 pos new_y in
      let odd = move_within_set self odd in
      odd pos
    else 0
  | odd -> \ pos self.
    let max_x = project_var self max_x in
    let old_x = get_field 0 pos in
    let is_lower = old_x < max_x in
    if is_lower then
      let v_odd = project_var self v_odd in
      let odd_x = get_field 0 v_odd in
      let new_x = old_x + odd_x in
      let _ = set_field 0 pos new_x in
      let old_y = get_field 1 pos in
      let odd_y = get_field 1 v_odd in
      let new_y = old_y + odd_y in
      let _ = set_field 1 pos new_y in
      let even = move_within_set self odd in
      even pos
    else 0
  (* values *)
  | v_even -> v_even
  | v_odd -> v_odd
  | max_x -> max_x
in
```

Enfin, la valeur initiale est représentée par un bloc d'étiquette 0 et la fonction `even` est récupérée depuis l'ensemble de fermeture `set` au moyen de l'instruction `project_closure`.

```
let x = 1 in let y = 1 in
let initial_pos = make_block 0 x y in

let even = project_closure set even in
even initial_pos
```

## 6.4 Sémantique de Flambda

Nous définissons une sémantique opérationnelle par réduction, dans le style de Wright et Felleisen [28]. Nos réductions sont de la forme $V^*, M, t$ où $V^*$ est une pile d'environnement correspondant aux différents appels de fonction, $M$ est la mémoire et $t$ le terme en train d'être évalué.

**Contexte de réduction** Nous définissons nos contextes de réductions comme suit. Ils permettent d'expliciter l'ordre d'évaluation des termes.

$$
\begin{aligned}
E ::= {} & \square \\
& \mid \texttt{let}\ id = E\ \texttt{in}\ t \\
& \mid \texttt{while}_t\ E\ \texttt{do}\ t \\
& \mid \texttt{scatch}\ E\ \texttt{with}\ id\ id^*\ t \\
& \mid \texttt{pop}\ E
\end{aligned}
$$

**Réduction au sein d'un contexte** La réduction au sein d'un contexte s'effectue en appliquant une réduction de tête à la fois.

$$\frac{V^*, M, t_1 \to V'^*, M', t_2}{V^*, M, E(t_1) \to V'^*, M', E(t_2)}$$

**Réductions de tête** Nous présentons maintenant les différentes réductions de tête.

$$\text{identifiant}\ \frac{V_0(id) = v}{V_0 V^*, id \to v}$$

Un identifiant se réduit simplement à la valeur correspondante dans l'environnement courant.

$$\text{égalité 1}\ \frac{V_0(id_1) = v_1 = n_1 \qquad V_0(id_2) = v_2 = n_2 \qquad n_1 = n_2}{V_0 V^*, id_1 = id_2 \to 1}$$

L'égalité de deux identifiants s'évalue en 1 qui est `true : bool` en OCaml lorsque les deux identifiants ont pour valeur le même entier dans l'environnement courant.

$$\text{égalité 2}\ \frac{V_0(id_1) = v_1 = n_1 \qquad V_0(id_2) = v_2 = n_2 \qquad n_1 \neq n_2}{V_0V^*, id_1 \texttt{ = } id_2 \rightarrow \texttt{0}}$$

L'égalité de deux identifiants s'évalue en 0 qui est `false : bool` en OCaml lorsque les deux identifiants ont pour valeur deux entiers différents dans l'environnement courant.

$$\text{addition}\ \frac{V_0(id_1) = v_1 = n_1 \qquad V_0(id_2) = v_2 = n_2 \qquad n = n_1 + n_2}{V_0V^*, id_1 \texttt{ + } id_2 \rightarrow n}$$

L'addition de deux identifiants s'évalue en l'addition des valeurs de ces deux identifiants dans l'environnement lorsque celles-ci sont des entiers.

$$\text{soustraction}\ \frac{V_0(id_1) = v_1 = n_1 \qquad V_0(id_2) = v_2 = n_2 \qquad n = n_1 - n_2}{V_0V^*, id_1 \texttt{ - } id_2 \rightarrow n}$$

La soustraction de deux identifiants s'évalue en la soustraction des valeurs de ces deux identifiants dans l'environnement lorsque celles-ci sont des entiers.

$$\texttt{let}\ \frac{}{V_0V^*, \texttt{let } id \texttt{ = } v \texttt{ in } t \rightarrow V_0[id \leftarrow v]V^*, t}$$

Un lieur `let` $id$ `=` $v$ `in` $t$ se réduit en $t$ avec l'environnement courant dans lequel $id$ est lié à $v$.

$$\text{application}\ \frac{V_0(id_1) = S\,.\,id \qquad V_0(id_2) = v \qquad S = V, F \qquad F(id) = \lambda\ id_3\ id_4\ t}{V_0V^*, id_1 id_2 \rightarrow [id_3 \leftarrow v; id_4 \leftarrow S\,.\,id]V_0V^*, \texttt{pop}\ t}$$

Une application $id_1 id_2$ consiste à trouver la fermeture $S\,.\,id$ correspondant à $id_1$ dans l'environnement courant. Cette fermeture a pour parent $S$ qui contient l'ensemble de fonctions $F$. La valeur de $id$ dans $F$ est $\lambda\ id_3\ id_4\ t$. Le terme se réduit alors en `pop` $t$ et un nouvel environnement où $id_3$ est lié à la valeur courant de $id_2$ et $id_4$ est lié à $S\,.\,id$.

$$\texttt{pop}\ \frac{}{V_0V^*, \texttt{pop } v \rightarrow V^*, v}$$

Cette règle modélise la pile d'appels en se débarrassant de l'environnement courant. Un `pop` est inséré autour de chaque application de fonction et retiré une fois la fonction évaluée.

$$\text{mue}\ \frac{V_0(id_2) = v_2}{V_0V^*, id_1\texttt{<-}id_2 \rightarrow V_0[id_1 \leftarrow v_2]V^*, \texttt{0}}$$

Une mue $id_1$ `<-` $id_2$ remplace la valeur de $id_1$ par celle de $id_2$ dans l'environnement courant, et se réduit en `0`, ce qui correspond à `() : unit` en OCaml.

$$\texttt{if}\ 1 \frac{V_0(id) = n \qquad n \neq \texttt{0}}{V_0 V^*, \texttt{if}\ id\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \to t_1}$$

Une conditionnelle `if` $id$ `then` $t_1$`else` $t_2$ se réduit en $t_1$ lorsque la valeur de $id$ n'est pas égal à `0` (*i.e.* quand elle vaut `true : bool` en OCaml).

$$\texttt{if}\ 2 \frac{V_0(id) = n \qquad n = \texttt{0}}{V_0 V^*, \texttt{if}\ id\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \to t_2}$$

Dans le cas contraire, la conditionnelle se réduit en $t_2$ lorsque $id$ est égale à `0` (c'est-à-dire lorsque $id$ vaut `false : bool` en OCaml).

$$\texttt{switch}\ 1 \frac{V_0(id) = n \qquad B_1(n) = t}{V_0 V^*, \texttt{switch}\ id\ B_1\ B_2 \to t}$$

Lorsque la valeur de $id$ dans l'environnement courant est un entier $n$, alors, `switch` $B_1$ $B_2$ se réduit en l'image de $n$ par $B_1$.

$$\texttt{switch}\ 2 \frac{V_0(id) = a \qquad M(a) = n, v^* \qquad B_2(n) = t}{V_0 V^*, M, \texttt{switch}\ id\ B_1\ B_2 \to t}$$

De manière similaire, lorsque $id$ est une adresse $a$ et que la valeur à cette adresse dans la mémoire $M$ est un bloc dont l'étiquette vaut $n$, alors, `switch` $B_1$ $B_2$ se réduit en l'image de $n$ par $B_2$. Le filtrage a été compilé efficacement avant Flambda et ici ne restent plus que les cas correspondant à l'automate optimisé.

$$\texttt{sraise}\ 1 \frac{id \neq id'}{\texttt{scatch}(\texttt{sraise}\ id\ id^*)\ \texttt{with}\ id'\ id'^*\ t \to \texttt{sraise}\ id\ id^*}$$

Si une exception levée ne correspond pas à celle attendue par un `scatch`, alors le `scatch` est ignoré et l'exception est relancée.

$$\texttt{sraise}\ 2 \frac{id = id' \qquad id = id_0, ..., id_{n-1} \qquad id' = id'_0, ..., id'_{n-1} \qquad \forall i \in [0; n[, V_0(id_i) = v_i}{\texttt{scatch}(\texttt{sraise}\ id\ id^*)\ \texttt{with}\ id'\ id'^*\ t \to V_0[id'_0 \leftarrow v_0, ..., id'_{n-1} \leftarrow v_{n-1}]}$$

Dans le cas où l'exception levée correspond à celle attendue, les arguments sont capturés et insérés dans l'environnement avant de continuer l'exécution. Les valeurs sont prises dans l'environnement courant : si les identifiants étaient dans l'environnement au moment où l'exception a été levée, alors ils le sont aussi lorsque l'exception est rattrapée. Cela est dû au fait que les exceptions statiques ne traversent jamais les applications de fonctions. Finalement, le terme se réduit à celui attaché au `scatch`.

$$\texttt{sraise}\ 3 \frac{}{\texttt{while}_{t'}\ (\texttt{sraise}\ id\ id^*)\ \texttt{do}\ t \to \texttt{sraise}\ id\ id^*}$$

Un `while` lève simplement l'exception à nouveau sans la traiter.

$$\texttt{sraise}\ 4\ \frac{}{\texttt{let}\ id_1 = (\texttt{sraise}\ id_2\ id^*)\ \texttt{in}\ t \to \texttt{sraise}\ id_2\ id^*}$$

De même, un lieur lèvera à nouveau l'exception, sans évaluer le corps de la construction.

$$\texttt{while}\ 1\ \frac{n = \texttt{0}}{\texttt{while}_{t'}\ n\ \texttt{do}\ t \to \texttt{0}}$$

Une boucle $\texttt{while}_{t'}\ n\ \texttt{do}\ t$ se réduit en `0` lorsque la condition $n$ vaut `0`.

$$\texttt{while}\ 2\ \frac{n \neq \texttt{0}}{\texttt{while}_{t'}\ n\ \texttt{do}\ t \to \texttt{let}\ \_ = t\ \texttt{in}\ \texttt{while}_{t'}\ t'\ \texttt{do}\ t}$$

Au contraire, si $n \neq \texttt{0}$, la boucle se réduit à une exécution de $t$ suivie d'une réévaluation de la boucle avec la condition d'origine $t'$.

$$\texttt{get\_field}\ \frac{V_0(id) = a \qquad M(a) = n', v^* \qquad v^* = v_0, ..., v_{m-1} \qquad m > n \geq 0}{V_0V^*, M, \texttt{get\_field}\ n\ id \to v_n}$$

Le terme $\texttt{get\_field}\ n\ id$ se réduit en la $n^{\text{ième}}$ valeur du bloc à l'adresse $a$, où $a$ est la valeur de $id$ dans l'environnement courant. Cette opération permet de récupérer un élément particulier d'un bloc en mémoire.

$$\texttt{set\_field}\ \frac{\begin{array}{c} V_0(id_1) = a \qquad M(a) = n', v^* \qquad v^* = v_0, ..., v_{m-1} \\ m > n \geq 0 \qquad V_0(id_2) = v \\ a' = n', v_0, ..., v_{n-1}, v, v_{n+1}, ..., v_{m-1} \end{array}}{V_0V^*, M, \texttt{set\_field}\ n\ id_1\ id_2 \to M[a \leftarrow a'], \texttt{0}}$$

Le terme $\texttt{set\_field}\ n\ id_1\ id_2$ met à jour la $n^{\text{ième}}$ valeur du bloc à l'adresse $a$ en mémoire, où $a$ est la valeur de $id$ dans l'environnement courant. Cette nouvelle valeur est celle de $id_2$ dans l'environnement courant. Le bloc mis à jour est ensuite remplacé dans la mémoire, et l'évaluation renvoie $0$.

$$\texttt{make\_block}\ \frac{V_0(id_0) = v_0, ..., V_0(id_{n-1}) = v_{n-1} \qquad a \notin dom(M)}{V_0V^*, M, \texttt{make\_block}\ n'\ id_0, ..., id_{n-1} \to M[a \leftarrow n', v_0, ..., v_{n-1}], a}$$

Cette règle crée un nouveau bloc en mémoire à une adresse fraîche. L'étiquette est du bloc est donné comme une valeur $n$ tandis que les autres valeurs sont lues depuis l'environnement.

$$\texttt{project\_closure}\ \frac{V_0(id_1) = S}{V_0V^*, \texttt{project\_closure}\ id_1\ id_2 \to S.id_2}$$

Le terme $\texttt{project\_closure}\ id_1\ id_2$ se réduit en la fermeture $S\,.\,id_2$ si $id_1$ est liée à l'ensemble récursif $S$ dans l'environnement courant.

$$\texttt{project\_var}\ \frac{V_0(id_1) = S.id \qquad S = V, \_ \qquad V(id_2) = v}{V_0V^*, \texttt{project\_var}\ id_1\ id_2 \to v}$$

La règle associée au terme `project_var` $id_1$ $id_2$ consiste à rechercher la fermeture $S.id$ liée à $id_1$ dans l'environnement courant. Cette fermeture a pour parent $S$, lequel a pour environnement de variables $V$. Le terme se réduit alors en la valeur de $id_2$ dans $V$.

$$\texttt{move\_within\_set}\frac{V_0(id_1) = S.id}{V_0V^*, \texttt{move\_within\_set}\ id_1\ id_2 \rightarrow S.id_2}$$

Le terme `move_within_set` $id_1$ $id_2$ impose que $id_1$ soit une fermeture $S.id$ dans l'environnement courant et se réduit alors en la fermeture $S.id_2$. En d'autres termes, il permet de naviguer d'une fermeture à une autre au sein du même ensemble récursif.

$$\texttt{make\_set}\frac{V_0(id_{1r}) = v_1 \quad \ldots \quad V_0(id_{nr}) = v_n \quad S = F, [id_{1l} \leftarrow v_1; \ldots; id_{nl} \leftarrow v_n]}{V_0V^*, \texttt{make\_set}\ F\ (id_{1l}\ \texttt{->}\ id_{1r}\ \ldots\ id_{nl}\ \texttt{->}\ id_{nr}) \rightarrow S}$$

**Exemple de réduction** Nous donnons un exemple de réduction. Nous partons du programme OCaml suivant :

```
(fun x -> x) 2
```

Son équivalent Flambda est le suivant :

```
let set =
  make_set
  | f -> \ x self. x
in
let f = project_closure set f in
let x = 2 in
f x
```

Le premier contexte de réduction qui s'applique est celui consistant à réduire à l'intérieur du `let set`.

$$\begin{array}{l} [], \\ \texttt{make\_set}[\texttt{f -> } \lambda\ \texttt{x self. x}] \\ \rightarrow [], \\ ([], [\texttt{f} \rightarrow \lambda\ \texttt{x self. x}]) \end{array}$$

Le terme s'évalue en une réduction en une valeur qui est un ensemble de fermeture. Nous appliquons alors un autre contexte de réduction :

$$\begin{array}{l} [], \\ \texttt{let set = } ([], [\texttt{f} \rightarrow \lambda\ \texttt{x self. x}])\ \texttt{in} \ldots \\ \rightarrow [\texttt{set} \rightarrow ([], [\texttt{f} \rightarrow \lambda\ \texttt{x self. x}])], \\ \texttt{let f = } \ldots \end{array}$$

Nous appliquons alors un autre contexte de réduction qui consiste à réduire à l'intérieur du `let f`.

```
                                [set → ([], [f → λ x self. x])],
                                project_closure set f
                              →[set → ([], [f → λ x self. x])],
                                ([], [f → λ x self. x]).f
```

Le terme s'évalue en une réduction en une valeur qui est une fermeture. Nous appliquons alors un autre contexte de réduction :

```
                  [set → ([], [f → λ x self. x])],
                  let f = ([], [f → λ x self. x]).f in let x = ...
                →[set → ([], [f → λ x self. x]), f → ([], [f → λ x self. x]).f],
                  let x = ...
```

De même, nous évaluons à l'intérieur du `let x` pour finalement nous retrouver dans la configuration suivante, qui va effectuer un appel, empiler un nouvel environnement pour la fonction, insérer un `pop` autour du corps de la fonction avant de réduire au sein du `pop` jusqu'à obtention d'une valeur et que l'environnement de la fonction soit retiré :

```
  [set → ([], [f → λ x self. x]), f → ([], [f → λ x self. x]).f, x → 2, ],
  f x
→[x → 2, self → ([], [f → λ x self. x]).f]
  [set → ([], [f → λ x self. x]), f → ([], [f → λ x self. x]).f, x → 2, ],
  pop x
→[x → 2, self → ([], [f → λ x self. x]).f]
  [set → ([], [f → λ x self. x]), f → ([], [f → λ x self. x]).f, x → 2, ],
  pop 2
→[set → ([], [f → λ x self. x]), f → ([], [f → λ x self. x]).f, x → 2, ],
  2
```

# 7 Wasocaml

Dans ce chapitre, nous présentons le développement de Wasocaml, un compilateur OCaml vers WasmGC, un outil que nous avons conçu et qui est librement disponible [128]. Ce compilateur a été créé dans le but d'explorer et de démontrer les capacités de WasmGC, en particulier l'utilité des `ref i31`.

Nous détaillerons la conception de ce compilateur, en expliquant les choix effectués pour la gestion des différentes constructions du langage, ainsi que les techniques de représentation des valeurs que nous avons envisagées. Bien que cette description reste informelle, une formalisation pour un sous-ensemble du compilateur sera abordée dans le chapitre suivant (§8).

À l'origine, ce compilateur a été conçu comme un prototype pour convaincre le groupe de travail WasmGC de l'importance des `ref i31`. Il s'est avéré être le premier à démontrer leur utilité [143], suivi de près par le compilateur Guile [159]. Ensemble, ces deux projets ont réussi à convaincre le comité de maintenir les `ref i31` [151], évitant ainsi des dégradations potentielles des performances pour de nombreux langages si cette fonctionnalité avait été retirée.

Ce travail a fait l'objet de plusieurs présentations, notamment :

— au séminaire Dagstuhl *Foundations of WebAssembly* [147] ;
— au *35th Symposium on Implementation and Application of Functional Languages* [156] ;
— ainsi qu'au *Higher-order, Typed, Inferred, Strict : ML Family Workshop 2023* [144].

Un article a également été rédigé à ce sujet, bien qu'il n'ait pas encore été publié [145]. En outre, ce projet a conduit à la proposition d'une extension pour Wasm (§9), visant à surmonter certaines limitations rencontrées lors du développement de notre compilateur.

## 7.1 Représentation des valeurs

### 7.1.1 Valeurs de base

À l'instar du compilateur natif OCaml, nous utilisons une *représentation uniforme* des valeurs. Cependant, contrairement à OCaml, nous ne pouvons pas utiliser le type `i32`, car il ne permet pas de pointer vers les valeurs gérées par le glaneur de cellules. Il est donc nécessaire d'utiliser un type de référence. Le type le plus général disponible est `anyref`, mais dans notre contexte, nous pouvons être plus précis en utilisant `eqref`. Ce type permet de tester l'égalité

physique pour toutes les valeurs. C'est un *super-type* de toutes les valeurs OCaml. Cela évite la nécessité de convertir `anyref` vers `eqref` lors des tests d'égalité.

**Petits scalaires** Toutes les valeurs OCaml représentées par des entiers dans le compilateur natif sont codées par des `ref i31`. Cela inclut les types `unit`, `bool`, `char`, `int`, ainsi que les constructeurs constants des types sommes.

**Tableaux** Les tableaux OCaml sont directement représentés par des valeurs du type Wasm `array`.

**Blocs** Pour les autres types de blocs, deux options s'offrent à nous :
- utiliser une `struct` avec un champ pour l'étiquette et un champ supplémentaire pour chaque champ de la valeur OCaml ;
- utiliser un `array` de type `eqref`, avec l'étiquette positionnée à l'indice 0.

Les deux solutions sont détaillées ci-dessous.

### Blocs sous forme de structures

Dans cette approche, un bloc avec un champ est représenté par une valeur de type `$block1`, tandis qu'un bloc avec deux champs est représenté par une valeur de type `$block2`, et ainsi de suite :

```
(type $block1 (struct
  (field $tag i8)
  (field $field0 eqref)))

(type $block2 (sub $block1) (struct
  (field $tag i8)
  (field $field0 eqref)
  (field $field1 eqref)))
```

Le type `$block2` est déclaré comme un sous-type de `$block1`. Cette structuration est nécessaire, car dans les représentations intermédiaires du compilateur OCaml, l'accès aux champs d'un bloc est non-typé : lors de l'accès au $n^{\text{ième}}$ champ d'un bloc, la taille de celui-ci est inconnue, mais il est garanti qu'elle est au moins de $n+1$. Bien qu'il soit théoriquement possible de propager les métadonnées de taille de bloc à travers les différentes étapes de compilation, cette solution a été écartée car elle compromet certaines optimisations du compilateur[1]. De plus, elle suffit pas dans tous les cas. Par exemple, elle ne permet pas l'utilisation de `Obj.field`, qu'il est nécessaire de supporter pour maintenir une compatibilité avec le code existant.

Ainsi, lors de l'accès au champ $n$ d'un bloc, ce dernier est initialement de type `eqref` et est ensuite converti en un bloc de type `$bloc`$(n+1)$. Même si le bloc contient plus de champs, cette information n'est pas requise :

1. Lorsque les deux sous-expressions d'une conditionnelle sont identiques, le test peut être retiré. Cependant, en propageant plus d'informations issues du typage, leur type peut désormais différer, empêchant l'optimisation de se produire — bien qu'elles se comportent de façon identique à l'exécution.

```
(func $snd (param $x eqref) (result eqref)
  (struct.get $block2 1
    (ref.cast (ref $block2) (local.get $x)))
)
```

Cependant, cette approche a rencontré une limitation liée à un aspect non spécifié de Wasm, qui a été mise en œuvre de manière restrictive dans les navigateurs. Pour illustrer ce problème, prenons l'exemple de la bibliothèque OCaml *ocaml-emoji* [162], qui expose un grand nombre d'emojis sous forme de valeurs au niveau top-level.

Cette bibliothèque contient un module qui définit précisément $3,782$ emojis, chacun représenté par une valeur top-level. Un extrait en est montré ci-dessous :

```
(** ♐ (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"
```

Comme pour tous les modules OCaml, ce module est compilé en un seul bloc, qui contient alors les $3\,782$ champs correspondants, formant un type `$block3782`. Ce type a pour sous-type `$block3781`, qui lui-même a pour sous-type `$block3780`, et ainsi de suite.

Lors de l'exécution de ce module dans V8, le moteur Wasm a renvoyé l'erreur suivante : *error : too long subtyping chain*. Cela s'explique par l'existence d'une limite arbitraire imposée à la longueur des chaînes de sous-typage, initialement fixée à 100. Cette limite a par la suite été standardisée à 63.

Étant donné que de nombreux modules OCaml contiennent bien plus de 63 valeurs au niveau top-level, cette contrainte nous a poussés à explorer des alternatives pour représenter ces modules et plus généralement les blocs avec plus de 63 champs.

### Blocs sous forme de tableaux

Dans cette variante, un bloc est représenté par un tableau de valeurs de type `eqref` :

```
(type $block (array eqref))
```

L'étiquette se trouve à l'indice $0$ du tableau (une variante consiste à la placer à la fin). La lecture de l'étiquette est effectuée en accédant à la cellule concernée et en convertissant sa valeur en entier :

```
(func $tag (param $x eqref) (result i32)
  (i31.get_u (ref.cast i31ref
    (array.get $block
      (i32.const 0)
      (ref.cast (ref $block) (local.get $x))))))
```

L'accès au champ $n$ d'un bloc OCaml revient à accéder au champ $n + 1$ du tableau. Ainsi, la fonction `snd : 'a * 'b -> 'b`, qui accède au second champ de la paire (qui est donc le champ d'indice 1) doit donc accéder au champ 2 avec notre représentation :

```
(func $snd (param $x eqref) (result eqref)
  (array.get $block
    (i32.const 2)
    (ref.cast (ref $block) (local.get $x))))
```

#### Comparaison des deux représentations

La représentation sous forme de tableau est plus simple à mettre en œuvre, mais elle impose une vérification implicite des limites à chaque accès et une conversion dynamique lors de la lecture de l'étiquette.

En revanche, la représentation sous forme de structure nécessite une conversion dynamique plus complexe (un test de sous-typage) pour l'environnement d'exécution Wasm. Un compilateur capable de propager des informations de type plus fines pourrait cependant réduire le nombre de conversion dynamiques en fournissant un type plus précis qu'`eqref` pour chaque champ de la structure.

Du fait du problème rencontré lors de la présence de blocs très larges, nous avons opté pour la représentation sous forme de tableau, même si nous avons conservé l'autre version.

### 7.1.2 Nombres emballés

Les types scalaires Wasm tels que `i64` ne sont pas des sous-types d'`eqref`, et ne peuvent donc pas représenter directement les nombres emballés d'OCaml comme `int64` ou `float`. Ces types doivent être encapsulés dans une structure pour être compatibles avec notre représentation uniforme :

```
(type $boxed_float (struct (field $v f64)))
(type $boxed_int64 (struct (field $v i64)))
```

### 7.1.3 Fermetures

Les valeurs Wasm de type `funcref` ne sont que des fonctions, sans environnement capturé. Pour créer des fermetures, nous devons encapsuler la fonction ainsi que son environnement dans une structure. Par exemple, voici la représentation d'une fermeture avec deux variables capturées :

```
(type $closure1 (struct
  (field funcref)
```

```
  (field $v1 eqref)
  (field $v2 eqref)))
```

## 7.2 Flot de contrôle

Les constructions OCaml influençant le flot de contrôle ont un équivalent presque direct en Wasm via les constructions et instructions `block`, `loop`, `br_table` et `if`.

Pour les exceptions, les primitives de bas niveau sont similaires à celles proposées dans l'extension Wasm pour la gestion des exceptions. Toutefois, OCaml permet de générer de nouveaux types d'exceptions à l'exécution, notamment avec `let exception … in`, ou via des foncteurs et modules de première classe. Par conséquent, nous utilisons une exception unique côté Wasm, en discriminant nous-mêmes les différents types d'exceptions via un identifiant.

## 7.3 Fonctions, fermetures et curryfication

Les types de fonction en OCaml sont curryfiés : les fonctions ont toujours exactement un argument. Le système de type donne donc l'illusion que toutes les applications se font sur un seul argument à la fois. Si c'était le cas, cela impliquerait que la plupart des applications produisent des fermetures immédiatement appliquées à un autre paramètre. En réalité, en *bytecode*, les fonctions, bien que toujours appelées sur un seul argument à la fois, ne sont pas totalement curryfiées : il y a une construction particulière qui permet de construire efficacement les applications partielles. Pour ce qui est du compilateur natif, les fonctions peuvent être appelées partiellement mais également en passant plusieurs arguments à la fois, et ce pour des raisons d'optimisations. Sémantiquement, il n'y a pas de différence entre appliquer les arguments uns à uns plutôt que par groupe. Le compilateur natif essaie donc de reconstuire des applications complètes à partir des applications partielles autant que possible. Cela induit un mécanisme complexe permettant donc d'appliquer plusieurs fonctions à la fois tout en effectuant un traitement spécial pour les applications partielles. Les informations dues à ce mécanisme sont explicites dans Flambda. Elles disparaissent pendant l'étape *cmmgen*. Ce mécanisme est reproduit dans Wasocaml, nécessitant un sous-typage structurel sur les fermetures, de sorte que les fermetures de fonctions d'arité 1 soient des super-types de toutes les autres. Il est possible d'encoder cette relation de sous-typage en Wasm.

## 7.4 Représentation de la pile

Après la compilation des constructions précédentes, l'essentiel du travail restant consiste à transformer les expressions liées par un `let` en un langage basé sur une pile, sans générer de code excessivement naïf. Les optimisations spécifiques à Wasm ne sont pas une priorité ici, car nous nous appuyons sur Binaryen [86] pour cela.

## 7.5 Déballage

L'optimisation principale manquante par rapport au compilateur OCaml natif est le déballage des nombres. En OCaml, les valeurs ayant une représentation uniforme, seuls les petits

scalaires peuvent tenir dans un mot. Par conséquent, les types `nativeint`, `int32`, `int64` et `float` doivent être emballés.

Cela peut poser problème, notamment dans le code numérique, où de nombreuses valeurs flottantes intermédiaires sont créées. Dans ce cas, le temps d'allocation pour l'emballage des nombres peut dominer le temps de calcul effectif.

Pour atténuer ce problème, une optimisation appelée *déballage* est réalisée durant l'étape *cmmgen*. Cette passe, située après Flambda, n'était pas nécessaire pour produire un compilateur fonctionnel, et a donc été laissée comme une future optimisation. De plus, il est prévu d'utiliser Flambda2, qui gère correctement le déballage. En attendant, nous pouvons espérer que Binaryen puisse gérer certains cas de déballage.

## 7.6 Performances

**Avertissement** La plupart des informations liées au statut des différentes extensions dans les navigateurs qui sont données dans cette section datent de plusieurs mois. Certaines ne sont plus vraies aujourd'hui, de nombreuses extensions ayant depuis été mises en œuvre par les navigateurs.

### 7.6.1 Choix du moteur Wasm

Les différentes extensions Wasm dont nous avons besoin sont mises en œuvre par dessus l'interpréteur Wasm de référence, dans des dépôts distincts. Les fusionner demande un effort considérable et n'est de plus pas forcément utile pour mesurer les performances de notre compilateur, en effet l'interpréteur de référence ne se focalise que la correction et ses performances ne reflètent celles des mises en œuvre avancées que l'on peut trouver dans les navigateurs. Trois moteurs Wasm et JavaScript sont disponible dans les navigateurs :

- Webkit, le moteur de Safari, qui ne supporte pas WasmGC ;
- SpiderMonkey, le moteur de Firefox, qui ne supporte pas l'extension permettant les appels terminaux ;
- V8, le moteur de Chromium, qui supporte toutes les extensions dont nous avons besoin dans sa branche expérimentale.

Ainsi, toutes nos expérimentations ont été effectuées en utilisant V8.

### 7.6.2 Mesures effectuées

Tout d'abord, nous avons réussi à exécuter les microbenchmarks fonctionnels classiques et les résultats obtenus sont plutôt encourageants. Bien que nous ne nous attendions pas à être aussi performants que le compilateur natif, la perte de performance reste constance : notre code est environ deux fois plus lent que le code natif.

Nous avons également pu compiler une mise en œuvre OCaml de l'algorithme de Knuth-Bendix [12]. Cependant, nous avons observé que les exceptions dans V8 sont extrêmement lentes[2] par rapport au code OCaml natif : lever une exception est environ cent fois plus lent. Ce qui nous a empêché de continuer les expérimentations en utilisant des exceptions. Nous avons signalé cette limitation aux développeurs de V8 et ils n'ont pas indiqué avoir l'intention de modifier

2. Dans SpiderMonkey, en revanche, elles sont efficaces.

leur mise en œuvre[3]. Ils considèrent, comme c'est souvent le cas dans l'éco-système C++, que les exceptions sont faites pour les événements exceptionnels et non pas un moyen d'exprimer certains flots de contrôles aisément, comme c'est souvent le cas dans l'éco-système OCaml.

Pour pallier à cette lenteur, nous avons mis en place une stratégie de compilation alternative qui n'utilise pas les exceptions Wasm. À la place, chaque fonction renvoie une paire composée de la valeur de retour et d'un booléen indiquant si une exception a été levée ou non. Avec cette approche, nous observons également une dégradation de performance d'environ un facteur deux par rapport au compilateur natif pour l'algorithme de Knuth-Bendix.

Nous continuons de travailler sur des benchmarks plus représentatifs en utilisant des programmes de taille réelle. Cela reste complexe, car Wasocaml est encore un prototype et ne s'intègre pas encore aux systèmes de *build* existants.

### 7.6.3 Coût des conversions à l'exécution

Le moteur d'exécution V8 permet de considérer les conversions comme des *no-ops* (uniquement à des fins de test). En profitant de cette fonctionnalité, nous avons mesuré une amélioration des performances d'environ 10%, ce qui nous donne une estimation du coût réel des conversions à l'exécution.

### 7.6.4 Prédictibilité

Comparé à une machine virtuelle JavaScript, un compilateur Wasm est un outil bien plus simple, ce qui facilite la compilation *ahead-of-time*. De ce fait, les différents moteurs Wasm devraient offrir des performances relativement similaires entre eux. De plus, nous nous attendons à ce que le rapport de performances entre un compilateur Wasm et le compilateur OCaml soit constant pour n'importe quel programme. Cela n'est pas le cas des moteurs JavaScript. Par exemple, lorsque l'on compile un programme OCaml vers JavaScript avec js_of_ocaml [83], le temps d'exécution par le navigateur est généralement deux fois plus important que le temps d'exécution d'un binaire produit par le compilateur OCaml natif. Cependant, il arrive parfois que ce rapport varie de façon très importante (jusqu'à devenir soixante fois plus lent), et de manière difficile à prédire en regardant le code OCaml. Nos expériences préliminaires montrent que cela n'est pas le cas pour Wasm lors de l'exécution de code produit par notre compilateur.

## 7.7 Travaux en cours

### 7.7.1 Interface de fonction externe

Une grande partie du code OCaml existant interagit avec du code C ou JavaScript via des *bindings* créés à l'aide de mécanismes de FFI (*Foreign Function Interface*). Nous envisageons de permettre la réutilisation de ces *bindings* lors de la compilation vers Wasm.

Actuellement, lorsqu'on cible un environnement tel qu'un navigateur, il n'est pas possible de réutiliser directement les *bindings* C. Le programmeur doit alors choisir entre réécrire le

3. Depuis, Jérôme Vouillon, a amélioré les exceptions dans V8, elles sont désormais seulement dix fois plus lentes qu'en OCaml.

code en OCaml pur ou utiliser des *bindings* JavaScript. Compiler directement les *bindings* C vers Wasm offrirait la possibilité de les réutiliser tels quels.

***Bindings* C** Certaines extensions récentes à Clang [65] permettent de compiler du code C vers Wasm tout en autorisant la réutilisation des *bindings* C pour OCaml, avec très peu de modifications nécessaires. Pour ce faire, il suffirait de fournir une version modifiée des fichiers d'en-têtes OCaml natifs, où les macros habituelles seraient remplacées par des fonctions Wasm écrites manuellement. La principale limitation que nous anticipons concerne la macro `Field`, qui ne pourrait plus être utilisée comme une *l-value*. Une nouvelle macro, `Set_field`, sera donc nécessaire.

***Bindings* JavaScript** Pour réutiliser les *bindings* OCaml de code JavaScript, une extension supplémentaire de Wasm sera requise : la proposition *reference-typed string* [142]. La plupart des appels externes dans la FFI JavaScript utilisent la fonction `Js.Unsafe.meth_call`, dont le type est `'a -> string -> any array -> 'b`. Cette fonction pourrait être exposée à un module Wasm par l'hôte sous la forme d'une fonction ayant la signature suivante :

```
(func $meth_call
  (param $obj externref)
  (param $method stringref)
  (param $args $anyarray)
  (result externref))
```

Cette fonction appelle une méthode nommée `$method` sur l'objet `$obj`, avec les arguments `$args`. Le côté JavaScript prend en charge tout le typage dynamique.

### 7.7.2 Support des gestionnaires d'effets

Notre compilateur est basé sur OCaml 4.14. Le compilateur OCaml 5 introduit les gestionnaires d'effets [79], un mécanisme qui peut être vu comme une généralisation des exceptions, permettant notamment leur reprise après capture. Bien que ce mécanisme ne soit pas encore pris en charge par notre compilateur, nous décrivons ici trois stratégies potentielles pour les gérer à l'avenir.

**Compilation CPS** Il est possible de représenter les gestionnaires d'effets sous forme de continuations en utilisant une transformation CPS (Continuation-Passing Style) sur l'ensemble du programme [92]. Cependant, cette approche présente deux inconvénients majeurs. Premièrement, elle exige que le programme ne contienne que du code OCaml, excluant ainsi les interactions avec d'autres langages. Deuxièmement, cette méthode empêche l'utilisation de la pile du langage cible, obligeant à stocker cette pile dans le tas, ce qui entraîne un coût de performance non négligeable. Ce coût peut être atténué en n'appliquant la transformation qu'au code pour lequel nous ne pouvons pas prouver qu'il n'utilise pas de gestionnaires d'effets. Cependant, cela n'est plus compatible avec la compilation séparée et nécessite une optimisation réalisée sur l'ensemble du programme. C'est l'approche adoptée par js_of_ocaml.

**Wasm *Stack Switching*** Il existe une proposition Wasm en cours de développement, appelée *stack switching* [126], qui correspond précisément aux besoins des gestionnaires d'effets d'OCaml. Avec cette proposition, la traduction des gestionnaires d'effets serait assez simple et directe.

**JavaScript *Promise Integration*** Une autre stratégie est envisageable pour les environnements d'exécution qui supportent à la fois Wasm et JavaScript. Il existe une proposition en cours, appelée *JavaScript Promise Integration* [125]. Avec cette proposition, les gestionnaires d'effets pourraient être mis en œuvre via un mécanisme basé sur JavaScript. Cette proposition pourrait voir le jour avant celle du changement de pile et constituer une solution temporaire en attendant une prise en charge complète des gestionnaires d'effets via le *stack switching*.

## 7.8 Récapitulatif des fonctionnalités OCaml prises en charge par Wasocaml

La quasi-totalité du langage est prise en charge par Wasocaml. Les parties à compléter sont :

- les objets, une primitive Flambda n'ayant pas encore été traitée (`send`) par manque de temps ;
- les effets, comme décrit précédemment ;
- les FFI C et JavaScript, comme décrit précédemment ;
- l'implémentation du *runtime* est incomplète et quelques primitives sont manquantes.

## 7.9 Travaux connexes

### 7.9.1 Langages avec glaneur de cellules vers Wasm

#### Compilateurs ciblant Wasm1

**Go** Go se compile vers Wasm [102]. Pour réutiliser le glaneur de cellules de Go, celui-ci doit pouvoir inspecter la pile. Cela n'est pas possible dans Wasm. Ainsi, Go doit gérer lui-même la pile au moyen d'une *shadow stack* dans la mémoire linéaire. Cela est beaucoup plus lent que d'utiliser la pile native de Wasm.

**Haskell** Haskell se compile également vers Wasm [140]. Il rencontre des contraintes similaires à celles de Go et adopte des solutions similaires. Notamment, Haskell utilise aussi la mémoire linéaire pour représenter la pile.

#### Compilateurs ciblant WasmGC

**Dart** Dart [141] se compile vers WasmGC. Il n'utilise pas le type `ref i31` et emballe les scalaires dans une `struct`.

**Hoot** La dernière addition à la famille des compilateurs ciblant WasmGC est le compilateur Scheme de Guile. Le compilateur ciblant Wasm est nommé Hoot. Scheme rencontre de nombreuses contraintes similaires à OCaml, et Guile utilise des solutions proches de celles développées pour Wasocaml, bien que les deux outils aient été développés indépendamment. Le compilateur a été présenté dans les grandes lignes au groupe de travail WasmGC [159]. Une explication plus détaillée a été publiée en ligne [158]. La mise en œuvre et sa description technique approfondie sont également disponibles [160].

### 7.9.2 Compilateurs OCaml pour le Web

L'histoire des compilateurs OCaml ciblant les langages web est assez fournie. Cela s'explique probablement par le grand plaisir qu'apporte l'écriture de compilateurs en OCaml.

#### Compilateurs ciblant JavaScript

Il existe plusieurs compilateurs OCaml vers JavaScript, avec de nombreuses approches explorées. Les deux projets principaux actuellement maintenus sont *js_of_ocaml* et *melange*. La compilation naïve d'OCaml vers JavaScript est assez simple, se résumant presque à un *effacement de type*. Cependant, certaines limitations de JavaScript empêchent cette approche d'être complète. De plus, produire du code JavaScript efficace et de petite taille est une tâche plus complexe.

**Jsoo** Jsoo [83] tente d'être aussi proche que possible de la sémantique native. Il compile le bytecode OCaml vers JavaScript en les décompilant en un lambda-calcul simple non typé, suivi de plusieurs passes d'optimisation et de minimisation.

**Melange** Melange [138] produit du JavaScript plus lisible, avec une meilleure intégration dans le système de modules JavaScript. Melange part d'une version modifiée de la représentation intermédiaire *Lambda*, fournissant plus d'informations de typage, ce qui permet l'utilisation de fonctionnalités *JavaScript* correspondant aux usages des fonctionnalités source, au prix de quelques différences sémantiques mineures.

#### Compilateurs ciblant Wasm

**OCamlrun WebAssembly** OCamlrun WebAssembly [93] compile l'interpréteur et le runtime OCaml vers Wasm. Ceux-ci étant écrits en C, cela est possible sans aucune extension Wasm. Dans certains cas, cela permet de démarrer l'exécution plus rapidement qu'en compilant OCaml vers Wasm, car il y a moins de code Wasm à compiler pour l'hôte. Cependant, l'exécution est plus lente car il s'agit d'interpréter le bytecode OCaml à l'intérieur de Wasm, au lieu d'exécuter une version compilée.

**WASICaml** WASICaml [124] est assez similaire à OCamlrun WebAssembly, mais il ne se contente pas d'interpréter directement le bytecode ; il le traduit partiellement en Wasm, ce qui conduit à une exécution plus rapide.

Ces approches rencontrent les mêmes problèmes que les programmes C exécutés en Wasm : ils ne peuvent pas aisément manipuler nativement des valeurs externes, comme des

objets DOM dans le navigateur. En effet, dans les premières versions de Wasm, il n'était pas possible de manipuler directement les valeurs de l'environnement hôte (par exemple, des objets JavaScript). Il serait envisageable d'identifier ces objets par un entier et de les associer à leurs objets correspondants côté hôte. Cependant, cette approche présente les limitations habituelles d'avoir deux runtimes manipulant des valeurs ramassées par un GC, avec un risque de fuites de mémoire dû aux cycles entre les deux runtimes.

**Wasm_of_ocaml** Wasm_of_ocaml [157] est un fork de js_of_ocaml. Il compile le bytecode OCaml vers WasmGC. Il a été développé après Wasocaml et est basé sur les techniques que nous avons développées. Il ne bénéficie pas des optimisations du compilateur ou de Flambda. Il est aussi très limité dans ses choix de représentation des valeurs et doit forcément utiliser une représentation basé sur des tableaux.

# 8 Formalisation de la compilation et preuve de correction

Après avoir décrit de manière informelle notre compilateur OCaml vers WasmGC, nous proposons dans ce chapitre une formalisation pour un sous-ensemble du compilateur. Nous commençons par donner une définition complète de la fonction de compilation, avant de prouver sa correction.

## 8.1 Schéma de compilation

### 8.1.1 Prélude

Lors de la compilation d'un programme, le code généré est toujours accompagné d'un prélude commun à tous les programmes. Celui-ci contient principalement des définitions de types et des variables globales dont nous faisons usage lors de la compilation. Nous détaillons maintenant le code de ce prélude.

Nous définissons le type `$mlfun`, qui sera le type de toutes nos fonctions :

