Sorbonne Université

PSTL
Rapport

Un Langage "Pur" pour Web Assembly

Élève :
Lucas Fumard
Lauryn Pierre
Saïd Mohammad ZUHAIR

Enseignant :
Frédéric Peschanski

4 mai 2023

Table des matières

1 Introduction

Le but de notre projet est de concevoir un langage 100 % fonctionnel et “pur” pour WebAssembly en se basant sur cet article[3](fourni). L'article défini un langage fonctionnel dont la gestion de la mémoire se fait par un mécanisme de comptage de références. WebAssembly[10] ou Wasm définit un format de code binaire portable et un langage de type assembleur[9]. Tous les principaux navigateurs peuvent exécuter des programmes WebAssembly. Des langages comme C, C++, Rust, Go et bien d'autres peuvent être compilés en WebAssembly.

2 WebAsssembly

JavaScript est le seul langage de programmation native au Web. Pour de nombreuses raisons, JavaScript n'est pas idéal pour être une cible de compilation efficace pour les langages de bas niveau tels que C/C++ et Rust. WebAssembly a plusieurs objectifs[1]. Être sûre, les programmes WebAssembly sont isolées de leur environnement hôte. WebAssembly est conçu pour être lié à aucun runtime ou langages de programmation, de sorte qu'il peut être exécuté sur n'importe quel appareil qui le prend en charge et d'avoir le même comportement. Ce qui le rend intéressant pour la création d'applications multi-plateformes. Les programmes pouvant être exécutées sur des ordinateurs de bureau, des appareils mobiles, et même des serveurs. WebAssembly fonctionne avec les technologies Web existants, par exemple JavaScript, CSS, et HTML. WebAssembly est un langage bytecode portable de bas niveau pris en charge par les principaux navigateurs Web. Les utilisations de WebAssembly ne se limitent pas qu'au Web, il y a aussi un intérêt pour l'Internet des Objets, les serveurs, les systèmes embarqués.

WebAssembly a un format texte (.wat) et un format binaire (.wasm). WebAssembly peut être exécuté dans différents environnements, tels que les serveurs, les navigateurs web ou les applications. Dans les navigateurs web, WebAssembly est exécuté en même temps que le code JavaScript. Pour les serveurs, WebAssembly est exécuté en utilisant des runtimes tels que Node.js. Plusieurs compilateurs ont été développés pour WebAssembly[4, 5, 7, 8]. Wasmer[7] et wasmtime[8] sont les plus connues. Ils utilisent tous les deux la technique de la compilation à la volée. Wasmtime est développé en collaboration supervisée par la fondation Mozilla. Alors que wasmer est développé par une entreprise privée. Dans le cadre de notre projet, on utilise wabt[4] pour compiler et on exécute le compilé à l'aide de Node.js[2].

Les programmes sont composés d'un ou plusieurs modules. Un module contient la définition des fonctions, variables globales. Les définitions peuvent être importées ou exportées. Ils interagissent avec l'environnement à l'aide d'import et d'export explicites. Un module doit être validé avant l'exécution pour s'assurer qu'il est bien typé et sûr d'exécuter. WebAssembly possède un système de type statique centré autour de quatre valeurs : i32, i64, f32 et f64. Qui désignent les nombres entiers de 32 bits et 64 bits et les flottants de 32 et 64 bits. La spécification officielle de Wasm comprend une sémantique formelle pour le langage, avec une déclaration précise de la propriété de solidité des types prévue. Elle a d'abord été publiée dans un premier projet en 2017, puis dans la norme officielle, appelée WebAssembly 1.0 (Wasm 1.0), en 2019.

La mémoire d'un programme Wasm repose sur le modèle de mémoire linéaire[1]. La mémoire linéaire est un tampon continu d'octets non signés que JavaScript et Wasm

peuvent lire et modifier de manière synchrone. Au cours de l'exécution, l'espace mémoire peut grandir.

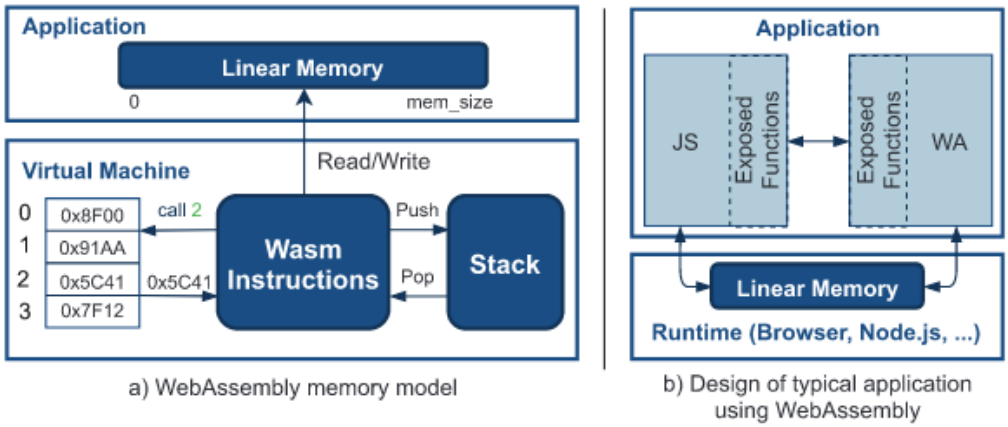

FIGURE 1 – WebAssembly Architecture [6]

