Rapport TER : Compilation vers Wasm

Hugo Barreiro LDD3 Magistère Info
Sous la direction de Thibaut Balabonski

Janvier - Avril 2024

Table des matières

1 Introduction

L'objectif de ce TER était d'explorer une machine d'exécution de bas niveau basée sur une pile, telle la machine de WebAssembly, à travers la réalisation d'un compilateur produisant du bytecode pour cette machine.

La réalisation d'un compilateur se base sur un langage source, depuis lequel on peut effectuer une traduction vers un autre langage souvent plus bas niveau. Dans mon cas, je suis parti d'un langage minimal nommé Imp.

Afin de mener à bien cette traduction, j'ai dû ensuite comprendre le fonctionnement de la machine de WebAssembly.

Pour conclure ce TER, la question de l'implémentation d'un Garbage Collector à été étudiée.

Le code est organisé en 3 répertoires :
— `wasm`, contenant des exemples de programmes Wasm mettant en avant diverses particularités du langage
— `imp`, dans lequel est implémenté le compilateur de Imp vers Wasm
— `gc`, la base d'un Garbage Collector

L'archive de code est disponible sur *GitHub*.

2 Le langage minimal Imp

2.1 Fonctionnalités

Le langage Imp présente plusieurs fonctionnalités :
— entiers, booléens, tableaux
— opérations arithmétiques (unaires et binaires)
— branchements conditionnels (if then else)
— boucles (while avec break et continue)
— fonctions (impératives et récursives)

2.2 Spécifications

Le langage Imp utilise des entiers signés 32 bits. Les tableaux sont homogènes et non-redimensionnables.

Il y a plusieurs fonctions dans la "bibliothèque standard" :
— `print`, attend une expression et affiche sa valeur (entière ou booléenne)
— `len`, attend un tableau et renvoie sa longueur
— `malloc`, attend un type (correspondant au type du tableau renvoyé) et un nombre supérieur ou égal à 0 d'entiers décrivant les dimensions du tableau à allouer.

Malgré l'obligation d'écrire les types dans son programme, le langage n'est actuellement pas typé. Néanmoins, il peut facilement le devenir en rajoutant un typechecker en utilisant les types présent dans l'AST (Arbre de la Syntaxe Abstraite) Imp.

3 Wasm

3.1 Présentation générale

WebAssembly est un langage à pile. C'est-à-dire qu'un programme est une séquence d'instructions dont son objectif est d'empiler des valeurs afin d'effectuer des opérations jusqu'à obtenir le résultat souhaité.

La syntaxe de Wasm s'inspire de la famille des langages Lisp. Un fichier Wasm est un module dans lequel on va pouvoir écrire tout notre programme.

Wasm est un langage typé. Ainsi, il existe notamment 4 types de valeurs courantes en Wasm : `i32`, `i64`, `f32`, `f64` (respectivement entier 32 et 64 bits, flottant 32 et 64 bits). On ne considérera que les entiers 32 bits (`i32`).

Ainsi, le typechecker vérifie que l'état de la pile est cohérent grâce aux annotations de types du programme. Par exemple, si une séquence doit empiler un entier sur la pile alors cela est vérifié.

De plus, le langage est muni de plusieurs instructions. On retrouve notamment les opérations usuelles d'arithmétiques (addition, soustraction, multiplication, etc...) et de comparaisons (et, ou, supérieur à, etc...). Ainsi que bien d'autres que l'on va présenter.

3.2 Un langage bas niveau assez riche

Des exemples de code Wasm sont présents dans l'archive de code.

3.2.1 Opérations de base

`i32.const n` place l'entier `n` sur le sommet de la pile.

Imaginons la pile suivante : `[3; 2; 1] <-- sommet`. Alors, les opérations binaires usuelles `binop` consomment 2 et 1 et place le résultat de `2 binop 1` sur la pile. Ainsi, pour les opérations non commutatives comme la soustraction et la division, l'ordre des éléments sur la pile est important. Par exemple, `i32.sub` place `1` sur la pile.

L'instruction `drop` supprime l'élément sur le sommet de la pile (lève une erreur si la pile est vide).

3.2.2 Branchements conditionnels

Il existe des branchements conditionnels en Wasm, notamment le `if then else`. La syntaxe est la suivante :

```
(if (then some_instructions) (else some_instructions))
```

Ainsi, un entier doit être présent sur le sommet de la pile. Il est alors consommé ; s'il est égal à 0 alors on exécute la séquence `else` sinon la séquence `then`.

Néanmoins, on ne pourrait pas empiler de valeur sur la pile avec un tel `if then else` car aucune annotation de types n'est présente.

C'est pour cela, qu'un `if then else` peut être paramétré par un type de retour grâce au mot-clef `result`. Ainsi, nous pouvons empiler un entier sur la pile avec cette syntaxe : `(if (result i32) (then i32.const 0) (else i32.const 1))`. Si l'annotation `result i32` n'était pas présente, alors il y aurait une erreur de typage.

