Sorbonne Université

PSTL
Rapport

Un Langage "Pur" pour Web Assembly

Élève :
Lucas Fumard
Lauryn Pierre
Saïd Mohammad ZUHAIR

Enseignant :
Frédéric Peschanski

30 mars 2023

Table des matières

1 Introduction

WebAssembly ou Wasm [5] est un nouveau langage bytecode pris en charge par les principaux navigateurs Web, conçu principalement pour être une cible de compilation efficace pour les langages de bas niveau tels que C/C++ et Rust[1]. WebAssembly a un format texte(.wat) et un format binaire(.wasm). Dans le cas de notre projet nous utilisons wabt[4] pour compiler et nous exécutons le code compilé à l'aide de Node.js [2]. Le but de notre projet est de concevoir un langage 100 % fonctionnel et "pur" pour WebAssembly en se basant sur le langage défini dans cet article[3]. L'article défini un langage fonctionnel dont la gestion de la mémoire se fait par un mécanisme de comptage de références.

2 Concepts fondamentaux

Dans cette partie, nous allons présentons les différentes notions nécessaire à la compréhension du projet.

2.1 WebAsssembly

2.2 Présentation de l'article

3 Cahier des charges

Les tâches que nous avons identifié sont les suivantes :
- Analyser le fonctionnement de WASM
- Programmer un parseur qui puisse lire le langage pur tel que défini dans l'article[3]
- Programmer un interpréteur en Rust du langage selon les sémantiques du langage pur
- Définir quelques tests unitaires couvrant les sémantiques définies dans l'article
- Ajouter la gestion des instructions `inc`, `dec`, `reset`, `reuse`
- Programmer un compilateur du langage agrandi vers WASM

4 Tâches Réalisées

4.1 Lecture de l'article

Le langage fonctionnel décrit dans l'article[3] alloue ses constructeurs dans la pile et manipule des adresses vers ces emplacements mémoire alloués. Il est donc primordial d'avoir un système d'allocation et réutilisation de mémoire performant afin d'éviter les fuites mémoires et un temps d'exécution faible.

Le système de la gestion de la mémoire par comptage de référence est bien plus vieux que des systèmes par garbage collector, mais aussi plus efficaces[3]. Cependant, la gestion de mémoire comptage de références ne fonctionne que si il n'est pas possible de créer de cycle de référence. C'est pourquoi les garbage collector sont plus utilisés de nos jours.

Les auteurs de l'article[3] ont créé un langage fonctionnel dans lequel les cycles de références sont impossibles afin d'implémenter un système de gestion de mémoire par comptage de référence et d'obtenir un langage fonctionnel dont l'exécution est optimisée. C'est ce langage que nous allons implémenter.

4.2 Parser/reader

Concernant la grammaire du langage, nous avons décidé de garder celle définie à la section 3 de l'article que voici.

$$
\begin{aligned}
&w, x, y, z \in \mathit{Var} \\
&\quad c \in \mathit{Const} \\
&\quad e \in \mathit{Expr} \quad ::= c\ \overline{y} \mid \mathbf{pap}\ c\ \overline{y} \mid x\ y \mid \mathbf{ctor}_i\ \overline{y} \mid \mathbf{proj}_i\ x \\
&\quad F \in \mathit{FnBody} ::= \mathbf{ret}\ x \mid \mathbf{let}\ x = e;\ F \mid \mathbf{case}\ x\ \mathbf{of}\ \overline{F} \\
&\quad f \in \mathit{Fn} \quad ::= \lambda\ \overline{y}.\ F \\
&\quad \delta \in \mathit{Program} = \mathit{Const} \rightharpoonup \mathit{Fn}
\end{aligned}
$$

Notre reader utilise la bibliothèque Chumsky[6], ainsi nous ne pouvons pas stoquer les noms définis variable ce qui nous empêche de distinguer directement les constantes des variables.