```
(type $mlfun (sub final
  (func (param eqref) (param eqref) (result eqref))))
```

Puis, nous définissons les types servant à représenter les blocs. Ceux-ci sont composés d'une étiquette et de données :

```
(type $data (sub final (array (mut (ref null eq)))))

(type $block (sub final
  (struct
    (field $tag i32)
    (field $data (ref null $data)))))
```

Ensuite, nous définissons les types servant à représenter les ensembles de fermetures :

```
;; 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))))))
```

Enfin, nous déclarons deux variables globales qui seront utilisées par le code généré :

```
(global $tmp_set_of_closures (mut (ref null $set_of_closures)) ref.null
↪  $set_of_closures)

(global $switch_v (mut i32) i32.const 0)
```

### 8.1.2 Passe d'analyse précédent la compilation

Avant de compiler un programme Flambda entier, nous allons parcourir celui-ci en profondeur pour extraire le code de toutes les fermetures. Chacune de ces fermetures sera compilée indépendamment vers une fonction Wasm. Le code du programme initial sera alors inséré dans le prélude, au début de la fonction `$start`. Celui-ci renverra forcément un `ref eq`, comme montré plus tard. Ainsi, il est nécessaire d'insérer un `drop` à la fin de la fonction `$start` pour que celle-ci ait le type idoine :

```
(func $start
 ;; …
 drop
)
```

De plus, chacune des fonctions sera marquée comme un élément déclaratif dans le prélude, afin d'autoriser leur utilisation lors de la création des références vers les fonctions servant à l'allocation des ensembles de fermeture :

```
(elem declare (ref $mlfun)
 ;; all the functions will be added here to allow forward references e.g. in
 ↪  the func.ref instruction
 …
)
```

De plus, lors de notre parcours en profondeur, nous allons collecter des informations sur chaque fonction et chaque variable capturée dans les ensembles de fermetures. En effet, pour chacune, il nous faut connaître son indice dans l'ensemble de fermetures. Chacun de ces indices sera stocké dans une variable globale, par exemple `$closure_offset_within_set_myfunc`. Enfin, chaque fonction est analysée afin de collecter l'ensemble des variables globales qu'il sera nécessaire de déclarer en Wasm.

### 8.1.3 Fonction de compilation

Nous définissons la fonction de compilation des termes Flambda vers les expressions de WasmGC. Il s'agit de la fonction $Comp : t \to i^*$.

$$Comp(n) ::= \texttt{i32.const}\ n \\ \texttt{i31.of\_i32}$$

Un littéral entier est converti en un `ref i31` afin de correspondre à la représentation uniforme.

$$Comp(id) ::= \texttt{local.get}\ \$id$$

Pour accéder à la valeur d'un identifiant, nous accédons simplement à la variable locale de la fonction courante.

$$Comp(id_1\ \texttt{+}\ id_2) ::= \texttt{local.get}\ \$id_1 \\ \texttt{i32.of\_i31} \\ \texttt{local.get}\ \$id_2 \\ \texttt{i32.of\_i31} \\ \texttt{i32.add} \\ \texttt{i31.of\_i32}$$

Nous compilons l'addition en plaçant le contenu des deux opérandes sur la pile après les avoir récupérées depuis les variables locales et converties en `i32`. Le résultat est reconverti en un `ref i31`.

$$Comp(id_1\ \texttt{-}\ id_2) ::= \texttt{local.get}\ \$id_1 \\ \texttt{i32.of\_i31} \\ \texttt{local.get}\ \$id_2 \\ \texttt{i32.of\_i31} \\ \texttt{i32.sub} \\ \texttt{i31.of\_i32}$$

Nous compilons la soustraction en plaçant le contenu des deux opérandes sur la pile après les avoir récupérées depuis les variables locales et converties en `i32`. Le résultat est reconverti en un `ref i31`.

$$
\begin{aligned}
Comp(id_1 = id_2) ::= \ & \texttt{local.get \$}id_1 \\
& \texttt{i32.of\_i31} \\
& \texttt{local.get \$}id_2 \\
& \texttt{i32.of\_i31} \\
& \texttt{i32.eq} \\
& \texttt{i31.of\_i32}
\end{aligned}
$$

Nous compilons le test d'égalité en plaçant le contenu des deux opérandes sur la pile après les avoir récupérées depuis les variables locales et converties en `i32`. Le résultat est reconverti en un `ref i31`.

$$
\begin{aligned}
Comp(\texttt{let } id = t_1 \texttt{ in } t_2) ::= \ & Comp(t_1) \\
& \texttt{local.set \$}id \\
& Comp(t_2)
\end{aligned}
$$

Une liaison `let` est compilée en évaluant $t_1$, en stockant son résultat dans la variable locale $id$ et en évaluant $t_2$.

$$
\begin{aligned}
Comp(id_1\ id_2) ::= \ & \texttt{local.get \$}id_1 \\
& \texttt{ref.cast (ref \$closure)} \\
& \texttt{local.get \$}id_2 \\
& \texttt{ref.as\_non\_null} \\
& \texttt{local.get \$}id_1 \\
& \texttt{ref.cast (ref \$closure)} \\
& \texttt{struct.get \$closure 0} \\
& \texttt{call\_ref \$mlfun}
\end{aligned}
$$

Pour appliquer une fonction $id_1$ à une valeur $id_2$, nous commençons par placer les deux paramètres de la fonction sur la pile. Le premier est la fermeture elle-même, nécessaire pour accéder à l'environnement et aux autres fonctions dans l'ensemble de fermetures. Le second est une valeur de type `ref eq`, qui est le paramètre effectif de la fonction. Ensuite, nous mettons la fonction sur la pile en accédant au champ 0 de la fermeture, qui est un `funcref`. Enfin, nous utilisons `call_ref` pour appeler la fonction avec ses deux arguments. Le type de la référence de la fonction est toujours `$mlfun`.

$$
\begin{aligned}
Comp(id_1 \texttt{ <- } id_2) ::= \ & \texttt{local.get \$}id_2 \\
& \texttt{local.set \$}id_1 \\
& \texttt{i32.const 0} \\
& \texttt{i31.of\_i32}
\end{aligned}
$$

Une mue place simplement le contenu de $id_1$ dans $id_2$ et dépose un 0 sur la pile (ce qui correspond à `() : unit` en OCaml).

```
Comp(if id then t1 else t2) ::= local.get $id
                                ref.cast (ref null i31)
                                i32.of_i31
                                (if (result (ref eq))
                                 (then Comp(t1))
                                (else Comp(t2)))
```

Une conditionnelle convertit la valeur de *id* en un `i32` puis utilise un `if` Wasm. Les deux branches sont simplement le résultat de la compilation de $t_1$ et $t_2$.

```
Comp(switch id B1 B2) ::=
 block $switch1 (result (ref null eq))
  block $switch2 (result (ref null eq))
   local.get $id
   ;; conditionally jump to scalar case
   br_on_cast $switch2 (ref null eq) (ref null i31)
   ;; handle block case
   ref.cast (ref $block)
   struct.get $block 0
   global.set $switch_v
   CompB(B2)
   ;; skip scalar case
   br $switch1)
  ;; handle scalar case
  ref.cast (ref i31)
  i32.of_i31
  global.set $switch_v
  CompB(B1))
```

Pour compiler un `switch`, nous commençons par tester si *id* est un bloc ou bien une valeur scalaire. Si c'est un bloc, nous stockons son étiquette dans la variable globale `$switch_v`, sinon nous stockons directement la valeur scalaire. Ensuite, nous utilisons une fonction auxiliaire *CompB* afin de compiler les différents cas de chaque branche. Soient $i_0$ ... $i_{n-1}$ tous les éléments de *dom*(B)$. Nous définissons alors la fonction auxiliaire comme suit :

$$
\begin{aligned}
&\mathit{CompB}(B) ::= \\
&\texttt{(i32.eq (i32.const } i_{n-1}\texttt{) (global.get \$switch\_v))} \\
&\texttt{(if (result (ref null eq)) (then } \mathit{Comp}(B(i_{n-1}))\texttt{))} \\
&\texttt{(else} \\
&\quad\texttt{;; …} \\
&\quad\texttt{(i32.eq (i32.const } i_0\texttt{) (global.get \$switch\_v))} \\
&\quad\texttt{(if (result (ref null eq)) (then } \mathit{Comp}(B(i_0))\texttt{))} \\
&\quad\texttt{(else unreachable))))}
\end{aligned}
$$

Nous produisons une suite de `if` imbriqués afin de trouver quel cas de $B$ nous devons évaluer[1]. Si aucun d'eux ne correspond à la valeur stockée dans `$switch_v` (qui a été écrite par le code généré lors de la compilation du `switch`) alors nous devons émettre un `unreachable` afin de générer du code bien typé. Le `unreachable` correspond à une exception `Match_failure` en OCaml. Ce cas ne doit pas arriver si tous les filtrages de motif sont exhaustifs dans le programme source et ce code ne peut donc effectivement pas être atteint.

$$
\begin{aligned}
\mathit{Comp}(\texttt{sraise } id_e\ id_0\ \ldots\ id_{n-1}) ::= \ & \texttt{local.get \$}id_{n-1} \\
& \ldots \\
& \texttt{local.get \$}id_0 \\
& \texttt{br \$exn\_}id_e
\end{aligned}
$$

Pour compiler un `sraise`, nous plaçons toutes les valeurs transportées par l'exception sur la pile et nous effectuons un branchement vers le bloc correspondant au `sraise`.

$$
\begin{aligned}
&\mathit{Comp}(\texttt{scatch } t_1 \texttt{ with } id_e\ id_0\ \ldots\ id_{n-1}\ t_2) ::= \\
&\texttt{(block \$after\_try\_catch (result (ref null eq))} \\
&\quad\texttt{(block \$exn\_}id_e \\
&\quad\quad\texttt{(result (ref null eq)) ;; 0} \\
&\quad\quad\texttt{(result (ref null eq)) ;; …} \\
&\quad\quad\texttt{(result (ref null eq)) ;; n - 1} \\
&\quad\quad \mathit{Comp}(t_1) \\
&\quad\quad\texttt{br \$after\_try\_catch)} \\
&\quad\texttt{local.set \$}id_0 \\
&\quad\ldots \\
&\quad\texttt{local.set \$}id_{n-1} \\
&\quad \mathit{Comp}(t_2))
\end{aligned}
$$

Un `scatch` est composé de deux blocs imbriqués. Le premier sert à traiter le cas où aucune exception n'est levée et renvoie directement la valeur de $t_1$. Le second correspond au cas où

1. Nous pourrions à la place utiliser une instruction `br_table` qui permettrait d'optimiser ce code, mais cela n'est pas fait dans la formalisation afin d'éviter d'avoir à rajouter une instruction supplémentaire.

une exception est levée. Il doit avoir plusieurs valeurs de retours qui correspondent aux valeurs transportées par l'exception. Toutes ces valeurs sont mises dans les variables locales idoines.

$$
\begin{aligned}
Comp(\texttt{while}_{t_1}\ t_1\ \texttt{do}\ t_2) ::= \texttt{(block \$exit\_while\_loop}\\
&\texttt{(loop \$while\_loop}\\
&\quad Comp(t_1)\\
&\quad \texttt{i32.of\_i31}\\
&\quad \texttt{i32.eqz}\\
&\quad \texttt{br\_if \$exit\_while\_loop}\\
&\quad Comp(t_2)\\
&\quad \texttt{drop}\\
&\quad \texttt{br \$while\_loop))}\\
&\texttt{i32.const 0}\\
&\texttt{i31.of\_i32}
\end{aligned}
$$

Une boucle OCaml `while` se compile presque directement vers une `loop` Wasm. Nous devons lui ajouter un bloc afin de sortir de la boucle lorsque la condition est fausse.

$$
\begin{aligned}
Comp(\texttt{get\_field}\ n\ id) ::= &\texttt{local.get \$}id\\
&\texttt{ref.cast (ref \$block)}\\
&\texttt{struct.get \$block 1}\\
&\texttt{ref.as\_non\_null}\\
&\texttt{i32.const}\ n\\
&\texttt{array.get \$data}
\end{aligned}
$$

Pour lire un champ d'un bloc, il faut d'abord le convertir en un `block`. Nous accédons ensuite à son second champ qui contient les valeurs du blocs (le premier contient l'étiquette). Finalement, nous lisons simplement l'élément à l'indice $n$.

$$
\begin{aligned}
Comp(\texttt{set\_field}\ n\ id_1\ id_2) ::= &\texttt{local.get \$}id_1\\
&\texttt{ref.cast (ref \$block)}\\
&\texttt{struct.get \$block 1}\\
&\texttt{ref.cast (ref \$data)}\\
&\texttt{i32.const}\ n\\
&\texttt{local.get}\ id_2\\
&\texttt{array.set \$data}\\
&\texttt{i32.const 0}\\
&\texttt{i31.of\_i32}
\end{aligned}
$$

De la même manière, pour écrire un champ d'un bloc, nous le convertissons, puis nous accédons à son champ `$data` avant d'écrire la valeur de $id_2$ à l'indice $n$. Le tableau étant muable,

il n'est nul besoin de mettre à jour la structure du bloc.

$$
\begin{aligned}
&\mathit{Comp}(\texttt{make\_block}\ n\ id_0, ..., id_{m-1}) ::= \\
&\texttt{i32.const}\ n \\
&\texttt{local.get}\ \$id_0 \\
&... \\
&\texttt{local.get}\ \$id_{m-1} \\
&\texttt{array.new\_fixed}\ \$\texttt{data}\ m \\
&\texttt{struct.new}\ \$\texttt{block}
\end{aligned}
$$

Pour créer un nouveau bloc, nous déposons son étiquette sur la pile. Ensuite, nous créons un nouveau tableau avec les données du bloc. Enfin, une structure est créée à partir de l'étiquette et du tableau de données.

$$
\begin{aligned}
&\mathit{Comp}(\texttt{project\_closure}\ id_1\ id_2) ::= \\
&\texttt{local.get}\ \$id_1 \\
&\texttt{ref.cast}\ (\texttt{ref}\ \$\texttt{set\_of\_closures}) \\
&\texttt{struct.get}\ \$\texttt{set\_of\_closures}\ \texttt{0} \\
&\texttt{global.get}\ \$\texttt{closure\_offset\_within\_set\_}id_2 \\
&\texttt{array.get}\ \$\texttt{closures}
\end{aligned}
$$

Pour récupérer une fermeture depuis un ensemble de fermeture, nous commençons par convertir $id_2$ en un `$set_of_closures`. Ensuite, nous accédons à son premier champ, qui est un tableau de fermetures. Il suffit alors d'accéder à l'élément situé au bon indice, lequel indice est accessible depuis une variable globale.

$$
\begin{aligned}
\mathit{Comp}(\texttt{project\_var}\ id_1\ id_2) ::= \ & \texttt{local.get}\ \$id_1 \\
& \texttt{ref.cast}\ (\texttt{ref}\ \$\texttt{closure}) \\
& \texttt{struct.get}\ \$\texttt{closure}\ \texttt{1} \\
& \texttt{ref.as\_non\_null} \\
& \texttt{struct.get}\ \$\texttt{set\_of\_closures}\ \texttt{1} \\
& \texttt{global.get}\ \$\texttt{var\_offset\_within\_set\_}id_2 \\
& \texttt{array.get}\ \$\texttt{vars}
\end{aligned}
$$

Ce cas est similaire au précédent, à la différence que $id_2$ est une fermeture et non un ensemble de fermetures. Nous devons donc commencer par accéder à son ensemble parent. Il suffit ensuite de procéder de la même manière mais en lisant cette fois dans le second champ de l'ensemble, qui contient les variables capturées.

```
Comp(move_within_set id1 id2) ::= local.get $id1
                                  ref.cast (ref $closure)
                                  struct.get $closure 1
                                  ref.as_non_null
                                  struct.get $set_of_closures 0
                                  global.get $closure_offset_within_set_id2
                                  array.get $closures
```

Ce cas est identique au précédent, nous lisons simplement dans le premier champ plutôt que dans le second afin de récupérer une fermeture et non une variable.

```
Comp(make_set (id_f1 → f1 ... id_fm → f_m) (id_1l -> id_1r ... id_nl -> id_nr)) ::=
;; empty closures
ref.null $closures
;; generating captured variables array
(ref.as_non_null (local.get $id_1r))
...
(ref.as_non_null (local.get $id_nr))
array.new_fixed $vars n
;; set of closures with empty closures but filled vars
struct.new $set_of_closures
global.set $tmp_set_of_closures
global.get $tmp_set_of_closures
;; generating closures
ref.func $id_f1
global.get $tmp_set_of_closures
struct.new $closure
...
ref.func $id_fm
global.get $tmp_set_of_closures
struct.new $closure
array.new_fixed $closures m
;; patching the set with closures
struct.set $set_of_closures $closures
global.get $tmp_set_of_closures
```

Nous commençons par définir un tableau vide de fermetures. Nous construisons ensuite le tableau composé de variables capturées. Puis, nous générons un ensemble de fermetures à partir des deux tableaux précédents. Chaque fermeture a besoin d'une référence vers l'ensemble parent, nous stockons donc la référence vers l'ensemble dans une variable globale temporaire

avant de constuire les fermetures. Enfin, nous construisons le tableau contenant les véritables fermetures. Le code de chacune ayant été compilé dans une fonction Wasm dont le nom est connu statiquement, nous pouvons créer une référence vers chacune. Nous pouvons finalement mettre à jour l'ensemble pour qu'il contienne le nouvel ensemble de fermeture.

### 8.1.4 Exemple de programme compilé

Considérons le programma Flambda suivant, similaire à celui présenté précedemment :

```
let xa = 21 in let ya = 32 in
let v_even = make_block 0 xa ya in

let xb = 43 in let yb = 54 in
let v_odd = make_block 0 xb yb in

let max_x = 1000 in

let set =
  make_set
  (* functions *)
  | even -> \ pos self.
    let max_x = project_var self max_x in
    let old_x = get_field 0 pos in
    let delta = old_x - max_x in
    let zero = 0 in
    let cond = delta = zero in
    if cond then
      let v_even = project_var self v_even in
      let even_x = get_field 0 v_even in
      let new_x = old_x + even_x in
      let _dummya = set_field 0 pos new_x in
      let old_y = get_field 1 pos in
      let even_y = get_field 1 v_even in
      let new_y = old_y + even_y in
      let _dummyb = set_field 1 pos new_y in
      let odd = move_within_set self odd in
      odd pos
    else 0
  | odd -> \ pos self.
    let max_x = project_var self max_x in
    let old_x = get_field 0 pos in
    let delta = old_x - max_x in
    let zero = 0 in
    let cond = delta = zero in
    if cond then
      let v_odd = project_var self v_odd in
```

```
      let odd_x = get_field 0 v_odd in
      let new_x = old_x + odd_x in
      let _dummya = set_field 0 pos new_x in
      let old_y = get_field 1 pos in
      let odd_y = get_field 1 v_odd in
      let new_y = old_y + odd_y in
      let _dummyb = set_field 1 pos new_y in
      let even = move_within_set self odd in
      even pos
    else 0
  (* values *)
  | v_even -> v_even
  | v_odd -> v_odd
  | max_x -> max_x
in
let x = 1 in let y = 1 in
let initial_pos = make_block 0 x y in

let even = project_closure set even in
even initial_pos
```

Après la passe d'analyse précédent la compilation décrite précédemment, nous obtiendrons :

```
(global $closure_offset_within_set_odd (mut i32) i32.const 1)
(global $closure_offset_within_set_even (mut i32) i32.const 0)
(global $var_offset_within_set_v_even (mut i32) i32.const 0)
(global $var_offset_within_set_v_odd (mut i32) i32.const 1)
(global $var_offset_within_set_max_x (mut i32) i32.const 2)
(elem declare (ref $mlfun) (item ref.func $even) (item ref.func $odd))
```

Puis, les deux fonctions `even` et `odd` seront compilées séparément :

```
(func $even (param $self eqref) (param $pos eqref) (result eqref)
  (local $max_x eqref) (local $old_x eqref) (local $delta eqref) (local $zero
  ↪  eqref) (local $cond eqref) (local $v_even eqref) (local $even_x eqref)
  ↪  (local $new_x eqref) (local $_dummya eqref) (local $old_y eqref) (local
  ↪  $even_y eqref) (local $new_y eqref) (local $_dummyb eqref) (local $odd
  ↪  eqref) (local $res eqref)
  local.get $self
  ref.cast (ref  $closure)
  struct.get $closure 1
  ref.as_non_null
  struct.get $set_of_closures 1
  global.get $var_offset_within_set_max_x
  ;; …
  local.set $cond
  local.get $cond
```

```
  ref.cast (ref i31)
  i31.get_u
  (if (result (ref null eq))
    (then
      local.get $self
      ref.cast (ref  $closure)
      struct.get $closure 1
      ref.as_non_null
      struct.get $set_of_closures 1
      global.get $var_offset_within_set_v_even
      array.get $vars
      local.set $v_even
      ;; …
      call_ref $mlfun
    )
    (else
      i32.const 0
      ref.i31
      local.set $res
      local.get $res
    )
  )
  ref.as_non_null
)

(func $odd (param $self eqref) (param $pos eqref) (result eqref)
  (local $max_x eqref) ;; …
  ;; …
)
```

Le terme issu du programme initial quant à lui, sera compilé comme la fonction `$start` avec un `drop` ajouté à la fin :

```
(func $start
  (local $xa (ref null eq)) ;; …
  i32.const 21
  ;; …
  call_ref $mlfun
  drop
)
(start $start)
```

## 8.2 Preuve de correction

### 8.2.1 Sous-ensemble considéré

Nous nous plaçons dans un sous-ensemble de notre définition de Flambda. Nous restreignons les valeurs aux entier et aux adresses :

$$
\begin{array}{rll}
v & ::= & n \quad \text{(entier)} \\
& | & a \quad \text{(adresse)}
\end{array}
$$

Les termes sont restreints également :

$$
\begin{array}{rll}
t & ::= & v \quad \text{(valeur)} \\
& | & id \quad \text{(identifiant)} \\
& | & \texttt{let}\ id\ \texttt{=}\ t\ \texttt{in}\ t \quad \text{(liaison)} \\
& | & id\ \texttt{<-}\ id \quad \text{(mue)} \\
& | & \texttt{if}\ id\ \texttt{then}\ t\ \texttt{else}\ t \quad \text{(conditionnelle)} \\
& | & \texttt{get\_field}\ n\ id \quad \text{(lecture de champ)} \\
& | & \texttt{set\_field}\ n\ id\ id \quad \text{(écriture de champ)} \\
& | & \texttt{make\_block}\ n\ id^* \quad \text{(création de bloc)} \\
& | & id\ op\ id \quad \text{(application binaire)}
\end{array}
$$

### 8.2.2 Validité des programmes générés

Nous commençons par montrer que tous les programmes générés sont valides au sens du typage de Wasm et de sa définition de bonne formation. La preuve se fait par induction structurelle sur $t$. L'argument permettant de mener la preuve à bien est le maintient d'un invariant affirmant que la compilation d'un terme $t$, $Comp(t)$ s'évalue toujours en une valeur de type `ref eq` et que toutes nos fonctions ont le type `$mlfun` (dans le cas où nous voudrions étendre la preuve aux fonctions). Nous ne détaillons pas la preuve ici qui ne présente aucune difficulté particulière en utilisant les invariants fournis.

### 8.2.3 Définition de l'équivalence sur les valeurs et les configurations

Nous définissons une équivalence sur les valeurs, pour cela il nous faut également définir une relation sur les configurations. Nous notons $v \approx v'$ pour indiquer qu'une valeur Flambda $v$ est équivalente à une valeurs Wasm $v'$.

**Équivalence sur les valeurs** Dans le cas où $v$ est un entier $n$, alors $v \approx v'$ si et seulement si $v'$ est un `ref i31` tel que une fois convertit en `i32`, alors les deux valeurs sont égales.

Dans le cas où $v$ est une adresse $a$, alors $v \approx v'$ ne peut se définir que dans une configuration comprenant $M$ et $E$. Si $M(a) = n, (v_0, ..., v_{m-1})$, alors, pour que $v \approx v'$, il faut que :

— $v'$ soit un `ref $block` ayant pour valeur `struct` $a'$

— et

$$E.\texttt{heap}(a') \;=\; \texttt{struct \$block}\; n\; a''$$

— et

$$E.\texttt{heap}(a'') \;=\; \texttt{array \$data}\; v'_0 \ldots v'_{m-1}$$

— et pour tout $i$, si $0 \leq i < m$, alors, $v_i \approx v'_i$.

**Relation entre les configurations** Deux configurations $V^*, M, t$ et $E, (L, v^*, (\mathit{Comp}(t), i^*))F^*$ sont en relation si et seulement si pour tout $\mathit{id}$, si $V_0(\mathit{id}) = v$, alors, $L.\texttt{env}(\mathit{id}) = v'$ et $v \approx v'$.

### 8.2.4 Préservation de la sémantique

Nous montrons que notre fonction de compilation préserve la sémantique.

**Théorème 8.1** *Pour tout terme Flambda t, si celui-ci se réduit vers une valeur v, alors l'évaluation de Comp(t) termine et produit une valeur équivalente et la dépose sur la pile.*

Plus formellement, nous voulons montrer que, pour tout $t$, si :

$$V^*, M, t \to V'^*, M', v$$

alors, $\forall i^*.\forall F^*.\forall v^*$, si $V^*, M, t$ et $E, (L, v^*, (\mathit{Comp}(t), i^*))F^*$ sont en relation nous avons :

$$E, (L, v^*, (\mathit{Comp}(t), i^*))F^* \to E', (L', (v', v^*), i^*)F^*$$

avec $v \approx v'$, et les configurations $V'^*, M', v$ et $E', (L', (v', v^*), i^*)F^*$ sont également en relation.

Nous effectuons un raisonnement par induction structurelle sur $t$.

**Cas où $t$ est une valeur** Dans ce cas, $t = v$ et $v$ est forcément un entier $n$ ou une adresse $a$ puisque ce sont le seul cas restant dans notre sous-ensemble. Il ne peut s'agir d'une adresse puisqu'elles n'apparaissent qu'au cours de l'exécution et que le programme source n'en contient pas (elles sont en quelque sorte des valeurs administratives). Nous avons donc $t = n$. Il n'y a pas de règle de réduction à appliquer, et donc l'environnement et la mémoire ne sont pas modifiés.

Par définition :

$$\mathit{Comp}(n) = \texttt{i32.const}\; n \\ \texttt{i31.of\_i32}$$

qui va bien s'évaluer en une constante $n$ qui sera déposée sur la pile. Nous avons donc $\forall i^*.\forall F^*.\forall v^*$ que :

$$E, (L, v^*, (\mathit{Comp}(n), i^*))F^* \to E, (L, (\texttt{i31}\; n, v^*), i^*)F^*$$

Nos deux configurations sont toujours en relation et nous avons bien déposé exactement une valeur équivalente sur la pile.

**Cas où $t$ est un identifiant** Nous appliquons la règle de réduction qui va simplement lire la valeur de $id$ dans l'environnement. Puisque le programme termine, elle est forcément dans l'environnement, sinon, aucune autre règle ne s'applique et le programme ne suis réduit pas en une valeur. Le programme se réduit alors en $V_0(id) = v$ :

$$V^*, M, id \to V^*, M, v$$

Par définition :

$$Comp(id) = \texttt{local.get \$}id$$

qui va s'évaluer en une valeur qui sera déposée sur la pile puisque nos deux environnements sont bien en relation. Nous avons donc $\forall i^*.\forall F^*.\forall v^*$ :

$$E, (L, v^*, (Comp(id), i^*))F^* \to E, (L, (L.\mathsf{env}(id), v^*), i^*)F^*$$

Nos deux configurations sont toujours en relation et nous avons bien déposé exactement une valeur équivalente sur la pile.

**Cas où $t$ est un opérateur binaire** Dans le cas de l'addition, nous appliquons la règle de réduction qui va simplement lire les valeur de $id_1$ et $id_2$ dans l'environnement et les additionner. Puisque le programme termine, les deux valeurs sont forcément dans l'environnement, sinon, aucune autre règle ne s'applique et le programme ne suis réduit pas en une valeur. Le programme se réduit alors en $V_0(id_1) + V_0(id_2) = v$ :

$$V^*, M, id_1 \texttt{ + } id_2 \to V^*, M, v$$

Par définition :

$$\begin{aligned} Comp(id_1 \texttt{ + } id_2) = \ & \texttt{local.get \$}id_1 \\ & \texttt{i32.of\_i31} \\ & \texttt{local.get \$}id_2 \\ & \texttt{i32.of\_i31} \\ & \texttt{i32.add} \\ & \texttt{i31.of\_i32} \end{aligned}$$

qui va bien trouver les deux valeurs $\texttt{\$}id_1$ et $\texttt{\$}id_2$ dans l'environnement puisque nos deux configurations sont en relation. Leur valeurs sont équivalentes à celles côté Flambda pour la même raison. Ainsi l'addition produira une valeur $v'$ équivalente à $v$. Nous avons donc $\forall i^*.\forall F^*.\forall v^*$ :

$$E, (L, v^*, (Comp(id), i^*))F^* \to E, (L, (v', v^*), i^*)F^*$$

Nos deux configurations sont toujours en relation et nous avons bien déposé exactement une valeur équivalente sur la pile.

Les cas liés à la soustraction et au test d'égalité sont similaires et nous ne les détaillons pas ici.

**Cas où $t$ est une liaison** Par définition de nos contextes de réduction, si $t$ s'évalue en une valeur, c'est d'abord que $t_1$ s'est évalué en une valeur $v_1$. Nous avons donc `let` *id* `=` $v_1$ `in` $t_2$. Ce terme se réduit en un pas de réduction vers $V_0[id \leftarrow v_1], t_2$. Puis, puisque le terme initial se réduit en une valeur, c'est que $t_2$ s'est évalué en une valeur $v_2$.

Par définition :

$$\begin{aligned} \mathit{Comp}(\texttt{let } id \texttt{ = } t_1 \texttt{ in } t_2) = \; & \mathit{Comp}(t_1) \\ & \texttt{local.set \$}id \\ & \mathit{Comp}(t_2) \end{aligned}$$

Par hypothèse d'induction, nous savons que $\mathit{Comp}(t_1)$ va déposer une valeur équivalente à $v_1$ sur la pile, nous nous retrouvons donc dans la configuration suivante :

$$\begin{aligned} & E, (L, v^*, (\mathit{Comp}(t), i^*))F^* \\ \rightarrow & E, (L, (v_1, v^*), (\texttt{local.set \$}id, \;\; \mathit{Comp}(t_2), i^*))F^* \end{aligned}$$

Puis, par définition de `local.set`, nous avons que :

$$\begin{aligned} & E, (L, (v_1, v^*), (\texttt{local.set \$}id \;\; \mathit{Comp}(t_2), i^*))F^* \\ \rightarrow & E, (L[env \leftarrow L.\mathsf{env}[id \leftarrow v_1]], v^*, (\mathit{Comp}(t_2), i^*))F^* \end{aligned}$$

Puis, par hypothèse d'induction, nous savons que $\mathit{Comp}(t_2)$ se réduit en une valeur équivalente à $v_2$ et la dépose sur la pile. Nous avons donc :

$$\begin{aligned} & E, (L[env \leftarrow L.\mathsf{env}[id \leftarrow v_1]], v^*, (\mathit{Comp}(t_2), i^*))F^* \\ \rightarrow & E, (L[env \leftarrow L.\mathsf{env}[id \leftarrow v_1]], (v_2, v^*), i^*)F^* \end{aligned}$$

Nos deux configurations sont toujours en relation et nous avons bien déposé exactement une valeur équivalente sur la pile.

**Cas où $t$ est une conditionnelle** Nous posons $t$ = `if` *id* `then` $t_1$ `else` $t_2$, puis par définition de la règle de réduction de tête, si $t$ s'évalue en une valeur, c'est que *id* était bien un entier dans l'environnement. Soit celui-ci vaut $0$, soit il est différent de $0$. Nous ne traitons que le cas où cet entier est différent de $0$, l'autre étant symétrique. Dans ce cas, nous savons que $t_1$ s'évalue en une valeur $v_1$, sinon le terme ne peut pas se réduire. Nous avons donc :

$$V^*, M, t \rightarrow V^*, M, v_1$$

Par définition :

$$
\begin{aligned}
\mathit{Comp}(\texttt{if } id \texttt{ then } t_1 \texttt{ else } t_2) = \ & \texttt{local.get \$}id \\
& \texttt{ref.cast (ref null i31)} \\
& \texttt{i32.of\_i31} \\
& \texttt{(if (result (ref eq))} \\
& \texttt{ (then } \mathit{Comp}(t_1)\texttt{)} \\
& \texttt{(else } \mathit{Comp}(t_2)\texttt{)))}
\end{aligned}
$$

Puisque nos configurations sont en relation, `$`$id$ va après conversion s'évaluer en un entier différent de $0$ et le résultat sera donc le résultat de la compilation de $t_1$.

Par hypothèse d'induction, nous savons que $\mathit{Comp}(t_1)$ dépose une valeur $v_1$ équivalente à la valeur en laquelle $t$ se réduit. Nous avons donc :

$$
\begin{aligned}
& E, (L, v^*, (\mathit{Comp}(t), i^*))F^* \\
\rightarrow & E, (L, (v_1, v^*), i^*)F^*
\end{aligned}
$$

Nos deux configurations sont toujours en relation et nous avons bien déposé exactement une valeur équivalente sur la pile.

**Cas où $t$ est une création de bloc** Nous appliquons la règle de réduction qui va récupérer les valeurs depuis l'environnement et les placer avec l'étiquette du bloc dans la mémoire à une adresse fraîche $a$. Nous notons $M'$ la nouvelle mémoire. Le programme se réduit alors en $a$.

$$
V^*, M, \texttt{make\_block} \ldots \rightarrow V^*, M', a
$$

Par définition :

$$
\begin{aligned}
& \mathit{Comp}(\texttt{make\_block } n \ id_0, \ldots, id_{m-1}) = \\
& \texttt{i32.const } n \\
& \texttt{local.get \$}id_0 \\
& \ldots \\
& \texttt{local.get \$}id_{m-1} \\
& \texttt{array.new\_fixed \$data } m \\
& \texttt{struct.new \$block}
\end{aligned}
$$

qui va bien trouver les valeurs dans l'environnement puisque nos deux configurations sont en relation. Par définition des instructions `array.new_fixed` et `struct.new`, la nouvelle configuration sera bien en relation avec celle obtenue en Flambda puisque toutes les valeurs et l'étiquette sont équivalentes. Le tableau et la structure seront ajoutés au tas $E.W$, produisant ainsi un état $E'$ :

$$
E, (L, v^*, (\mathit{Comp}(\mathit{make_b lock} \ldots), i^*))F^* \rightarrow E', (L, (\texttt{struct } n \ldots, v^*), i^*)F^*
$$

Nos deux configurations sont toujours en relation et nous avons bien déposé exactement une valeur équivalente sur la pile.

**Cas où $t$ est une lecture ou une écriture dans un bloc** Nous ne détaillons pas la preuve. Celle-ci se fait de la même manière que les cas précédents : grâce à la relation sur les configurations la valeur finale Wasm est équivalente à celle produite en Flambda et la relation est maintenue.

**Extension aux autres constructions de Flambda** Bien que cela ne soit détaillé ici, il est possible d'étendre la preuve à d'autres constructions. Dans le cas du `while` cela se fait assez directement en considérant que puisque le programme termine, il y a un nombre fini de fois où l'évaluation de la boucle Flambda va produire un booléen produisant un nouveau tour de boucle. Il est possible de montrer qu'un tour de boucle préserve la configuration en utilisant l'hypothèse d'induction et ce jusqu'au dernier tour de boucle où nous pouvons de nouveau utiliser l'hypothèse d'induction sur l'autre terme pour montrer que la valeur finale est équivalente. Dans le cas des fonctions et des exceptions statiques, cela se fait en étendant la notion d'équivalence sur les relations pour garantir par exemple que le nombres d'environnements dans la configuration correspond à un nombre de fenêtre de fonctions côté Wasm, lesquelles peuvent être entremêlées de fenêtres d'une autre forme (`block` et `loop`).

**Preuve sans condition de terminaison** Dans la preuve précédente, nous nous sommes placés dans le cadre où le programme termine et produit effectivement une valeur. Nous avons défini des sémantiques à petit pas pour les deux langages, qui sont généralement adaptées à prouver des propriétés de correction sur des programmes qui peuvent ne pas terminer. Notre choix de nous limiter à des programmes qui terminent peut donc paraître surprenant et nous en avons bien conscience. Une preuve plus générale est laissée à des travaux futurs.

# 9 Extension de WasmGC : valeurs gelées

Nous présentons ici une extension de Wasm proposée au comité de normalisation Wasm. Le passage de cette proposition en phase 1 a été accepté à l'unanimité. Bien que cette idée soit originale, de nombreux arguments décrits ici sont issus de discussions sur GitHub et lors des réunions du comité.

## 9.1 Motivation

### 9.1.1 Construction de valeurs récursives

Lors du développement de Wasocaml, nous avons observé que le code généré pour l'initialisation des ensembles de fermetures nécessite l'utilisation de structures avec des champs potentiellement nuls et muables. Cependant, ces champs ne sont jamais nuls ni modifiés après leur initialisation.

Prenons un type récursif `$t`. Actuellement, pour construire une valeur `$l` de type `$t` cyclique, ayant la représentation en mémoire représentée dans la Figure 9.1, nous pouvons procéder comme suit :

```
(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)))
```

Considérons maintenant un type similaire `$t'`, où le champ `$f` est immuable et non `null` :

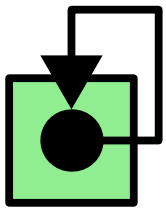


FIGURE 9.1 – Exemple de structure cyclique.