3.2.3 Boucles et blocs

Il existe 2 sortes de boucles en Wasm : `loop` et `block`. La syntaxe est la suivante `(loop $loop some_instructions)` et `(block $block some_instructions)`. Il n'est pas obligatoire de nommer une boucle même si cela s'avère souvent utile.

Le principal intérêt de `loop` et `block` est d'utiliser l'instruction `br` qui permet d'effectuer un jump sur une boucle. Néanmoins, il y a une différence, jump sur `loop` permet de revenir au début de la boucle alors que jump sur `block` permet d'ignorer la fin de la boucle et de continuer le programme.

Ainsi, en combinant `loop`, `block` et `if then else`, on peut simuler les boucles `for` et `while` contenant même les mots-clefs `break` et `continue`.

3.2.4 Variables et fonctions

On peut créer des variables globales de la façon suivante (l'initialisation est obligatoire) :

```
(global $v0 (mut i32) (i32.const 0))
```

On peut alors accéder à la valeur de cette variable avec `global.get $v0` et la modifier avec `global.set $v0 (i32.const 1)` par exemple.

Il existe des fonctions en Wasm (elles sont récursives). Il faut préciser le type de retour s'il y en a un (c'est-à-dire si on laisse un entier en plus sur la pile à la fin de l'exécution de la fonction).

Ainsi, on peut alors avoir des paramètres avec le mot-clef `param` et des variables locales avec `local`. Leur fonctionnement est semblable aux variables globales au détail près qu'on ne peut pas les initialiser lors de leur déclaration.

On a alors besoin de l'instruction `call` afin d'effectuer un appel de fonction. On dispose également de l'instruction `return` permettant de renvoyer une valeur de manière anticipée (un `return` en fin de fonction n'est pas obligatoire).

Petite subtilité lorsque l'on mélange `if then else` et `return`. En effet, si on a un branchement conditionnel qui contient un renvoi anticipé dans les 2 branches alors on doit l'annoter d'un type de résultat. Sinon, si notre branchement ne comporte qu'un renvoi anticipé (ou 0) alors on ne doit pas l'annoter de type de résultat.

Il existe une instruction pratique pour structurer un programme Wasm : `start`. Cette instruction permet d'indiquer quelle fonction débute l'exécution du programme parmi toutes celles du module.

3.2.5 Mémoire

Un module Wasm possède une mémoire unique que l'on doit déclarer au début du module. Elle est comptabilisée en pages (on doit préciser combien de pages on veut).

On possède donc des instructions permettant d'écrire dans les cases mémoires et de charger leur valeur. En effet, la mémoire Wasm est un grand tableau.

3.3 Format des fichiers

Il existe plusieurs formats de fichiers Wasm.

Celui par défaut est l'écriture binaire. Ce n'est pas ce que j'ai retenu.

Il existe alors les fichiers `.wat` qui supporte les syntaxes présentées jusqu'ici. Néanmoins, ce n'est pas ce que j'utilise non plus car il n'y a pas de fonction d'affichage en Wasm "pur".

Pour cela, nous devons utiliser une fonction fournie par notre interpréteur et faire du Wasm "script" qui permet d'utiliser des bouts de JavaScript sous le capot. C'est le format `.wast`, que j'utilise.

4 Traduction Imp vers Wasm

Le code concerné se trouve dans le répertoire `imp`.

4.1 Architecture du compilateur

Le compilateur Impc qui traduit un programme Imp vers un programme Wasm est organisé en plusieurs fichiers :

- `impc.ml` le fichier principal du compilateur qui prend en entrée un fichier .imp et produit un fichier .wast
- `implexer.mll` et `impparser.mly` pour obtenir l'AST Imp d'un programme Imp
- `imp2wasm.ml` qui produit un AST Wasm à partir d'un AST Imp

— `print.ml` qui écrit le fichier .wast correspondant au programme décrit par un AST Wasm
— `imp.ml` et `wasm.ml` qui contiennent respectivement la description des AST de Imp et Wasm

Il existe 2 options de compilation :
— `--run` pour exécuter directement le programme Wasm produit
— `--debug` pour exécuter directement en mode débogage le programme Wasm produit

On utilise l'interpréteur *Owi* pour exécuter nos fichiers `.wast`.

4.2 Traduire vers Wasm

4.2.1 Traductions basiques

On représente nos entiers 32 bits Imp par les entiers 32 bits Wasm. Quant aux booléens, ceux-ci sont représentés par des entiers, en particulier 0 pour `false`, et 1 pour `true`.

Les opérations de bases sont traduites de façon directe. Par exemple l'addition 1 + 2 représentée `Add(Int 1, Int 2)` dans l'AST Imp est traduite de la façon suivante :

```
S( S( I(I32 1), I(I32 2) ), I(Op Add) )
```

En effet, on doit d'abord empiler 1 et 2 sur la pile, puis calculer le résultat de l'addition. On retrouve des constructions similaires pour les autres opérations.

Les traductions du `if then else`, des variables et des fonctions sont également assez élémentaires car Wasm contient déjà ces types de constructions.

4.2.2 Boucle while

Pour obtenir un `while` il suffit de combiner `loop`, `block` et `if then else`.