3 Cahier des charges

Les tâches que nous avons identifiées sont les suivantes :
- Analyser le fonctionnement de WASM et étudier son écosystème.
- 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 ne 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. Ils définissent un langage source λ_{pure}. Qui, après une étape de compilation, deviendra λ_{RC}, notre langage de destination. λ_{RC} est une extension de λ_{pure} auquel on a ajouté

les instructions de gestion de la mémoire : `inc`, `dec`, `reset`, `reuse`. Notre interpréteur implémentera λ_{pure}.

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

FIGURE 2 – Grammaire du langage source λ_{pure}

$$
\begin{aligned}
e \in \mathit{Expr} \quad &::= \ldots \mid \mathbf{reset}\ x \mid \mathbf{reuse}\ x\ \mathbf{in}\ \mathbf{ctor}_i\ \overline{y} \\
F \in \mathit{FnBody} &::= \ldots \mid \mathbf{inc}\ x;\ F \mid \mathbf{dec}\ x;\ F
\end{aligned}
$$

FIGURE 3 – Grammaire du langage λ_{RC}

4.2 Parser/reader

Concernant la grammaire du langage, nous reprenons celle qui est définie dans la section 3 de l'article et retranscrit dans la Figure 2.

Notre reader utilise la bibliothèque Chumsky[11], ainsi, nous ne pouvons pas stoker les noms définis variables, ce qui nous empêche de distinguer directement les constantes des variables, car sans historique de définition de variable, tous les mots en paramètres de fonction sont lus en tant que `Const`.

C'est pourquoi nous avons besoin de transformer certaines constantes et certains appels de fonctions en variables et applications partielles à l'aide de fonction d'explorations définies dans le fichier `transform_var.rs`. Les variables définies avec un `let` sont donc ainsi représentées par `Var` au lieu de `Const` dans l'AST à interpréter.

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`).

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

FIGURE 4 – Structures de gestion de la mémoire

`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.

4.4 Pile

La pile est implémentée par un énumérateur `Ctxt` contenant les champs pour le nom de la variable, l'emplacement de la variable dans le tas, ainsi que la suite de la pile. On agrandi cette pile en l'englobant dans un nouvel énumérateur `Ctxt` contenant les champs de la nouvelle variable ainsi que l'ancienne pile.

Nous ne transmettons pas la pile crée à l'appelant, ainsi la pile est automatiquement dépilée lorsque l'on quitte le block d'affectation de variable.

Pour chercher une variable, nous parcourons la pile de l'exterieur vers l'intérieur de façon récursive jusqu'à obtenir la variable souhaitée ou arriver à la fin de la pile, dans quel cas l'interpréteur produit une erreur.

Ainsi, nous obtenons une structure de type FILO caractérisant le comportement d'une pile.

4.5 Tas

Nous avons choisi de diverger un peu de l'article[3] en introduisant un tas changé par effet de bord, et non passé en sortie de block afin de rendre une seule valeur de retour.

Ce tas `Heap` est une structure contenant le nombre totale d'allocations ainsi qu'une table de hashage d'un entier vers un `Value`.

Cette structure permet de lier des emplacements et les valeurs et fonctionne avec `Ctxt`.

Les auteurs de l'artice stockent un couple `(Valeur, entier)`, où l'entier est le nombre de références de l'objet. Nous l'implémentons dans le compilateur, mais comme l'interpréteur n'adapte que la partie pure du langage, le nombre de références n'est jamais modifié. De ce fait, nous avons préféré enlevé la présence de ce nombre dans l'interpréteur pour simplifier la lecture du code.

Ainsi, `Ctxt` permet de prendre une référence associée à un nom de variable et `Heap` permet de prendre la valeur associée à cette référence.

4.6 AST

Nous avons implémenté en Rust l'AST du langage proposé par les auteurs, en en étendant les expression (`Expr`) pour ajouter la notion d'entiers à notre interpréteur, sou la forme `Expr::Num(i32)` en Rust afin de tester concrètement notre interpréteur à partir de langages lus de fichiers.