```
(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
  )
)
```

Dans l'état actuel de WasmGC, il est impossible de construire une valeur de type `$t'`. Nous voudrions pouvoir construire une valeur de type `$t'` qui aurait aussi la représentation mémoire présentée dans la Figure9.1. De manière similaire, il arrive que des variables globales doivent être potentiellement `null` pour pouvoir être initialisées, ce qui impose l'utilisation de types muables et potentiellement `null` :

```
(module
  (type $some_ref_type
    ;; …
  )

  ;; here g must be `null` because we don't know yet how to fill it
  (global $g (mut (ref null $some_ref_type)) (ref.null $some_ref_type))

  (func $mk (result (ref $some_ref_type))
    ;; …
  )

  (func $main

    ;; we initialize g
    call $mk
    global.set $g

    ;; …
```

```
    ;; we are forced to use `ref.as_non_null`
    global.get $g
    ref.as_non_null

    ;; …
  )

  (start $main))
```

### 9.1.2 Avantages des valeurs immuables et non `null`

Pouvoir créer des valeurs de ce type présente plusieurs avantages, notamment en termes d'optimisation. En éliminant les tests de nullité, tels que `ref.as_non_null`, et en supprimant les barrières en lecture, nous pourrions améliorer l'efficacité des compilateurs, par exemple lors de la création de tableaux immuables.

En dehors des optimisations, cette extension facilite également les bonnes pratiques de programmation. Elle permet de proposer des API plus propres, de garantir plus facilement la préservation des invariants de code, et de clarifier l'usage des types ou des variables globales.

Plusieurs compilateurs ont manifesté un intérêt pour cette extension, notamment Wasocaml [128], et wasm_of_ocaml [157]. En dehors de l'écosystème OCaml, le compilateur Hoot [160] pourrait également en bénéficier, de par son système d'objets similaire à CLOS [13]. Dans ce système, la classe d'un objet est elle-même un objet, entraînant une récursion jusqu'à une classe racine qui est sa propre classe. Le champ correspondant à la classe d'un objet doit donc être potentiellement `null` pour pouvoir initialiser cette classe racine avec elle-même. De plus, certaines variables globales sont initialisées à la compilation, mais doivent être relocalisées à l'exécution, nécessitant qu'elles soient muables et potentiellement `null`. Cela peut être évité dans les deux cas avec notre extension.

## 9.2 Proposition d'extension

### 9.2.1 Gel de valeurs

Nous proposons d'étendre Wasm pour permettre la création de valeurs récursives immuables comme suit :

```
(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))
  )
)
```

L'idée est de permettre la déclaration d'un type muable ou potentiellement `null`, mais avec la possibilité de le marquer comme `freezable`. Nous définissons ensuite un second type, immuable et non `null`, qui est la version gelée du premier. Une fois la valeur initialisée avec le premier type, elle peut être convertie en sa version gelée grâce à l'instruction `ref.freeze`.

Le type gelé doit respecter des propriétés similaires à celles du sous-typage :

— le type gelé doit avoir un nombre de champs inférieur ou égal à celui du type initial ;
— les annotations `mut` et `null` peuvent être retirées dans la version gelée.
— chaque champ du type gelé doit être un sous-type du champ correspondant dans le type initial, et ce sous-type doit également être gelé si le champ original était d'un type que l'on peut geler (les conversions vers le haut pourraient être autorisés).

À l'exécution, l'instruction `ref.freeze` doit lever un `trap` si une valeur est nulle mais que le type gelé ne l'autorise pas.

Une fois la valeur gelée, la valeur originale ne doit plus être lue. Pour garantir cela, un bit est réservé dans les valeurs de type `freezable`, qui est activé lors de l'exécution de `ref.freeze`. Si ce bit est activé, toute tentative d'accès à la valeur lève un `trap`.

Nous espérons que le coût supplémentaire lié au bit de gel, ainsi qu'aux vérifications d'exécution, sera amorti par le fait que les valeurs non gelées ne sont utilisées que lors de l'initialisation.

### 9.2.2 Phases

Pour permettre également le gel de variables globales, nous introduisons la notion de *phases*. Chaque phase dispose de sa propre fonction `start`, et ces fonctions sont exécutées dans l'ordre. À la fin de chaque phase, les variables globales de la phase suivante sont gelées. La phase par défaut est la phase 0.

Si le gel est impossible (par exemple, parce qu'une valeur est toujours nulle), un `trap` est levé. De même, l'accès à une variable globale non gelée d'une phrase précédente lève un `trap`. Il est interdit d'exporter des variables marquées comme `freezable`.

Une fonction de la phase $n$ ne peut être appelée qu'après la fin de la phase $n-1$, et ne peut faire référence qu'à des valeurs des phases $\leq n$.

Une exemple de programme contenant des `phase` est donné ci-dessous :

```
(module

  ;; some reference type
  (type $t …)

  ;; a mutable and nullable global
  (global $g  (mut (ref null $t)) freezable (ref.null $t))
  ;; its immutable and non-nullable counterpart
  (global $g'      (ref      $t) (freeze $g) (phase 1))
```

```
  ;; the first phase, initializing g
  (start $start0 (phase 0))
  (func $start0 (phase 0)
    call $cycle
    global.set $g
  )
  (func $cycle (phase 0) (result (ref $t))
    …
  )

  ;; g is frozen at the end of phase 0

  ;; the second phase
  (start $start1 (phase 1))
  (func $start1 (phase 1)
    call $f
    drop
  )
  ;; here f can return a non-nullable t without using ref.as_non_null
  (func $f (phase 1) (result (ref $t))
    global.get $g'
  )
)
```

Pour éviter d'avoir plusieurs fonctions `start`, nous pourrions envisager une instruction `call_and_freeze` qui avancerait à la phase suivante.

## 9.3 Questions ouvertes

### 9.3.1 Encodage

L'encodage des phases reste à déterminer, mais il semble possible de le rendre relativement compact.

Tous les moteurs d'exécution Wasm ne disposent pas forcément d'un bit libre pour stocker le bit de gel. Dans certains cas, cela pourrait nécessiter un mot supplémentaire pour les objets `freezable` ou `frozen`, ce qui pourrait poser problème. En effet, cela pourrait impacter négativement la consommation mémoire et ce, dans des proportions rédhibitoire.

### 9.3.2 Coût et complexité de l'opération de gel

L'instruction `ref.freeze` doit parcourir un graphe potentiellement très large d'objets, ce qui en fait une opération dont le coût est non-borné et dépend de l'état global du programme. De plus, il faut gérer les cycles dans ce graphe, qui sont pour l'instant interdits par la sémantique pour les types immuables.

Cette opération doit également fonctionner de manière transitive, en traitant plusieurs types sumultanément, ce qui rend difficile la détermination statique de la profondeur du parcours. En d'autres termes, cette information est déterminée par la relation de sous-typage définie sur les valeurs gelées.

L'instruction `ref.freezable` ne respecte donc pas le principe du nombre fixe d'instructions matérielles, comme c'est le cas pour les autres instructions Wasm. Cependant, le temps combiné de construction et de gel de la valeur pourrait être considéré comme adhérant à une une version amortie de ce principe, ce qui pourrait être suffisant.

### 9.3.3 Bénéfices

Les gains en performance de cette extension restent incertains tant qu'une mise en œuvre complète n'aura pas été réalisée. Par exemple, dans SpiderMonkey, l'accès à des pointeurs nuls est détecté via une approche basée sur les signaux, et il n'y a actuellement pas de barrière en lecture. Le coût d'accès à un objet potentiellement `null` est donc généralement faible. Cependant, nos mesures sur du code généré par Wasocaml ont montré une amélioration de 10% du temps d'exécution sous V8 en désactivant les vérifications d'accès sur des valeurs potentiellement `null`.

### 9.3.4 Multiples fils d'exécution

Une extension en cours de développement pour Wasm permet l'exécution concurrente de plusieurs fils d'exécution. Cette propositon, connue sous le nom de *Threads Proposal* [97], inclura à terme des fonctionnalités liées à WasmGC. Cepdendant, cela soulève des questions importantes concernant la gestion des valeurs gelées dans un environnement avec de multiples fils d'exécution.

Premièrement, un problème clé est de savoir s'il est possible d'autoriser un fil d'exécution à modifier le graphe d'objets pendant que celui-ci est en cours de gel par un autre fil. Cette situation pourrait entraîner des incohérences, et il pourrait être nécessaire d'introduire un verrou en écriture pour les objets `freezable` afin d'assurer que l'opération de gel soit atomique. Une question subséquente est de déterminer si ce verrou doit être global, ou s'il serait envisageable d'en avoir un par objet.

D'un autre côté, si le mécanisme de gel n'est pas compatible avec les objets partagés, il suffirait d'interdire aux objets marqués comme `shared` (comme défini dans le *Thread Proposal*) d'être gelés. Cette approche simplifie le problème en excluant les objets partagés du processus de gel, au prix de certaines limitations fonctionnelles.

## 9.4 Propositions alternatives

### 9.4.1 Attribut `readonly`

Une alternative à la proposition des valeurs gelées serait l'introduction d'un attribut `readonly`, qui serait un super-type de `mut` [119]. Cette approche statique présente plusieurs avantages : elle est plus simple, purement définie au niveau du typage et n'affecte pas l'exécution à proprement parler. Cependant, cette solution ne résout par le problème des coûts liés à la lecture, particulièrement dans le cadre de l'accès aux objets muables. Une analyse des

performances serait nécessaire pour justifier ou non cette approche par rapport à la proposition initiale.

### 9.4.2 Un système de types pour l'initialisation

Plus récemment, nous avons découvert un article décrivant un système de type léger pour l'initialisation des objets [72]. Cette approche est similaire à notre proposition, dans le sens où elle permet d'éviter les coûts liés à la lecture de valeurs potentiellement `null`. Mais contrairement à notre approche, elle ne requiert aucun support durant l'exécution, ce qui la rend similaire à la proposition d'un attribut `readonly`. Elle permettrait donc d'obtenir le meilleur des deux propositions. Une limitation est cependant à prévoir concernant les tableaux, comme les auteurs le décrivent dans leur article. Nous devons encore nous pencher sur le système de types proposé et nous assurer que son intégration à Wasm est possible avec les contraintes du langages (notamment, permettre une validation et une compilation en une seule passe).

### 9.4.3 Modules imbriqués et déclarations ordonnées

Deux autres idées indépendantes de l'extension des valeurs gelées pourraient potentiellement résoudre les problèmes soulevés par la gestion des phases dans notre proposition :

- **Déclaration ordonnées au niveau des modules** : cette approche permettrait de spécifier des dépendances explicites entre les champs des modules Wasm (via des sections répétées dans le format binaire) ; cela simplifierait la gestion des phases en explicitant les relations d'ordre entre les différentes déclarations.
- **Modules imbriqués** : les modules imbriqués permettraient de déclarer des sous-modules au sein d'un module, ce qui permettrait d'ordonner leur ordre d'évaluation.

Nous pourrions simplifier et adapter notre proposition si l'une de ces deux idées venait à être développée.

### 9.4.4 Valeurs récursives

Dans le même esprit que les déclarations de types mutuellement récursifs :

```
(rec
  (type $t1 (struct (field (ref $t2))))
  (type $t2 (struct (field (ref $t1))))
)
```

Nous pourrions envisager l'autorisation de créer des valeurs mutuellement récursives, où les structures seraient initialisées de manière récursive :

```
(rec
  (value $s1 (struct.new $t1 (ref $s2)))
  (value $s2 (struct.new $t2 (ref $s1)))
)
```

Dans ce cas le moteur d'exécution serait responsable de gérer l'initialisation des valeurs là où `null` auraient été nécessaires auparavant. Toutefois, cette solution présente plusieurs problèmes importants :

- **Non-compositionnalité** : la gestion de la préallocation et des mises à jour des valeurs pendant la création récursive impose une modification de la sémantique et de la compilation de toutes les instructions intervenant durant la récursion.
- **Visibilité des opérandes** : pour que cette approche fonctionne, toutes les instructions d'allocation ainsi que leurs opérandes doivent être visibles directement par la construction récursive. Cela impose des restrictions sévères, interdisant notamment les appels de fonctions prenant ou renvoyant des valeurs initialisées dans la récursion. Cette limitation réduit considérablement l'utilité de cette approche, car elle empêche de factoriser la logique d'initialisation dans des fonctions réutilisables.
- **Validation complexe** : la validation des modules Wasm serait également plus complexe, car il faudrait s'assurer que les références créées pendant la récursion ne peuvent pas s'échapper du contexte avant la fin de l'initialisation. Cette tâche est compliquée par le fait que les références sont implicites dans la pile d'opérande. Par exemple, si une instruction `dup` était ajoutée à Wasm, les règles de validation deviendraient beaucoup plus complexes.
- **Complexité cachée** : enfin, cette approche encapsule un nombre arbitraire de pas d'exécution — correspondant aux mises à jour nécessaires, dont le nombre est linéaire en la somme des champs des objets allouées — dans une seule construction monolithique. Dans un langage de bas niveau comme Wasm, il est généralement attendu que ces opérations soient programmées explicitement, ce qui rend cette approche peu intuitive et potentiellement sous-optimale.

# Troisième partie

# Éxécution symbolique

# 10 Exécution concrète de Wasm

Lors du développement de Wasocaml, nous avons parallèlement conçu un interpréteur Wasm, nommé Owi, dans le but de mieux comprendre le langage et de fournir un environnement compatible avec les extensions spécifiques nécessaires à notre compilateur. À l'époque, aucun moteur d'exécution ne prenait en charge l'ensemble de ces extensions. Owi est disponible en accès libre sur un dépôt Git [122].

Owi prend en charge différents formats liés à Wasm, notamment le format textuel (`.wat`), le format binaire (`.wasm`) et le format de scripts utilisé pour la suite de tests de référence (`.wast`).

Plusieurs sous-commandes sont disponibles pour manipuler ces différents fichiers :

- `owi fmt` : un formateur de code Wasm ;
- `owi opt` : un optimiseur de code Wasm ;
- `owi run` : un interpréteur concret Wasm ;
- `owi script` : un interpréteur de scripts Wasm ;
- `owi validate` : un validateur de modules Wasm ;
- `owi wasm2wat` : un convertisseur du format binaire vers le format textuel ;
- `owi wat2wasm` : un convertisseur du format textuel vers le format binaire.

La commande `owi validate` effectue la vérification de type telle que décrite précédemment 3.2.2,5.3. La commande `owi opt` quant à elle n'effectue pour le moment que des optimisations simples (élimination de code mort, propagation de constantes) et ne sera donc pas décrite. Le lecteur curieux pourra trouver une documentation extensive et des exemples d'utilisation de ces commandes sur le dépôt d'Owi.

Owi gère le standard initial de Wasm, ainsi que la quasi-totalité des extensions ajoutées au standard depuis, comme décrit dans le tableau 10.1 (l'icône 🤷 indique que l'extension n'est pas applicable à Owi).

En outre, Owi supporte un certain nombre d'extensions encore en discussion pour devenir des standards, comme indiqué dans le tableau 10.2.

## 10.1 Interface de fonctions étrangère (FFI) sûre

Un module Wasm, par défaut, ne peut pas interagir directement avec son environnement. Pour ce faire, il doit importer des fonctions fournies pas l'hôte, qui, dans le cas d'Owi, est un interpréteur écrit en OCaml. Owi propose un ensemble de fonctions OCaml permettant d'exécuter des scripts de la suite de tests de référence, comme l'affichage de valeurs scalaires.

| 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 | ⏳ |
| JavaScript BigInt to WebAssembly i64 integration | 🤷‍♀️ |

Table 10.1 – Extensions du standard Wasm supportées par Owi.

| Extension | Statut |
|---|---|
| Tail call | ✅ |
| Typed Function References | ✅ |
| Extended Constant Expressions | ✅ |
| Custom Annotation Syntax in the Text Format | 🔜 |
| Garbage collection | 🔜 |

Table 10.2 – Extensions du standard Wasm encore en développement mais déjà supportées par Owi.

Plutôt que de se limiter à un ensemble restreint de fonctions prédéfinies, nous avons développé un mécanisme flexible permettant à l'utilisateur de définir ses propres fonctions en OCaml et d'utiliser Owi comme une bibliothèque pour exécuter des programmes Wasm s'appuyant sur ces fonctions.

Supposons que l'on veuille exposer un module OCaml `Intref` permettant de travailler sur des références vers des entiers et ayant la signature suivante :

```
type t

val fresh : int32 -> unit
val get   : t     -> int32
val set   : t     -> int32 -> unit
val print : t     -> unit
```

Un programme OCaml pourrait utiliser ce module ainsi :

```
let () =
  let r = Intref.fresh 42 in
  Intref.print r;
  Intref.set r 13;
  Intref.print r
```

Or, nous ne souhaitons pas utiliser le module `Intref` depuis du code OCaml, mais depuis du code Wasm. Prenons par exemple une version Wasm du programme précédent, contenue dans un fichier `purewasm.wat` :

```
(module $purewasm

  ;; we import all the functions we expect from the host
  (import "intref" "fresh" (func $fresh (param i32)       (result externref)))
  (import "intref" "get"   (func $get   (param externref) (result i32)))
  (import "intref" "set"   (func $set   (param externref) (param i32)))
  (import "intref" "print" (func $print (param i32)))

  ;; we translate our previous OCaml program using the imported functions
  (func $start (local $r externref)

    ;; let r = Intref.fresh 42
    (local.set $r (call $fresh (i32.const 42)))

    ;; Intref.print r
    (call $print (call $get (local.get $r)))

    ;; Intref.set r 13
    (call $set (local.get $r) (i32.const 13)  )

    ;; Intref.printf r
    (call $print (call $get (local.get $r))))

  (start $start))
```

Nous voulons maintenant utiliser Owi comme une bibliothèque pour exposer ces fonctions à notre module Wasm. Cependant, il est facile pour le programmeur de se tromper, soit en définissant des fonctions OCaml avec un type qui ne correspond pas aux types attendus par le programme Wasm, soit dans les types indiqués lors de l'import des fonctions dans le module Wasm.

Dans la plupart des moteurs Wasm, une erreur dans la déclaration des types des fonctions attendues par Wasm ou dans le langage hôte ne se manifeste souvent qu'à l'exécution, voire pas du tout, conduisant à des comportements erronés. Au contraire, notre approche vise à garantir *par construction* qu'une fonction OCaml donnée respecte le type attendu par une signature Wasm spécifique. Pour cela nous offrons une interface basée sur des GADTs dont un extrait simplifié est donné ci-dessous :

```
open Types

(* The type of values, we simplified it to get only integers and extern
↪  references *)
type _ telt =
  | I32 : int32 telt
  | Externref : 'a Type.Id.t -> 'a telt
```

```
(* The type to represent the result part of a function's signature. Either a
↪  function returns nothing, either it returns one value, either it return
↪  two values. In the real implementation we allow returning more values. *)
type _ rtype =
  | R0 : unit rtype
  | R1 : 'a telt -> 'a rtype
  | R2 : 'a telt * 'b telt -> ('a * 'b) rtype

(* The type to represent the arguments part of a function's signature. Arg is
↪  used when there is one more parameter to add and Res is used when there is
↪  no more parameter. *)
type (_, _) atype =
  | Arg : 'a telt * ('b, 'r) atype -> ('a -> 'b, 'r) atype
  | Res : ('r, 'r) atype

(* The type of functions *)
type _ func_type =
  | Func : ('f, 'r) atype * 'r rtype -> 'f func_type

(* The type of extern functions *)
type extern_func =
  | Extern_func : 'a func_type * 'a -> extern_func
```

À partir de cette interface, nous pouvons définir les fonctions attendues par ce module :

```
open Owi

(* An extern module defined in OCaml that will be linked with our pure Wasm
↪  module *)
let extern_module : Value.Func.extern_func Link.extern_module =

  (* a type with an unique identifier, which will be seen as an `externref` in
  ↪  Wasm *)
  let intref : int32 ref Type.Id.t = Type.Id.make () in

  (* some custom functions *)
  let fresh i = ref i in
  let set r i = r := i in
  let get r = !r in
  let print r = Printf.printf "%ld\n%!" !r in

  (* we need to describe their types *)
  let functions =
    [ ( "print"
      , Value.Func.Extern_func (Func (Arg (Externref intref, Res), R0), print)
      ↪  )
```

```
    ; ( "fresh"
      , Value.Func.Extern_func (Func (Arg (I32, Res), R1 (Externref rint)),
      ↪  fresh) )
    ; ( "set"
      , Value.Func.Extern_func (Func (Arg (Externref intref, Arg (I32, Res)),
      ↪  R0), set) )
    ; ( "get"
      , Value.Func.Extern_func (Func (Arg (Externref intref, Res), R1 I32),
      ↪  get) )
    ]
  in
  { functions }
```

Intéressons-nous maintenant sur les garanties apportées par notre mécanisme. Tout d'abord, nous garantissons lors du typage du programme que les fonctions OCaml ont bien le type attendu. Par exemple, pour la fonction `print`, cela s'effectue ici :

```
Value.Func.Extern_func (Func (Arg (Externref intref, Res), R0), print)
```

Nous savons initialement que `intref : int32 ref Type.Id.t`. Par définition du constructeur :

```
| Externref : 'a Type.Id.t -> 'a telt
```

Nous déduisons que `Externref intref : int32 ref telt`. Par définition, `Res : ('r, 'r) atype`. Par définition du constructeur :

```
| Arg : 'a telt * ('b, 'r) atype -> ('a -> 'b, 'r) atype
```

Nous déduisons que `Arg (Externref intref, Res) : (int32 ref -> 'r, 'r) atype`. Par définition, `R0 : unit rtype`. Par définition du constructeur :

```
| Func : ('f, 'r) atype * 'r rtype -> 'f func_type
```

Nous déduisons que `Func (Arg (Externref intref, Res)) : (int32 ref -> unit) func_type`. Enfin, par définition du constructeur :

```
| Extern_func : 'a func_type * 'a -> extern_func
```

Nous déduisons que `Extern_func (Func (Arg (Externref intref, Res), R0), print) : externref` car `print : int32 ref -> unit`.

Si le type de `print` n'avait pas correspondu au type de `Func (Arg (Externref intref, Res))`, le compilateur aurait rejeté notre définition. Ainsi, nous avons une représentation des types des fonctions dont nous savons qu'elle est correcte. Nous pouvons alors utiliser ce module externe et procéder à l'édition de lien avec notre module Wasm :

```
(* a link state that contains our custom module, available under the name
↪  `intref` *)
```

```
let link_state = Link.extern_module Link.empty_state ~name:"intref"
↪  extern_module

(* the pure wasm module refering to `intref` *)
let pure_wasm_module = Parse.Text.Module.from_file "purewasm.wat"

(* our pure wasm module, linked with `intref` *)
let module_to_run, link_state = Compile.Text.until_link link_state
↪  pure_wasm_module
```

L'édition de lien entre les deux modules s'effectue à la dernière ligne. Durant cette étape, nous vérifions par un simple filtrage de motif que le type déclaré par l'utilisateur au niveau de l'import de la fonction en Wasm (`((func $print (param i32)))`) correspond bien à la représentation donnée en OCaml (`Func (Arg (Externref intref, Res))`). Nous avons alors la garantie que la fonction `print` peut être utilisée de façon sûre depuis Wasm.

Nous pouvons finalement exécuter le module Wasm ainsi :

```
(* let's run it ! it will print the values as defined in the print function *)
let () = Interpret.Concrete.modul link_state module_to_run
```

L'exécution de ce code produit bien :

```
$ ./extern.exe
42
13
```

L'interface actuelle permet à l'utilisateur de définir des fonctions manipulant des scalaires, des références externes, et la mémoire linéaire du module. D'autres constructions seront rendues accessibles ultérieurement. Bien que la mise en œuvre de ce mécanisme puisse sembler complexe, son utilisation par le programmeur est étonnamment simple[1]. En effet, celui-ci interagit avec les types de manière très semblable à la manipulation des types algébriques classiques.

Cette approche, qui consiste à intégrer des preuves de correction directement dans le code, a été introduite initialement dans les travaux de Necula [34]. Notre utilisation des GADTs s'inspire de nombreux travaux antérieurs dans ce domaine [44, 47, 49, 64]. Pour les lecteurs souhaitant approfondir ce sujet, nous recommandons vivement la lecture de *Typed Type Checking and Typed Compilation* [58].

## 10.2 Fuzzing

Ce travail a été réalisé dans le cadre du stage de recherche de Master 2 d'Eric Patrizio, que j'ai encadré pendant six mois.

Le fuzzing est une technique de test intensif qui consistant à injecter un grand nombre de données aléatoires dans un programme. L'objectif de ce travail est d'utiliser un *fuzzer* pour

1. Ce constat reste vrai tant que le programmeur évite les erreurs, car les messages d'erreur générés par le compilateur OCaml peuvent parfois être déroutants en présence de GADTs.

vérifier si Owi interprète correctement les programmes Wasm. Les fuzzers peuvent être classés en trois grandes catégories, selon le niveau de connaissance qu'ils ont du code source du programme cible. Voici un aperçu de ces catégories.

**Fuzzing en boîte noire** Le fuzzing en boîte noire [18] ne requiert aucune connaissance du code source. Un exemple de bibliothèque permettant ce type de fuzzing est QuickCheck pour Haskell [38]. L'utilisateur peut définir des générateurs d'entrées aléatoires en utilisant les combinateurs fournis par la bibliothèque. La génération d'entrées se base uniquement sur les sorties du programme, sans avoir accès à l'exécution interne. Par conséquent, certains points du programme peuvent être atteints avec une très faible probabilité.

**Fuzzing en boîte grise** Le fuzzing en boîte grise est une approche similaire, mais avec un accès partiel aux informations d'exécution. Le programme est compilé de manière à produire un binaire instrumenté. Lors de l'exécution, les différents branchements pris par le programme guident la génération des prochaines entrées, ce qui améliore la couverture du code. Le fuzzer en boîte grise le plus populaire est AFL (American Fuzzy Lop) [84].

**Fuzzing en boîte blanche** Le fuzzing en boîte blanche, quant à lui, utilise une exécution symbolique (ou plus exactement concolique) du programme [63, 5]. Un solveur de contraintes ou bien un solveur SMT est employé pour tester l'atteignabilité des différentes branches du programme et générer des vecteurs de test. Cette approche permet d'obtenir la couverture de code la plus complète.

### 10.2.1 Approche choisie

**Choix de l'outil** Étant donné qu'Owi est écrit en OCaml, seules les approches de fuzzing en boîte noire et en boîte grise sont applicables, via les bibliothèques QCheck[2] et Crowbar[3]. À notre connaissance, il n'existe pas d'interpréteur symbolique pour OCaml qui permettrait de faire du fuzzing en boîte blanche. Nous avons opté pour le fuzzing en boîte grise, qui tend à offrir de meilleurs résultats en termes de couverture. Pour cela, nous utilisons la bibliothèque Crowbar avec un compilateur qui permet l'instrumentation via AFL. De plus, Crowbar permet aussi d'effectuer du fuzzing en boîte noire à la QuickCheck.

**Choix de la méthode de génération** Notre objectif étant de tester l'ensemble d'Owi (validation, optimisation et exécution), générer simplement des chaînes de caractères aléatoires ne permet d'atteindre certaines passes qu'avec une faible probabilité. Pour que les programmes générés passent les étapes de parsing et de validation, nous avons choisi de ne générer que des programmes syntaxiquement valides et correctement typés. Avec Crowbar, cela implique d'utiliser des combinateurs pour définir un générateur de modules.

### 10.2.2 Génération de modules

Générer des programmes Wasm conformes revient à construit un générateur de modules bien typés. Un module est défini par une liste de champs :

2. https://github.com/c-cube/qcheck
3. https://github.com/stedolan/crowbar

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

La fonction `fields` constitue le cœur du générateur de modules. Il est ensuite nécessaire de définir les autres générateurs, comme `memory`, `data`…

### Environnement muable et syntaxe monadique

Une difficulté est que la génération des éléments du module ne peut pas se faire dans n'importe quel ordre. Par exemple, certains types de `data` (celles qui sont *actives*) ne peuvent être déclarés que si une `memory` existe déjà. Crowbar gère ce cas avec les fonctions `Crowbar.dynamic_bind` et `Crowbar.map`. Le code suivant montre l'utilisation de ces fonctions dans la fonction `fields` en supposant que seuls les champs `data` et `memory` sont nécessaires :

```
let fields env =
  Crowbar.dynamic_bind
    (list (memory env))
    (fun memories ->
      Crowbar.map
        [ list (data env) ]
        (fun datas ->
          memories @ data ))
```

La génération des `memories` modifie l'environnement muable, permettant ainsi de générer les `data`s en fonction des mémoires déjà créées. Bien qu'il soit possible de passer à un environnement immuable, notre expérience montre que cela complique le code et entraîne des erreurs de typage plus difficiles à gérer.

Lorsque le nombre de générateurs devient important, comme c'est le cas ici[4], l'imbrication de `dynamic_bind` rend la lecture et la maintenance du code plus complexe. Pour pallier ce problème, nous avons défini un module `Syntax` contenant des opérateurs monadiques `let*`, `let+` et `and+` :

```
module Syntax = struct
  let ( let* ) = Crowbar.dynamic_bind
  let ( let+ ) gen map_fn = Crowbar.map [ gen ] map_fn
  let ( and+ ) = Crowbar.pair
end
```

4. Notre fuzzer contient 83 appels à `Crowbar.map` et 43 appels à `Crowbar.dynamic_bind`.

Ce module, proposé et intégré à Crowbar, facilite grandement l'écriture de générateurs, rendant l'utilisation de cette bibliothèque plus accessible. En effet, une fois cette syntaxe ajoutée, la fonction `fields` ressemble à cela :

```
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
```

### Générateur d'expressions

La difficulté principale pour générer des programmes Wasm valides réside dans la création de fonctions bien typées. Une fonction est une liste d'instructions, où chaque instruction opère sur une pile dont le type des éléments est connu à chaque point du programme.

```
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 to match
    ↪  the expected block_type *)
  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
```

La génération est effectuée de manière récursive. L'objectif est de produire une expression de type `~block_type` avec une pile de type `~stack`. Pour éviter de générer des fonctions trop longues, l'environnement est initialement doté d'un *carburant* (*fuel*). À chaque appel récursif, le carburant est décrémenté. Lorsque celui-ci est épuisé, nous procédons à ce qui pourrait s'apparenter à une vidange : nous insérons des instructions `drop` pour vider la pile, puis nous ajoutons des constantes pour correspondre au type `~block_type`. Si du carburant reste, nous choisissons une instruction parmi celles compatibles avec le type courant de la pile, et nous

continuons récursivement.

Lorsque le carburant est épuisé, une alternative à la vidange complète de la pile suivie de l'insertion de constantes pourrait être envisagée. Une approche plus sophistiquée consisterait à réutiliser certains éléments déjà présents dans la pile ou à employer des instructions qui consommeraient ces éléments pour générer des valeurs conformes au type attendu. Cette fonctionnalité est envisagée pour une future version.

### 10.2.3 Fuzzing différentiel

Une fois un module généré, il est utilisé pour tester l'interpréteur. Puisque nous générons du code bien typé, le module doit passer la validation. Si ce n'est pas le cas, cela indique un bogue dans la génération ou dans la validation.

Après validation, le module est exécuté. Si l'exécution échoue (via un `trap`), ou s'effectue sans erreur, il est difficile de déterminer si le résultat est correct. Pour cela, nous utilisons le *fuzzing différentiel* [36], où le module est exécuté par plusieurs interpréteurs pour comparer les résultats. Le programme de référence est appelé *l'Oracle*.

Nous utilisons cette méthode pour comparer Owi avec l'interpréteur de référence de Wasm. Une fois le module généré, il est écrit dans un fichier, puis parsé par l'interpréteur de référence. Le code de sortie indique si le programme a échoué et comment. Cela permet de vérifier que le comportement est identique dans les deux interpréteurs.

Nous utilisons également cette méthode pour comparer Owi avec une version optimisée de lui-même, en utilisant le module `Optimize`, pour tester que les optimisations préservent la sémantique.

Dans le cas où les deux exécutions ne produisent pas d'erreur, nous ne comparons pas directement les résultats, car Wasm ne renvoie pas de résultat au niveau du module. Cependant, les valeurs sont testées indirectement, par exemple en vérifiant que certaines branches provoquent un `trap` ou non.

Il est envisageable des comparer directement les valeurs produites, par exemple en générant des fonctions d'affichage et en comparant les sorties produites, en vérifiant l'état de la mémoire à la fin de l'exécution, ou en testant individuellement des fonctions qui laissent des valeurs sur la pile en sortie.

Une autre difficulté réside dans la gestion des cas de non-terminaison. En effet, il est courant de générer des boucles ou des fonctions (mutuellement) récursives qui ne se terminent pas ou bien dont l'exécution est simplement trop longue pour permettre de tester un grand nombre de modules rapidement. Pour y remédier, un timeout a été mis en place. Si les deux exécutions mènent à un timeout, le test est ignoré. Si un seul des deux cas produit un timeout, une option permet de choisir s'il faut signaler ou non l'erreur. En effet, le recours à un fichier intermédiaire pour l'interpréteur de référence ralentit considérablement son exécution, ce qui peut entraîner de nombreuses divergences si le timeout est trop court. Pour cette raison, nous envisageons d'utiliser l'interpréteur de référence comme une bibliothèque et d'écrire une fonction de conversion entre notre type de module et celui de la bibliothèque. En pratique, avec un timeout ajusté après quelques expérimentations, nous constatons environ 10% de timeout. Ce pourcentage semble offrir un bon compromis entre un nombre excessif de tests ignorés et des tests trop longs qui limitent le nombre de modules testables sur une période de temps donnée.

### 10.2.4 Résultats

Le fuzzing d'Owi a permis de détecter plusieurs bogues. La majorité d'entre eux concernent le pretty printer, qui était jusqu'à présent utilisé principalement à des fins de débogage. L'utilisation de l'interpréteur de référence lors du fuzzing différentiel, en passant par un fichier intermédiaire, a révélé de nombreuses inexactitudes et des parties manquantes.

D'autres bogues ont été découverts dans le typechecker, notamment dans les parties subtiles liées à la gestion des instructions dites *stack-polymorphic* en Wasm. Fait intéressant, même en générant des programmes correctement typés, ces bogues ont été révélés, tant dans les cas où le typechecker rejetait des programmes valides que dans ceux où il acceptait des programmes invalides.

Un autre bogue a été trouvé dans la passe d'optimisation, en particulier lors de l'élimination des variables locales inutilisées. La suppression d'une variable représentée par un indice à ce stade de l'interpréteur, nécessite de modifier l'indice des autres variables, ce qui n'était pas correctement géré dans certains cas.

Plusieurs bogues ont été découverts dans l'exécution proprement dite, souvent liés à des vérifications d'accès hors des bornes de la mémoire et des tables, à une mauvaise gestion des *overflow*, ou à des erreurs d'interprétation entre entiers signés et non signés.

Par la suite, une réécriture majeure d'Owi a été entreprise pour permettre au choix une exécution concrète ou symbolique des programmes Wasm, comme décrit dans la section 11.2. Le fuzzer a rapidement détecté des erreurs introduites lors de ce refactoring.

### 10.2.5 Perspectives

**Mesure de la couverture** La suite de tests d'Owi utilise la bibliothèque `bisect_ppx` pour mesurer la couvertude du code par les tests. Cependant, il semble que le mécanisme d'instrumentation utilisé par cette bibliothèque soit incompatible avec celui produit par le compilateur modifié pour AFL. En effet, l'outil signale une couverture de 0% lorsque les deux sont combinés. Une solution envisagée serait de générer des modules et de les sauvegarder dans des fichiers. Nous pourrions ainsi les utiliser avec un compilateur non instrumenté pour obtenir des mesures de couverture. Cela permettrait de vérifier que certaines instructions ne sont pas oubliées et sont bien générées, ou d'identifier si certaines instructions sont plus présentes que d'autres, afin d'équilibrer la génération.

**Introduction de bogues** En utilisant une bibliothèque comme `mutaml`, nous pourrions introduire volontairement des bogues dans l'interpréteur et mesurer l'efficacité du fuzzer pour les détecter.

**Génération de modules invalides** Pour tester d'autres aspects de l'interpréteur, il serait pertinent de générer des modules invalides et de vérifier qu'ils sont bien rejetés.

**Fuzzing intensif** Jusqu'à présent, le fuzzing n'a été lancé que pendant quelques minutes avec un carburant minimal. Il est prévu de l'exécuter sur une période beaucoup plus longue, avec des valeurs de carburant plus élevées, afin d'essayer de découvrir davantage de bogues.

**Amélioration de la génération de module** Certaines limitations existent dans la génération actuelle : la taille de la mémoire est limitée pour éviter une consommation excessive de ressources, ce qui ralentit considérablement le fuzzing ; et les expressions générées de manière rudimentaire une fois le carburant épuisé (en insérant des `drop` pour vider la pile avant d'y placer des constantes). Corriger ces limitations pourrait permettre de générer une plus grande variété de programmes.

**Comparaison fine avec d'autres interpréteurs** Une comparaison plus détaillée que la simple détection d'erreurs lors de l'exécution, ainsi que l'utilisation d'autres interpréteurs, pourrait augmenter les chances de découvrir des bogues, que ce soit dans Owi ou dans d'autres interpréteurs.

**Support des futures fonctionnalités d'Owi** De nombreuses fonctions sont régulièrement ajoutées à Owi, notamment les différentes extensions de Wasm (la prochaine devrait être WasmGC). Nous envisageons d'étendre le fuzzer pour tester ce nouveau code.

### 10.2.6 Travaux connexes

**Alt-ergo-fuzz** Un outil partageant de nombreuses similitudes avec le fuzzing d'Owi est Alt-ergo-fuzz [134]. Nous y retrouvons l'utilisation de Crowbar, l'emploi d'autres solveurs SMT comme oracles, la génération de code bien typé pour arriver à atteindre les parties intéressantes, ainsi que la gestion des timeouts… Les principales différences résident dans la méthode utilisée pour la génération du code bien typé (Owi se base sur le type de la pile), et dans l'utilisation directe des combinateurs de Crowbar plutôt que des opérateurs monadiques, ce qui rend le code plus difficile à lire et à maintenir.

**Stack-driven program generation of WebAssembly** Dans l'article *Stack-driven program generation of WebAssembly* [116], les auteurs réalisent également du fuzzing d'interpréteurs Wasm en utilisant des techniques similaires aux nôtres. La différence principale réside dans le type de fuzzing : les auteurs adoptent une approche en boîte noire, qui atteint certains chemins avec une probabilité plus faible. De plus, les auteurs affirment générer des expressions de façon *backward*, même si à la lecture de l'article, il semble qu'il s'agisse en fait d'une génération *forward* comme dans notre approche (*i.e.* nous générons d'abord la première instruction, puis la deuxième…). Enfin, leur approche peut échouer à générer une expression. Quand cela se produit, une nouvelle tentative de génération est effectuée et ce jusqu'à obtenir un succès. En comparaison, notre approche réussit toujours à générer une expression et ne peut pas échouer.

# 11 Exécution symbolique pour Wasm

Ce travail a fait l'objet d'un article qui sera publié dans le journal *The Art, Science, and Engineering of Programming*. Une pré-publication est disponible en ligne [161]. Il a également été présentés à plusieurs occasions, notamment :

- au *ICFP 2023, OCaml Users and Developers Workshop* ;
- au *Wasm Research Day 2023* ;
- au *meetup OUPS (OCaml Users in PariS)* d'avril 2024.

## 11.1 Exécution symbolique

L'exécution symbolique est une technique d'analyse de programmes utilisée pour explorer tous les chemins d'exécution possibles d'un programme jusqu'à une certaine limite [5]. Au lieu d'utiliser des entrées concrètes, l'exécution symbolique fonctionne avec des entrées symboliques, qui représentent plusieurs valeurs possibles. Chaque fois que le moteur d'exécution symbolique rencontre une branche conditionnelle dont la condition dépend de la valeur d'un ou plusieurs symboles, il scissionne l'exécution, explorant les deux branches possibles. Pour chaque chemin, le moteur construit une formule logique appelée condition de chemin, qui représente les contraintes sur les entrées nécessaires pour atteindre ce chemin.

Lorsqu'un branchement est rencontrée, la condition de chemin est mise à jour avec la garde correspondante pour la branche "then" ou avec la négation de la garde pour la branche "else". Pour vérifier l'atteignabilité de ces chemins et valider les assertions dans le programme, les moteurs d'exécution symbolique s'appuient sur un solveur SMT (Satisfiability Modulo Theories) tel que Z3 [62], Colibri2 [169], Bitwuzla [155], Alt-Ergo [100] ou CVC5 [129].

Un chemin d'exécution est considéré comme atteignable si le solveur SMT peut trouver au moins un ensemble concret d'entrées satisfaisant la condition de chemin. De plus, une assertion à un point particulier du programme est valide si elle est impliquée par la condition de chemin à ce point.

### 11.1.1 Exemple d'exécution symbolique

Pour illustrer l'exécution symbolique en pratique, considérons la fonction `$test_swap` dans le programme Wasm présenté dans le code 11.1. Dans un des chemins, la fonction se termine par une instruction `unreachable`, qui ne devrait jamais être exécutée, pour n'importe

```
(module
  ;; This function does nothing from the point of view of its caller but
  ↪  internally swaps x and y if needed so that x <= y and assert that the
  ↪  swap happened correctly
  (func $test_swap (param $x i32) (param $y i32)
    ;; if x > y
    (if (i32.gt_s (local.get $x) (local.get $y))
    (then
      ;; Swap x and y using integer arithmetic
      ;; and not a temporary variable
      ;; x <- x + y
      (local.set $x
        (i32.add (local.get $x) (local.get $y)))
      ;; y <- x - y
      (local.set $y
        (i32.sub (local.get $x) (local.get $y)))
      ;; x <- x - y
      (local.set $x
        (i32.sub (local.get $x) (local.get $y)))
    ))

    ;; Check now that x <= y by raising an exception if x - y > 0
    (if (i32.gt_s
      (i32.sub (local.get $x) (local.get $y))
      (i32.const 0))
    (then
        ;; raise en exception (ie. "trap")
        unreachable
    ))
  )
)
```

Code source 11.1 : Fonction Wasm `$swap`, il s'agit d'une traduction d'un exemple originellement écrit en C et provenant de Khurshid [45].

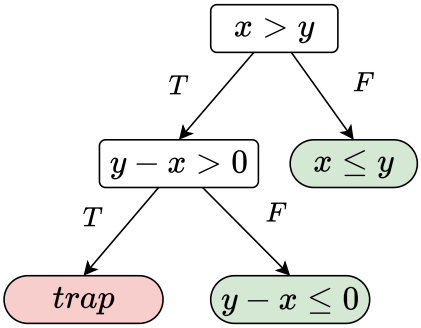



FIGURE 11.1 – Arbre d'exécution de la fonction `$test_swap`. L'exécution symbolique explore exhaustivement tous ces chemins d'exécution. Lorsqu'on atteint le nœud `trap`, le SMT fournit les valeurs menant à ce nœud.

quelle entrée, et déclenche une exception si elle l'est. L'exécution symbolique est utilisée pour vérifier cela en explorant exhaustivement tous les chemins d'exécution atteignables, comme le montre l'arbre d'exécution dans la figure 11.1. Dans cette figure, les feuilles vertes représentent des résultats d'exécution valides, tandis que les feuilles rouges indiquent des chemins qui déclenchent l'instruction `unreachable`.

Considérons les entrées $x = 8388481$ et $y = -2147483648$, qui provoquent un dépassement de capacité d'entier 32 bits lors de la soustraction $x - y$ dans la deuxième branche (c'est-à-dire $-2147483648 - 8388481 \bmod 2^{32} = 2139095167$), conduisant au `trap unreachable`. Ci-dessous, nous expliquons comment cette entrée peut être découverte.

Puisqu'il y a trois chemins d'exécution à travers la fonction `$test_swap`, chacun doit être exécuté symboliquement. Pour cet exemple, nous supposons une stratégie d'exploration en largeur. Initialement, l'exécution symbolique assigne des valeurs symboliques aux variables locales $x$ et $y$. Lors de l'exécution de la première branche conditionnelle, l'exécution se scissionne en deux chemins : l'un où $x > y$, et l'autre où $x \leq y$. Les conditions de chemin sont mises à jour en conséquence :

$$\text{PC}_T \equiv x > y \quad \text{and} \quad \text{PC}_F \equiv x \leq y$$

L'exécution se poursuit alors le long du chemin décrit par $\text{PC}_T$. Dans ce chemin, les valeurs de $x$ et $y$ sont échangées, et la deuxième branche conditionnelle (`$x-y > 0`) est exécutée, générant deux nouvelles conditions de chemin :

$$x' \mapsto x + y \quad y \mapsto x' - y \quad x'' \mapsto x' - y$$

$$\text{PC}_{TT} \equiv x > y \wedge y'' - x'' > 0 \quad \text{and} \quad \text{PC}_{TF} \equiv x > y \wedge y'' - x'' \leq 0$$

Les deux conditions de chemin sont atteignables, menant à une autre scission de l'exécution. Finalement, $\text{PC}_{TT}$ atteint l'instruction `unreachable`. Un solveur SMT peut être interrogé pour trouver des entrées concrètes pour $\text{PC}_{TT}$, produisant :

$$\texttt{\$x} \mapsto 8388481 \quad \texttt{\$y} \mapsto -2147483648$$

En utilisant Owi, nous pouvons simplement obtenir ce modèle de la façon suivante :

```
$ owi sym swap.wat
Trap: unreachable
Model:
  (model
    (symbol_0 (i32 8388481))
    (symbol_1 (i32 -2147483648)))
Reached problem!
```

Cela démontre la puissance de l'exécution symbolique dans l'identification de cas limites, comme un dépassement de capacité d'entier, qui peuvent provoquer des comportements inattendus. Bien que d'autres chemins, tels que $\text{PC}_{TF}$, restent à explorer, la découverte d'un chemin erroné démontre déjà l'utilité de l'analyse.

## 11.2 Généralisation d'un interpréteur concret en un interpréteur monadique pour permettre l'exécution symbolique

Dans cette section, nous présentons comment nous avons transformé Owi, historiquement un interpréteur de référence robuste pour Wasm, en un interpréteur modulaire et général capable d'effectuer à la fois des interprétations concrètes et symboliques. Ce choix est motivé par le fait que nous ne souhaitions pas perdre cette capacité à exécuter du code concret et à ne plus avoir qu'un interpréteur symbolique. En effet, cela permet notamment de réutiliser la suite de tests existante (ce qui facilite la mise en œuvre des nouvelles extensions), de fuzzer l'interpréteur pour y trouver des bogues et même de simplifier la mise en œuvre de certaines optimisations (si une opération sur des constantes peut potentiellement mener à un `trap`, nous nous contentons d'appeler l'interpréteur concret ; si nous arrivons à un `trap`, nous n'effectuons pas la simplification).

Cette section est organisée comme suit : tout d'abord, nous donnons un aperçu de la mise en œuvre de l'interpréteur concret (§11.2.1). Ensuite, nous décrivons comment paramétrer l'interpréteur pour différentes versions de ses valeurs de base (§11.2.2). Enfin, nous montrons comment paramétrer l'interpréteur en fonction de sa stratégie d'évaluation, laquelle sera représentée sous forme de monade, ce qui nous permet de le transformer en un moteur complet d'exécution symbolique (§11.2.3).

### 11.2.1 L'interpréteur concret

Le code de l'interpréteur concret est basé la sémantique de Wasm. Il contient quelques optimisations et ne met donc pas en œuvre celle-ci de manière mécanique[1], mais produit des résultat équivalents. Pour des raisons de simplicité, nous nous concentrons dans cette thèse sur une syntaxe Wasm réduite qui ne comprend que des types entiers.

Elle est représentée par les types de données algébriques OCaml suivants :

1. Une transcription mécanique est disponible dans l'interpréteur de référence Wasm [98].