En effet, on imbrique un `loop` dans un `block`. Puis, on empile la condition du `while`. Si la condition renvoie `true`, alors on exécute la séquence d'instructions puis on jump sur la `loop`, sinon on jump sur le `block` ce qui permet de sortir du `while`.

Ainsi, on traduit `continue` par un jump sur la `loop` et `break` par un jump sur le `block`.

4.2.3 Les tableaux

La traduction des tableaux est la première traduction un peu plus subtile, en effet cela demande de manipuler la mémoire d'un module Wasm, ce qui peut être délicat.

Tout d'abord, par défaut, aucune mémoire n'est disponible. Ainsi, il faut au début du module réserver de la mémoire. Pour simplifier l'implémentation, j'ai choisi une taille

arbitraire fixe.

Ensuite, il faut savoir que la mémoire est simplement un grand tableau. De plus, nous disposons uniquement d'une instruction pour charger la valeur d'une case et une autre pour écrire dans une case.

Ainsi, notre propre allocateur doit être implémenté. J'ai choisi d'utiliser la première case de la mémoire pour stocker l'offset courant. Puis, j'alloue les tableaux les uns à la suite des autres en incrémentant l'offset courant (on réserve une case supplémentaire par tableau pour stocker leur taille).

De cette façon, on peut allouer de la mémoire pour obtenir des tableaux mais on ne peut à aucun moment la libérer, ce qui nous amènera à nous poser la question d'un Garbage Collector.

On peut remarquer que cette implémentation autorise les tableaux de tableaux. En effet, comme toutes les valeurs sont des entiers, un tableau est un entier représentant l'adresse de la case de départ dans la mémoire. Ainsi, pour un tableau à 2 dimensions, un premier tableau stocke l'adresse des autre tableaux qui eux contiennent les valeurs.

4.3 Traduction plus complexe

J'ai voulu avoir dans ma "bibliothèque standard" une fonction qui alloue un tableau de dimensions $n_1 \times n_2 \times ... \times n_k$ (comme en Java). Néanmoins, cela n'est pas du tout pratique à traduire directement pour des raisons techniques. J'ai dû alors, pour la première fois re-travailler sur l'AST Imp au lieu de le transformer directement en AST Wasm.

Ainsi, lorsque je rencontre une allocation de tableaux à plus d'une dimension, je la décompose en plusieurs allocations à une dimension à l'aide de boucles `while` (comme on le ferait en C par exemple). Cette forme d'allocation à plusieurs dimensions est elle facile à transformer en AST Wasm puisqu'elle n'est composée que d'allocations simples et de boucles `while`.

5 Garbage Collector

Il a été question d'implémenter un Garbage Collector pour gérer la mémoire. Néanmoins, des problèmes ont été rencontrés.

5.1 Difficultés

La première question à se poser lorsque l'on souhaite implémenter un Garbage Collector devrait être : ai-je accès aux variables de mon programme ? En effet, cela peut s'avérer être utile notamment si on se lance dans des travaux d'envergure alors que ce n'est pas le cas...

En Wasm, il y a deux piles : celle de travail dans laquelle on empile nos valeurs pour effectuer nos calculs et une cachée à laquelle nous n'avons pas accès qui contient les variables de notre programme. Ainsi, nous ne pouvons pas implémenter notre GC.

De plus, même si on avait accès à ces variables, on ne pourrait toujours pas implémenter notre Garbage Collector car on ne peut pas inspecter une pile en Wasm. On ne peut alors pas mettre à jour les éventuels pointeurs empilés.

5.2 Utiliser OCaml

Une première idée a été d'utiliser l'interpréteur Owi et sa fonctionnalité permettant d'écrire des fonctions en OCaml afin de les utiliser dans notre programme Wasm.

Néanmoins, cela ne résout pas vraiment notre problème et ne semble que le déplacer.

5.3 Solution

Une solution est de simuler 2 shadow stacks. C'est-à-dire de gérer deux piles dans notre mémoire et de n'utiliser que celles-ci. Ainsi, on aura accès à toutes les variables et à tous les pointeurs du programme.

Une partie de l'implémentation à été effectuée. Les fonctions principales de Garbage Collection de type Stop and Copy ont été codées en Wasm. La mémoire est bien sectionnée en 4 zones équivalentes (2 zones pour le Stop and Copy et 2 zones pour les shadow stacks).

Tout cela a été laissé inachevé dans le répertoire `gc`.

6 Conclusion

Ce TER m'a permis d'approfondir les connaissances acquises lors du cours de compilation. Notamment la manipulation d'Arbre de Syntaxe Abstraite, et plus particulièrement la compréhension de la traduction d'un langage vers un autre. J'ai également pu davantage m'intéresser à la manipulation plus bas niveau de la mémoire à travers l'implémentation des tableaux et du Garbage Collector.

Cela a été aussi l'occasion de m'exercer en programmation tout en développant mes qualités d'autonomie et de recherche dans l'objectif de trouver des renseignements sur un sujet peu documenté.

Pour conclure, je tiens à remercier M. Balabonski de m'avoir encadré et aidé pendant toute la durée de ce TER.