4.3 Interpréteur

Afin d'implémenter un interpréteur du langage décrit, il nous a d'abord fallu implémenter les structures permettant d'accéder à la mémoire, dont les structures du tas et de la pile (`Heap` et `Ctxt`)

$$
\begin{aligned}
&l \in \mathit{Loc} \\
&\rho \in \mathit{Ctxt} \quad = \mathit{Var} \rightharpoonup \mathit{Loc} \\
&\sigma \in \mathit{Heap} \quad = \mathit{Loc} \rightharpoonup \mathit{Value} \times \mathbb{N}^+ \\
&v \in \mathit{Value} ::= \mathbf{ctor}_i\ \overline{l} \mid \mathbf{pap}\ c\ \overline{l}
\end{aligned}
$$

`Loc` est l'adresse de la case mémoire allouée pour la valeur dans le tas. C'est la valeur de retour de toutes les sémantiques du langage, tel que définies dans la figure 1 de l'article[3]. Contrairement à l'article, nous avons choisi de ne pas retourner un nouveau tas à chaque fois, mais de garder un même tas que l'on modifie par effet de bord.

Nous avons testé cet interpréteur en créant plusieurs tests unitaires sur les sémantiques mais aussi quelques programmes simples, tel que le calcul de fibonacci, ou le programme `swap` défini à la page 5 de l'article[3].

WebAssembly ne possède que des types numériques (i32, f32, ...) ainsi il est compliqué d'implémenter les applications partielles. Ainsi, nous nous réservons le droit de ne pas les implémenter.

Afin d'implémenter le tas en WebAssembly, nous utilisons une `Table` qui nous permet de définir une table dans la mémoire que nous pouvons manipuler de WebAssembly ainsi que de JavaScript. Cette table nous permettra de réaliser des diagnostiques ainsi que des évaluations de l'empreinte mémoire. Nous avons commencé à définir le schéma du tas du code compilé : La case mémoire 0 est réservée pour indiquer le prochain espace mémoire libre. Ensuite, chaque constructeur suit le format suivant : `<type> <nb_ref> <args>`. Ainsi, un constructeur `FALSE` référencé deux fois est représenté sous la forme "0 2" dans la mémoire. Un entier 10 référencé 3 fois sous la forme "4 3 10".

5 Tâches Restantes

Il nous faut choisir un schéma de mémoire pour les objets dans la mémoire. Il nous faut implémenter `inc`, `dec`, `reset`, `reuse`. Il nous faut faire le compilateur en WASM

WASM n'a qu'un seul type de variable : les nombres (entiers ou flottant, 32 bit ou 64 bit). Ainsi, il faut que l'on interprète différement ces nombres selon le contexte dans lequel on les prend. Dans la mémoire, nos constructeurs et applications partielles seront stoqués sous la forme `<type> <nombre de références> <arguments>`. Par exemple, une application partielle est sous la forme `6 1 7 2 12 13` où 6 est le type "application partielle", 1 est le nombre de références à cette application partielle, 7 est l'identifiant de la fonction à appeler (dans l'interpréteur, on utilise des `string`), 2 est le nombre d'arguments fixés et 12 et 13 sont les arguments fixés.

Dû à la difficulté d'une telle tâche, nous laissons l'implémentation des applications partielles pour plus tard.

Références

[1] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In : *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA : Association for Computing Machinery, 14 juin 2017, p. 185-200. ISBN : 978-1-4503-4988-8. DOI : 10.1145/3062341.3062363. URL : https://dl.acm.org/doi/10.1145/3062341.3062363 (visité le 16/03/2023).

[2] *Node.js Et WebAssembly*. Node.js Et WebAssembly. Section : Node.js. URL : https://nodejs.dev/fr/learn/nodejs-with-webassembly/ (visité le 16/03/2023).

[3] Sebastian Ullrich et Leonardo de Moura. *Counting Immutable Beans : Reference Counting Optimized for Purely Functional Programming*. 5 mars 2020. DOI : 10.48550/arXiv.1908.05647. arXiv : 1908.05647[cs]. URL : http://arxiv.org/abs/1908.05647 (visité le 08/03/2023).

[4] *WABT : The WebAssembly Binary Toolkit*. original-date : 2015-09-14T18 :14 :23Z. 16 mars 2023. URL : https://github.com/WebAssembly/wabt (visité le 16/03/2023).

[5] *WebAssembly*. URL : https://webassembly.org/ (visité le 12/02/2023).

[6] zesterer. *Chumksy*. URL : https://docs.rs/chumsky/latest/chumsky/ (visité le 08/03/2023).