```
type value =
  | I32 of int32
  | I64 of int64
```

Dans cette représentation, les valeurs de type `value` représentent un entier Wasm de 32 ou 64 bits. Une valeur OCaml représentant l'entier Wasm de 32 bits 42 est écrite `I32 42l`, où le `l` indique qu'il s'agit d'un entier de 32 bits et non des entiers utilisés par défaut en OCaml, qui peuvent être de 63 ou 31 bits.

Les instructions sont également définies comme un type de données OCaml, illustré par le sous-ensemble suivant :

```
type binop =
  | Add
  | Gt

type instr =
  | Binop of binop
  | If of instr list * instr list
```

Les instructions peuvent être :

— soit une opération binaire `Binop op`, où `op` peut être `Add` ou `Gt` ;

— soit une conditionnelle `If (if_true, if_false)`, où `if_true` (resp. `if_false`) est la liste des instructions à exécuter si la condition au sommet de la pile est vraie (resp. fausse).

Nous mettons en œuvre un interpréteur pour cette syntaxe en définissant un évaluateur pour les blocs Wasm (listes d'instructions). L'évaluateur est une fonction ayant la signature suivante :

```
val eval : instr list -> value list -> value list
```

Elle prend une `instr list` (représentant un bloc d'instructions) et une `value list` (représentant la pile de valeurs de Wasm) comme arguments et renvoie une `value list` (la pile modifiée après l'évaluation).

Cet évaluateur d'expression est construit sur un évaluateur d'instruction, qui met également à jour la pile en conséquence. Il a la signature suivante et sa mise en œuvre est montrée dans le Code Source 11.2.

```
val eval_instr : instr -> value list -> value list
```

La fonction `eval_instr` effectue un filtrage de motifs sur `(i, stack)`, où `i` est l'instruction à évaluer et `stack` est la pile de valeurs. Les cas clés sont les suivants :

1. `Binop Add` : Ce cas correspond à une pile avec deux entiers 32 bits au sommet, `I32 x` et `I32 y`. Le résultat de l'addition est calculé à l'aide du module `Int32` d'OCaml, qui fournit des opérations sur les entiers 32 bits, comme `Int32.add` pour l'addition. Une nouvelle pile est ensuite renvoyée avec le résultat de l'addition au début, suivi du reste.
2. `Binop Gt` : De manière similaire au cas précédent, deux entiers `I32 x` et `I32 y` sont dépilés. Nous effectuons une comparaison "supérieur à" sur eux, et le résultat est empilé comme une valeur de vérité (1 pour vrai, 0 pour faux).

```ocaml
let select = function
  | I32 x -> x <> 0l
  | _ -> false

let eval_instr i stack =
  match (i, stack) with
  | Binop Add, I32 x :: I32 y :: stack ->
    I32 (Int32.add x y) :: stack
  | Binop Gt, I32 x :: I32 y :: stack ->
    I32 (Int32.gt x y) :: stack
  | If (if_true, if_false), cond :: stack ->
    let cond = select cond in
    if cond then eval if_true stack
            else eval if_false stack
```

Code source 11.2 : Évaluateur pour les instructions Wasm simplifiées. Nous omettons les cas qui entraîneraient des erreurs de type Wasm (par exemple, une pile vide).

3. `If (if_true, if_false)` avec une valeur `cond` au sommet de la pile. Si `cond` est vraie (c'est-à-dire, `cond` est `I32 x` où $x \neq 0$), le bloc `if_true` est exécuté ; sinon, le bloc `if_false` est exécuté.

### 11.2.2 L'interpréteur paramétrique

Pour généraliser l'interpréteur concret, nous introduisons une version paramétrique en abstrayant sur les valeurs. Cela se fait à l'aide d'un *foncteur* OCaml, qui permet de paramétrer l'interpréteur par un module représentant les différents types de valeurs. En OCaml, un module est une collection de valeurs ou de fonctions regroupées sous un espace de noms. Un foncteur est une fonction au niveau des modules. Le système de modules d'OCaml est décrit dans un article de Leroy [39].

La signature `Value_intf`, montrée dans la figure 11.3, définie les opérations requises pour `Int32` et `Int64`. Elle indique qu'un module `Int32` doit spécifier les quatre éléments suivants :

1. `type t` : Le type principal du module, par exemple `int32` pour l'exécution concrète.
2. `val add` : Une fonction pour effectuer la somme de deux éléments.
3. `val gt` : Une fontion pour effectuer une comparaison "plus grand que" entre deux éléments.
4. `val eqz` : Une fonction pour tester si une valeur vaut zéro.

Elle indique également qu'un module `Int64` doit fournir un type et des fonctions équivalentes pour des entiers 64-bits.

L'extrait de code suivant montre le foncteur `Interpreter`. Il prend comme paramètre un module `Value` qui doit avoir la signature `Value_intf`. L'interpréteur utilise ensuite les opérations fournies par le module `Value`. Cela nous permet de remplacer les opérations concrètes `Int32` et `Int64` par des contreparties abstraites. Nous utilisons la construction `open Value` au

```
module type Value_intf = sig
  module Int32 : sig
    type t
    val add : t -> t -> t
    (* greater than *)
    val gt : t -> t -> t
    (* is equal to zero *)
    val eqz : t -> t
  end
  module Int64 : sig (* Same as Int32 *) end
end
```

Code source 11.3 : Signature `Value_intf` servant à paramétrer le foncteur `Interpreter`.

début du module afin de remplacer les valeurs anciennement accessibles au sein de celui-ci par leur nouvelle version.

```
module Interpreter (Value : Value_intf) = struct

  (* Makes items in the namespace of the Value module directly available here.
  ↪  *)
  open Value

  type value =
  | I32 of Int32.t
  | I64 of Int64.t

  let select = function
  | I32 x -> not (Int32.eqz x)
  | _ -> false

  let eval_instr i stack =
    match i, stack with
    | Binop Add, I32 x :: I32 y :: stack ->
      I32 (Int32.add x y) :: stack
    | Binop Gt, I32 x :: I32 y :: stack ->
      I32 (Int32.gt x y) :: stack
    | If (if_true, if_false), cond :: stack ->
      let cond = select cond in
      if cond then eval_expr if_top stack
              else eval_expr if_bot stack
end
```

Grâce à cette paramétrisation, nous pouvons facilement mettre en œuvre des variations de notre interpréteur dans lesquelles les opérations de base seraient différentes. Nous pourrions

par exemple émettre une erreur lors d'un débordement de capacité ou utiliser des fonctions qui collectent des statistiques sur leur usage durant l'exécution. Dans le cas concret, le module `Values.Int32` sera le même que précédemment. Dans le cas symbolique, il sera un module qui opère non plus sur des valeurs de type `int32` mais sur des *expressions* similaires à celles dont est composée la condition de chemin.

### 11.2.3 L'interpréteur monadique

Cependant, pour effectuer une exécution symbolique, il ne suffit pas d'abstraire les types de valeurs ; il faut également abstraire le mode d'exécution lui-même. Pour ce faire, nous devons transformer l'interpréteur en une forme monadique. Les monades, en programmation fonctionnelle, représentent des calculs, et les utiliser nous permet de supporter de manière transparente différent modes d'exécution, y compris l'exécution symbolique [16, 19, 30].

L'interpréteur monadique, présenté dans le Code Source 11.4, est paramétré par un module ayant l'interface monadique suivante :

```
module type Choice_intf = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
  val select : Value.Int32.t -> bool t
end
```

Les fonctions principales de cette signature sont :

— `return : 'a -> 'a t`, qui est la fonction de retour monadique utilisée pour injecter une valeur dans la monade, elle prend une valeur "nue" et la transforme dans la valeur monadique correspondant "la plus simple" comme illustré dans la figure 11.2 ;

— `bind : 'a t -> ('a -> 'b t) -> 'b t`, qui est la fonction de liaison monadique utilisée pour chaîner des calculs ;

— `val select : Value.Int32.t -> bool t`, qui prend un `Int32.t` abstrait et renvoie une valeur monadique contenant un booléen. Comme précédemment, la valeur renvoyée indique si l'entier passé est différent de zéro. Le fait que cette valeur soit monadique signifie qu'elle représente un calcul, et comme elle encapsule une valeur booléenne, cela signifie que ce calcul s'évaluera en un booléen. Le fait que `select` renvoie une valeur monadique témoigne du fait que lorsqu'on atteint un point de branchement et qu'il faut décider de la valeur de vérité d'une expression composée de symboles, l'exécution symbolique peut décider quels cas explorer, et dans quel ordre.

L'évaluateur monadique diffère de l'évaluateur paramétrique sur deux points clés :

1. **Liaison monadique** : l'opérateur personnalisé `let*` encapsule la fonction monadique `bind`, permettant un chaînage concis des calculs. Cela est similaire à la *do notation* en Haskell ou à la notation *for* en Scala.
2. **Évaluation conditionnelle** : la condition est évaluée dans un contexte monadique, supportant ainsi l'exécution symbolique en déportant le mode d'évaluation à la fonction `select`, qui sélectionnera la branche à exécuter. Cette fonction renvoie un calcul monadique de type `bool Choice.t`. En utilisant `let*`, nous lions ce résultat à une variable que nous utilisons ensuite pour choisir entre l'exécution de `if_true` ou `if_false`.

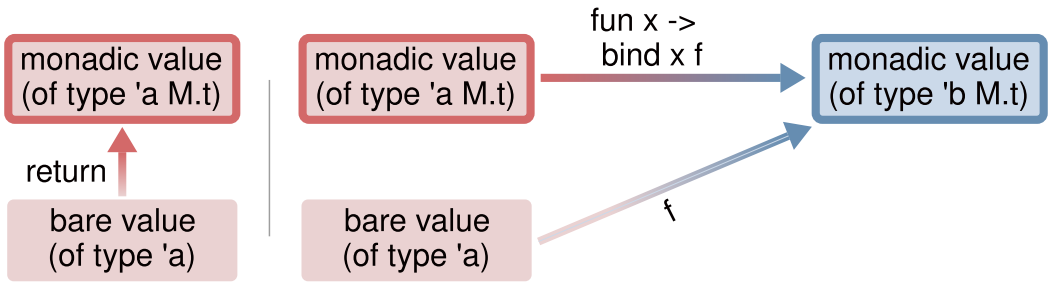



FIGURE 11.2 – Une représentation graphique de deux opérations de la monade, `bind` et `return`. Les valeurs monadiques sont représentées avec une bordure, symbolisant la façon dont la monade "enrobe" les valeurs.

```
module Interpreter (Value : Value_intf) (Choice : Choice_intf) = struct

  open Value
  (* This is a syntax trick that allows to write: `let* x = e1 in e2` wich
  ↪  will be desugared into: `Choice.bind e1 (fun x -> e2)` *)

  let ( let* ) v f = Choice.bind v f

  let rec eval_instr i stack =
    match i, stack with
    | Binop Add, I32 x :: I32 y :: stack ->
      Choice.return (I32 (Int32.add x y) :: stack)
    | Binop Gt, I32 x :: I32 y :: stack ->
      Choice.return (I32 (Int32.gt x y) :: stack)
    | If (if_true, if_false), cond :: stack ->
      let* cond = Choice.select cond in
      if cond then eval if_true stack
              else eval if_false stack
end
```

CODE SOURCE 11.4 : Évaluateur monadique pour la syntaxe simplifiée de Wasm.

Dans le cas concret, la mise en œuvre de la monade `Choice` sera la monade identité, où les calculs sont simplement des valeurs :

```
module Concrete_choice : Choice_intf = struct

  type 'a t = 'a

  let return x = x [@@inline]
  let bind x f = f x [@@inline]
  let select = function
    | I32 x -> x <> 0l
    | _ -> false
    [@@inline]

end
```

Ici, le code se comporte exactement de la même manière qu'il le ferait sans aucune abstraction, et il n'y a aucun coût d'exécution grâce aux annotations `[@@inline]` et à l'optimiseur Flambda [2].

Pour l'exécution symbolique, la monade implique une complexité supplémentaire, car elle doit gérer le raisonnement symbolique, en combinant des fonctionnalités issues à la fois des monades de continuation et d'état. La mise en œuvre détaillée de la monade de choix symbolique sera abordée dans la section suivante.

En résumé, la fonctorisation monadique permet de partager la majeure partie du code entre les interpréteurs concrets et symboliques, offrant ainsi des modes d'exécution flexibles avec un surcoût minimal. Nous pensons que cette approche offre un moyen pratique de dériver un interpréteur symbolique à partir d'un interpréteur concret dans d'autres contextes.

## 11.3 La monade de choix

Comme détaillé dans la section 11.1, l'interpréteur symbolique doit fréquemment faire face à des choix lors de la présence de branchement, comme déterminer si la condition d'une construction *if-then-else* est vraie ou fausse. Chaque choix correspond à une trace d'exécution différente, et ces traces forment un arbre préfixe où chaque nœud représente une décision. Les bogues, tels que les `trap` Wasm ou les violations d'assertion, se trouvent dans les feuilles de cet arbre. Bien que prouver l'absence totale de bogues nécessiterait d'explorer l'ensemble de l'arbre — une tâche généralement irréalisable —, une exploration efficace des branches clés peut aider à identifier rapidement les bogues.

Pour y parvenir, nous pouvons viser deux objectifs : (a) maximiser le taux d'exploration, c'est-à-dire combien de traces sont évaluées par unité de temps, et (b) utiliser des heuristiques pour explorer en priorité les branches les plus susceptibles de contenir des bogues. Bien que notre mise en œuvre actuelle supporte les aspects fondamentaux nécessaires à l'ajout de

2. Les annotations `[@@inline]` sont des indices pour le compilateur indiquant que cette fonction doit être *inlinée*. L'inlining de ces petites fonctions garantit que le compilateur pourra les optimiser davantage, faisant de cette monade d'identité une abstraction *zero cost*.

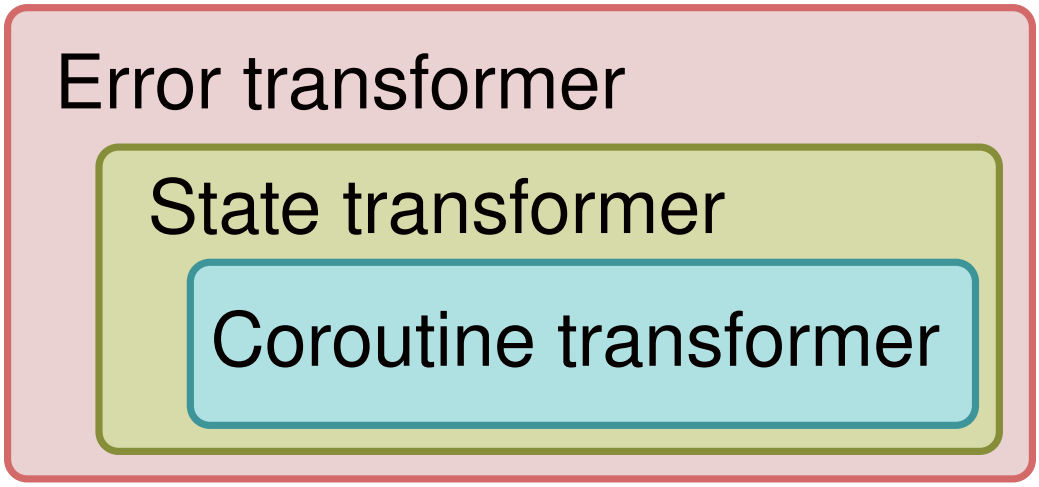



FIGURE 11.3 – Notre monade de choix est composée de trois transformateurs empilés.

priorités, la définition des heuristiques elles-mêmes est laissée à des travaux futures. Dans cette section, nous expliquons comment notre monade de choix assure une exploration efficace des branches en tirant parti de l'exécution parallèle.

Nous décrivons d'abord la structure de la monade de choix et son rôle dans l'exécution symbolique (§11.3.1). Ensuite, nous discutons de notre modèle de mémoire paresseux, qui atténue les problèmes de surconsommation de mémoire qui surgissent lorsque l'on se déploie sur plusieurs cœurs de CPU (§11.4.1).

### 11.3.1 Une monade de choix multi-cœur

#### Vue d'ensemble

Notre monade de choix symbolique et parallèle expose la même interface que celle décrite dans la section précédente, mais intègre plusieurs fonctionnalités clés, comme illustré dans la figure 11.3 :

- Une monade de coroutine coopérative scissionnable, où l'exécution peut rendre la main à son parent et se scissionner elle-même. En se scissionnant, la coroutine duplique l'exécution et fournit la notion de choix. De plus, ces coroutines ont accès à un stockage local au *worker* (WLS), une valeur accessible uniquement au *worker* exécutant la coroutine, ce qui permet d'avoir des valeurs "globales" de types qui sinon ne seraient pas sûrs dans un contexte de multiples fils d'exécution.
- Une monade d'état, qui gère l'état interne de Wasm de l'interpréteur et l'état recquis par l'exécution symbolique, en particulier la condition de chemin.
- Une monade d'erreur qui gère les `trap` des programmes Wasm et les violations d'assertions symboliques. Ces erreurs se propagent à travers la structure monadique.

Les monades d'état et d'erreur ressemblent étroitement aux constructions existantes dans la littérature [25], et nous ne les expliquerons pas davantage ici. La monade de coroutine a quatre constructions de base, illustrées dans la figure 11.4 :

- Les nœuds `Choice` indiquent que le calcul peut prendre deux directions, avec des résultats différents.

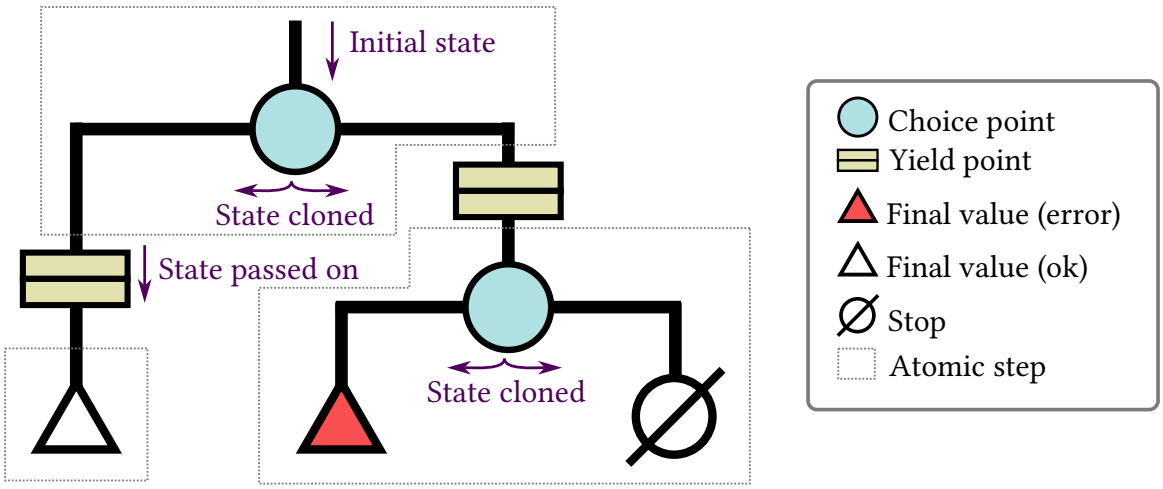



FIGURE 11.4 – Une représentation graphique d'une valeur de notre monade de choix.

— Les nœuds `Yield` permettent à l'ordonnanceur de suspendre la coroutine et de commencer à en exécuter une autre.
— Les feuilles `Value` indiquent une valeur finale pour cette (sous-)coroutine. Dans notre cas, cette valeur sera une valeur de la monade d'erreur.
— Les feuilles `Stop` indiquent que l'exécution s'est arrêtée sans une `Value` car cette branche est en fait inatteignable.

Les opérations monadiques exposées par notre monade de choix construisent une valeur représentant l'arbre des coroutines. Cette valeur décrivant une coroutine peut ensuite être passée à un ordonnanceur qui l'exécutera en étapes atomiques délimitées par des points de cession (`Yield`). Lorsqu'une feuille `Value` est atteinte, notre ordonnanceur exécute un *callback* prédéterminé pour gérer cette valeur de manière appropriée.

### Mise en œuvre de la monade de choix multi-cœur

Le reste de cette section est consacré à une présentation approfondie de notre monade de coroutine. Correctement mettre en œuvre cette monade a été l'un des défis techniques qui, une fois surmonté, a permis à Owi d'atteindre les performances présentées dans la section 13. Cependant, comprendre cette monade de manière exhaustive n'est pas nécessaire pour suivre le reste de cette thèse. Le lecteur qui le souhaite peut directement passer à la section 11.4.1 (p. 174).

**La monade de couroutine** La monade de coroutine est mise en œuvre comme suit en OCaml :

```
(* Two mutually recursive datatypes *)

(* The type of coroutines themselves, it contains a function that takes the
↪  worker local storage and returns a status *)
type ('a, 'wls) t =
  Schedulable of ('wls -> ('a, 'wls) status)
```

```
(* The type of one step of the coroutine *)
and ('a, 'wls) status =
  | Now of 'a (* Final value of type 'a *)
  | Yield of Prio.t * ('a, 'wls) t (* Coroutine with a priority *)
  | Choice of (('a, 'wls) status * ('a, 'wls) status) (* Choice point *)
  | Stop (* End of execution *)
```

Les coroutines, de type `t`, s'exécutent par étapes. Chaque valeur de (sous-)coroutine (de type `t`) lit un stockage local au *worker* (WLS) et peut produire :

— `Now`, indiquant une valeur finale ;
— `Yield`, qui planifie une continuation de la coroutine avec une priorité associée ;
— `Choice`, indiquant un choix entre deux chemins d'exécution ;
— `Stop`, signalant la fin de la coroutine sans aucune valeur.

Les primitives de base de cette monade incluent :

```
(* Convert a value to a coroutine that returns it immediately *)
let return (x : 'a) : ('a,'wls) t =
  (* A Schedulable value containing a function ignoring the WLS and returning
  ↪  x *)
  Schedulable (fun _wls -> Now x)

(* Executes one step of the coroutine with the provided WLS *)
let run (Schedulable mxf : ('a, 'wls) t) (wls : 'wls) : ('a, 'wls) status
  = mxf wls

(* Yields control back to the scheduler with a specified priority *)
let yield (prio: Prio.t) : (() ,'wls) t = u
  Schedulable (fun _wls -> Yield (prio, return ()))

(* Creates a choice between two coroutines *)
let choose (a : ('a, 'wls) t) (b : ('a, 'wls) t) : ('a, 'wls) t =
  Schedulable (fun wls -> Choice (run a wls, run b wls))

(* Accesses the worker-local storage *)
let wls : ('wls,'wls) t =
  (* This coroutine inner function simply returns the WLS *)
  Schedulable (fun wls -> Now wls)

(* Forks the current coroutine, yielding back control to the scheduler. The
↪  parent coroutine will resume execution with a priority prio_parent, and
↪  the child coroutine with a priority prio_child. In the parent (resp.
↪  child) coroutine, fork is false (resp. true).*)
  let fork (prio_parent: Prio.t) (prio_child : Prio.t): bool t =
    Schedulable (fun _wls ->
      Choose
```

```
        (Yield (prio_parent, Now false))
        (Yield (prio_child, Now true)))
```

L'opération centrale dans cette monade est la fonction de liaison (`bind`) :

```
let rec bind (mx : ('a, 'wls) t) (f : 'a -> ('b, 'wls) t) : ('b, 'wls) t
  = Schedulable
    (fun wls ->
      (* A closure whose purpose is to traverse nested statuses and return the
      ↪  final value of one step of (bind mx f) *)
      let rec unfold_status (x : ('a, 'wls) status) : ('b, 'wls) status
        = match x with
        (* The final value is that of f applied to x *)
        | Now x -> run (f x) wls
        (* If the coroutine is yielding, we return a yield. The resulting next
        ↪  coroutine will be the result of recursively binding f to the
        ↪  initial next coroutine. *)
        | Yield (prio, lx) -> Yield (prio, bind lx f)
        (* When encountering a choice, we simply continue unfolding both
        ↪  branches. *)
        | Choice (mx1, mx2) ->
          let mx1' = unfold_status mx1 in
          let mx2' = unfold_status mx2 in
          Choice (mx1', mx2')
        (* We stop if mx stops *)
        | Stop -> Stop
      in
      unfold_status (run mx wls) )
```

**Ordonnancement des coroutines** Pour exécuter les coroutines, nous utilisons un ordonnanceur qui gère la distribution du travail. Cet ordonnanceur utilise une file d'attente FIFO synchronisée pour stocker les coroutines et distribue le travail aux *worker threads* disponibles. Il dispose de l'interface suivante.

```
(** Synchronized FIFO queues *)

(** The main and only type of this module. *)
type !'a t

(** Create a new queue *)
val make : unit -> 'a t

(** Add a new element to the queue *)
val push : 'a -> 'a t -> unit
```

```
(** Get an element from the queue. The boolean shall be true to atomically
↪  start a new pledge (cf make_pledge) while popping. *)
val pop : 'a t -> bool -> 'a option

(** Make a new pledge, i.e. indicate that new elements may be pushed to the
↪  queue and that calls to pop should block waiting for them *)
val make_pledge : 'a t -> unit

(** End one pledge *)
val end_pledge : 'a t -> unit

(** Mark the queue as closed: all threads trying to pop from it will get no
↪  element *)
val close : 'a t -> unit

(** Call in a loop the provided function while there is elements in the queue.
↪  The provided function should take a queue element as first parameter, and
↪  a callback allowing to push new elements to the queue as second parameter.
↪  *)
val work_while : ('a -> ('a -> unit) -> unit) -> 'a t -> unit
```

En utilisant ces éléments, nous pouvons mettre écrire l'ordonnanceur comme suit :

```
(* Our work queue type: a queue containing the Schedulable.t coroutines
↪  defined above *)
type ('a, 'wls) work_queue = ('a, 'wls) Schedulable.t Wq.t

(* Scheduler type containing a work queue *)
type ('a, 'wls) t = { work_queue : ('a, 'wls) work_queue }

(* Create a new scheduler with an empty queue  *)
let make_scheduler () =
  { work_queue = Wq.make () }

(* Add a new task to the scheduler *)
let submit_task sched task = Wq.push task sched.work_queue

(* Main loop of a worker thread. Initialized with its WLS, it runs steps of
↪  the coroutines scheduled on sched and calls callback on their final value.
↪  *)
let work wls sched callback =
  (* Handles the coroutine's status. The second argument is the Queue callback
  ↪  allowing to push new elements. *)
  let rec handle_status (t : _ Schedulable.status) write_back =
    match t with
    (* No more execution needed, return () *)
```

```
    | Stop -> ()
    (* A final value, pass it to the call back *)
    | Now x -> callback x
    (* The coroutine yielded, push the follow-up coroutine to the queue. For
    ↪  now the priority is ignored. *)
    | Yield (_prio, f) -> write_back f
    (* The coroutine forked. Evaluate each of the two sub coroutines
    ↪  sequentially. *)
    | Choice (m1, m2) ->
      handle_status m1 write_back;
      handle_status m2 write_back
  in
  (* Use the queue work_while function to run each coroutine and handle their
  ↪  status while the scheduler work queue is not empty *)
  Wq.work_while
    (fun f write_back ->
      handle_status (Schedulable.run f wls) write_back)
    sched.work_queue

(* Spawn a worker thread for scheduler `sched`. It will call callback on each
↪  final value. callback_init and callback_close are the callback initializer
↪  and deinitializer respectively. *)
let spawn_worker sched wls_init callback callback_init callback_close =
  callback_init ();
  (* Start a Domain, an OCaml concept similar to a thread *)
  Domain.spawn (fun () ->
      Fun.protect
        (* This will be executed even if the other closure raises an exception
        ↪  *)
        ~finally:callback_close
        (* This closure is the actual thread code *)
        (fun () ->
          (* Initialize the worker local storage *)
          let wls = wls_init () in
          (* Run the worker loop defined above *)
          try work wls sched callback
          with e ->
            let bt = Printexc.get_raw_backtrace () in
            (* If this worker loop fails, mark the queue as closed so that the
            ↪  scheduler stops *)
            Wq.close sched.work_queue;
            Printexc.raise_with_backtrace e bt ) )
```

L'ordonnanceur retire continuellement des coroutines de la file de travail, les exécute, et soit termine leur exécution, soit les replanifie en fonction de leur état actuel, comme illustré sur les figures 11.5 et 11.6. Pour le moment, nous traitons toutes les cessions avec une priorité égale,

bien que le support de l'ordonnancement prioritaire soit une extension simple[3]. Déterminer quelle priorité attribuer à chaque branche est la partie délicate et est laissée pour de futurs travaux.

**La monade de choix complète** En appliquant des transformateurs de monades pour l'état et les erreurs, et en fixant certains des paramètres de type, nous obtenons le type monadique final :

```
type 'a t = St of (Thread.t ->
  Schedulable of ( Solver.t ->
    ('a eval * Thread.t, Solver.t) status
  )
)
```

Où :

- `Thread.t` représente l'état de l'interpréteur Wasm et la condition de parcours ;
- `Solver.t` représente l'état du solveur SMT ;
- `eval` représente le résultat potentiel d'une évaluation qui peut échouer, et est défini comme suit :

```
type 'a eval =
  | EVal of 'a (* success *)
  | ETrap of (* metadata describing the Wasm trap *)
  | EAssert of (* metadata representing the assertion failure *)
```

- `status` est l'état de la coroutine défini plus haut.

Nous faissons passer toutes les opérations requises à travers les transformateurs, ce qui nous permet de définir des routines complexes d'exécution symbolique, comme `check` qui prend une valeur booléenne et utilise le solveur SMT pour vérifier sa faisabilité compte tenu des hypothèses actuelles, et `assume` qui prend une valeur booléenne et enregistre qu'elle a été supposée vraie. En utilisant ces constructions, nous pouvons maintenant écrire une fonction `select` symbolique :

```
(** Checks if a boolean value is feasible, and if so record its value in the
↪  assumptions. Otherwise, stops the execution of the current branch. *)
let check_and_record (b : Symbolic_value.bool)  =
  let* () = yield in
  if check b then
    record b
  else
    stop

let select (v : Symbolic_value.bool) : bool Choice.t =
```

3. Pour planifier avec une priorité, nous devrions simplement changer notre module de file d'attente : pour l'instant, il est basé sur une file d'attente FIFO, mais nous pourrions le changer en une file de priorité tout en conservant le même code de synchronisation multi-cœur.

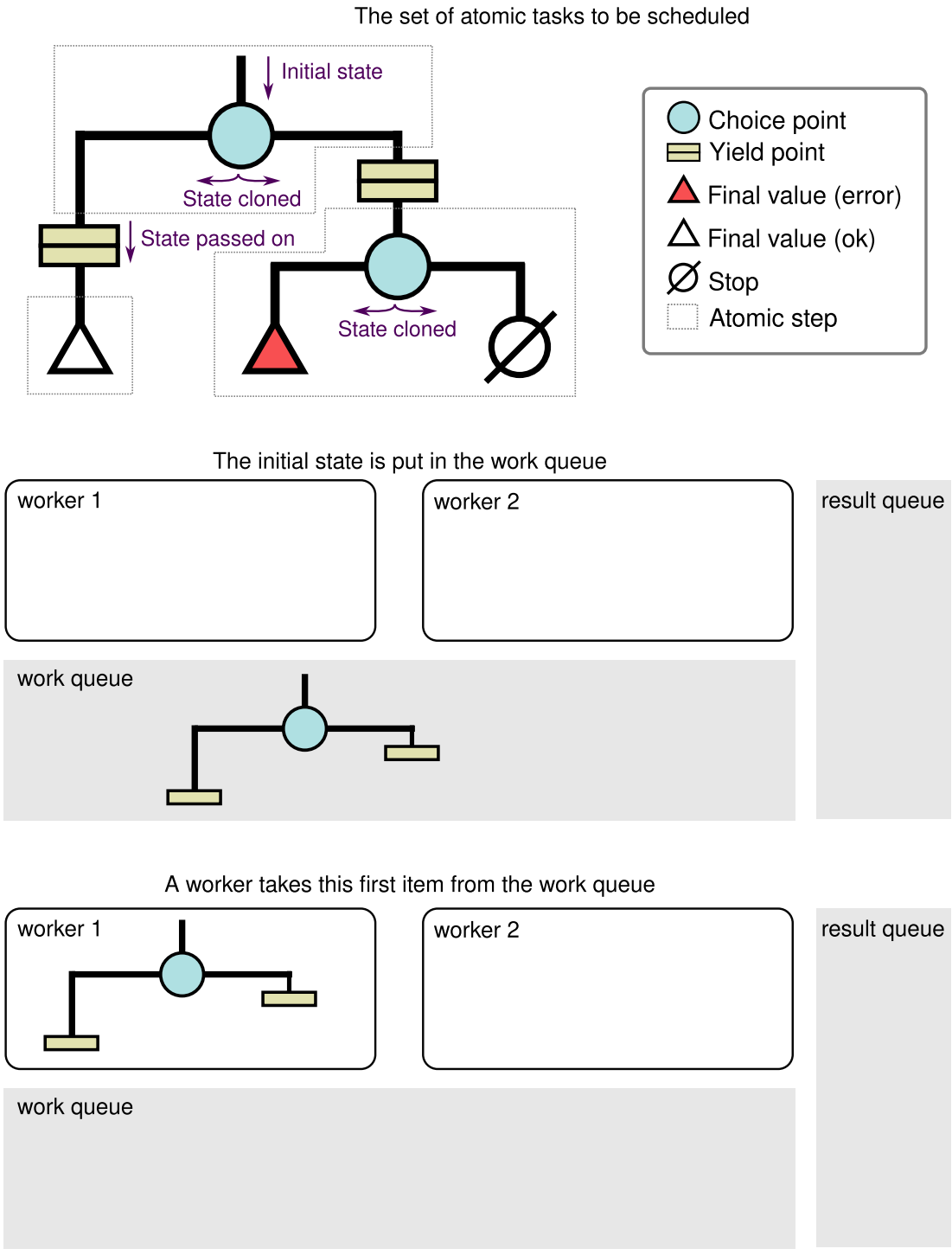



FIGURE 11.5 – Première étape illustrant l'ordonnancement.

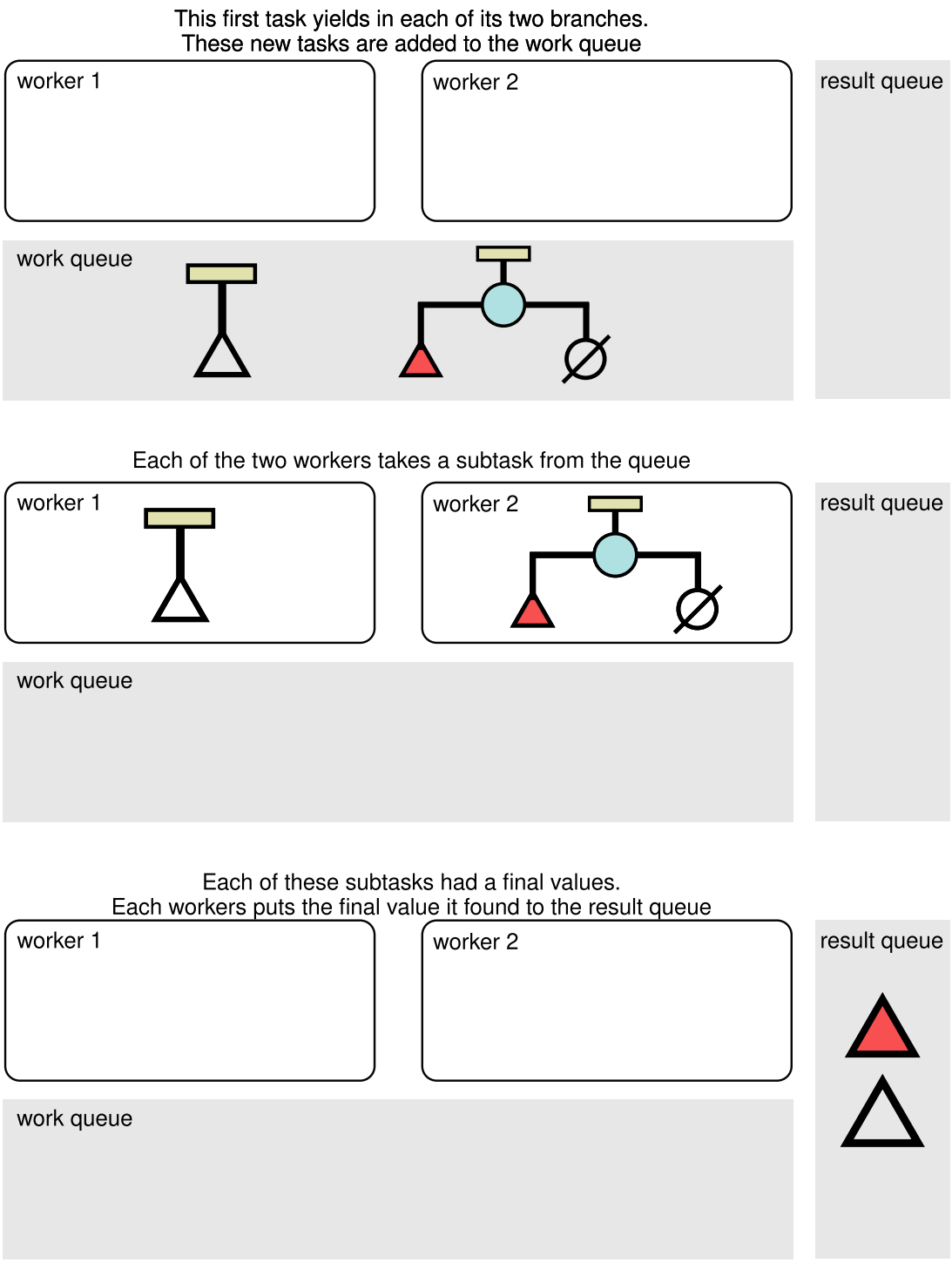



FIGURE 11.6 – Seconde étape illustrant l'ordonnancement.

```
(* We first create two monadic values, each corresponding to a possible
↪  value of v (either true or false). Each of these branches simply checks
↪  the reachability, records the chosen value, and returns it as part of
↪  the monad. *)
let top_branch =
  (* The let+ notation is similar to Haskell's do notation. This can be read:
  ↪  bind (check_and_record v) (fun () -> return true) *)
  let+ () = check_and_record v
  true
in
let bot_branch =
  let+ () = check_and_record (Symbolic_value.not v) in
  false
in
(* Finally, we create the coroutine combining the two possibilities defined
↪  above *)
choose top_branch bot_branch
```

## 11.4 Modèle mémoire

### 11.4.1 Un modèle paresseux

Un problème classique avec l'exécution symbolique est celui de la consommation de mémoire excessive due à l'explosion du nombre de chemins [70, 99] et ce, même pour des petits programmes. En évaluant Owi sur les programmes issus de Test-Comp [146] avec un timeout de 30 secondes, nous avons observé un nombre maximal de 156052 branchements dans un unique programme. En adoptant une approche naïve consistant à copier tout l'état à chaque branchement, nous aurions à stocker 156052 copies de la mémoire linéaire Wasm, dont la taille varie généralement entre 64KiB et 4GiB, ce qui est impratiquable. Cette explosion rapide met en avant l'importance d'une gestion efficace du stockage des états.

Owi utilise un modèle mémoire paresseux similaire à une approche de copie à l'écriture (*copy-on-write*). Lorsqu'un nouvel état est créé, il ne provoque pas une duplication de la mémoire. À la place, une nouvelle mémoire est créée. Cette dernière pointe vers la mémoire précédente mais comporte aussi une table des modifications. Les écritures sont toujours effectuées dans la table des modifications. Lors d'une lecture à un emplacement mémoire donné, Owi vérifie d'abord si cet emplacement est présent dans la table des modifications. S'il l'est, la valeur est renvoyée ; sinon, la valeur est récupérée depuis la mémoire précédente. Ce procédé est récursif, la mémoire précédente pouvant elle aussi contenir une table de modifications et pointer vers une mémoire encore plus ancienne.

Cette stratégie est similaire à l'approche utilisée par KLEE [60]. Cependant, KLEE représente la mémoire comme une table d'objets, ce qui requiert de copier un objet entier dans la table des modifications même pour une petite modification. Au contraire, Owi représente la mémoire comme un grand tableau de mots, lui permettant de ne copier que les mots modifiés, ce qui est plus efficace en termes d'utilisation mémoire.

Cette stratégie laisse penser que les tableaux persistants de Baker [7] ou leur version

```
type t = expr hash_consed
and  expr =
  | Val of Value.t
  | Unop of unop * t
  | Binop of binop * t * t
  | Triop of triop * t * t * t
  | Relop of relop * t * t
  | Cvtop of cvtop * t
  | Symbol of Symbol.t
  | Extract of t * int * int
  | Concat of t * t
```

Code source 11.5 : Syntaxe abstraite de *Smt.ml*

semi-persistante décrite par Conchon [61] pourraient être moyen efficace de mettre en œuvre notre approche. Cependant, il ne nous est pas possible de les utiliser simplement du fait de notre monade multi-cœur. En effet, il serait nécessaire d'introduire des verrous ce qui aurait probablement un impact négatif sur les performances. Les travaux récents d'Allain+24 indiquent pouvoir proposer une mise en œuvre efficace de la persistence *partielle* où seule la dernière version de la structure est susceptible d'être écrite tandis que toutes les autres peuvent être lues. Les auteurs indiquent ne pas avoir de cas d'utilisation à cette variante, et il serait intéressant d'expérimenter avec.

### 11.4.2 Gestion des adresses symboliques

Actuellement, notre modèle mémoire gère les valeurs symboliques de façon naïve. Dès qu'une lecture ou une écriture est faite à une adresse symbolique, il va effectuer un branchement pour toutes les valeurs possibles, en les énumérant grâce au SMT. Il est intéressant de noter que KLEE quant à lui, ne sait pas gérer les adresses symboliques non bornées, et ne semble pas effectuer un parcours exhaustif de toutes les possibilités lorsque l'adresse est bornée. Un travail est en cours pour améliorer notre modèle en utilisant une approche plus efficace basée sur les travaux de Coppa [89].

## 11.5 Interaction avec les solveurs SMT

La bibliothèque OCaml *Smt.ml* [152] est une couche d'abstraction pour les solveurs de contraintes et les solveurs SMT. Son objectif premier est de faciliter la transition d'un solveur à un autre dans les analyses de programmes, certains solveurs pouvant se montrer plus efficaces pour gérer des logiques ou des formules spécifiques. Pour le moment, *Smt.ml* supporte Z3 [62], Colibri2 [169] et Bitwuzla [155]. L'ajout du support d'Alt-Ergo [100] et de CVC5 [129] est en cours.

### 11.5.1 Syntaxe abstraite

Le principal moyen d'interaction avec *Smt.ml* est au travers de sa grammaire abstraite. En premier lieu, les termes logiques doivent être encodés comme des expressions *Smt.ml* afin de pouvoir être utilisés lors de futurs tests de satisfiabilité par l'un des solveurs. La grammaire abstraite est définie par le type algébrique dans la figure 11.5.

La grammaire permet de représenter des valeurs concrètes des opérations unaires, binaires, ternaires, relationnelles, des opérations de conversion ainsi que des opérations spécifiques telles que l'extraction et la concaténation. Elle permet de représenter succintement la représentation de termes logiques tels que :

```
Relop (Ge, Symbol "x", Val (Int 42))
```

Dans cet exemple, la grammaire abstraite est employée pour exprimer la contrainte logique $x \geq 42$. Le constructeur `Relop` dénote une opération relationnelle. Ici il s'agit de l'opérateur "supérieur ou égal" (`Ge`). Le constructeur `Symbol "x"` représente la variable $x$, et le constructeur `Val` dénote une valeur concrète, ici l'entier $42$. *Smt.ml* fournit également des *smart constructors* permettant de construire des termes *hash-consé* :

```
relop Ge (symbol "x") (int 42)
```

### 11.5.2 Solveurs paramétriques

*Smt.ml* fournit une manière générique d'interagir avec des solveurs SMT. Indépendemment du solveur choisi, la signature du module `Solver` est toujours la même. Ce module fournit un ensemble cohérent de fonctions facilitant l'intégration avec les SMT solveurs. Cela inclut les fonctions `check`, `push`, `pop`, `model` et `get_value`.

Afin de fournir une intégration facile avec de multiples solveurs SMT, nous fonctorisons le module `Solver` par un ensemble bien défini de fonctions fournies par chaque solveur SMT :

```
module Make (M : Prover_mappings_intf.S) = struct … end
```

Ce foncteur, appelé `Make`, prend un module `M` en argument. Le module `M` doit la signature définie par `Prover_mappings_intf.S`, qui contient l'ensemble standard des fonctions attendues par chaque solveur SMT.

### 11.5.3 Intégration avec Owi

Grâce à l'interpréteur fonctorisé d'Owi, y intégrer *Smt.ml* est un processus simple. Cela requiert de fournir un module `Values` qui adhère à l'interface attendue par l'interpréteur. À l'intérieur de ce module, les expressions sont construites en utilisant les *smart constructors* de *Smt.ml*. En reprenant le module `Values` de la section 11.2, le module utilisé pour l'exécution symbolique prend maintenant la forme suivante :

```
open Smtml
module Values = struct
  module Int32 =
    type t = Expr.t
```

```
    let v i = Expr.value (I32 i)
    let add lhs rhs = Expr.relop Add lhs rhs
    let gt lhs rhs = Expr.relop Gt lhs rhs
  end
  (* … *)
end
```

Il est intéressant de noter que nos expressions restent toujours *décidables* et que Z3 est toujours en mesure de répondre aux requêtes qui lui sont envoyées.

## 11.6 Démonstration des capacité d'exécution symbolique d'Owi

Après avoir détaillé la manière dont Owi est mis en œuvre, nous allons maintenant démontrer ses capacités en tant qu'outil d'exécution symbolique.

Nous illustrons d'abord les capacités d'Owi en effectuant une exécution symbolique sur la fonction `$test_swap` définie dans le code 11.1. Pour ce faire, nous écrivons le code suivant qui appelle la fonction `$test_swap` avec deux paramètres symboliques. Rappelons que la fonction `$test_swap` vérifie déjà sa propre cohérence en utilisant un `if` qui mène à une instruction `unreachable`.

```
(func $main
  call $i32_symbol ;; Creates a fresh symbol
  call $i32_symbol ;; Creates a fresh symbol
  call $test_swap
)
```

En appelant Owi sur ce code, nous pouvons effectivement identifier l'entrée problématique qui conduit à un débordement de capacité.

```
$ owi sym test_swap.wat
Trap: unreachable
Model:
  (model
    (symbol_0 (i32 2147483646))
    (symbol_1 (i32 -2147483647)))
Reached problem!
```

Au-delà de cet exemple simpliste, Owi peut également détecter des bogues dans des bases de code plus réalistes. Par exemple, nous avons pris une mise en œuvre Wasm d'une structure de données d'arbres B [3] développée par C. Costa [114]. Ensuite, nous avons ajouté quelques assertions supplémentaires pour vérifier la cohérence de la structure de données après quelques opérations. En exécutant Owi, il est déterminé que tous les chemins d'exécution possibles sont exempts de bogues, comme le montre la sortie suivante où Owi affiche "All OK" :

```
$ owi sym tree_3o3u.wat
All OK
```

Ensuite, nous introduisons un bogues dans une section critique du code en modifiant une comparaison “supérieur à” en une comparaison “inférieur à”, et Owi parvient à trouver qu’un chemin d’exécution conduit désormais à un échec d’assertion :

```
$ owi sym tree_3o3u_buggy.wat
Assert failure: false
Model:
  (model
    (symbol_0 (i32 -71872612))
    (symbol_1 (i32 -72221176))
    (symbol_2 (i32 -1543220269))
    (symbol_3 (i32 -72326636))
    (symbol_4 (i32 -205803988))
    (symbol_5 (i32 71802348)))
Reached problem!
```

De même, Owi est capable d’analyser des programmes C complexes, comme décrit dans la section 12.1. Ces codes peuvent être complexes. Un exemple est la suite de tests symboliques utilisée pour l’évaluation dans le chapitre (§13). Ces tests contiennent par exemple des programmes extrait du code source du noyau Linux [27].

## 11.7 Exécution concolique

Nous avons également intégré une exécution *concolique* dans Owi. Ce terme est un néologisme qui combine les mots “concret” et “symbolique” pour désigner une méthode qui fait coopérer ces deux types d’exécutions [54]. L’idée d’hybrider l’exécution concrète avec l’exécution symbolique a été introduite pour la première fois en 2003 [46]. La méthode que nous utilisons ici a été raffinée et décrite plus en détail en 2005 [52].

### 11.7.1 Principes de l’exécution concolique

Le fonctionnement de l’exécution concolique repose sur un principe proche de celui de l’exécution symbolique. Lorsqu’un symbole est créé, il est associé à une valeur concrète initiale, choisir aléatoirement. Ensuite, l’exécution du programme suit le chemin produit en évaluant les expressions à l’aide de cette valeur concrète tout en collectant en parallèle les conditions de chemin symboliques pour les branches prises. Autrement dit, là où une exécution symbolique ne manipulerait que des expressions symboliques, une expression concolique manipule des paires, constituées d’une valeur concrète et d’une expression symbolique.

Deux scénarios se présentent alors :

1. **L’exécution se termine par un bogue**. Dans ce cas, les valeurs concrètes qui ont mené à ce bogue sont déjà connues, car elles ont été utilisées tout au long de l’exécution. Ces valeurs constituent un modèle direct des entrées ayant provoqué l’erreur.

2. **L'exécution se termine sans erreur**. Dans ce cas, les contraintes symboliques collectées lors de l'exécution sont ajustées pour explorer d'autres chemins. Un solveur SMT est alors sollicité pour fournir de nouvelles valeurs concrètes qui satisferont ces contraintes modifiées, permettant ainsi d'explorer un chemin différent du programme.

Ce processus est répété de manière récursive jusqu'à ce qu'un bogue soit trouvé ou que toutes les branches possibles aient été explorées.

L'exécution concolique est souvent plus efficace que l'exécution symbolique pure, et ce pour plusieurs raisons :

— **Évaluation directe des conditions de branchement** : lorsqu'un branchement est rencontré, il n'est plus nécessaire d'interroger le solveur SMT pour vérifier si une condition est satisfiable. Grâce aux valeurs concrètes disponibles, il suffit d'évaluer directement la condition.

— **Exploration d'une seule branche à la fois** : contrairement à l'exécution symbolique qui explore simultanément toutes les branches possibles, l'exécution concolique ne suit qu'une seule branche, celle dictée par les valeurs concrètes. Il n'y a alors qu'un seul état concret maintenu à tout moment, au lieu d'un grand nombre d'états comme c'est le cas avec l'exécution symbolique classique. En effet, même pour se rappeler qu'il reste une branche à explorer, il n'est pas nécessaire de stocker son état. Il suffit de reprendre l'exécution à partir de l'état initial avec les bonnes valeurs concrètes pour se retrouver dans le même état que celui qu'on aurait pu inutilement stocker.

En résumé, l'exécution concolique combine les avantages de la gestion concrète et symbolique des chemins d'exécution, permettant une exploration plus rapide et plus efficace des chemins critiques du programme, tout en réduisant la complexité et la charge mémoire.

### 11.7.2 Monade de choix concolique

Notre mise en œuvre repose sur une monade dédiée à l'exécution concolique. Dans cette monade, chaque valeur et chaque état est représenté sous la forme d'une paire concrète et symbolique. La condition de chemin, qui capture les contraintes associées aux décisions symboliques prises durant l'exécution, est maintenue dans la monade d'état. Lorsqu'un branchement symbolique est rencontré, celui-ci est ajouté à la condition de chemin.

Pour garantir une exploration systématique des chemins d'exécution, une structure arborescente est utilisée. Celle-ci permet de représenter tous les chemins déjà parcourus et ceux restant à explorer. Lorsque le moment est venu de choisir un nouveau chemin, un nœud non exploré [4] de l'arbre est sélectionné. Un solveur SMT est alors utilisé pour produire un modèle qui permet d'explorer une branche du programme encore inexplorée. Si le nombre de chemins d'exécution est fini, alors l'algorithme finira par explorer toutes les branches possibles.

Un chemin dans cet arbre correspond à l'ensemble des conditions de chemin satisfaites par les modèles générés pour chaque branche explorée. Toutefois, il existe certaines complexités spécifiques, notamment lorsqu'il s'agit de manipuler des adresses symboliques lors d'une lecture mémoire. Dans ce cas, l'utilisation de la valeur concrète de l'adresse pourrait masquer certaines exécutions potentielles. Il est donc nécessaire de tester toutes les valeurs concrètes possibles pour cette adresse symbolique, comme dans une exécution symbolique classique, afin d'éviter de rater des cas d'erreur.

4. *I.e.* qui a encore au moins une branche non explorée.

Nous ne détaillons pas davantage la mise en œuvre de la monade concolique pour deux raisons principales : premièrement, elle contient encore des bogues qui doivent être corrigés, et deuxièmement, elle n'est pas encore optimisée pour le multi-cœur. Sa mise en œuvre actuelle étant encore sujette à de nombreux changements, nous avons préféré ne pas la présenter en détail dans cette thèse. Le lecteur curieux peut se référer au dépôt d'Owi pour obtenir les détails de la version actuelle. Enfin, il est déjà possible d'utiliser l'exécution concolique dans Owi au moyen de la commande `l@tex:~$ owi conc file.wat`.

## 11.8 Travaux connexes

L'objectif de ce travail est de développer un interpréteur Wasm évolutif et maintenable, capable d'effectuer une exécution symbolique efficace. Nos contributions s'alignent étroitement avec les recherches sur les interpréteurs monadiques et paramétriques, ainsi que sur les outils d'exécution symbolique spécifiquement conçus pour Wasm.

### 11.8.1 Interpréteurs monadiques et paramétriques

Notre approche présente des similitudes avec les travaux de Mensing [110], qui dérivent un moteur d'exécution symbolique à partir d'un interpréteur définitionnel. Comme notre interpréteur Wasm, leur approche débute avec une mise en œuvre concrète et évolue vers une version paramétrique avec une interface de valeurs abstraites. Ils instancient l'interpréteur dans des modes à la fois concrets et symboliques. La principale différence entre notre travail et le leur réside dans le fait qu'ils se concentrent sur des langages dynamiquement typés avec des fonctions récursives et des motifs de correspondance. En outre, ils adoptent une stratégie de recherche en largeur dans leur interpréteur symbolique pour explorer l'espace des états du programme. En revanche, notre mise en œuvre utilise une monade de coroutines, ce qui nous permet d'explorer en priorité les branches d'exécution les plus pertinentes.

Necro [139], un cadre formel pour la sémantique des langages de programmation, génère des interpréteurs ou des preuves en Coq [32] à partir de la sémantique des langages. Necro génère un foncteur OCaml paramétrique sur une monade d'interprétation qui gère les applications et les branchements. Comme notre approche, il utilise des monades pour gérer les effets computationnels, bien que Necro se concentre principalement sur l'interprétation concrète plutôt que sur l'extension à l'exécution symbolique.

### 11.8.2 Exécution symbolique pour Wasm

L'exécution symbolique a été largement utilisée pour découvrir des erreurs critiques et des vulnérabilités dans divers langages de programmation, notamment C [52], C++ [60], Java [54] et Python [80]. Pour le Web, plusieurs outils à la pointe de l'exécution symbolique pour le code JavaScript [103, 112, 68, 81, 85] témoignent de la demande croissante pour valider et tester les applications web modernes.

Les moteurs d'exécution symbolique se divisent généralement en deux classes : les moteurs d'exécution statiques et concoliques/dynamiques [99]. Les outils d'exécution symbolique statique [5, 67, 45, 82, 103, 112] explorent l'ensemble de l'arbre d'exécution symbolique jusqu'à une profondeur prédéfinie. En revanche, les outils d'exécution concolique [60, 52, 54, 73, 68, 81, 85] combinent des exécutions concrètes et symboliques, en se concentrant sur l'exploration

d'un chemin à la fois. De nombreux outils d'exécution symbolique ont été proposés pour divers langages de programmation, comme recensé dans plusieurs articles[99, 70, 77]. Ci-dessous, nous explorons les outils d'exécution symbolique actuels pour Wasm, en dehors d'Owi.

**WANA** WANA [118] est un outil multiplateforme qui utilise l'exécution symbolique statique pour détecter les vulnérabilités dans les contrats intelligents compilés en bytecode Wasm. Cependant, il ne dispose pas d'un moteur d'exécution symbolique autonome pour le code Wasm général.

**Manticore** Manticore [111], un cadre flexible d'exécution symbolique pour les binaires et les contrats intelligents, inclut le support du bytecode Wasm, mais repose sur des scripts Python complexes et manuellement créés pour chaque test. Manticore n'est plus activement maintenu.

**WASP** WASP [137], basé sur un ancien interpréteur de référence Wasm, utilise l'exécution concolique pour réduire les interactions avec le solveur et simplifier la modélisation de la mémoire. Cependant, il reste limité à Wasm 1.0, car la mise à jour de l'interpréteur pour supporter les nouvelles fonctionnalités du langage a été jugée irréalisable par l'équipe de développement.

**SeeWasm** SeeWasm[164], présenté dans [150], introduit une nouvelle approche d'exécution symbolique avec des stratégies de recherche locale à grain fin, mais sa faisabilité pratique est limitée par le besoin d'une compréhension approfondie du programme.

Aucun de ces outils, y compris WANA, Manticore, WASP et SeeWasm, ne prend en charge l'exploration parallèle ou concurrente de l'espace d'états pour les programmes Wasm. Owi est le premier moteur d'exécution symbolique pour Wasm à introduire l'exploration parallèle de l'espace d'états, en exploitant la puissance d'OCaml multi-cœur.

### 11.8.3 Moteurs d'exécution symbolique parallèle

Une vaste littérature existe sur la parallélisation de l'exécution symbolique [69, 127, 148]. Tous les articles que nous avons consultés utilisent un algorithme similaire au nôtre, où les branches de l'arbre d'exécution sont distribuées à différents fils d'exécution. Cependant, aucun ne semble utiliser notre modèle de mémoire paresseux permettant une passage à l'échelle efficace. Ces outils n'ayant pas participé à TestComp, il est difficile de les tester sur nos ensembles de benchmarks. Cela nécessiterait de les modifier pour qu'ils fournissent une interface C compatible avec les fichiers attendus par TestComp. De plus, à notre connaissance, il n'existe pas d'outil à la fois bien documenté et utilisable pour Wasm, C et/ou Rust, capable de fonctionner en parallèle. Enfin, aucun des outils que nous avons trouvés n'utilise une architecture logicielle basée sur des monades.

# 12 Exécution symbolique de code C, Rust et cross-langage

Nous avons démontré la capacité d'Owi à identifier des bogues dans des bases de code Wasm. Cependant, de nombreux langages, y compris C et Rust, peuvent être compilés vers Wasm. Nous pouvons donc utiliser Owi pour détecter des bogues dans tout programme qui peut être compilé vers Wasm. Pour analyser un programme écrit dans un langage $L$ en utilisant Owi, il faut :

- **Modéliser les primitives du langage *L*** : Cela implique de modéliser les interactions du programme avec son environnement (comme les entrées/sorties disque et réseau ou les appels système) pour une analyse précise. Cela inclut également la modélisation correcte de la gestion dynamique de la mémoire (par exemple, `malloc`, `realloc` et `free` en C).
- **Exposer les primitives d'Owi** : Cette étape consiste à connecter les primitives de base d'Owi à $L$, permettant ainsi la génération de valeurs symboliques, les vérifications d'assertions et l'exploration contrôlée.

Une fois ces étapes effectuées, le programme écrit en $L$ peut être compilé vers Wasm et exécuté dans Owi pour la détection de bogues. Toutefois, il est important de noter que certains bogues, en particulier ceux liés aux comportements indéfinis, peuvent être masqués par le compilateur. Un programme Wasm représente une interprétation possible du programme original, qui peut avoir plusieurs interprétations si un comportement indéfini est présent. Les sous-sections suivantes décrivent notre travail dans l'application d'Owi à C et Rust.

## 12.1 Vérification de code C

### 12.1.1 Utilisation via des harnais

Nous avons ajouté une sous-commande à l'exécutable Owi qui facilite la compilation des programmes C vers Wasm. Avec ces composants en place, les programmes C peuvent être testés en utilisant Owi, comme illustré dans les exemples suivants.

Une approche courante pour trouver des bogues consiste à écrire un harnais de test. Par exemple, pour tester une fonction `f`, nous définissons des entrées symboliques correspondant aux types de ses paramètres et effectuons des assertions sur sa sortie. Le harnais est défini dans la fonction `main`, avec `f` étant la fonction à tester :

```c
#include <owi.h>

int f(int x, float y) {
  …
}

void main(void) {
  int x = owi_i32();
  float y = owi_f32();

  int result = f(x, y);

  owi_assert(result == 42);
}
```

Dans cet exemple, nous définissons des valeurs symboliques pour un entier `x` et une valeur en virgule flottante `y` en utilisant les fonctions exposées par Owi. Nous appliquons ensuite `f` à ces entrées symboliques et nous nous assurons que le résultat est égal à 42. La fonction `owi_assert` d'Owi vérifie que l'expression symbolique fournie en entrée est toujours vraie. De plus, Owi offre une fonction `owi_assume`, qui permet aux développeurs de restreindre la condition de chemin à certains scénarii, en excluant les comportements non pertinents de l'exploration.

Alternativement, les utilisateurs peuvent remplacer les assertions par des contrats de fonction, comme ceux vus dans des outils de vérification d'assertions à l'exécution tels qu'E-ACSL [95] ou des outils de vérification déductive comme Why3 [78]. Ces contrats sont écrits dans une logique de spécification distincte du langage exécutable, et leur traduction en code symboliquement exécutable nécessite un travail supplémentaire. Nous avons intégré cette fonctionnalité pour C en réutilisant E-ACSL. Notre approche consiste à prendre un fichier C annoté avec des spécifications E-ACSL, à générer un fichier exécutable instrumenté via E-ACSL, et à utiliser un runtime symbolique pour gérer le code instrumenté. Ce travail est décrit ultérieurement dans la thèse, dans le chapitre 15.

### 12.1.2 Exposition des primitives d'Owi

Nous avons écrit des portions de la bibliothèque standard C, ainsi qu'un fichier d'en-tête C qui permet l'interaction avec Owi depuis le C. Les primitives d'Owi sont exposées au travers d'un fichier d'en-tête `owi.h`. Des attributs permettent d'indiquer où trouver les fonctions au moment de l'édition de lien Wasm, en fournissant le nom du module et le nom de la fonction correspondants à chaque fonction C exposée :

```c
__attribute__((import_module("summaries"), import_name("alloc"))) void
↪  *owi_malloc(void *, unsigned int);
__attribute__((import_module("summaries"), import_name("dealloc"))) void
↪  owi_free(void *);
```

```
__attribute__((import_module("symbolic"), import_name("i8_symbol"))) char
↪  owi_i8(void);
__attribute__((import_module("symbolic"), import_name("i32_symbol"))) int
↪  owi_i32(void);
__attribute__((import_module("symbolic"), import_name("i64_symbol"))) long
↪  long owi_i64(void);
__attribute__((import_module("symbolic"), import_name("f32_symbol"))) float
↪  owi_f32(void);
__attribute__((import_module("symbolic"), import_name("f64_symbol"))) double
↪  owi_f64(void);
__attribute__((import_module("symbolic"), import_name("bool_symbol"))) _Bool
↪  owi_bool(void);

__attribute__((weak, import_module("symbolic"), import_name("assume"))) void
↪  owi_assume(int);
__attribute__((import_module("symbolic"), import_name("assert"))) void
↪  owi_assert(int);
```

Certaines fonctions doivent être marquées comme étant `weak` afin de ne pas créer des incompatibilités dans le cas où des noms d'attributs similaires existent.

### 12.1.3 Bibliothèque standard

Notre mise en œuvre de la bibliothèque standard est basée sur `dietlibc` [41]. Il s'agit d'une `libc` optimisée pour produire du code de petite taille. Elle présente l'avantage d'être bien plus courte (en termes de nombre de ligne de code) que les autres mises en œuvre de la `libc`, ce qui la rend plus facile à comprendre et à modifier.

### 12.1.4 Allocation

Nous ne pouvons pas directement utiliser la fonction `malloc` fournie par `dietlibc`. À la place nous avons modifié `stdlib.c` ainsi :

```
extern unsigned char __heap_base;
unsigned int bump_pointer = &__heap_base;

void *malloc(size_t size) {
  unsigned int start = bump_pointer;
  unsigned int closest_pow2 = 1 « (sizeof(size_t)*8 - (__builtin_clz(size) +
  ↪  1));
  unsigned int align = (closest_pow2 <= 16) ? closest_pow2 : 16;
  unsigned int off_align = bump_pointer % align;

  if ( off_align != 0 ) {
    start += (align - off_align);
  }
```

```
  bump_pointer = size + start;
  return (void *)owi_malloc(start, size);
}
```

Cela nous permet de correctement utiliser la mémoire linéaire tout en utilisant la fonction `owi\_malloc` exposée par Owi qui permet de détecter des classes de bogues supplémentaires, tels que les *use after free*, les *double free* ou encore de détecter proprement les *out of bounds access* plutôt que de renvoyer des valeurs fausses ou bien d'obtenir un `trap` à l'exécution sans plus d'informations sur la source de l'erreur.

### 12.1.5 Exemple des pointeurs de fonctions

Owi peut gérer toutes les fonctionnalités de Wasm (sauf les instructions SIMD) et donc des programmes C complexes. Par exemple, l'exemple de code C suivant illustre qu'Owi peut gérer les pointeurs de fonction dans l'exécution symbolique :

```c
#include <stddef.h>
#include <owi.h>

// fold(f, a, len, init) is f(f(… f(init, a[0]), a[1] …), a[len-1])
int fold(int (*f)(int, int), int *array, size_t len, int init) {
  for (size_t i = 0; i < len; i++) {
    init = f(init, array[i]);
  }
  return init;
}

// a function whose result is expected to be positive if acc is positive
int step(int acc, int x) {
  return (acc + x * x);
}

void main(void) {
  // Create an array of unknown (symbolic) size containing symbolic integers
  size_t len = owi_i32();
  int *array = malloc(sizeof(int) * len);
  for (size_t i = 0; i < len; i++) {
    array[i] = owi_i32();
  }

  // Apply the `fold` function on the symbolic array with `step` as a function
  ↪  pointer
  int init = 42;
  int res = fold(step, array, len, init);

  // Check that the result is greater than the initial value
```

```
  // This should be the case because the output of `step` is expected to be
  ↪  positive
  // when the accumulator is positive, which is the case of `init` here
  owi_assert(res >= init);

}
```

Bien que le code semble correct, Owi révèle un contre-exemple causé par un débordement de capacité dans la fonction `step` :

```
$ owi c function_pointer.c --fail-on-assertion-only -O0
Assert failure: (i32.ge (i32.add (i32 42) (i32.mul symbol_2 symbol_2)) (i32
↪  42))
Model:
  (model
    (symbol_0 (i32 1))
    (symbol_2 (i32 57344)))
Reached problem!
```

L'exécution du code en mode non optimisé (`-O0`) garantit que l'appel indirect n'est pas optimisé (Owi trouverait tout de même le bogue sans cela, mais il n'y aurait plus de référence de fonction dans le code Wasm). Nous utilisons l'option `--fail-on-assertion-only` pour concentrer l'analyse sur les violations d'assertion plutôt que sur d'autres bogues potentiels, tels que le dépassement des limites de taille de la mémoire de Wasm. Dans cet exemple, `step` est un pointeur concret. Cependant, s'il avait été symbolique, Owi aurait également été en mesure d'effectuer une exécution symbolique.

### 12.1.6 Vérification de l'équivalence de fonctions

De même, nous pouvons utiliser Owi pour vérifier l'équivalence de deux fonctions :

```
#include <owi.h>

// check that f1 and f2 produce the same result for any inputs
void check_function_equivalence(unsigned int (*f1)(unsigned int, unsigned int),
↪  unsigned int *(f2)(unsigned int, unsigned int)) {
  unsigned int x = owi_i32();
  unsigned int y = owi_i32();
  owi_assert(f1(x, y) == f2(x, y));
}

// two different implementations of the mean function
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) {
  check_function_equivalence(mean1, mean2);
}
```

Owi identifie que `mean1` et `mean2` ne sont pas équivalentes en raison d'un débordement de capacité dans `mean2` :

```
$ owi c ./function_equiv.c --fail-on-assertion-only -O0
Assert failure: (bool.eq (i32.add (i32.and symbol_0 symbol_1) (i32.shr_u
↪  (i32.xor symbol_0 symbol_1) (i32 1))) (i32.shr_u (i32.add symbol_0
↪  symbol_1) (i32 1)))
Model:
  (model
    (symbol_0 (i32 -922221680))
    (symbol_1 (i32 1834730321)))
Reached problem!
```

## 12.2 Vérification simultanée de code Rust et C

Nous décrivons dans cette section comment le support de Rust a été ajouté à Owi. Ensuite, nous cherchons à démontrer comment cela permet à Owi d'effectuer de l'exécution symbolique de code constitué de plusieurs langages sources.

### 12.2.1 Support de Rust dans Owi

Un travail similaire a celui effectué pour C a été effectué pour Rust. Pour le moment, seulement une partie de la bibliothèque standard été modélisée. Nous exposons ainsi l'interface d'Owi :

```
mod owi_sym {

    pub mod sys {
        #[link(wasm_import_module = "symbolic")]
        extern "C" {
            pub(super) fn i8_symbol() -> u8;
            pub(super) fn i32_symbol() -> u32;
            pub(super) fn assert(condition: bool);
            pub(super) fn assume(condition: bool);
        }

        #[link(wasm_import_module = "summaries")]
        extern "C" {
            pub fn alloc(base: *mut u8, size: u32) -> *mut u8;
            pub fn dealloc(base: *mut u8);
```

```
        }
    }

    pub fn stop_exploration() -> ! {
        unsafe {
            sys::assume(false);
            panic!("")
        }
    }

    pub fn u8_symbol() -> u8 { unsafe { sys::i8_symbol() } }
    pub fn u32_symbol() -> u32 { unsafe { sys::i32_symbol() } }
    pub fn assert(b: bool) { unsafe { sys::assert(b) } }
    pub fn assume(b: bool) { unsafe { sys::assume(b) } }
}
```

Nous pouvons alors utiliser Owi ainsi :

```
fn main()  {
    let x: i32 = owi_sym::u32_symbol() as i32;
    let y: i32 = owi_sym::u32_symbol() as i32;

    if (x > y) {
        x = x + y;
        y = x - y;
        x = x - y;
    }
    if (x - y > 0) {
        owi_sym::assert(false)
    }
}
```

Ce qui produit le résultat attendu :

```
$ cargo build --target wasm32-unknown-unknown
$ owi sym target/wasm32-unknown-unknown/release/rust-owi-opt.wasm
Assert failure: false
Model:
  (model
    (symbol_0 (i32 8388481))
    (symbol_1 (i32 -2147483648)))
Reached problem!
```

### 12.2.2 Vérification de l'équivalence de fonctions Rust et C

Dans un scénario où une base de code est en cours de migration de C vers Rust, considérons la fonction C suivante qui calcule le produit scalaire de deux vecteurs 2D :

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

Un développeur Rust pourrait traduire cette fonction en un code Rust idiomatique utilisant des itérateurs :

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

Pour comparer le comportement de la fonction C et son pendant en Rust, nous pouvons lier la fonction C dans Rust de la manière suivante :

```rust
extern "C" {
    pub fn dot_product(x: *const f32, y: *const f32) -> f32;
}

fn dot_product_c(x: &[f32; 2], y: &[f32; 2]) -> f32 {
    unsafe { dot_product(x.as_ptr(), y.as_ptr()) }
}
```

Ensuite, nous utilisons Owi pour vérifier que les deux mises en œuvre se comportent de manière équivalente :

```rust
fn main() {
    let x = std::array::from_fn(|_| owi_sym::f32_symbol());
    let y = std::array::from_fn(|_| owi_sym::f32_symbol());
    let c_val = dot_product_c(&x, &y);
    let rust_val = dot_product_rust(&x, &y);
    owi_sym::assert(
      (c_val.is_nan() && rust_val.is_nan()) ||
      (c_val.to_bits() == rust_val.to_bits())
    )
}
```

Étonnamment, Owi trouve un contre-exemple :

```
Model:
  (model
    (symbol_0 (f32 -0.))
    (symbol_1 (f32 -0.))
    (symbol_2 (f32 0.))
    (symbol_3 (f32 0.)))
```

Ce contre-exemple provient du comportement des zéros dans l'arithmétique à virgule flottante IEEE 754. Le tableau 12.1 illustre les règles d'addition.

Le véritable élément neutre de l'addition est $-0$ et non $+0$. Or, la mise en œuvre en Rust de la somme se comporte de manière similaire au code C suivant :

| + | +0 | -0 |
|---|---|---|
| +0 | +0 | +0 |
| -0 | +0 | -0 |

TABLE 12.1 – Règles de l'addition dans l'arithmétique à virgule flottante IEEE 754.

```
float sum(float *x, int n) {
    float sum = 0.0;
    for (int i = 0; i < n; i++) {
        sum += x[i];
    }
    return sum;
}
```

Dans cet exemple, additionner $-0$ à $+0$ conduit à des résultats différents : la somme en Rust donne $+0$, tandis que la fonction C aboutit à $-0$. Bien que $+0$ et $-0$ soient égaux lors de la comparaison, ils peuvent entraîner des différences significatives dans les calculs ultérieurs. Après avoir découvert ce problème, nous l'avons identifié comme un bogue dans la bibliothèque standard de Rust et avons soumis une *Pull Request* pour le corriger, qui a été acceptée et fusionnée [1].

### 12.2.3 Vérification de propriétés sur du code Rust appelant du code C

De nombreuses bibliothèques fondamentales sont écrites en C, couvrant divers domaines d'utilisation : cryptographie (NaCl et OpenSSL), réseau (libcurl), graphisme (GTK), formats de fichiers courants (git, zlib, gzip), etc. Les nouveaux langages de programmation exposent souvent des moyens de se lier à ces bibliothèques. Cependant, cela pose un difficulté pour la vérification, car la base de code mélange plusieurs langages de programmation. Étant donné qu'Owi utilise Wasm pour la vérification, il peut gérer du code où un langage (ici Rust) appelle un autre (ici C), à condition que les deux soient compilés vers Wasm.

Pour illustrer cela, nous avons écrit le code Rust suivant, qui appelle une primitive `sha512` de la populaire bibliothèque `libsodium`. Dans ce code, la fonction `user_entry` prend un tampon, un nom d'utilisateur et un mot de passe, puis écrit dans le tampon le nom d'utilisateur, un octet nul et le hachage sha512 du mot de passe.

```
extern "C" {
    fn c_sha512(out: *mut u8, msg: *const u8, bytes: u64) -> c_int;
}

fn hash(out: &mut [u8], msg: &[u8]) {
    let ret_code = unsafe {
      c_sha512(out.as_mut_ptr(),msg.as_ptr(),msg.len() as u64)
    };
    if ret_code != 0 { panic!("Hash failed!") }
}
```

1. https://github.com/rust-lang/rust/pull/129321

```
fn user_entry(mut output: &mut [u8], user_name: &str, password: &str) {
    use std::io::Write;
    write!(&mut output, "{}\0", user_name.to_uppercase()).unwrap();
    hash(output, password.as_bytes());
}
```

Pour le tester, nous utilisons la fonction principale suivante. Dans cette fonction, nous allouons un tampon suffisamment grand pour stocker quatre octets, l'octet nul et les 64 octets du hachage. Nous générons ensuite un nom d'utilisateur composé de quatre caractères et appelons la fonction `user_entry` avec ces paramètres.

```
let user_name_size = 4;
let mut buffer = vec![0u8; user_name_size + 1 + 64];
let user_name =
    String::from_iter(std::iter::repeat_with(||
    ↪  owi_sym::char_symbol()).take(user_name_size));
user_entry(buffer.as_mut_slice(), &user_name, "password");
```

Bien que nous n'ayons ajouté aucune assertion dans notre fonction principale, Owi trouve tout de même un problème :

```
Trap: memory heap buffer overflow
Model:
  (model
    (symbol_18 (i32 14))
    (symbol_19 (i32 8079))
    (symbol_20 (i32 0))
    (symbol_21 (i32 32)))
Reached problem!
```

En effet, cet exemple démontre également la gestion de la mémoire intégré d'Owi, et un dépassement de tampon a été correctement identifié. Cela est dû au fait que nous allouons un tampon de 69 octets, oubliant que les quatre caractères Unicode (comme donnés par `char_symbol`) peuvent occuper jusqu'à quatre octets chacun, soit seize octets au total. Ce dépassement de tampon ne peut pas être détecté par Rust lui-même (malgré ses propriétés de sûreté), car il se produit dans le code C appelé : le nom d'utilisateur et l'octet nul s'insèrent sans problème au début du tampon de 69 octets, mais c'est lors de l'écriture des 64 octets du hachage que le dépassement se produit.

## 12.3 Travaux connexes

### 12.3.1 Vérification de bases de code cross-langages

La vérification des bases de code cross-langages nécessite que les deux langages partagent une cible de compilation commune supportée par l'outil de vérification. Les langages assembleurs sont notoirement difficiles à vérifier, notamment parce que le flux de contrôle est souvent

complexe et difficile à reconstruire dans le code assembleur généré par les compilateurs. LLVM IR est la cible de certains outils de vérification (par exemple KLEE), qui pourrait être utilisé pour la vérification de code cross-langage, mais nous n'avons pas trouvé d'exemples dans la littérature. Une explication possible est le manque d'outils pour générer et manipuler facilement le bytecode LLVM. En revanche, Wasm est supporté par un nombre croissant de langages, et sa conception se prête bien à la vérification et à l'exécution symbolique. De plus, les bibliothèques standard et les runtimes des langages sont déjà distribués en Wasm, ce qui n'est pas le cas de LLVM IR.

# 13 Évaluation expérimentale

Dans ce chapitre, nous évaluons les performances et l'efficacité d'Owi sur des programmes Wasm et des programmes C du monde réel. Plus précisément, notre évaluation vise à répondre aux deux questions de recherche suivantes :

1. Quelles sont les performances d'Owi et comment se comparent-elles à celles d'autres outils d'exécution symbolique existants pour Wasm ?
2. Quelle est l'efficacité d'Owi dans la détection de bogues et comment se compare-t-elle à KLEE et Symbiotic ?

Notre banc d'essai se composait d'un serveur Ubuntu 22.04 avec un processeur AMD EPYC 7451 à 24 cœurs, offrant 48 threads et 128 Go de RAM. Owi a été compilé avec le compilateur OCaml 5.2.0 en utilisant l'optimiseur Flambda1. Pour le solveur de contraintes, tous les outils comparés ont utilisé le solveur SMT Z3 [62] version 4.13.0.

Pour chaque exécution d'Owi, nous avons utilisé l'option `-w24`, fixant ainsi le nombre de *workers* à 24. L'utilisation de ce nombre de *workers* est justifiée expérimentalement dans la section 13.2.

Le code de benchmarking, les scripts de reproductibilité, de génération de diagrammes, et de comparaisons entre outils sont disponibles dans le dépôt Git d'Owi [122].

## 13.1 Évaluation des performances sur du code Wasm

### 13.1.1 Protocole expérimental

Pour répondre à la première question de recherche nous utilisons un ensemble de programmes Wasm utilisé pour l'évaluation des performances dans des travaux antérieurs [137, 164]. Cet ensemble comprend 22 tests symboliques Wasm mettant en œuvre une structure de données d'arbres B [3], créée sur mesure [114]. Ces tests ont déjà été utilisés pour évaluer les performances de l'interpréteur symbolique dans l'article de Marques [137]. Tous les tests partagent le même modèle de code, mais varient en fonction du nombre de valeurs symboliques, certaines étant contraintes à être ordonnées. Nous désignons le nombre de valeurs symboliques ordonnées et non ordonnées par $n_o$ et $n_u$, respectivement.

Pour comparer Owi avec les travaux antérieurs, nous avons utilisé les trois seuls moteurs symboliques Wasm autonomes disponibles : WASP [137], SeeWasm [164] et Manticore [111].

TABLE 13.1 – Table d'accélération calculée avec $S_{tool} = \frac{T_{tool}}{T_{\texttt{Owi}-24}}$. Nous comparons Owi avec 24 *workers* (`Owi-24`) par rapport aux autres outils : Manticore, SeeWasm, WASP, et Owi avec un seul *worker* Owi (`Owi-1`). Chaque entrée indique le facteur par lequel `Owi-24` est plus rapide que l'interpréteur indiqué en haut de chaque colonne.

| $n_o$ | $n_u = 1$ | | | | | $n_u = 2$ | | | | | $n_u = 3$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $s_{\texttt{Owi}-24}$ | $S_{\texttt{Mcore}}$ | $S_{\texttt{SW}}$ | $S_{\texttt{WASP}}$ | $S_{\texttt{Owi}-1}$ | $s_{\texttt{Owi}-24}$ | $S_{\texttt{Mcore}}$ | $S_{\texttt{SW}}$ | $S_{\texttt{WASP}}$ | $S_{\texttt{Owi}-1}$ | $s_{\texttt{Owi}-24}$ | $S_{\texttt{Mcore}}$ | $S_{\texttt{SW}}$ | $S_{\texttt{WASP}}$ | $S_{\texttt{Owi}-1}$ |
| 2 | 1.0 | 17.2 | 2.5 | 0.4 | 0.6 | 1.0 | 122.2 | 15.8 | 1.4 | 1.5 | 1.0 | 635.7 | 64.5 | 5.0 | 5.7 |
| 3 | 1.0 | 33.1 | 5.3 | 0.5 | 0.6 | 1.0 | 281.3 | 33.2 | 2.4 | 2.9 | 1.0 | 842.7 | 89.7 | 7.8 | 8.9 |
| 4 | 1.0 | 48.3 | 10.0 | 0.9 | 0.9 | 1.0 | 444.1 | 52.6 | 3.7 | 4.9 | 1.0 | 844.3 | 101.6 | 9.1 | 10.8 |
| 5 | 1.0 | 75.7 | 13.4 | 1.4 | 1.6 | 1.0 | 589.2 | 69.0 | 5.0 | 7.2 | 1.0 | 772.3 | 92.7 | 10.5 | 12.2 |
| 6 | 1.0 | 134.6 | 20.3 | 1.7 | 2.2 | 1.0 | 647.0 | 88.1 | 5.9 | 9.1 | 1.0 | 688.8 | 98.8 | 12.3 | 13.1 |
| 7 | 1.0 | 170.5 | 35.3 | 2.5 | 3.4 | 1.0 | 626.7 | 96.9 | 6.1 | 9.8 | 1.0 | 629.6 | 94.0 | 16.6 | 14.0 |
| 8 | 1.0 | 201.0 | 40.5 | 2.9 | 4.1 | 1.0 | 597.1 | 95.3 | 6.9 | 10.7 | – | – | – | – | – |
| 9 | 1.0 | 212.2 | 47.2 | 3.4 | 5.1 | 1.0 | 564.0 | 89.1 | 6.8 | 11.2 | – | – | – | – | – |

### 13.1.2 Résultats

Le Tableau 13.1 montre l'accélération d'Owi avec 24 *workers* (`Owi-24`) par rapport aux autres outils : Manticore, SeeWasm et WASP, ainsi que par rapport à Owi exécuté avec un seul *worker* (`Owi-1`). Les tests varient à la fois en valeurs symboliques ordonnées ($n_o$ : de 2 à 9) et non ordonnées ($n_u$ : de 1 à 3). Pour chaque combinaison $(n_o, n_u)$, la table fournit le facteur d'accélération de `Owi-24` par rapport au temps d'exécution brute de chaque outil présenté dans la Table 13.2. L'accélération est calculée comme suit : $S = \frac{T_{tool}}{T_{\texttt{Owi}-24}}$, ce qui signifie que si $S$ est supérieur à 1, Owi était $S\times$ plus rapide que *tool*.

Notamment, comme le montre la Table 13.1, Owi-24 surpasse constamment les autres outils :

- Comparé à Manticore, Owi réalise une accélération allant de $17,2\times$ à $844,3\times$, avec une moyenne de $312,6\times$.
- Comparé à SeeWasm, Owi réalise une accélération allant de $2,5\times$ à $101,6\times$, avec une moyenne de $57,1\times$.
- Comparé à WASP, Owi réalise une accélération allant de $0,4\times$ à $16,4\times$, avec une moyenne de $4,1\times$.
- Comparé à Owi avec un seul *worker*, Owi réalise une accélération allant de 0,6× à 14,0×, avec une moyenne de 4,5×.

De plus, la Table 13.1 révèle trois observations mineures :

1. Pour $2 \leq n_o \leq 4$ et $n_u = 1$, `Owi-24` est plus lent que WASP et `Owi-1`. Ce ralentissement est causé par le surcoût de création et de destruction des *workers*, ainsi que par la synchronisation entre eux.
2. En général, l'accélération obtenue par `Owi-24` est plus importante par rapport à `Owi-1` que par rapport à WASP. Cela s'explique par le fait qu'Owi n'est pas optimisé pour des performances avec un seul *worker*, ce qui entraîne des performances parfois plus lentes en mode mono-*worker* par rapport à WASP, suggérant ainsi un potentiel d'amélioration.
3. L'accélération s'améliore avec l'augmentation du nombre de variables symboliques en raison de la croissance exponentielle du nombre des chemins explorés, ce qui profite à l'exploration parallèle d'Owi.

TABLE 13.2 – Temps mesurés pour Manticore ($T_{\texttt{Mcore}}$), SeeWasm ($T_{\texttt{SW}}$), WASP ($T_{\texttt{WASP}}$), Owi avec un seul *worker* Owi ($T_{\texttt{Owi}}$), et Owi avec 24 *workers* ($T_{\texttt{Owi24}}$) sur la suite de tests arbres B.

| $n_o$ | $n_u = 1$ | | | | | $n_u = 2$ | | | | | $n_u = 3$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $T_{\texttt{Mcore}}$ | $T_{\texttt{SW}}$ | $T_{\texttt{WASP}}$ | $T_{\texttt{Owi}}$ | $T_{\texttt{Owi24}}$ | $T_{\texttt{Mcore}}$ | $T_{\texttt{SW}}$ | $T_{\texttt{WASP}}$ | $T_{\texttt{Owi}}$ | $T_{\texttt{Owi24}}$ | $T_{\texttt{Mcore}}$ | $T_{\texttt{SW}}$ | $T_{\texttt{WASP}}$ | $T_{\texttt{Owi}}$ | $T_{\texttt{Owi24}}$ |
| 2 | 4.356 | 0.634 | 0.104 | 0.155 | 0.254 | 35.058 | 4.534 | 0.391 | 0.441 | 0.287 | 382.687 | 38.829 | 3.062 | 3.466 | 0.602 |
| 3 | 11.151 | 1.791 | 0.166 | 0.218 | 0.337 | 110.546 | 13.053 | 0.953 | 1.159 | 0.393 | 1,117.439 | 118.925 | 10.399 | 11.863 | 1.326 |
| 4 | 17.659 | 3.648 | 0.330 | 0.339 | 0.366 | 261.156 | 30.920 | 2.178 | 2.895 | 0.588 | 2,713.705 | 326.637 | 29.403 | 34.852 | 3.214 |
| 5 | 33.305 | 5.897 | 0.607 | 0.712 | 0.440 | 507.321 | 59.453 | 4.337 | 6.197 | 0.861 | 5,722.794 | 686.920 | 77.568 | 90.649 | 7.410 |
| 6 | 67.175 | 10.137 | 0.826 | 1.081 | 0.499 | 896.068 | 121.961 | 8.179 | 12.593 | 1.385 | 11,118.566 | 1,595.489 | 198.750 | 211.120 | 16.142 |
| 7 | 97.692 | 20.221 | 1.458 | 1.952 | 0.573 | 1,584.262 | 244.923 | 15.520 | 24.684 | 2.528 | 20,727.641 | 3,096.070 | 538.423 | 460.280 | 32.924 |
| 8 | 152.363 | 30.762 | 2.165 | 3.117 | 0.758 | 2,567.524 | 409.891 | 29.648 | 45.938 | 4.300 | – | – | – | – | – |
| 9 | 210.069 | 46.709 | 3.390 | 5.062 | 0.990 | 3871.196 | 611.926 | 46.851 | 76.744 | 6.865 | – | – | – | – | – |

## 13.2 Évaluation de l'efficacité dans la recherche de bogues dans du code C

### 13.2.1 Protocole expérimental

Pour répondre à la seconde question de recherche, l'ensemble de données utilisé est une collection de programmes C du monde réel, utilisée pour évaluer les outils de test logiciel dans la compétition Test-Comp [146].

L'édition 2024 de Test-Comp [146] comprend 11 042 tâches. La compétition comporte deux types de tâches de test : **(1)** *les tâches* Cover-Branches, dont l'objectif est de générer un ensemble de tests concrets qui couvrent le plus grand nombre possible de branches du programme, et **(2)** *les tâches* Cover-Error, dont l'objectif est de générer au moins un ensemble d'entrées qui conduit à un bogue dans l'exécution du programme. Nous nous concentrons sur les tâches Cover-Error, qui comprennent 1 217 tâches sur les 11 042. Cependant, nous avons exclu deux tâches : l'une contient du code assembleur X86 *inline*, et l'autre est du code invalide. Chaque tâche est un programme C unique.

Pour comparer Owi avec les travaux antérieurs, nous avons évalué les deux meilleurs outils d'exécution symbolique ayant participé à l'édition 2024 de Test-Comp : KLEE [60] et Symbiotic [123]. Owi a toujours été lancé avec l'option `-O3` (qui indique à `clang` quel niveau d'optimisation utiliser). Ce choix est justifié plus loin dans la section 13.2.5.

### 13.2.2 Résultats

Nous évaluons l'efficacité d'Owi dans la détection des bogues et la comparons à KLEE et Symbiotic en exécutant tous les outils sur les tâches Cover-Error de Test-Comp. Bien que Test-Comp impose une limite de 900 secondes par tâche, nous utilisons une limite de 30 secondes pour maintenir le temps total d'exécution en dessous de 8 heures (car ces tâches consomment beaucoup de mémoire). Cette limite de 30 secondes était tout de même suffisante pour que KLEE et Symbiotic résolvent plus de 80 % des tâches dans la compétition originale [1]. Les résultats sont résumés dans la Table 13.3.

1. https://test-comp.sosy-lab.org/2024/results/results-verified/META_Cover-Error.table.html#/quantile?selection=cpu

Table 13.3 – Tâches résolues par KLEE, Owi et Symbiotic sur les tests de la catégorie *Cover-Error* issus de Test-Comp.

| Tool | Reached | Timeout | Nothing | $\frac{\text{Nothing}}{\text{Reached+Nothing}}$ |
|---|---|---|---|---|
| KLEE | 782 | 368 | 65 | 7.67% |
| Owi | 676 | 539 | 0 | 0.00% |
| Symbiotic | 489 | 657 | 69 | 12.37% |

### Bogues détectés

Un bogue est considéré comme "détecté" lorsqu'un outil génère un ensemble concret d'entrées qui provoque l'atteinte d'une localisation du programme marquée comme défaillante, par exemple un `assert(false)`.

La colonne "Reached" dans la Table 13.3 montre que KLEE détecte le plus grand nombre de bogues, tandis qu'Owi en trouve 0,86 fois moins. KLEE bénéficie de diverses stratégies d'exploration, alors qu'Owi manque actuellement d'heuristiques d'exploration efficaces et explore les chemins au fur et à mesure qu'ils sont découverts. Cette différence stratégique a un impact significatif sur les performances des moteurs d'exécution symbolique, ce qui explique probablement l'écart de détection entre les deux outils. De plus, KLEE utilise le solveur SMT STP et non Z3, car ils ont remarqué qu'il offrait de bien meilleurs résultats pour l'exécution symbolique. Nous prévoyons d'intégrer le solveur STP à Owi à l'avenir.

Malgré cela, Owi surpasse Symbiotic, détectant 1,38 fois plus de bogues. Étant donné que KLEE et Symbiotic se sont classés respectivement 2ème et 3ème[2] parmi les 20 outils testés lors de Test-Comp 2024, les performances d'Owi démontrent que ses capacités de détection de bogues sont compétitives avec les outils de test de logiciels de haut niveau.

### Bogues non détectés

Un bogue est considéré comme non détecté lorsqu'un outil termine sans dépasser le temps imparti et sans produire d'entrée causant un bogue, même si un bogues est présent. Cela est appelé un faux négatif.

La colonne "Nothing" dans la Table 13.3 montre le nombre de faux négatifs rapportés par chaque outil, tandis que la dernière colonne indique le pourcentage de faux négatifs par rapport à toutes les réponses sans dépassement de temps (c'est-à-dire Reached + Nothing). Notamment, KLEE présente un taux de faux négatifs de 7,67%, et Symbiotic un taux de 12,37%. En revanche, Owi ne produit aucun faux négatif.

Cela s'explique par le fait que KLEE sous-approxime délibérément certaines fonctions de la bibliothèque standard C. Par exemple, lorsqu'un programme alloue de la mémoire avec une taille symbolique en utilisant `malloc`, KLEE fournit un seul bloc de mémoire concret [107]. Cette approche peut accélérer l'exécution et augmenter le nombre de bogues détectés dans le délai imparti — à condition que le bogue soit indépendant de la taille du bloc — mais elle introduit également des faux négatifs potentiels. Étant donné que Symbiotic utilise KLEE pour l'exécution symbolique, il souffre également des mêmes faux négatifs potentiels.

2. Deux variantes de FuseBMC se sont classées premières.

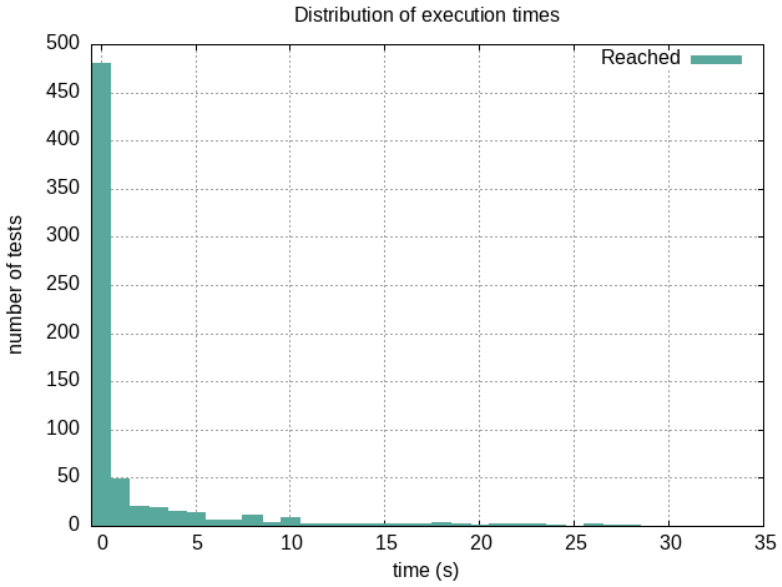



FIGURE 13.1 – Distribution des temps d'exécution d'Owi sur Test-Comp.

### 13.2.3 Distribution des temps d'exécution

Dans la figure 13.1, nous pouvons voir la distribution des temps d'exécution. Il apparaît que la plupart des problèmes sont résolus par Owi dans un très court laps de temps : plus de la moitié des problèmes dans la catégorie *reached* sont résolus en moins d'une seconde.

Le temps d'exécution moyen sur les tâches atteintes par les deux outils est de 1,43 secondes pour KLEE et de 2,92 secondes pour Owi. La distribution des temps d'exécution dans la Figure 13.2 est assez similaire à celle d'Owi.

### 13.2.4 Évaluation de la monade multi-cœur

Pour évaluer l'efficacité de la monade multicœur d'Owi, nous avons fait varier le nombre de *workers* de `-w1` à `-w48` et observé les résultats sur les benchmarks Cover-Error de Test-Comp, présentés dans le tableau suivant :

| Nombre de *workers* | 1 | 2 | 4 | 8 | 12 | 16 | 20 | **24** | 36 | 48 |
|---|---|---|---|---|---|---|---|---|---|---|
| Bogues trouvés | 612 | 645 | 663 | 664 | 675 | 675 | 675 | **676** | 675 | 672 |

Les résultats indiquent qu'à mesure que le nombre de *workers* augmente, le nombre de tâches résolues augmente également, atteignant un pic à 24 *workers*. Au-delà de ce point optimal, les performances diminuent légèrement, probablement en raison de l'augmentation de la contention du cache des *threads* virtuels partageant le même cœur logique. En effet, la machine sur laquelle les tests ont été effectués possède exactement 24 cœur logiques et 48 *threads*. Une comparaison des exécutions utilisant 1 et 24 *workers* montre que la version multicœur a résolu 62 tâches de plus, représentant une amélioration de 10%. La distribution des temps d'exécution est restée similaire dans toutes les configurations de *workers*.

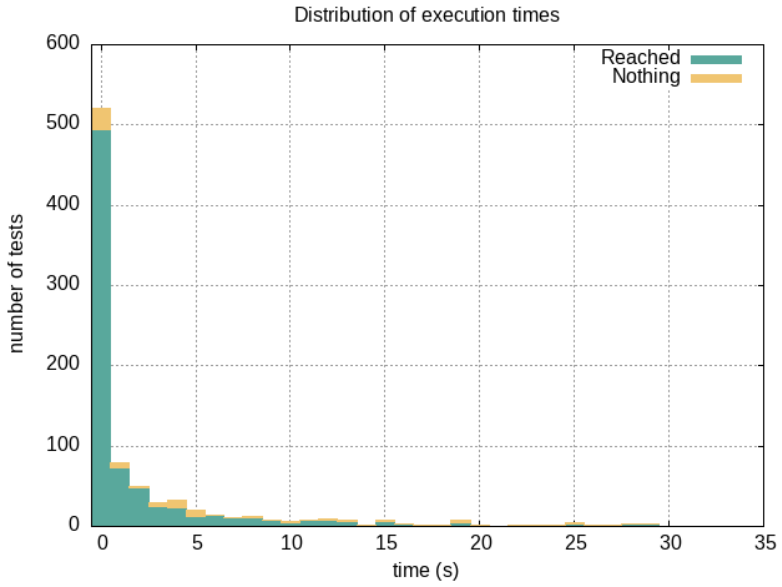



FIGURE 13.2 – Distribution des temps d'exécution de KLEE sur Test-Comp.

### 13.2.5 Niveaux d'optimisations du compilateur

Nous avons également expérimenté différents niveaux d'optimisation de `clang` (de `-O0` à `-O3`) pour évaluer leur impact sur les performances. Les résultats sur les benchmarks de Test-Comp sont résumés dans le tableau ci-dessous :

| Niveau d'optimisaiton de Clang | O0 | O1 | O2 | O3 |
|---|---|---|---|---|
| Bogues trouvés | 554 | 674 | 674 | **676** |

Au niveau d'optimisation `-O0`, Owi a résolu moins de tâches. Il semble que tous les exemples qui ne sont plus résolus sont liés à des boucles qui ne sont plus optimisées, ce qui conduit Owi à être bloqué lors de l'exploration. À partir de `-O1`, Owi a résolu un plus grand nombre de tâches, ce qui suggère que la plupart des optimisations conçues pour l'exécution concrète sont également de bonnes optimisations pour l'exécution symbolique, même en tenant compte du temps de compilation accru.

# 14 Owi comme moteur de programmation par contraintes

Jusqu'à présent, nous avons présenté Owi principalement comme un outil de détection de bogues dans les programmes. Ces bogues proviennent soient de `trap` indiquant des erreurs de programmation, soit de violations d'assertions explicitement définies par le programmeur. Dans ce cadre, les assertions servent à garantir que certains états du programme ne sont pas atteints, et les modèles produits par Owi correspondent aux entrées qui mènent à ces état indésirables.

En changeant de perspective, nous pouvons réinterpréter les assertions non plus comme une protection empêchant l'accès à des états indésirables, mais comme des moyens de s'assurer qu'un état spécifique est bien atteint. Les modèles, dans cette optique, deviennent des ensembles d'entrées qui conduisent à des états souhaités. En d'autres termes, Owi peut être utilisé pour résoudre des problèmes encodés sous forme de contraintes.

Il est important de noter que, si Owi n'arrive pas à trouver de solution, cela indique que le problème en question ne possède effectivement pas de solution. Dans ce chapitre, nous illustrons cette capacité de résolution de contraintes d'Owi à travers plusieurs exemples concrets.

## 14.1 Résolution de polynôme

Considérons le fichier `poly.c` suivant :

```c
#include <owi.h>

int main() {
  int x = owi_i32();
  int x2 = x * x;
  int x3 = x * x * x;

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

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

  owi_assert(poly != 0);
```

```
  return 0;
}
```

Dans cet exemple, nous définissons une variable symbolique `x` à l'aide de la fonction `owi_i32(void)`. Nous construisons ensuite le polynôme `poly`, qui correspond à l'expression $x^3 - 7x^2 + 14x - 8$. En utilisant l'assertion `owi_assert(poly != 0)`, nous cherchons à trouver une racine du polynôme. Ici, l'assertion fonctionne comme une contrainte à résoudre : Owi interprète cette contrainte comme une "erreur" à éviter, et va donc chercher à prouver que l'assertion est fausse. Ce processus conduit à la résolution de l'équation et à la découverte d'une racine du polynôme, comme le montre l'exemple suivant :

```
$ owi c ./poly.c -w1
...
Model:
  (model
    (symbol_0 (i32 4)))
Reached problem!
```

Dans ce cas, $4$ est effectivement une racine du polynôme, confirmant que le polynôme vaut $0$ pour cette valeur de $x$. Pour explorer d'autres racines, nous pouvons forcer Owi à exclure cette solution. Cela peut être fait en utilisant la fonction `owi_assume(bool)`, comme dans le code suivant :

```
#include <owi.h>

int main() {
  int x = owi_i32();
  int x2 = x * x;
  int x3 = x * x * x;

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

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

  owi_assume(x != 4); // exclusion de la solution précédente

  owi_assert(poly != 0);

  return 0;
}
```

En lançant Owi sur cette nouvelle version du programme :

```
$ owi c ./poly.c
...
Model:
  (model
```

```
    (symbol_0 (i32 2)))
Reached problem!
```

Owi trouve bien cette fois une deuxième racine du polynôme : 2.

**Exercice 1** Ce polynôme étant de degré 3, pensez-vous qu'il existe une troisième racine qu'Owi peut décrouvrir ? Si oui, laquelle, et comment procéder pour la trouver ?

**Exercice 2** Étonnamment, bien que ce polynôme soit de degré 3, il possède en réalité au moins cinq racines. Trouvez deux autres racines supplémentaires et expliquez l'origine de ces deux solutions.

## 14.2 Trouver la sortie d'un labyrinthe

Cet exemple est repris d'un article de blog[1] par Felipe Andres Manzano [66], dont nous recommandons vivement la lecture. L'idée est de représenter un labyrinthe sous la forme d'un tableau de caractères :

```
char maze[H][W] = {
  "+-+---+---+",
  "|X|     |#|",
  "| | --+ | |",
  "| |   | | |",
  "| +-- | | |",
  "|     |   |",
  "+-----+---+"
};
```

Dans ce labyrinthe, la position de départ est marquée par le caractère `X`, et la sortie est indiquée par `#`. L'utilisateur peut se déplacer à l'aide des touches de direction (`w`, `a`, `s` et `d`) pour avancer dans le labyrinthe et atteindre la sortie. Cependant, il lui est interdit de traverser des murs ou de repasser sur des positions déjà visitées.

Plutôt que de faire ces déplacements manuellement, Owi peut être utilisé pour automatiser la recherche de la sortie. Le programme suivant encode cette logique :

```
#include <owi.h>

// example from https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

#define H 8
#define W 11
#define ITERS 40

char maze[H][W] = {
```

1. https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

```
  "+-+---+---+",
  "| |     |#|",
  "| | --+ | |",
  "| |   | | |",
  "| +-- | | |",
  "|     |   |",
  "|     |   |",
  "+-----+---+"
};

int main (void) {

  // initial player position
  int x = 1, y = 1;

  // creating a symbolic array corresponding to the user inputs on its
  ↪  keyboard
  char program[ITERS];
  for (int i = 0; i < ITERS; i++) { program[i] = owi_i32(); }

  // iterates over the user input
  for (int i = 0; i < ITERS; i++) {

    // saving user position
    int old_x = x, old_y = y;

    // moving the player according to one input
    switch (program[i]) {
      case 'w': y--; break;
      case 's': y++; break;
      case 'a': x--; break;
      case 'd': x++; break;
      default: return 1; }

    // the player found the exit!
    if (maze[y][x] == '#') { owi_assert(0); }

    // fail if the player goes to an invalid position or could not move
    if ((maze[y][x] != ' ') || (old_x == x && old_y == y)) { return 1; }

    // mark the position as visited
    maze[y][x] = 'X';
  }
  return 1; }
```

Dans ce programme, Owi génère un ensemble de déplacements possibles, puis teste ces

séquences pour voir si elles mènent à la sortie. Owi est capable de trouver une première solution, qui, une fois traduite en séquence de caractères donne : `"ssssdsdwddwwaawwddddssssddwwww"`.

**Exercice 1** Vérifiez que la séquence de caractères donnés par Owi est bien une solution.

**Exercice 2** Générez une deuxième solution, traduire le modèle donné par Owi en séquence de caractères et vérifiez qu'il s'agit bien d'une solution.

**Exercice 3** Combien de solutions différentes existe-t-il ?

## 14.3 Dobble

Cet exemple repose sur le jeu Dobble[2]. Il a été proposé par Arthur Carcano dans le cadre de tests de performance sur une mise en œuvre symbolique de la fonction `popcount`, que nous décrirons plus loin. Le jeu Dobble nécessite de disposer d'un certain nombre `N_CARDS` de cartes, où chaque carte comporte un nombre `CARD_SIZE` de symboles. Les contraintes que doivent respecter ces cartes sont les suivantes :

- chaque carte doit contenir des symboles différents (*i.e.* il n'existe pas deux symboles identiques sur une même carte) ;
- deux cartes quelconques doivent avoir exactement un symbole en commun.

Le but du jeu est de trouver ce symbole commun entre deux cartes, selon les différentes variantes de Dobble.

Nous allons utiliser Owi pour générer un ensemble de cartes respectant ces contraintes. Chaque carte est représentée par un entier où chaque bit de cet entier indique la présence ou non d'un symbole spécifique sur la carte. Par exemple, si le troisième bit d'un entier est à 1, cela signifie que le troisième symbole est présent sur cette carte.

L'instruction `popcount` (qui compte le nombre de bits à 1 dans un entier) est incluse dans l'ensemble d'instructions de Wasm. Cependant, elle n'est pas encore prise en charge dans *Smt.ml*, ce qui nous empêche de l'utiliser directement. Ainsi, lors de la compilation du code C vers Wasm, nous devons utiliser l'option `-O1` pour éviter que Clang génère du code Wasm contenant cette fonction. Nous devons donc écrire la fonction `popcount` en C, bien que cela soit moins efficace.

Voici le code complet :

```c
// An encoding representing the problem of finding a suitable set of cards for
↪  Dobble. Cards are encoded on integers, with each position representing one
↪  of the possible symbols.
#include <owi.h>
#include <stdlib.h>

// Dobble is actually a finite projective plan where each card is represented
↪  by a line and each symbol by a point where two lines intersect.

// the *order* of the projective plane
```

2. https://fr.wikipedia.org/wiki/Dobble

```c
#define N 2
// the number of symbols per card = the number of points on each line = the
↪  number of lines through each point = n + 1 = 3
#define CARD_SIZE (N + 1)
// the number of cards = the number of lines = the number of points = N*N + N
↪  + 1 = (N+1)^2 - N + 1 = CARD_SIZE^2 - CARD_SIZE + 1 = 7
#define N_CARDS ((CARD_SIZE * CARD_SIZE) - CARD_SIZE + 1)

int popcount(unsigned int x) {
  int count = 0;
  while (x != 0) {
    count += x & 1;
    x »= 1;
  }
  return count;
}

int main() {
  unsigned int cards[N_CARDS];
  for (int i = 0; i < N_CARDS; i++) {
    unsigned int x = owi_i32();
    owi_assume((x » N_CARDS) == 0);
    owi_assume(popcount(x) == CARD_SIZE);
    cards[i] = x;
    if (i > 0) {
      owi_assume(cards[i] > cards[i-1]);
    }
  }
  unsigned int acc = 1;
  for (int i = 0;i < N_CARDS; i++) {
    for(int j = i + 1; j < N_CARDS;j++) {
      owi_assume(cards[i] != cards[j]);
      unsigned int z = cards[i] & cards[j];
      acc = acc & (z != 0); // the two cards have one common symbol
      acc = acc & ((z & (z-1)) == 0); // and at most one
    }
  }
  owi_assert(!acc);
}
```

L'utilisation de la fonction `owi_assume` permet de contraindre Owi à respecter certaines conditions. Par exemple, dans la première boucle, `owi_assume(popcount(x) == CARD_SIZE)` force Owi à trouver des cartes ayant exactement `CARD_SIZE` symboles. Owi génère un modèle contenant les cartes suivantes :

```
Assert failure:  …
Model:
  (model
    (symbol_0 (i32 14))
    (symbol_1 (i32 21))
    (symbol_2 (i32 35))
    (symbol_3 (i32 56))
    (symbol_4 (i32 73))
    (symbol_5 (i32 82))
    (symbol_6 (i32 100)))
Reached problem!
```

Le premier symbole s'écrit $0001110$ en base 2. Cela signifie que la première carte contient les deuxième, troisième et quatrième symboles.

**Exercice 1** Écrivez la représentation en base 2 de toutes les cartes. Vérifiez que chaque paire de carte a exactement deux symboles en commun.

**Exercice 2** En enlevant des cartes du paquet, nous pouvons toujours jouer au jeu. Pour une solution donnée de 7 cartes, trouvez combien de sous-ensembles d'au moins deux cartes permettent de jouer au jeu ?

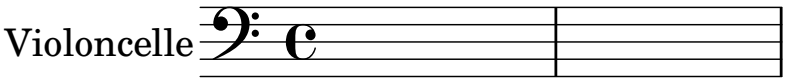
## 14.4 Harmonisation à quatre voix

Dans cette section, nous présentons un exemple d'utilisation d'Owi pour la génération de partitions respectant un ensemble de règles d'harmonie musicale. Notre but est de créer une partition pour un quatuor à cordes, en suivant les règles de base de l'harmonie, telles qu'elles sont généralement enseignées dans les conservatoires, cours de musicologie, et traités d'harmonie.

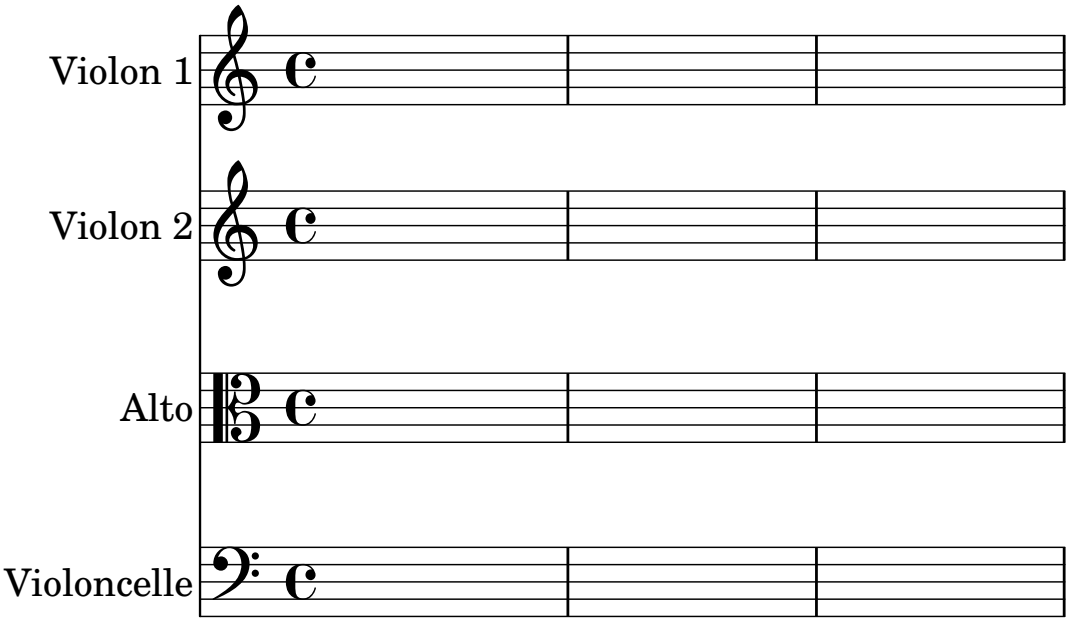
Un quatuor à cordes est composé de quatre instruments à cordes, le plus souvent il s'agira de deux violons, d'un alto et d'un violoncelle. C'est exactement ceux nous utiliserons ici pour générer quatre voix distinctes.

**La notion de voix** Une voix est une succession de notes formant une mélodie. Dans ce contexte, chaque instrument du quatuor joue une seule voix, ce qui les rend monodiques. Cela signifie qu'à tout moment, un instrument ne joue qu'une seule note, par opposition à des instruments polyphoniques tels que le piano ou la guitare, capables de jouer plusieurs notes simultanément. Bien que les instruments à cordes comme le violon puissent techniquement jouer plusieurs voix à la fois, nous nous limiterons ici à une écriture monodique. Ainsi, chaque instrument représentera une et une seule voix indépendante.

**La portée** Les notes jouées par chaque instrument sont représentées sur une portée, un ensemble de lignes horizontales permettant de visualiser la hauteur des notes. Chaque portée commence par une clé qui associe les notes aux lignes. Par exemple, la voix du violoncelle est généralement notée avec une clé de fa notée 𝄢 :




La signature rythmique **C** indique ici une mesure à quatre temps (*i.e.* pulsation), avec chaque temps d'une durée de noire. Notre partition contiendra donc quatre portées, chacune correspondant à un des instruments du quatuor, et chaque portée comportera trois mesures. La structure de base de la partition ressemblera à ceci :




**Contraintes harmoniques** Nous allons à présent introduire plusieurs contraintes basées sur des règles d'harmonie, ou des versions simplifiées de celles-ci, afin de faciliter leur mise en œuvre. Voici la liste de ces contraintes, sans les expliquer en détail :

- le violoncelle ne doit jamais jouer plus haut que l'alto, l'alto ne doit pas dépasser le violon 2, et le violon 2 doit rester en dessous du violon 1 ;
- aucun instrument ne doit se déplacer de plus d'une octave d'une pulsation à l'autre ;
- chaque note jouée par les instruments doit appartenir à la tonalité ;
- la sensible doit se résoudre sur la tonique (*i.e.* si un instrument joue la septième note de la tonalité, il doit jouer la première note de la tonalité à la pulsation suivante) ;
- à chaque pulsation, les notes jouées par les différents instruments doivent former un accord, composé de la basse, la tierce et la quinte ;
- si l'intervalle entre deux instruments à une pulsation donnée est une quinte, il est interdit qu'ils forment une autre quinte à la pulsation suivante (afin d'éviter les quintes parallèles) ;
- de manière similaire, les octaves parallèles sont également interdites.

**Représentation des notes** Pour chaque instrument, nous allons générer une note par pulsation, et chaque note sera définie par deux paramètres : un numéro de note et une octave.

Il existe douze notes différentes, comme do, ré, mi, fa, sol, la, et si, ainsi que leurs variantes altérées (dièses ou bémols). Par exemple, do3 représente un do joué dans la troisième octave.

Les notes seront représentées par deux entiers : le premier (de 0 à 11) indiquera la note et le second (de 0 à 2) indiquera l'octave. La fonction Wasm suivante génère une note valide :

```
(func $gen_note (result i32 i32)

  (local $note i32) (local $octave i32)

  ;; creating two symbols for the note
  (local.set $note (call $symbol))
  (local.set $octave (call $symbol))

  ;; putting constraint on the note so that it is a valid note
  (call $assume (i32.le_u (local.get $note) (i32.const 11)))
  ;; putting constraint on the octave so that it is not too high
  (call $assume (i32.le_u (local.get $octave) (i32.const 2)))

  ;; returning the new note
  (local.get $note) (local.get $octave))
```

Dans cette approche, nous devons générer 96 symboles au total, correspondant à quatre instruments, chacun jouant trois mesures, avec quatre notes par mesure (chaque note nécessite deux symboles).

**Gestion des notes en mémoire** Nous utilisons trois fonctions pour écrire et lire les notes et les octaves dans la mémoire du programme. La fonction `$offset` calcule la position mémoire pour un instrument à une pulsation donnée :

```
;; compute the offset where a note for a given time/instrument is stored in
↪  memory
(func $offset (param $instrument i32) (param $time i32) (result i32)
  (i32.mul (i32.const 8)
    (i32.add
      (i32.mul (local.get $time) (global.get $nb_instr))
      (local.get $instrument))))
```

La fonction `$set_note` enregistre une note et une octave pour un instrument à une pulsation donnée :

```
;; set a note in memory for a given instrument and time
(func $set_note (param $instrument i32) (param $time i32) (param $note i32)
↪  (param $octave i32)
   (local $pos i32)
   (local.set $pos (call $offset (local.get $instrument) (local.get $time)))
   (i32.store (local.get $pos) (local.get $note))
   (i32.store (i32.add (i32.const 4) (local.get $pos)) (local.get $octave)))
```

Les fonctions `$get_note` et `$get_octave` récupèrent respectivement la note et l'octave à partir de la mémoire pour un instrument à une pulsation donnée :

```
;; get a note in memory for a given instrument and time
(func $get_note (param $instrument i32) (param $time i32) (result i32)
   (local $pos i32)
   (local.set $pos (call $offset (local.get $instrument) (local.get $time)))
   (i32.load (local.get $pos)))

;; get the octave in memory for a given isntrument and time
(func $get_octave (param $instrument i32) (param $time i32) (result i32)
   (local $pos i32)
   (local.set $pos (call $offset (local.get $instrument) (local.get $time)))
   (i32.load (i32.add (i32.const 4) (local.get $pos))))
```

Nous utilisons une variable globale `$length` indiquant le nombre de pulsations que l'on souhaite avoir. Ces fonctions permettent alors de générer des notes aléatoires pour chaque instrument et chaque pulsation :

```
;; generate a random note for a single instrument and store it in memory
(func $init_note (param $instrument i32) (param $time i32)
 (local $note i32)
 (local $octave i32)
 (call $gen_note)
 (local.set $octave)
 (local.set $note)
 (call $set_note (local.get $instrument) (local.get $time)
                 (local.get $note) (local.get $octave))
)

;; generate a random note for each instrument and store it in memory
(func $init_time (param $time i32)
 (call $init_note (i32.const 0) (local.get $time))
 (call $init_note (i32.const 1) (local.get $time))
 (call $init_note (i32.const 2) (local.get $time))
 (call $init_note (i32.const 3) (local.get $time))
)

;; generate all random notes
(func $init_length
(local $time i32)
(loop $loop
  (call $init_time (local.get $time))
  (local.set $time (i32.add (local.get $time) (i32.const 1)))
  (br_if $loop (i32.lt_u (local.get $time) (global.get $length)))
))
```

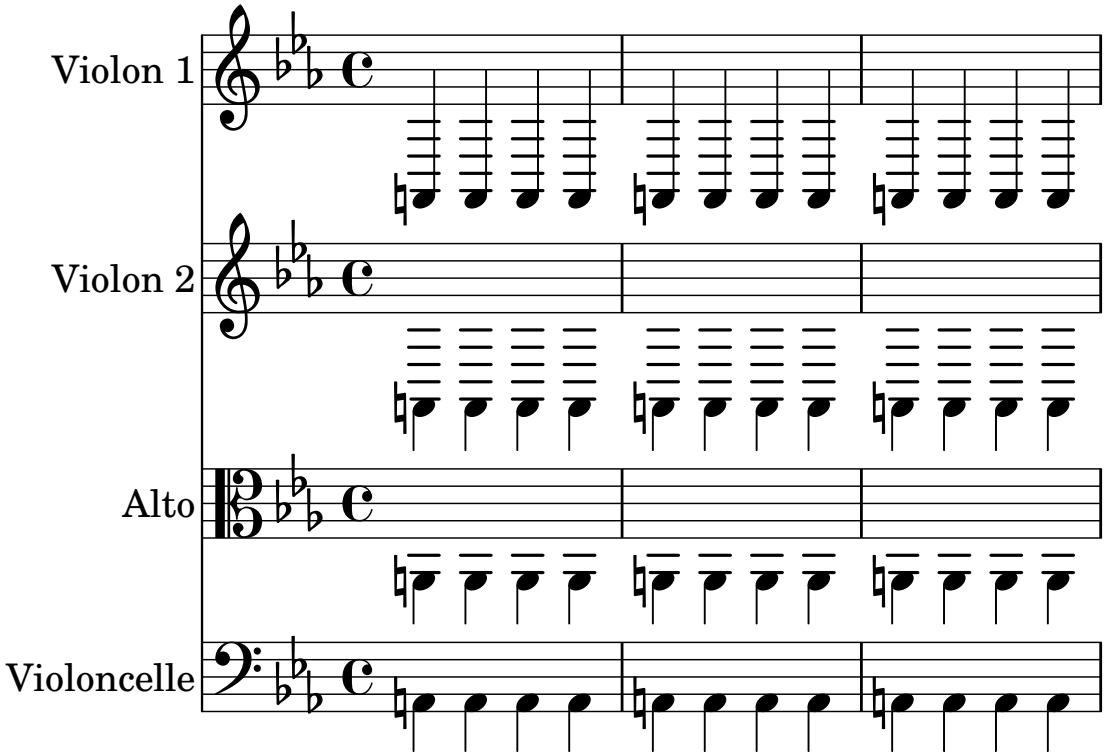
En utilisant la fonction `$init_length` suivie d'une instruction `unreachable`, nous pouvons déjà générer un modèle avec Owi. Ce modèle contient toutes les informations nécessaires à la création de la partition souhaitée, mais reste difficilement exploitable en l'état. C'est pourquoi nous avons développé un programme en OCaml pour analyser ce modèle, puis générer un fichier LilyPond à partir duquel nous pourrons produire une partition lisible et un fichier MIDI.

Nous rajoutons aussi un symbole pour la tonalité. Celui-ci ne sert pas actuellement à la génération, mais sera utilisé par la suite. Le rajouter dès maintenant nous permet de partager le code OCaml entre les deux versions. Nous fixons la tonalité à do mineur. Il n'est pas nécessaire de détailler ici le programme OCaml utilisé pour générer le fichier LilyPond à partir du modèle. Il est fourni uniquement pour assurer la reproductibilité des explications qui suivent. Le code complet est disponible en annexe C.

Une fois cette étape accomplie, la partition peut être générée sous forme de fichier PDF et le fichier MIDI créé grâce aux commandes suivantes :

```
$ owi sym ./gen.wat > model.txt
$ ocaml gen.ml model.txt
$ lilypond ./out.ly
$ evince ./out.pdf # open the pdf
$ timidity ./out.midi # play the midi file
```

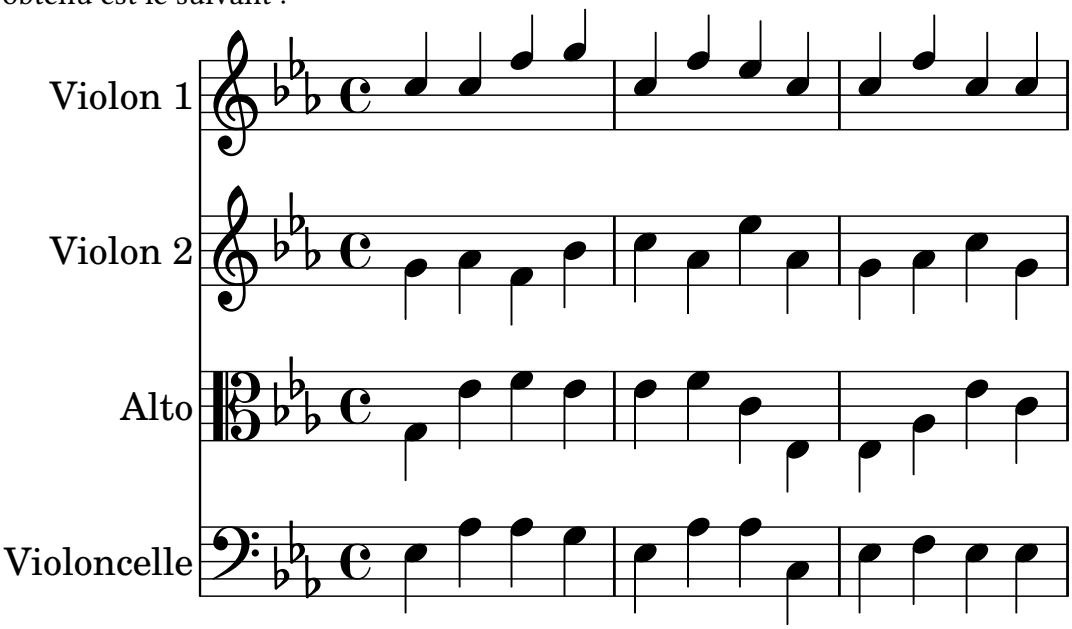
Ainsi, nous obtenons une partition générée automatiquement. Le code Wasm complet est donné en annexe A. Voici le résultat obtenu :




Le résultat peut être écouté en cliquant-droit sur l'icône suivante avec un lecteur PDF tel qu'Evince, en enregistrant le fichier, puis en l'ouvrant avec un lecteur compatible avec le format MP3 : ▶️.

Le code complet intégrant les contraintes est fourni en annexe (voir B). Le résultat final

obtenu est le suivant :




Le résultat peut être écouté en cliquant-droit sur l'icône suivante avec un lecteur PDF tel qu'Evince, en enregistrant le fichier, puis en l'ouvrant avec un lecteur compatible avec le format MP3 : ▶️.

Bien que le résultat obtenu ne soit pas encore entièrement satisfaisant, il est déjà nettement meilleur que celui produit sans contraintes. L'encodage manuel de toutes ces contraintes peut s'avérer laborieux, surtout lorsqu'il est effectué directement en Wasm. Toutefois, il est envisageable d'obtenir un résultat bien plus abouti en utilisant un langage de plus haut niveau, permettant une expression plus simple des contraintes.

**Exercice** Générez votre propre partition pour un quatuor à cordes. Observez la partition produite et écoutez l'enregistrement correspondant.

# 15 Vérification et génération d'assertions (symboliques) à partir de spécifications formelles

Dans ce chapitre, nous décrivons comment un langage de spécification existant pour le langage C peut être utilisé pour effectuer de l'exécution symbolique à partir de ces spécifications. Nous montrons comment l'exécution symbolique permet d'améliorer la génération d'assertions. Enfin, nous introduisons Weasel, un langage de spécification que nous avons conçu pour Wasm, et expliquons comment il est utilisé pour générer des assertions exécutables symboliquement. Ce travail a été réalisé dans le cadre du stage de recherche de Zhicheng Hui, étudiant en deuxième année du cycle ingénieur à l'École Polytechnique, que j'ai encadré pendant trois mois.

## 15.1 Utilisation d'E-ACSL pour la vérification d'assertions dans du code C

### 15.1.1 Introduction à ACSL et E-ACSL

ACSL [168] est un langage de spécification pour le C, tandis qu'E-ACSL [95] est un outil qui génère des assertions exécutables à partir d'un sous-ensemble d'ACSL. Ces deux outils sont intégrés dans Frama-C [74]. E-ACSL prend en entrée un fichier C annoté avec des spécifications et génère un fichier C instrumenté. Ce fichier instrumenté contient des assertions correspondant aux spécifications et qui vont interrompre l'exécution si elles ne sont pas respectées.

Prenons l'exemple d'une fonction incorrecte `primes`, qui met en œuvre l'algorithme du crible d'Ératosthène pour identifier les nombres premiers inférieurs à un entier donné $n$ :

```c
#include <stdbool.h>

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) { is_prime[i * j] = false; } } }
```

Cette fonction marque d'abord tous les nombres comme premiers, puis élimine les multiples de chaque nombre premier. Si un nombre reste marqué comme premier, alors il est effectivement premier.

Pour vérifier cette fonction, nous l'annotons avec des spécifications en utilisant E-ACSL. Les annotations sont placées directement au-dessus de la fonction, dans des commentaires :

```
#define MAX_SIZE 100

/*@
  requires 2 <= n <= MAX_SIZE;
  requires \valid(is_prime + (0 .. (n - 1)));
  ensures
    \forall integer i; 0 <= i < n ==>
    (is_prime[i] <==> (i >= 2 && \forall integer j; 2 <= j < i ==> i % j !=
    ↪  0));
*/
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) { is_prime[i * j] = false; } } }
```

Les préconditions (via `requires`) et postconditions (via `ensures`) spécifient que :

— $n$ doit être compris entre 2 et `MAX_SIZE` ;
— chaque élément du tableau `is_prime` doit être accessible de manière sûre (au sens des lectures et écritures dans la mémoire) ;
— chaque élément du tableau indique effectivement si l'entier correspondant à son indice est un nombre premier.

Une fois l'instrumentation effectuée, des assertions sont générées automatiquement à partir des spécifications. Toutefois, celles-ci ne seront vérifiées que sur une exécution concrète, ce qui limite les garanties sur la correction du programme.

### 15.1.2 Exécution symbolique de code instrumenté par E-ACSL

Les programmes instrumentés par E-ACSL sont généralement exécutés avec des valeurs concrètes, mais il est possible de combiner cette approche avec l'exécution symbolique pour vérifier des assertions sur des valeurs symboliques, augmentant ainsi la couverture des chemins d'exécution.

Nous avons ajouté une option `--e-acsl` à Owi, permettant de traiter un fichier instrumenté E-ACSL comme un fichier C classique. Considérons maintenant un exemple avec un harnais de test symbolique :

```
#define MAX_SIZE 100

#include <owi.h>
#include <stdlib.h>
```

```
/*@
  requires 2 <= n <= MAX_SIZE;
  requires \valid(is_prime + (0 .. (n - 1)));
  ensures
    \forall integer i; 0 <= i < n ==>
    (is_prime[i] <==> (i >= 2 && \forall integer j; 2 <= j < i ==> i % j !=
    ↪  0));
*/
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) { is_prime[i * j] = false; } } }

int main(void) {
  bool *is_prime = malloc(MAX_SIZE * sizeof(int));

  int n = owi_i32();
  owi_assume(n >= 2 && n <= MAX_SIZE);
  primes(is_prime, n);
  free(is_prime);
  return 0; }
```

L'exécution de ce programme avec Owi génère un modèle symbolique où une assertion échoue pour $n = 2$ :

```
$ owi c --e-acsl primes.c
Assert failure: false
Model:
  (model
    (symbol_0 (i32 2)))
Reached problem!
```

Le problème vient du fait que $0$ et $1$ ne sont pas marqués comme non-premiers. Nous corrigeons la fonction :

```
void primes(bool *is_prime, int n) {
  for (int i = 0; i < n; ++i) { is_prime[i] = true; }
  is_prime[0] = is_prime[1] = false; // fix the function
  for (int i = 2; i * i < n; ++i) {
    if (!is_prime[i]) continue;
    for (int j = i; i * j < n; ++j) { is_prime[i * j] = false; } } }
```

Après correction, toutes les assertions sont respectées :

```
$ owi c --e-acsl primes2.c
All OK
```

Cette approche présente plusieurs avantages :

— elle permet de réutiliser du code annoté sans avoir à écrire d'assertions spécifiques pour Owi ;
— elle rend possible la vérification d'une nouvelle base de code sans avoir à intégrer des assertions dans le programme ;
— elle permet d'être plus expressive : il est parfois plus agréable d'écrire des spécifications que des assertions ;
— elle facilite la génération automatique de harnais de test à partir des spécifications ;
— elle ouvre la voie à la vérification fonction par fonction dans Owi, au lieu d'analyser tout un programme à la fois.

Enfin, pour rendre E-ACSL compatible avec l'exécution symbolique, nous avons dû adapter son environnement d'exécution. En particulier, nous avons dû adapter certains composants, comme la gestion de la mémoire et les fonctions d'assertions, pour réutiliser les mécanismes d'Owi. Nous avons même découvert et corrigé des bogues dans le code d'E-ACSL, notamment liés aux fonctions variadiques et à des suppositions sur leur représentation en mémoire une fois compilée (qui n'étaient pas vraies en ciblant Wasm).

Ainsi, nous avons pu exécuter symboliquement toutes les constructions supportées par E-ACSL.

### 15.1.3 Utilisation de symboles pour améliorer l'instrumentation

Prenons l'exemple de l'annotation suivante qui vérifie que les dix premiers éléments d'un tableau sont inférieurs ou égaux à 100 :

```
//@ assert \forall int i; 0 <= i <= 10 ==> a[i] <= 100;
```

E-ACSL génère le code C suivant :

```
int __gen_e_acsl_forall = 1;
int __gen_e_acsl_i = 0;
__e_acsl_assert_data_t __gen_e_acsl_assert_data = {.values = (void *)0};
while (1) {
  if (__gen_e_acsl_i <= 10) ; else break;
  {
    int __gen_e_acsl_valid_read;
    // …
    if (*(a + __gen_e_acsl_i) <= 100) ;
    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,& __gen_e_acsl_assert_data);
```

Ce code est nécessaire pour l'exécution concrète. Toutefois, avec l'exécution symbolique, nous pouvons optimiser le code généré en introduisant un symbole pour la variable `i` et encoder cette spécification de manière plus efficace. Le code résultant pourrait alors être simplifié ainsi :

```
int __e_acsl_i = owi_i32();
if (0 <= i && i <= 10) {
  owi_assert(a[i] <= 100);
}
```

### 15.1.4 Extension d'E-ACSL aux quantificateurs non bornés via des symboles

Étant donné qu'E-ACSL génère du code destiné à une exécution concrète, il est limité à un sous-ensemble d'ACSL, notamment avec des restrictions sur les quantificateurs qui doivent être bornés. L'exécution symbolique permet de lever certaines de ces contraintes. Par exemple, la spécification suivante vérifiant qu'un entier `x` du programme est impair :

```
//@ assert \forall int i; x != 2*i;
```

n'est pas gérée par E-ACSL. Celui-ci émet un avertissement :

```
Warning:
E-ACSL construct `unguarded \forall quantification' is not yet supported.
Ignoring annotation.
```

Cependant, elle pourrait être traduite en utilisant un symbole :

```
int __e_acsl_i = owi_i32();
owi_assert(x != 2*i);
```

Pour effectuer cette transformation dans le cas général, il faut correctement gérer les quantificateurs existentiels ainsi que les quantifications imbriquées. Pour ce faire, il faut suivre plusieurs étapes :

- convertir les propositions en forme normale prénexe ;
- éventuellement effectuer une skolémisation ;
- associer chaque quantificateur à un symbole du type idoine ;
- encoder la proposition interne sans quantificateur ;
- ajouter des fonctions comme `model_exist(var, prop)` et `model_forall(var, prop)` pour itérer sur les quantificateurs de tête ;
- finalement, faire une assertion sur le résultat.

Nous n'avons pas encore mis en oœuvre ces modifications dans E-ACSL.

## 15.2 Vérification et génération d'assertions (symboliques) à partir des spécifications formelles Wasm

Même si Wasm est souvent utilisé comme cible de compilation, il arrive fréquemment que du code Wasm soit écrit manuellement, par exemple pour créer un environnement d'exécution ou pour remplacer des segments d'assembleur. Pouvoir annoter et vérifier du code Wasm dans ce contexte devient donc bien plus pertinent. Dans cette section, nous exposons la conception d'un langage de spécification dédié à Wasm et la mise en œuvre de la génération d'assertions.

### 15.2.1 L'extension *Custom Annotations*

Pour annoter des programmes Wasm écrits manuellement dans le format textuel, nous avons choisi d'étendre ce dernier à l'aide de l'extension *Custom Annotations* [106]. Celle-ci permet l'ajout d'annotations personnalisées tout en respectant les standards Wasm. Elles seront ignorées par tout outil ne reconnaissant pas l'annotation en question. Elles sont similaires aux *attributs* en OCaml (par exemple `[@@inline]`). En Wasm, elles commencent par un `@`, comme le montre cet exemple :

```
(module
  (@custom "my-fancy-section" "Hello, I'm useless don't try to understand
  ↪  me"))
(module (@name "I'm a module with a fancy name")
  (func (@inline) $lambda (@name "λ") (param $x (@name "α") i32) (result i32)
    (local.get $x)))
```

Nous avons mis en œuvre cette extension dans Owi, ce qui nous permet d'utiliser des champs personnalisés pour ajouter des spécifications.

### 15.2.2 Weasel : *WEbAssembly SpEcification Language*

**Syntaxe** Weasel repose sur l'extension Custom Annotations pour permettre l'expression de contrats dans Wasm. Voici un exemple de contrat pour une fonction `$sum` :

```
(module

  (@contract $sum
    (ensures (= result (+ $p1 (+ $p2 (+ $p3 $p4))))))

  (func $sum (param $p1 i32) (param $p2 i32) (param $p3 i32) (param $p4 i32)
  ↪  (result i32)
    (i32.add
      (i32.add
        (i32.add
          (local.get $p3)
          (local.get $p4))
        (local.get $p2))
```

```
      (local.get $p1)))

  (func $start
    (call $sum (i32.const 42) (i32.const 42) (i32.const 42) (i32.const 42))
    drop )

  (start $start))
```

Weasel utilise une syntaxe en S-expressions pour rester cohérent avec celle de Wasm, facilitant ainsi la mise en œuvre dans le cadre d'un prototype. La syntaxe abstraite complète de Weasel est présentée ci-dessous.

Tout d'abord, le seul opérateur unaire disponible est l'opposé :

| | | |
|---|---|---|
| *unop* ::= | `-` | (opposé) |

Ensuite, plusieurs opérateurs binaires sont proposés :

| | | |
|---|---|---|
| *binop* ::= | `+` | (addition) |
| \| | `-` | (soustraction) |
| \| | `*` | (multiplication) |
| \| | `/` | (division) |

Les termes peuvent être des *pterm*, un indice (entier ou textuel), ou encore un résultat :

| | | |
|---|---|---|
| *term* ::= | `(` *pterm* `)` | (pterm) |
| \| | *i32* | (indice entier) |
| \| | `$`*ind* | (indice textuel) |
| \| | `result` | (result) |

Les *pterm* peuvent représenter différentes formes de constantes, d'identifiants ou de résultats, voire des mémoires :

| | | |
|---|---|---|
| *pterm* ::= | `i32` *i32* | (constante i32) |
| | \| `f32` *f32* | (constante f32) |
| | \| `i64` *i64* | (constante i64) |
| | \| `f64` *f64* | (constante f64) |
| | \| `param` *ind* | (indice de paramètre) |
| | \| `global` *ind* | (indice de variable globale) |
| | \| `binder` *ind* | (indice de variable quantifiée) |
| | \| *unop term* | (opération unaire) |
| | \| *binop term term* | (opération binaire) |
| | \| `result` *i32* | (résultat) |
| | \| `memory` *term* | (mémoire) |

Les prédicats binaires suivants sont disponibles pour les comparaisons :

| | | |
|---|---|---|
| *binpred* ::= | `>=` | (supérieur ou égal) |
| | \| `>` | (supérieur strict) |
| | \| `<=` | (inférieur ou égal) |
| | \| `<` | (inférieur strict) |
| | \| `=` | (égal) |
| | \| `!=` | (différent de) |

Le seul connecteur unaire est la négation :

| | | |
|---|---|---|
| *unconnect* ::= | `!` | (négation) |

Les connecteurs binaires suivants sont également disponibles :

| | | |
|---|---|---|
| *binconnect* ::= | `&&` | (conjonction) |
| | \| `\|\|` | (disjonction) |
| | \| `==>` | (implication) |
| | \| `<==>` | (équivalence) |

Une propriété peut être soit une *pprop*, soit une constante booléenne :

| | | |
|---|---|---|
| *prop* ::= | `(` *pprop* `)` | (pprop) |
| | \| `true` | (vrai) |
| | \| `false` | (faux) |

Les deux quantificateurs usuels sont présents dans la syntaxe :

| | | |
|---|---|---|
| *binder* ::= | `forall` | (quantificateur universel) |
| \| | `exists` | (quantificateur existentiel) |

Quatre types sont disponibles pour les quantificateurs :

| | | |
|---|---|---|
| *binder_type* ::= | `i32` | (i32) |
| \| | `i64` | (i64) |
| \| | `f32` | (f32) |
| \| | `f64` | (f64) |

Enfin, une *pprop* combine des propriétés à l'aide de prédicats, de connecteurs, de quantificateurs et de types :

| | | |
|---|---|---|
| *binder* ::= | *binpred term term* | (prédicat binaire) |
| \| | *unconnect prop* | (connecteur unaire) |
| \| | *binconnect prop* | (connecteur binaire) |
| \| | *binder binder_type prop* | (quantificateur sans indice) |
| \| | *binder binder_type ind prop* | (quantificateur avec indice) |

### 15.2.3 Génération d'assertions (symboliques) exécutables à partir de Weasel

Grâce à la syntaxe que nous avons définie, nous pouvons désormais générer des assertions à partir de spécifications de contrats. Prenons l'exemple suivant :

```
(module
  (@contract $plus_three
    (ensures (= result (+ $x 3)))
  )
  (func $plus_three (param $x i32) (result i32)
    local.get $x
    i32.const 3
    i32.add
  )
  (func $start
    i32.const 42
    call $plus_three
    drop
  )
  (start $start)
)
```

En exécutant la commande suivante :

```
$ owi instrument plus_three.wat
```

Le fichier Wasm généré sera :

```
(module
  (import "symbolic" "assert" (func $assert (param i32)))
  (func $plus_three (param $x i32) (result i32)
    local.get $x
    i32.const 3
    i32.add
  )
  (func $start
    i32.const 42
    call $__weasel_plus_three
    drop
  )
  (func $__weasel_plus_three (param $x i32) (result i32)
      (local $__weasel_temp i32) (local $__weasel_res_0 i32)
    local.get $x
    call $plus_three
    local.set $__weasel_res_0
    local.get $__weasel_res_0
    local.get $x
    i32.const 3
    i32.add
    i32.eq
    call $assert
    local.get $__weasel_res_0
  )
  (start $start)
)
```

Ce fichier est pleinement exécutable dans un environnement concret, à condition que l'hôte fournisse une fonction `$assert`. En plus de cela, une option `--rac` est disponible pour les commandes `owi sym`, `owi run` et `owi conc`, permettant l'exécution et la génération directe du code instrumenté, tout en fournissant les fonctions requises par le module.

La commande `owi instrument` propose également une option `--symbolic` qui permet de générer des assertions basées sur une exécution symbolique. De même, les commandes `owi sym` et `owi conc` incluent des options `--srac` pour exécuter directement du code instrumenté qui repose sur cette exécution symbolique.

Pour le moment, aucune validation des contrats n'est réalisée. La partie de la mise en œuvre concernant la génération du code comprend environ 700 lignes de code OCaml, que nous ne détaillerons pas ici.

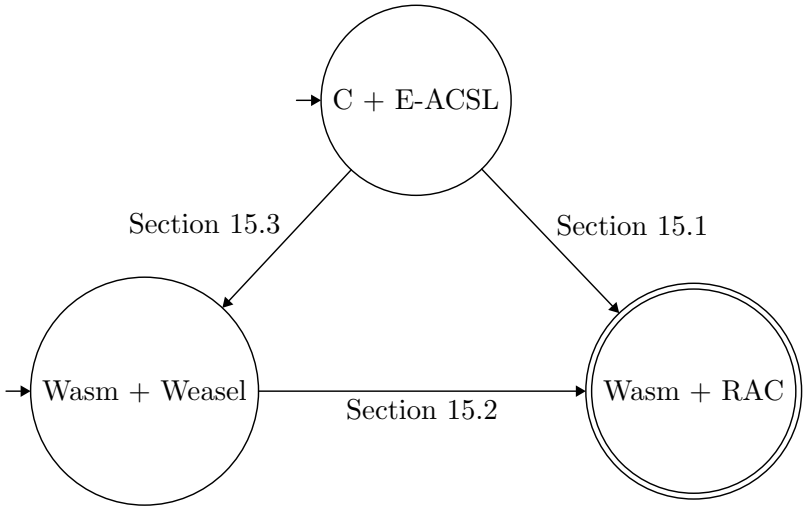



FIGURE 15.1 – Différents chemins allant des spécifications à la vérification d'assertions.

## 15.3 Travaux futurs

Plusieurs pistes d'évolution sont envisagées pour approfondir ce travail. Premièrement, nous comptons mettre en œuvre diverses améliorations pour la génération de code dans E-ACSL. Cela pourrait renforcer l'efficacité et l'expressivité de la vérification via Owi.

Deuxièmement, nous prévoyons de poursuivre le développement de Weasel, qui est encore à un stade préliminaire. De nombreuses améliorations restent à apporter, notamment pour étendre le langage de spécification à un plus large éventail de constructions Wasm.

Une autre perspective intéressante concerne la vérification de code combinant plusieurs langages, chacun annoté avec ses propres spécifications. Par exemple, il serait pertinent de vérifier une bibliothèque optimisée écrite en Wasm, intégrée dans un algorithme en C, où les preuves de correction de l'un dépendent des assertions générées à partir des spécifications de l'autre.

De plus, nous envisageons d'appliquer cette approche à d'autres langages, comme Rust. En réutilisant Pearlite, le langage de spécification de Creusot [132] (un outil de vérification déductive pour Rust), il serait possible de générer du code instrumenté, ensuite exécutable via Owi.

Enfin, une autre voie prometteuse serait de compiler des langages de spécification, tels que ceux utilisés en C, vers Weasel, ce qui permettrait d'unifier les outils de vérification pour différents langages à travers une infrastructure commune. Une représentation des différentes possibilités est donnée dans la figure 15.1.

## 15.4 Travaux connexes

Les travaux de MUNUERA MAZZARO [154] proposent un langage de spécification pour Wasm, nommé VerifiWasm, où les spécifications sont rédigées dans un fichier séparé. Leur outil vérifie que les spécifications sont respectées pour un *binaire Wasm* donné. À l'instar de notre approche, ils utilisent l'exécution symbolique pour générer des conditions de vérification qui sont ensuite transmises à des solveurs SMT.

Cependant, leur méthode présente plusieurs limitations par rapport à la nôtre. Premièrement, ils ne produisent pas de module Wasm intégrant des assertions exécutables issues du programme original. De plus, leur système est limité à des modules binaires distincts des spécifications, ce qui complique leur écriture et maintenance.

Ensuite, ils ne peuvent prouver que la correction partielle du programme (*i.e.*, s'il termine, alors il est correct). En effet, leur outil ne possède pas de mécanisme pour prouver la terminaison des programmes, alors que notre approche inclut implicitement cette preuve, puisque l'exécution symbolique qui termine est elle-même une preuve que le programme se termine.

Leur mise en œuvre est également restreinte à une petite partie du standard Wasm : les opérations sur les entiers sont incomplètes, les nombres flottants ne sont pas pris en charge, et la gestion des variables globales ainsi que de la mémoire est absente. Ces limitations contrastent fortement avec Owi, qui offre un support bien plus large.

Enfin, leur outil ne prend en charge qu'un seul solveur SMT (Z3), tandis qu'Owi, grâce à l'utilisation de *Smt.ml*, peut interagir avec plusieurs solveurs, offrant ainsi une plus grande flexibilité.

À notre connaissance, aucun autre outil n'exploite les langages de spécification existants, comme E-ACSL, pour générer des assertions en vue d'une exécution symbolique. Cependant, il convient de mentionner l'outil de vérification déductive construit autour du langage Viper [87, 88], qui permet de compiler plusieurs langages de spécification (comme ceux pour Java et Scala) vers un langage intermédiaire. Ils y réalisent ensuite de l'exécution symbolique ainsi que la vérification déductive. Toutefois, Viper n'instrumente pas directement les langages source avec des langages de spécification existants.

De plus, Viper doit compiler lui-même les langages vers sa représentation intermédiaire, alors que notre approche tire parti des compilateurs déjà en place, qui bénéficient généralement de plus d'attention et de ressources pour l'optimisation, ce qui est important pour une exécution symbolique performante, comme nous l'avons démontré.

# 16 Conclusion et perspectives

Nous avons présenté *Wasocaml*, un compilateur d'OCaml vers WasmGC. Ce travail démontre la pertinence de WasmGC comme cible de compilation pour des langages fonctionnels complexes.

Nous avons également présenté *Owi*, un moteur d'exécution symbolique performant pour Wasm, qui permet d'exécuter symboliquement des programmes écrits dans d'autres langages comme C ou Rust.

Nous avons aussi introduit *Weasel*, un langage de spécification conçu pour Wasm, qui facilite la vérification individuelle des fonctions et ouvre de nouvelles perspectives de vérification, notamment par des méthodes déductives. Nous souhaitons approfondir ce travail pour étendre les capacités de vérification de Wasm.

Nous pensons que ce travail a permis de montrer une propriété inattendue et intéressante de Wasm : son adéquation lorsqu'il s'agit d'appliquer des méthodes formelles et d'effecteur des analyses statiques.

Notre première ambition est de porter Wasocaml de *Flambda1* à *Flambda2*, ce qui permettrait de tirer parti d'optimisations plus avancées et, ainsi, d'améliorer les performances globales. Par ailleurs, il souhaitons étendre le compilateur pour couvrir l'ensemble du langage source.

Quant à Owi notre prochain objectif est d'améliorer les performances du moteur en intégrant un modèle mémoire plus précis et en optimisant l'exploration des chemins via des heuristiques avancées.

Ensuite, nous souhaitons adapter les mécanismes de FFI existants pour permettre l'intégration de logiciels complets capables d'interagir avec du code C ou JavaScript, ce qui ouvrirait la voie à des applications plus interopérables.

Enfin, une perspective intéressante consisterait à réunir les deux contributions majeures de cette thèse. En intégrant la prise en charge de WasmGC dans Owi, et en rendant possible l'exécution symbolique du code utilisant cette extension, nous pourrions ainsi développer le premier moteur d'exécution symbolique pour WasmGC. Cela permettrait également d'envisager l'exécution symbolique de plusieurs langages avec glaneur de cellules. En particulier, cela permettrait, au moyen de Wasocaml, d'effectuer de l'exécution symbolique de notre langage favori : OCaml. Il n'existe actuellement pas de moteur d'exécution symbolique pour OCaml. Utiliser Wasocaml et Owi à cette fin serait donc un apport des plus bénéfique à la communauté.

# A

# Code Wasm permettant de générer un modèle correspondant à une partition pour quatuor à cordes sans contrainte

```
(module
  (import "symbolic" "i32_symbol" (func $symbol (result i32)))
  (import "symbolic" "assume" (func $assume (param i32)))
  (import "symbolic" "assert" (func $assert (param i32)))

  ;; must be in the [0; 12[ interval
  (global $tonalite (mut i32) (i32.const 3))

  ;; must not be changed otherwise the lilypond generation won't be working
  ↪  for now
  (global $nb_instr (mut i32) (i32.const 4))

  ;; number of generated chords
  (global $duree i32 (i32.const 12))

  (memory $mem 1)

  (func $init_tonalite

    (local $note i32)

    ;; create a symbol
    (local.set $note (call $symbol))

    ;; set this symbol to tonality, just so it is printed on output and can be
    ↪  parsed when generating lilypond file with the correct key
    (call $assume (i32.eq (local.get $note) (global.get $tonalite))))

  (func $gen_note (result i32 i32)
```

```
  (local $note i32) (local $octave i32)

  ;; creating two symbols for the note
  (local.set $note (call $symbol))
  (local.set $octave (call $symbol))

  ;; putting constraint on the note so that it is a valid note
  (call $assume (i32.le_u (local.get $note) (i32.const 11)))
  ;; putting constraint on the note so that it is not too high
  (call $assume (i32.le_u (local.get $octave) (i32.const 2)))

  ;; returning the new note
  (local.get $note) (local.get $octave))

;; compute the offset where a note for a given time/instrument is stored in
↪  memory
(func $offset (param $instrument i32) (param $temps i32) (result i32)
  (call $assert (i32.lt_u (local.get $instrument) (global.get $nb_instr)))
  (i32.mul (i32.const 8)
    (i32.add
      (i32.mul (local.get $temps) (global.get $nb_instr))
      (local.get $instrument)))
)

;; set a note in memory for a given instrument and time
(func $set_note (param $instrument i32) (param $temps i32) (param $note i32)
↪  (param $octave i32)
   (local $pos i32)
   (local.set $pos (call $offset (local.get $instrument) (local.get
   ↪  $temps)))
   (i32.store (local.get $pos) (local.get $note))
   (i32.store (i32.add (i32.const 4) (local.get $pos)) (local.get $octave)))

;; get a note in memory for a given instrument and time
(func $get_note (param $instrument i32) (param $temps i32) (result i32)
   (local $pos i32)
   (local.set $pos (call $offset (local.get $instrument) (local.get
   ↪  $temps)))
   (i32.load (local.get $pos)))

;; get the octave in memory for a given isntrument and time
(func $get_octave (param $instrument i32) (param $temps i32) (result i32)
   (local $pos i32)
```

```
    (local.set $pos (call $offset (local.get $instrument) (local.get
    ↪  $temps)))
    (i32.load (i32.add (i32.const 4) (local.get $pos))))

;; get a number ~~corresponding to the absolute "pitch" of the note
(func $get_niveau (param $instrument i32) (param $temps i32) (result i32)
  (i32.add
    (call $get_note (local.get $instrument) (local.get $temps))
    (i32.mul (i32.const 12) (call $get_octave (local.get $instrument)
    ↪  (local.get $temps))))
)

;; generate a random note for a single instrument and store it in memory
(func $init_note (param $instrument i32) (param $temps i32)
    (local $note i32)
    (local $octave i32)
    (call $gen_note)
    (local.set $octave)
    (local.set $note)
    (call $set_note (local.get $instrument) (local.get $temps)
                    (local.get $note) (local.get $octave))
)

;; generate a random note for each instrument and store it in memory
(func $init_temps (param $temps i32)
  (call $init_note (i32.const 0) (local.get $temps))
  (call $init_note (i32.const 1) (local.get $temps))
  (call $init_note (i32.const 2) (local.get $temps))
  (call $init_note (i32.const 3) (local.get $temps))
)

;; generate all random notes
(func $init_duree
  (local $temps i32)
  (loop $loop
    (call $init_temps (local.get $temps))
    (local.set $temps (i32.add (local.get $temps) (i32.const 1)))
    (br_if $loop (i32.lt_u (local.get $temps) (global.get $duree)))
  ))

(func $start
      (call $init_tonalite)
      (call $init_duree)
      (unreachable))
```

```
  (start $start)
)
```

# B

# Code Wasm permettant de générer un modèle correspondant à une partition pour quatuor à cordes avec différentes contraintes

```
(module
  (import "symbolic" "i32_symbol" (func $symbol (result i32)))
  (import "symbolic" "assume" (func $assume (param i32)))
  (import "symbolic" "assert" (func $assert (param i32)))

  ;; must be in the [0; 12[ interval
  (global $tonalite (mut i32) (i32.const 3))

  ;; must not be changed otherwise the lilypond generation won't be working
  ↪  for now
  (global $nb_instr (mut i32) (i32.const 4))

  ;; number of generated chords
  (global $duree i32 (i32.const 12))

  (memory $mem 1)

  (func $init_tonalite

    (local $note i32)

    ;; create a symbol
    (local.set $note (call $symbol))

    ;; set this symbol to tonality, just so it is printed on output and can be
    ↪  parsed when generating lilypond file with the correct key
    (call $assume (i32.eq (local.get $note) (global.get $tonalite))))

  (func $gen_note (result i32 i32)
```

```
  (local $note i32) (local $octave i32)

  ;; creating two symbols for the note
  (local.set $note (call $symbol))
  (local.set $octave (call $symbol))

  ;; putting constraint on the note so that it is a valid note
  (call $assume (i32.le_u (local.get $note) (i32.const 11)))
  ;; putting constraint on the note so that it is not too high
  (call $assume (i32.le_u (local.get $octave) (i32.const 2)))

  ;; returning the new note
  (local.get $note) (local.get $octave))

;; compute the offset where a note for a given time/instrument is stored in
↪  memory
(func $offset (param $instrument i32) (param $temps i32) (result i32)
  (call $assert (i32.lt_u (local.get $instrument) (global.get $nb_instr)))
  (i32.mul (i32.const 8)
    (i32.add
      (i32.mul (local.get $temps) (global.get $nb_instr))
      (local.get $instrument)))
)

;; set a note in memory for a given instrument and time
(func $set_note (param $instrument i32) (param $temps i32) (param $note i32)
↪  (param $octave i32)
   (local $pos i32)
   (local.set $pos (call $offset (local.get $instrument) (local.get
   ↪  $temps)))
   (i32.store (local.get $pos) (local.get $note))
   (i32.store (i32.add (i32.const 4) (local.get $pos)) (local.get $octave)))

;; get a note in memory for a given instrument and time
(func $get_note (param $instrument i32) (param $temps i32) (result i32)
   (local $pos i32)
   (local.set $pos (call $offset (local.get $instrument) (local.get
   ↪  $temps)))
   (i32.load (local.get $pos)))

;; get the octave in memory for a given isntrument and time
(func $get_octave (param $instrument i32) (param $temps i32) (result i32)
   (local $pos i32)
```

```
    (local.set $pos (call $offset (local.get $instrument) (local.get
    ↪  $temps)))
    (i32.load (i32.add (i32.const 4) (local.get $pos))))

;; get a number ~~corresponding to the absolute "pitch" of the note
(func $get_niveau (param $instrument i32) (param $temps i32) (result i32)
  (i32.add
    (call $get_note (local.get $instrument) (local.get $temps))
    (i32.mul (i32.const 12) (call $get_octave (local.get $instrument)
    ↪  (local.get $temps))))
)

;; generate a random note for a single instrument and store it in memory
(func $init_note (param $instrument i32) (param $temps i32)
   (local $note i32)
   (local $octave i32)
   (call $gen_note)
   (local.set $octave)
   (local.set $note)
   (call $set_note (local.get $instrument) (local.get $temps)
                   (local.get $note) (local.get $octave))
)

;; generate a random note for each instrument and store it in memory
(func $init_temps (param $temps i32)
  (call $init_note (i32.const 0) (local.get $temps))
  (call $init_note (i32.const 1) (local.get $temps))
  (call $init_note (i32.const 2) (local.get $temps))
  (call $init_note (i32.const 3) (local.get $temps))
)

;; generate all random notes
(func $init_duree
  (local $temps i32)
  (loop $loop
    (call $init_temps (local.get $temps))
    (local.set $temps (i32.add (local.get $temps) (i32.const 1)))
    (br_if $loop (i32.lt_u (local.get $temps) (global.get $duree)))
  ))

;; tells if a note is the VIIth degree
(func $is_sensible (param $note i32) (result i32)
  (i32.or
    (i32.eq (i32.rem_u (i32.add (global.get $tonalite) (i32.const 11))
    ↪  (i32.const 12)) (local.get $note))
```

```
    (i32.eq (i32.rem_u (i32.add (global.get $tonalite) (i32.const 10))
    ↪  (i32.const 12)) (local.get $note)))
)

;; gives the degree of a note according to current key
(func $note_nbr (param $note i32) (result i32)
  (local $mod_note i32)
  (local.set $mod_note
     (i32.rem_u
       (i32.add (local.get $note)
                (i32.sub (i32.const 12) (global.get $tonalite)))
       (i32.const 12)))
  (i32.const 8)
  (i32.const 7)
  (i32.const 6)
  (i32.const 5)
  (i32.const 4)
  (i32.const 3)
  (i32.const 2)
  (i32.const 1)
  (i32.const 0)
  (i32.eq (local.get $mod_note) (i32.const 0))
  select
  (i32.eq (local.get $mod_note) (i32.const 2))
  select
  (i32.eq (local.get $mod_note) (i32.const 3))
  select
  (i32.eq (local.get $mod_note) (i32.const 5))
  select
  (i32.eq (local.get $mod_note) (i32.const 7))
  select
  (i32.eq (local.get $mod_note) (i32.const 8))
  select
  (i32.eq (local.get $mod_note) (i32.const 10))
  select
  (i32.eq (local.get $mod_note) (i32.const 11))
  select
)

;; tell if a note is the fifth of another
(func $is_quinte_note_1 (param $note_nbr1 i32) (param $note_nbr2 i32)
↪  (result i32)
  (i32.and (i32.eq (local.get $note_nbr1) (i32.const 1)) (i32.eq (local.get
  ↪  $note_nbr2) (i32.const 5)))
```

```
    (i32.and (i32.eq (local.get $note_nbr1) (i32.const 2)) (i32.eq (local.get
    ↪  $note_nbr2) (i32.const 6)))
    (i32.and (i32.eq (local.get $note_nbr1) (i32.const 3)) (i32.eq (local.get
    ↪  $note_nbr2) (i32.const 7)))
    (i32.and (i32.eq (local.get $note_nbr1) (i32.const 4)) (i32.eq (local.get
    ↪  $note_nbr2) (i32.const 1)))
    (i32.and (i32.eq (local.get $note_nbr1) (i32.const 5)) (i32.eq (local.get
    ↪  $note_nbr2) (i32.const 2)))
    (i32.and (i32.eq (local.get $note_nbr1) (i32.const 6)) (i32.eq (local.get
    ↪  $note_nbr2) (i32.const 3)))
    (i32.and (i32.eq (local.get $note_nbr1) (i32.const 7)) (i32.eq (local.get
    ↪  $note_nbr2) (i32.const 4)))
    i32.or i32.or i32.or i32.or i32.or i32.or
  )
  ;; tell if a note is the fifth of another or the another is the fifth of the
  ↪  note :-)
  (func $is_quinte_note (param $note_nbr1 i32) (param $note_nbr2 i32) (result
  ↪  i32)
    (i32.or (call $is_quinte_note_1 (local.get $note_nbr1) (local.get
    ↪  $note_nbr2))
            (call $is_quinte_note_1 (local.get $note_nbr2) (local.get
            ↪  $note_nbr1)))
  )

  ;; tell if two instruments are forming a fifth at a given time
  (func $is_quinte (param $instrument1 i32) (param $instrument2 i32) (param
  ↪  $temps i32) (result i32)
    (call $is_quinte_note
      (call $note_nbr (call $get_note (local.get $instrument1) (local.get
      ↪  $temps)))
      (call $note_nbr (call $get_note (local.get $instrument2) (local.get
      ↪  $temps))))
  )

  ;; tell if a note is the octave of another
  (func $is_octave_note (param $note_nbr1 i32) (param $note_nbr2 i32) (result
  ↪  i32)
    (i32.eq (local.get $note_nbr1) (local.get $note_nbr2))
  )

  ;; tell if two instruments are forming an octave at a given time
  (func $is_octave (param $instrument1 i32) (param $instrument2 i32) (param
  ↪  $temps i32) (result i32)
    (call $is_octave_note
      (call $get_note (local.get $instrument1) (local.get $temps))
```

```
    (call $get_note (local.get $instrument2) (local.get $temps)))
)

;; forbid parallel fifth at a given time between two instruments
(func $contrainte_quinte_instr (param $instrument1 i32) (param $instrument2
↪  i32) (param $temps i32)
  (call $assert (i32.ge_u (local.get $temps) (i32.const 1)))
  (i32.eq (i32.const 0)
    (i32.and
      (call $is_quinte (local.get $instrument1) (local.get $instrument2)
      ↪  (i32.sub (local.get $temps) (i32.const 1)))
      (call $is_quinte (local.get $instrument1) (local.get $instrument2)
      ↪  (local.get $temps))))
  (call $assume)
)

;; forbid parallel fifth at a given time for all instrument pairs
(func $contrainte_quinte (param $temps i32)
  (call $contrainte_quinte_instr (i32.const 0) (i32.const 1) (local.get
  ↪  $temps))
  (call $contrainte_quinte_instr (i32.const 0) (i32.const 2) (local.get
  ↪  $temps))
  (call $contrainte_quinte_instr (i32.const 0) (i32.const 3) (local.get
  ↪  $temps))
  (call $contrainte_quinte_instr (i32.const 1) (i32.const 2) (local.get
  ↪  $temps))
  (call $contrainte_quinte_instr (i32.const 1) (i32.const 3) (local.get
  ↪  $temps))
  (call $contrainte_quinte_instr (i32.const 2) (i32.const 3) (local.get
  ↪  $temps))
)

;; forbid parallel octave at a given time between two instruments
(func $contrainte_octave_instr (param $instrument1 i32) (param $instrument2
↪  i32) (param $temps i32)
  (call $assert (i32.ge_u (local.get $temps) (i32.const 1)))
  (i32.eq (i32.const 0)
    (i32.and
      (call $is_octave (local.get $instrument1) (local.get $instrument2)
      ↪  (i32.sub (local.get $temps) (i32.const 1)))
      (call $is_octave (local.get $instrument1) (local.get $instrument2)
      ↪  (local.get $temps))))
  (call $assume)
)
```

```
;; forbid parallel octave at a given time for all instrument pairs
(func $contrainte_octave (param $temps i32)
  (call $contrainte_octave_instr (i32.const 0) (i32.const 1) (local.get
  ↪  $temps))
  (call $contrainte_octave_instr (i32.const 0) (i32.const 2) (local.get
  ↪  $temps))
  (call $contrainte_octave_instr (i32.const 0) (i32.const 3) (local.get
  ↪  $temps))
  (call $contrainte_octave_instr (i32.const 1) (i32.const 2) (local.get
  ↪  $temps))
  (call $contrainte_octave_instr (i32.const 1) (i32.const 3) (local.get
  ↪  $temps))
  (call $contrainte_octave_instr (i32.const 2) (i32.const 3) (local.get
  ↪  $temps))
)

;; ensure the note of an instrument at a given time is one of the degree
↪  given in parameter
(func $contrainte_groupe_instr (param $n1 i32) (param $n2 i32) (param $n3
↪  i32) (param $instrument i32) (param $temps i32) (result i32)
  (local $note i32)
  (local.set $note (call $note_nbr (call $get_note (local.get $instrument)
  ↪  (local.get $temps))))
  (i32.or
  (i32.or
    (i32.eq (local.get $n1) (local.get $note))
    (i32.eq (local.get $n2) (local.get $note)))
    (i32.eq (local.get $n3) (local.get $note)))
)

;; ensure the notes of all instruments at a given time are one of the degree
↪  given in parameter
(func $contrainte_groupe (param $n1 i32) (param $n2 i32) (param $n3 i32)
↪  (param $temps i32) (result i32)
  (call $contrainte_groupe_instr (local.get $n1) (local.get $n2) (local.get
  ↪  $n3) (i32.const 0) (local.get $temps))
  (call $contrainte_groupe_instr (local.get $n1) (local.get $n2) (local.get
  ↪  $n3) (i32.const 1) (local.get $temps))
  (call $contrainte_groupe_instr (local.get $n1) (local.get $n2) (local.get
  ↪  $n3) (i32.const 2) (local.get $temps))
  (call $contrainte_groupe_instr (local.get $n1) (local.get $n2) (local.get
  ↪  $n3) (i32.const 3) (local.get $temps))
  i32.and
  i32.and
  i32.and
```

```
  )

  ;; ensure all instruments are forming a chord at a given time
  (func $contraintes_groupe (param $temps i32)
    (call $contrainte_groupe (i32.const 1) (i32.const 3) (i32.const 5)
    ↪  (local.get $temps))
    (call $contrainte_groupe (i32.const 2) (i32.const 4) (i32.const 6)
    ↪  (local.get $temps))
    (call $contrainte_groupe (i32.const 3) (i32.const 5) (i32.const 7)
    ↪  (local.get $temps))
    (call $contrainte_groupe (i32.const 4) (i32.const 6) (i32.const 1)
    ↪  (local.get $temps))
    (call $contrainte_groupe (i32.const 5) (i32.const 8) (i32.const 2)
    ↪  (local.get $temps))
    (call $contrainte_groupe (i32.const 6) (i32.const 1) (i32.const 3)
    ↪  (local.get $temps))
    (call $contrainte_groupe (i32.const 7) (i32.const 2) (i32.const 4)
    ↪  (local.get $temps))
    i32.or i32.or i32.or i32.or i32.or i32.or
    (call $assume)
  )

  ;; make sure the note of an instrument is the Ith degree if the previous one
  ↪  was a VIIth degree
  (func $contrainte_sensible_tonique (param $instrument i32) (param $temps
  ↪  i32)
    (local $prev_note i32)
    (local $note i32)
    (br_if 0 (i32.lt_u (local.get $temps) (i32.const 1)))
    (local.set $prev_note (call $get_note (local.get $instrument) (i32.add
    ↪  (local.get $temps) (i32.const -1))))
    (local.set $note (call $get_note (local.get $instrument) (local.get
    ↪  $temps)))
   (select
     (i32.and
       (i32.eq (local.get $note) (global.get $tonalite))
       (i32.eq
          (call $get_octave (local.get $instrument) (local.get $temps))
          (call $get_octave (local.get $instrument) (i32.add (local.get
          ↪  $temps) (i32.const -1)))))
     (i32.const 1)
     (call $is_sensible (local.get $prev_note)))
   (call $assume)
  )
```

```
  ;; tells if a note is part of the key
  (func $note_is_ok (param $note i32) (result i32)
    (local $mod_note i32)
    (local.set $mod_note
       (i32.rem_u
         (i32.add (local.get $note)
                  (i32.sub (i32.const 12) (global.get $tonalite)))
         (i32.const 12)))
    (i32.eq (local.get $mod_note) (i32.const 0))
    (i32.eq (local.get $mod_note) (i32.const 2)) i32.or
    (i32.eq (local.get $mod_note) (i32.const 3)) i32.or
    (i32.eq (local.get $mod_note) (i32.const 5)) i32.or
    (i32.eq (local.get $mod_note) (i32.const 7)) i32.or
    (i32.eq (local.get $mod_note) (i32.const 8)) i32.or
    (i32.eq (local.get $mod_note) (i32.const 10)) i32.or
    (i32.eq (local.get $mod_note) (i32.const 11)) i32.or
  )

  ;; make sure the note of an instrument at a given time is part of the key
  (func $note_ok (param $instrument i32) (param $temps i32)
    (call $assume
      (call $note_is_ok
        (call $get_note (local.get $instrument) (local.get $temps))))
  )

  ;; make sure we don't jump higher than an octave from a time to the next on
  ↪  a given instrument
  (func $contrainte_saut (param $instrument i32) (param $temps i32)
    (local $diff i32)
    (call $assert (i32.ge_u (local.get $temps) (i32.const 1)))
    (local.set $diff
      (i32.sub
        (call $get_niveau (local.get $instrument) (local.get $temps))
        (call $get_niveau (local.get $instrument) (i32.sub (local.get $temps)
        ↪  (i32.const 1)))))
    (call $assume
        (i32.le_s (local.get $diff) (i32.const 12)))
    (call $assume
        (i32.ge_s (local.get $diff) (i32.const -12)))
  )

  ;; make sure one instrument doesn't go higher than another
  (func $contrainte_rel_instruments_1 (param $instrument1 i32) (param
  ↪  $instrument2 i32) (param $temps i32)
      (i32.le_u
```

```
          (call $get_niveau (local.get $instrument1) (local.get $temps))
          (call $get_niveau (local.get $instrument2) (local.get $temps)))
        (call $assume)
  )

  ;; make sure each instrument doesn't go higher then the one above him
  (func $contrainte_rel_instruments (param $temps i32)
    (call $contrainte_rel_instruments_1 (i32.const 1) (i32.const 0) (local.get
    ↪  $temps))
    (call $contrainte_rel_instruments_1 (i32.const 2) (i32.const 1) (local.get
    ↪  $temps))
    (call $contrainte_rel_instruments_1 (i32.const 3) (i32.const 2) (local.get
    ↪  $temps))
  )

  ;; applies a bunch of constraint defined previously
  (func $applique_constraintes
    (local $instrument i32)
    (local $temps i32)
    (loop $loop_temps
      (call $contraintes_groupe (local.get $temps))
      (if (local.get $temps) (then
        (call $contrainte_quinte (local.get $temps))
        )
      )
      (local.set $instrument (i32.const 0))
      (loop $loop_instrument

        (call $contrainte_sensible_tonique (local.get $instrument) (local.get
        ↪  $temps))
        (call $note_ok (local.get $instrument) (local.get $temps))

        (if (local.get $temps) (then
          (call $contrainte_saut (local.get $instrument) (local.get $temps))))

        (local.set $instrument (i32.add (local.get $instrument) (i32.const
        ↪  1)))
        (br_if $loop_instrument (i32.lt_u (local.get $instrument) (global.get
        ↪  $nb_instr)))
      )
      (local.set $temps (i32.add (local.get $temps) (i32.const 1)))
      (br_if $loop_temps (i32.lt_u (local.get $temps) (global.get $duree)))
    )
  )
```

```
  ;; applies a bunch of constraint defined previously
  (func $applique_constraintes_2
    (local $instrument i32)
    (local $temps i32)
    (loop $loop_temps
      (if (local.get $temps) (then
        (call $contrainte_octave (local.get $temps))
        )
      )
      (call $contrainte_rel_instruments (local.get $temps))
      (local.set $temps (i32.add (local.get $temps) (i32.const 1)))
      (br_if $loop_temps (i32.lt_u (local.get $temps) (global.get $duree)))
    )
  )

  (func $fix_notes
    (call $assume
      (i32.eq (call $get_note (i32.const 0) (i32.const 0))
              (i32.const 3)))
    (call $assume
      (i32.eq (call $get_octave (i32.const 0) (i32.const 0))
              (i32.const 2)))

    (call $assume
      (i32.eq (call $get_note (i32.const 0) (i32.const 7))
              (i32.const 3)))
    (call $assume
      (i32.eq (call $get_octave (i32.const 0) (i32.const 0))
              (i32.const 2))))

  (func $start
        (call $init_tonalite)
        (call $init_duree)
        (call $applique_constraintes)
        (call $applique_constraintes_2)
        (call $fix_notes)
        (unreachable))

  (start $start)
)
```

# C

# Code OCaml permettant la génération d'un fichier LilyPond à partir d'un modèle produit par Owi

```
let lines =
  let fd =
    if Array.length Sys.argv < 2 then
      stdin
    else
      open_in_bin Sys.argv.(1)
  in
  let lines = In_channel.input_lines fd in
  close_in fd;
  lines

let symbol_line line =
  Scanf.sscanf line "    (symbol_%i (i32 %i))" (fun id v -> id, v)

let nb_instr = 4

type echantillon = {
  temps : int;
  instrument : int;
  note : int;
  octave : int;
}

let rec pairs l =
  match l with
  | [] -> []
  | [_] -> assert false
  | (x, note) :: (_, octave) :: t ->
    let id = x/2 in
    let temps = id / nb_instr in
```

```
    let instrument = id mod nb_instr in
    { temps; instrument; note; octave } :: pairs t

let variables lines =
  List.filter_map (fun l ->
      try Some (symbol_line l) with _ -> None)
    lines |>
  List.sort (fun (x, _) (y, _) -> compare x y)

let variables = variables lines

let tonalite, variables =
  match variables with
  | (0, tonalite) :: variables ->
    tonalite, (List.map (fun (i, x) -> i-1, x) variables)
  | _ -> assert false

let e = pairs variables
let r = List.init nb_instr (fun instrument ->
    List.filter (fun x -> x.instrument = instrument) e
  )

let note = function
  | 0 -> "a"
  | 1 -> "ais"
  | 2 -> "b"
  | 3 -> "c"
  | 4 -> "cis"
  | 5 -> "d"
  | 6 -> "dis"
  | 7 -> "e"
  | 8 -> "f"
  | 9 -> "fis"
  | 10 -> "g"
  | 11 -> "gis"
  | _ -> assert false

let note_with_tonality tonality note =
  let has_sharp = match tonality with
  | 0 -> true
  | 1 -> false
  | 2 -> true
  | 3 -> false
  | 4 -> true
  | 5 -> false
```

```
  | 6 -> false
  | 7 -> true
  | 8 -> false
  | 9 -> true
  | 10 -> false
  | 11 -> false
  | _ -> assert false
  in
  match note with
  | 0 -> "a"
  | 1 -> if has_sharp then "ais" else "bes"
  | 2 -> "b"
  | 3 -> "c"
  | 4 -> if has_sharp then "cis" else "des"
  | 5 -> "d"
  | 6 -> if has_sharp then "dis" else "ees"
  | 7 -> "e"
  | 8 -> "f"
  | 9 -> if has_sharp then "fis" else "ges"
  | 10 -> "g"
  | 11 -> if has_sharp then "gis" else "aes"
  | _ -> assert false

let note_with_tonality = note_with_tonality tonalite

let base_octave = 1
let octave i =
  if i >= base_octave then
    String.concat "" @@ List.init (i - base_octave) (fun _ -> "'")
  else
    String.concat "" @@ List.init (base_octave - i) (fun _ -> ",")
let mk t =
  let oct = if t.note < 3 then t.octave else t.octave + 1 in
  note_with_tonality t.note ^ octave oct

let notes l = String.concat " " (List.map mk l)
let instruments = List.map notes r

let violon1 = List.nth instruments 0
let violon2 = List.nth instruments 1
let alto = List.nth instruments 2
let violoncelle = List.nth instruments 3

let pattern : _ format6 = {|
\version "2.24.1"
```

```
Violonun = \absolute {
 \key %s \minor
 %s
}
Violondeux = \absolute {
 \key %s \minor
 %s
}
Alto = \absolute {
 \key %s \minor
 %s
}
Violoncelle = \absolute {
 \key %s \minor
 %s
}

\score {
  «
    \new Staff \with {
      midiInstrument = "violin"
      instrumentName = "Violon 1"
    }
    «
      \new Voice { \voiceOne \Violonun }
    »
    \new Staff \with {
      midiInstrument = "violin"
      instrumentName = "Violon 2"
      \consists Merge_rests_engraver
    }
    «
      \new Voice { \voiceTwo \Violondeux }
    »
    \new Staff \with {
      midiInstrument = "viola"
      instrumentName = "Alto"
      \clef C
      \consists Merge_rests_engraver
    }
    «
      \new Voice { \voiceTwo \Alto }
    »
    \new Staff \with {
```

```
      midiInstrument = "cello"
      instrumentName = "Violoncelle"
      \clef F
      \consists Merge_rests_engraver
    }
    «
      \new Voice { \voiceTwo \Violoncelle }
    »
  »
\layout { }
\midi { }
}
|}

let res =
  let tonalite = note tonalite in
  Printf.sprintf pattern
  tonalite
    violon1
    tonalite
    violon2
    tonalite
    alto
    tonalite
    violoncelle

let () =
  let out = Out_channel.open_bin "out.ly" in
  Out_channel.output_string out res;
  Out_channel.close out
```

# Bibliographie


[1] Kurt Gödel. « Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I ». In : *Monatshefte für mathematik und physik* 38 (1931), p. 173-198.

[2] Henry Gordon Rice. « Classes of recursively enumerable sets and their decision problems ». In : *Transactions of the American Mathematical society* 74.2 (1953), p. 358-366.

[3] Rudolf Bayer et Edward McCreight. « Organization and maintenance of large ordered indices ». In : *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 1970, p. 107-141.

[4] Nf Solntseff et Alexander Yezerski. « A survey of extensible programming languages ». In : *International Tracts in Computer Science and Technology and Their Application*. T. 7. Elsevier, 1974, p. 267-307.

[5] James C. King. « Symbolic Execution and Program Testing ». In : *Communications of the ACM* (1976).

[6] Patrick Cousot et Radhia Cousot. « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints ». In : *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, p. 238-252.

[7] Henry G Baker Jr. « Shallow binding in lisp 1.5 ». In : *Communications of the ACM* 21.7 (1978), p. 565-569.

[8] John Nestor et Mary Van Deusen. *RED language reference manual*. Rapp. tech. INTERMETRICS INC CAMBRIDGE MA, 1979.

[9] John G. P. Barnes. « An overview of Ada ». In : *Software : Practice and Experience* 10.11 (1980), p. 851-887.

[10] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[11] Adele Goldberg et David Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[12] Donald E Knuth et Peter B Bendix. « Simple word problems in universal algebras ». In : *Automation of Reasoning : 2 : Classical Papers on Computational Logic 1967–1970* (1983), p. 342-376.

[13] Linda G DeMichiel et Richard P Gabriel. « The common lisp object system : An overview ». In : *European Conference on Object-Oriented Programming*. Springer. 1987, p. 151-170.

[14] Richard M Stallman et Zachary Weinberg. « The C preprocessor ». In : *Free Software Foundation* (1987), p. 8.

[15] Hans-Juergen Boehm et Mark Weiser. « Garbage collection in an uncooperative environment ». In : *Software : Practice and Experience* 18.9 (1988), p. 807-820.

[16] Eugenio Moggi. *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, Laboratory for ..., 1988.

[17] Bjarne Stroustrup. « Parameterized Types for C++. » In : *C++ Conference*. 1988, p. 1-18.

[18] Barton P Miller, Lars Fredriksen et Bryan So. « An empirical study of the reliability of UNIX utilities ». In : *Communications of the ACM* 33.12 (1990), p. 32-44.

[19] Philip Wadler. « Comprehending monads ». In : *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 1990, p. 61-78.

[20] Benjamin Goldberg. « Tag-free garbage collection for strongly typed programming languages ». In : *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1991, p. 165-176.

[21] Ronald Morrison et al. « An ad hoc approach to the implementation of polymorphism ». In : *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.3 (1991), p. 342-371.

[22] Amr Sabry et Matthias Felleisen. « Reasoning about programs in continuation-passing style. » In : *ACM SIGPLAN Lisp Pointers* 1 (1992), p. 288-298.

[23] Tim Berners-Lee. *A Brief History of the Web*. 1993. url : https://www.w3.org/DesignIssues/TimBook-old/History.html.

[24] Cormac Flanagan et al. « The essence of compiling with continuations ». In : *ACM Sigplan Notices* 28.6 (1993), p. 237-247.

[25] Mark P Jones et Luc Duponcheel. *Composing monads*. Rapp. tech. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale ..., 1993.

[26] Lal George, Florent Guillame et John H Reppy. « A portable and optimizing back end for the SML/NJ compiler ». In : *International Conference on Compiler Construction*. Springer. 1994, p. 83-97.

[27] Linus Torvalds. « Linux kernel implementation ». In : *Proceedings of Open Systems. Looking into the future. AUUG'94*. 1994, p. 9-14.

[28] Andrew K Wright et Matthias Felleisen. « A syntactic approach to type soundness ». In : *Information and computation* 115.1 (1994), p. 38-94.

[29] Robert Harper et Greg Morrisett. « Compiling polymorphism using intensional type analysis ». In : *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995, p. 130-141.

[30] Sheng Liang, Paul Hudak et Mark Jones. « Monad transformers and modular interpreters ». In : *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995, p. 333-343.

[31] David Tarditi et al. « TIL : A type-directed optimizing compiler for ML ». In : *ACM Sigplan Notices* 31.5 (1996), p. 181-192.

[32] Bruno Barras et al. « The Coq proof assistant reference manual : Version 6.1 ». Thèse de doct. Inria, 1997.

[33] Xavier Leroy. « The effectiveness of type-based unboxing ». In : *TIC 1997 : Workshop Types in Compilation*. 1997.

[34] George C Necula. « Proof-carrying code ». In : *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1997, p. 106-119.

[35] Gilad Bracha et al. « Making the future safe for the past : Adding genericity to the Java programming language ». In : *Acm sigplan notices* 33.10 (1998), p. 183-200.

[36] William M McKeeman. « Differential testing for software ». In : *Digital Technical Journal* 10.1 (1998), p. 100-107.

[37] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[38] Koen Claessen et John Hughes. « QuickCheck : a lightweight tool for random testing of Haskell programs ». In : *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 2000, p. 268-279.

[39] Xavier Leroy. « A modular module system ». In : *Journal of Functional Programming* 10.3 (2000), p. 269-303.

[40] Andrew Kennedy et Don Syme. « Design and implementation of generics for the .NET Common language runtime ». In : *ACM SIGPLAN Notices* 36.5 (mai 2001), p. 1-12. doi : 10.1145/381694.378797. url : https://doi.org/10.1145/381694.378797.

[41] Felix Von Leitner. *diet libc*. 2001. url : https://www.fefe.de/dietlibc/talk.pdf.

[42] Michael D. Ernst, Greg J. Badros et David Notkin. « An empirical analysis of C preprocessor use ». In : *IEEE Transactions on Software Engineering* 28.12 (2002), p. 1146-1170.

[43] Hayo Thielecke. « Comparing control constructs by double-barrelled CPS ». In : *Higher-Order and Symbolic Computation* 15 (2002), p. 141-160.

[44] Chiyan Chen et Hongwei Xi. « Implementing typeful program transformations ». In : *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. 2003, p. 20-28.

[45] Sarfraz Khurshid, Corina S Păsăreanu et Willem Visser. « Generalized symbolic execution for model checking and testing ». In : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2003, p. 553-568.

[46] Eric Larson et Todd Austin. « High Coverage Detection of {Input-Related} Security Faults ». In : *12th USENIX Security Symposium (USENIX Security 03)*. 2003.

[47] Hongwei Xi, Chiyan Chen et Gang Chen. « Guarded recursive datatype constructors ». In : *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2003, p. 224-235.

[48] David F Bacon, Perry Cheng et VT Rajan. « A unified theory of garbage collection ». In : *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2004, p. 50-68.

[49] Emir Pasalic. *The role of type equality in meta-programming*. Oregon Health & Science University, 2004.

[50] Andreas Rossberg et al. « Alice ML through the looking glass ». In : *Trends in functional programming* 5 (2004).

[51] Patrick Cousot et al. « The ASTRÉE analyzer ». In : *Programming Languages and Systems : 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14*. Springer. 2005, p. 21-30.

[52] Patrice Godefroid, Nils Klarlund et Koushik Sen. « DART : Directed automated random testing ». In : *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005, p. 213-223.

[53] Roberto Ierusalimschy, Luiz Henrique De Figueiredo et Waldemar Celes Filho. « The Implementation of Lua 5.0. » In : *J. Univers. Comput. Sci.* 11.7 (2005), p. 1159-1176.

[54] Koushik Sen, Darko Marinov et Gul Agha. « CUTE : A concolic unit testing engine for C ». In : *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), p. 263-272.

[55] Mike Barnett et al. « Boogie : A modular reusable verifier for object-oriented programs ». In : *Formal Methods for Components and Objects : 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*. Springer. 2006, p. 364-387.

[56] Andreas Rossberg. « The missing link : dynamic components for ML ». In : *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*. 2006, p. 99-110.

[57] Stephen Weeks. « Whole-program compilation in MLton ». In : *ML* 6 (2006), p. 1-1.

[58] Oleg Kiselyov. *Typed Type Checking and Typed Compilation*. 2007. url : https://okmij.org/ftp/tagless-final/typed-compilation.html.

[59] Andreas Rossberg. « Typed open programming ». Thèse de doct. Citeseer, 2007.

[60] Cristian Cadar, Daniel Dunbar et Dawson Engler. « KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs ». In : *USENIX Conference on Operating Systems Design and Implementation*. 2008.

[61] Sylvain Conchon et Jean-Christophe Filliâtre. « Semi-persistent data structures ». In : *European Symposium on Programming*. Springer. 2008, p. 322-336.

[62] Leonardo De Moura et Nikolaj Bjørner. « Z3 : An Efficient SMT Solver ». In : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008.

[63] Patrice Godefroid, Michael Y Levin, David A Molnar et al. « Automated whitebox fuzz testing. » In : *NDSS*. T. 8. 2008, p. 151-166.

[64] Louis-Julien Guillemette et Stefan Monnier. « A type-preserving compiler in Haskell ». In : *ACM Sigplan Notices* 43.9 (2008), p. 75-86.

[65] Chris Lattner. « LLVM and Clang : Next generation compiler technology ». In : *The BSD conference*. T. 5. 2008, p. 1-20.

[66] Felipe Andres Manzano. *The Symbolic Maze !* 2010. url : https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/.

[67] Corina S. Păsăreanu et Neha Rungta. « Symbolic PathFinder : Symbolic Execution of Java Bytecode ». In : *IEEE/ACM International Conference on Automated Software Engineering*. 2010.

[68] Prateek Saxena et al. « A Symbolic Execution Framework for JavaScript ». In : *IEEE Symposium on Security and Privacy*. 2010.

[69] Junaid Haroon Siddiqui et Sarfraz Khurshid. « ParSym : Parallel symbolic execution ». In : *2010 2nd international conference on software technology and engineering*. T. 1. IEEE. 2010, p. V1-405.

[70] Cristian Cadar et al. « Symbolic execution for software testing in practice : preliminary assessment ». In : *International Conference on Software Engineering*. 2011.

[71] Jean-Christophe Filliâtre. « Deductive program verification ». In : *These d'habilitation, Université Paris* 11 (2011).

[72] Alexander J Summers et Peter Müller. « Freedom before commitment : a lightweight type system for object initialisation ». In : *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 2011, p. 1013-1032.

[73] Sang Kil Cha et al. « Unleashing Mayhem on Binary Code ». In : *IEEE Symposium on Security and Privacy*. 2012.

[74] Pascal Cuoq et al. « Frama-C : A software analysis perspective ». In : *International conference on software engineering and formal methods*. Springer. 2012, p. 233-247.

[75] Antoine Miné et Xavier Leroy. *Zarith*. 2012. url : https://github.com/ocaml/Zarith/.

[76] Julien Pauli, Nikita Popov et Anthony Ferrara. *PHP Internals Book - Basic structure - Types and values*. 2012. url : https://www.phpinternalsbook.com/php7/zvals/basic_structure.html.

[77] Cristian Cadar et Koushik Sen. « Symbolic Execution for Software Testing : Three Decades Later ». In : *Communications of the ACM* (2013).

[78] Jean-Christophe Filliâtre et Andrei Paskevich. « Why3—where programs meet provers ». In : *Programming Languages and Systems : 22nd European Symposium on Programming, ESOP 2013, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, p. 125-128.

[79] Gordon D Plotkin et Matija Pretnar. « Handling algebraic effects ». In : *Logical methods in computer science* 9 (2013).

[80] Ting Chen et al. « Conpy : Concolic Execution Engine for Python Applications ». In : *Algorithms and Architectures for Parallel Processing*. 2014.

[81] Guodong Li, Esben Andreasen et Indradeep Ghosh. « SymJS : Automatic Symbolic Testing of JavaScript Web Applications ». In : *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014.

[82] Emina Torlak et Rastislav Bodik. « A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages ». In : *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014.

[83] Jérôme Vouillon et Vincent Balat. « From bytecode to JavaScript : the Js_of_ocaml compiler ». In : *Software : Practice and Experience* 44.8 (2014), p. 951-972.

[84] Michal Zalewski. « Technical "whitepaper" for afl-fuzz ». In : *URl : http://lcamtuf.coredump.cx/afl/technical_details.txt* (2014).

[85] Koushik Sen et al. « MultiSE : Multi-path Symbolic Execution Using Value Summaries ». In : *Joint Meeting on Foundations of Software Engineering*. 2015.

[86] WebAssembly Community Group participants. *Binaryen*. 2015. url : https://github.com/WebAssembly/binaryen.

[87] Peter Müller, Malte Schwerhoff et Alexander J Summers. « Viper : A verification infrastructure for permission-based reasoning ». In : *Verification, Model Checking, and Abstract Interpretation : 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*. Springer. 2016, p. 41-62.

[88] Malte H Schwerhoff. « Advancing automated, permission-based program verification using symbolic execution ». Thèse de doct. ETH Zurich, 2016.

[89] Emilio Coppa, Daniele Cono D'Elia et Camil Demetrescu. « Rethinking pointer reasoning in symbolic execution ». In : *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, p. 613-618.

[90] Douglas Gregor, Slava Pestov et John McCall. *Implementing Swift Generics*. 2017. url : https://www.llvm.org/devmtg/2017-10/#talk15.

[91] Andreas Haas et al. « Bringing the web up to speed with WebAssembly ». In : *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, p. 185-200.

[92] Daniel Hillerström et al. « Continuation passing style for effect handlers ». In : (2017).

[93] Sebastian Markbåge. *OCamlrun WebAssembly*. 2017. url : https://github.com/sebmarkbage/ocamlrun-wasm.

[94] Samuel A. Rebelsky. *SamR's Assorted Musings and Rants : Using macros for generic structures in C*. 2017. url : https://rebelsky.cs.grinnell.edu/musings/cnix-macros-generics.

[95] Julien Signoles, Nikolai Kosmatov et Kostyantyn Vorobyov. « E-ACSL, a runtime verification tool for safety and security of C programs (tool paper) ». In : *RV-CuBES 2017-International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*. 2017.

[96] WebAssembly Community Group participants. *GC Proposal for WebAssembly*. 2017. url : https://github.com/WebAssembly/gc.

[97] WebAssembly Community Group participants. *Threads Proposal for WebAssembly*. 2017. URL : https://github.com/WebAssembly/threads.

[98] WebAssembly Community Group participants. *WebAssembly Reference Interpreter*. 2017. URL : https://github.com/WebAssembly/spec/tree/main/interpreter.

[99] Roberto Baldoni et al. « A Survey of Symbolic Execution Techniques ». In : *ACM Computing Surveys* (2018).

[100] Sylvain Conchon et al. « Alt-Ergo 2.2 ». In : *SMT Workshop : International Workshop on Satisfiability Modulo Theories*. 2018.

[101] The Guide To Rustc Development Contributors. *Guide to Rustc Development - Monomorphization*. Accessed : 2022-12-23. 2018. URL : https://rustc-dev-guide.rust-lang.org/backend/monomorph.html.

[102] Richard Musiol. *WebAssembly architecture for Go*. https://docs.google.com/document/d/131vjr4DH6JFnb-blm_uRdaC0_Nv3OUwjEY5qVCxCup4. 2018.

[103] José Fragoso Santos et al. « Symbolic Execution for JavaScript ». In : *International Symposium on Principles and Practice of Declarative Programming*. 2018.

[104] Axel Souchet. *Introduction to SpiderMonkey exploitation*. 2018. URL : https://doar-e.github.io/blog/2018/11/19/introduction-to-spidermonkey-exploitation/.

[105] Akira Tanaka, Reynald Affeldt et Jacques Garrigue. « Safe low-level code generation in Coq using monomorphization and monadification ». In : *Journal of Information Processing* 26 (2018), p. 54-72.

[106] WebAssembly Community Group participants. *Custom Annotations Proposal*. 2018. URL : https://github.com/WebAssembly/annotations.

[107] Luca Borzacchiello et al. « Memory models in symbolic execution : key ideas and new thoughts ». In : *Software Testing, Verification and Reliability* (2019). DOI : https://doi.org/10.1002/stvr.1722.

[108] « IEEE Standard for Floating-Point Arithmetic ». In : *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), p. 1-84. DOI : 10.1109/IEEESTD.2019.8766229.

[109] Yannis Lilis et Anthony Savidis. « A survey of metaprogramming languages ». In : *ACM Computing Surveys (CSUR)* 52.6 (2019), p. 1-39.

[110] Adrian D. Mensing et al. « From definitional interpreter to symbolic executor ». In : *International Workshop on Meta-Programming Techniques and Reflection*. 2019.

[111] Mark Mossberg et al. *Manticore : A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. 2019. arXiv : 1907.03890 [cs.SE].

[112] José Fragoso Santos et al. « JaVerT 2.0 : compositional symbolic execution for JavaScript ». In : *Proceedings of the ACM on Programming Languages* (2019).

[113] Bastien Abadie et Sylvestre Ledru. *Engineering code quality in the Firefox browser : A look at our tools and challenges*. 2020. URL : https://hacks.mozilla.org/2020/04/code-quality-tools-at-mozilla/.

[114] Carolina Costa. « Concolic Execution for WebAssembly ». Mém. de mast. https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/846778572212567 : Instituto Superior Técnico, nov. 2020.

[115] Robert Griesemer et al. « Featherweight go ». In : *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), p. 1-29.

[116] Árpád Perényi et Jan Midtgaard. « Stack-driven program generation of WebAssembly ». In : *Programming Languages and Systems : 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30–December 2, 2020, Proceedings 18*. Springer. 2020, p. 209-230.

[117] Igor Sheludko et Aboy Solanes. *GC Post-v1 Extensions - Readonly Fields*. 2020. url : https://v8.dev/blog/pointer-compression#value-tagging-in-v8.

[118] Dong Wang, Bo Jiang et W. K. Chan. *WANA : Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection*. 2020. arXiv : 2007.15510 [cs.SE].

[119] WebAssembly Community Group participants. *GC Post-v1 Extensions - Readonly Fields*. 2020. url : https://github.com/WebAssembly/gc/blob/main/proposals/gc/Post-MVP.md#readonly-fields.

[120] Allen Wirfs-Brock et Brendan Eich. « JavaScript : the first 20 years ». In : *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), p. 1-189.

[121] David Wood. « Polymorphisation : Improving Rust compilation times through intelligent monomorphisation ». Mém. de mast. University of Glasgow, avr. 2020. url : https://davidtw.co/media/masters_dissertation.pdf.

[122] Léo Andrès et al. *Owi*. 2021. url : https://github.com/ocamlpro/owi.

[123] Marek Chalupa, Jakub Novák et Jan Strejcek. « Symbiotic 8 : Parallel and Targeted Test Generation ». In : *Fundamental Approaches to Software Engineering* (2021).

[124] Gerd Stolpmann. *WASICaml*. 2021. url : https://github.com/remixlabs/wasicaml.

[125] WebAssembly Community Group participants. *JavaScript-Promise Integration Proposal for WebAssembly*. 2021. url : https://github.com/WebAssembly/js-promise-integration.

[126] WebAssembly Community Group participants. *Stack Switching Proposal for WebAssembly*. 2021. url : https://github.com/WebAssembly/stack-switching.

[127] Guannan Wei et al. « LLSC : A parallel symbolic execution compiler for LLVM IR ». In : *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, p. 1495-1499.

[128] Léo Andrès et Pierre Chambart. *Wasocaml*. 2022. url : https://github.com/ocamlpro/wasocaml.

[129] Haniel Barbosa et al. « cvc5 : A versatile and industrial-strength SMT solver ». In : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, p. 415-442.

[130] ocaml.org Contributors. *OCaml - Metaprogramming - PPXs*. 2022. url : https://ocaml.org/docs/metaprogramming#ppxs.

[131] *CPython. PyObject header file*. 2022. url : https://github.com/python/cpython/blob/main/Include/object.h.

[132] Xavier Denis, Jacques-Henri Jourdan et Claude Marché. « Creusot : a foundry for the deductive verification of rust programs ». In : *International Conference on Formal Engineering Methods*. Springer. 2022, p. 90-105.

[133] The Rust Project Developers. *The Rust Language Reference - Procedural Macros*. 2022. url : https://doc.rust-lang.org/reference/procedural-macros.html.

[134] Hichem Rami Ait El Hara, Guillaume Bury et Steven de Oliveira. « Alt-Ergo-Fuzz : A fuzzer for the Alt-Ergo SMT solver ». In : *33èmes Journées Francophones des Langages Applicatifs*. 2022.

[135] Stephen Ellis et al. « Generic go to go : dictionary-passing, monomorphisation, and hybrid ». In : *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022), p. 1207-1235.

[136] D Language Foundation. *String Mixins*. 2022. url : https://dlang.org/articles/mixin.html.

[137] Filipe Marques et al. « Concolic execution for webassembly ». In : *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2022.

[138] Antonio Nuno Monteiro. *Melange*. 2022. url : https://github.com/melange-re/melange.

[139] Louis Noizet et Alan Schmitt. « Necro ML : Generating OCaml Interpreters ». In : (2022).

[140] Cheng Shao. *WebAssembly backend merged into GHC*. https://www.tweag.io/blog/2022-11-22-wasm-backend-merged-in-ghc. 2022.

[141] The Dart Project Authors. *dart2wasm*. 2022. url : https://github.com/dart-lang/sdk/tree/main/pkg/dart2wasm.

[142] WebAssembly Community Group participants. *Reference-Typed Strings Proposal for WebAssembly*. 2022. url : https://github.com/WebAssembly/stringref.

[143] Léo Andrès et Pierre Chambart. *OCaml on WasmGC*. https://github.com/WebAssembly/meetings/blob/main/gc/2023/GC-01-10.md. 2023.

[144] Léo Andrès et Pierre Chambart. *Wasocaml : a compiler from OCaml to WebAssembly*. https://icfp23.sigplan.org/details/ocaml-2023-papers/13/Wasocaml-a-compiler-from-OCaml-to-WebAssembly. 2023.

[145] Léo Andrès, Pierre Chambart et Jean-Christophe Filliâtre. « Wasocaml : compiling OCaml to WebAssembly ». working paper or preprint. Juill. 2023. url : https://inria.hal.science/hal-04311345.

[146] Dirk Beyer. « Software Testing : 5th Comparative Evaluation : Test-Comp 2023 ». In : *International Conference on Fundamental Approaches to Software Engineering*. 2023, p. 309-323.

[147] Karthikeyan Bhargavan et al. « Foundations of WebAssembly ». In : (2023).

[148] AmirMohammad Deilami. « KLEEWT : A parallel symbolic execution engine ». In : (2023).

[149] Armaël Guéneau et al. « Melocoton : A program logic for verified interoperability between ocaml and c ». In : *Proceedings of the ACM on Programming Languages* 7.OOPSLA2 (2023), p. 716-744.

[150] Ningyu He et al. « Eunomia : Enabling User-Specified Fine-Grained Search in Symbolically Executing WebAssembly Binaries ». In : *ISSTA*. 2023.

[151] Ryan Hunt. *Pushing i31ref to post-MVP*. https://github.com/WebAssembly/gc/issues/320. 2023.

[152] Filipe Marques. *Smt.ml*. 2023. url : https://github.com/formalsec/smtml.

[153] Raphaël Monat, Abdelraouf Ouadjaout et Antoine Miné. « Mopsa-c : Modular domains and relational abstract interpretation for C programs (competition contribution) ». In : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2023, p. 565-570.

[154] David Munuera Mazarro. « Specification and verification of WebAssembly programs ». Thèse de doct. ETSI_Informatica, 2023.

[155] Aina Niemetz et Mathias Preiner. « Bitwuzla ». In : *International Conference on Computer Aided Verification*. Springer. 2023, p. 3-17.

[156] João Saraiva et João Paulo Fernandes. *Accepted Papers for presentation at IFL*. https://ifl23.github.io/accepted_papers.html. 2023.

[157] Jérôme Vouillon. *Wasm_of_ocaml*. 2023. url : https://github.com/ocaml-wasm/wasm_of_ocaml.

[158] Andy Wingo. *a world to win : webassembly for the rest of us*. https://www.wingolog.org/archives/2023/03/20/a-world-to-win-webassembly-for-the-rest-of-us. 2023.

[159] Andy Wingo. *Compiling Scheme to WasmGC*. https://github.com/WebAssembly/meetings/blob/main/gc/2023/GC-04-18.md. 2023.

[160] Andy Wingo. *Hoot*. 2023. url : https://gitlab.com/spritely/guile-hoot/-/blob/main/design/ABI.md.

[161] Léo Andrès et al. « Owi : Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly ». In : (2024).

[162] Edgar Aroutiounian et Olivier Pierre. *ocaml-emoji*. 2024. url : https://github.com/Swrup/ocaml-emoji.

[163] Nicolas Chataing et al. « Unboxed Data Constructors : Or, How cpp Decides a Halting Problem ». In : *Proceedings of the ACM on Programming Languages* 8.POPL (2024), p. 1509-1539.

[164] Ningyu He et al. « SeeWasm : An Efficient and Fully-Functional Symbolic Execution Engine for WebAssembly Binaries ». In : *arXiv preprint arXiv:2408.08537* (2024).

[165] Open Hub. *Chromium - Language Breakdown*. 2024. url : https://openhub.net/p/chrome/analyses/latest/languages_summary.

[166] Open Hub. *Google V8 JavaScript Engine - Language Breakdown*. 2024. url : https://openhub.net/p/v8-js/analyses/latest/languages_summary.

[167] Open Hub. *Mozilla Firefox - Language Breakdown*. 2024. URL : https://openhub.net/p/chrome/analyses/latest/languages_summary.

[168] Patrick Baudin et al. « Acsl : Ansi c specification language ». In : ().

[169] Colibri2 Team. *Colibri2*. URL : https://colibri.frama-c.com/.