Les énumérateurs feuilles dans notre AST sont les variables `Var` et les constantes `Const`, qui contiennent chacunes une chaîne de charactère décrivant le nom de la fonction pour les constantes et le nom de la variable pour les variables. La valeur de la variable peut être récupérée dans le tas une fois que l'on a l'adresse, obtenue à partir de la pile et de la chaîne de charactère.

Une expression peut être un appel de fonction (`FnCall(Const, Vec<Var>)`), un appel de fonction partielle (`PapCall(Var, Var)`), une fonction partielle (`Pap(Const, Vec<Var>)`), un constructeur (`Ctor(i32, Vec<Var>)`), l'obtention d'un champ d'un objet ((`Proj(i32, Var)`)), ou un entier (`Num(i32)`).

Chaque constructeur est défini par un entier différent, détaillé par les constantes globales ci-dessous.

```
CONST_FALSE : i32 = 0;
CONST_TRUE  : i32 = 1;
CONST_NIL   : i32 = 2;
CONST_LIST  : i32 = 3;
CONST_NUM   : i32 = 4;
```

Ainsi, nous avons défini les booléens `False` et `True` par les entiers `0` et `1` respectivement.

Nous utilisons des identifiants plutôt qu'une énumération où l'on pourrait définir chaque type proprement car c'est comme cela que nous allons stocker les types en mémoire en WASM.

Une fois le contructeur créé, nous pouvons accéder aux champs du contructeur avec `Proj`. Il est à noter que seules les listes ont des champs, car ce sont les seuls constructeur qui prennent des variables en paramètres pour être créés.

Une application partielle `Pap` est un appel de fonction dans lequel il manque des arguments, et un appel de fonction partiel `PapCall` est une application partielle désignée par une variable à laquelle nous ajoutons une autre variable, augmentant le nombre d'arguments de un. Une fois qu'il y a assez d'arguments dans le vecteur d'arguments de l'application partielle, la fonction désignée est exécutée, avec tous les arguments.

En remonttant dans l'AST, nous trouvons les corps de fonction qui peuvent être le retour de fonction d'une variable `Ret(Var)`, une affectation de variable `Let(Var, Expr, Box<FnBody>)` ou un match du type d'une variable `Case(Var, Vec<FnBody>)`.

L'affectation interprète son expression afin d'injecter la valeur en résultant dans le tas, donc l'adresse est associée au nom de sa variable, sur la pile. Ainsi, cette variable référencera la valeur de l'expression dans la suite de la fonction.

Pour savoir quel branche du `Case` il faut exécuter, il suffit tout simplement de regarder l'identifiant du type du constructeur. Ainsi, si l'on veut exécuter du code pour les listes, il faut d'abord remplir les branches de `False`, `True` et `Nil`. Il n'est pas nécessaire de remplir la branche pour les entiers si on considère qu'aucun entier ne sera jamais évalué dans ce `Case`. Le cas échéant, l'interprète lance une erreur et le comportement du langage compilé en WASM est indéterminé.

Le corps de fonction à la racine de la fonction, encapsulant tous les corps suivant, est lui-même encapsulé dans la définition d'une fonction `Fn` composée d'une liste (un vecteur en Rust) de variables `Vec<Var>` ainsi que d'un corps de fonction `FnBody`.

Enfin, à la racine de notre AST, nous définissons un `Program` en Rust comme une `IndexMap<Const, Fn>` liant des constantes `Const` à des définitions de fonction `Fn`.

4.7 Interprétation

TODO : interprétation des différents éléments de l'AST

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

4.8 Compilateur

WebAssembly ne possède que des types numériques (i32, f32, i64, f64) 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 diagnostics 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 différemment ces nombres selon le contexte dans lequel on les prend. Dans la mémoire, nos constructeurs et applications partielles seront stockés 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] *Wasm3*. original-date : 2019-10-01T17 :06 :03Z. 30 mars 2023. URL : https://github.com/wasm3/wasm3 (visité le 30/03/2023).

[6] *Wasmati : An efficient static vulnerability scanner for WebAssembly | Elsevier Enhanced Reader*. DOI : 10.1016/j.cose.2022.102745. URL : https://reader.elsevier.com/reader/sd/pii/S0167404822001407?token=1A424725E4364C14A4E8F87B68C935originRegion=eu-west-1&originCreation=20230316195601 (visité le 16/03/2023).

[7] *Wasmer - The Universal WebAssembly Runtime*. URL : https://wasmer.io/ (visité le 16/03/2023).

[8] *Wasmtime*. URL : https://wasmtime.dev/ (visité le 16/03/2023).

[9] *WebAssembly*. In : *Wikipedia*. Page Version ID : 1133857733. 15 jan. 2023. URL : https://en.wikipedia.org/w/index.php?title=WebAssembly&oldid=1133857733 (visité le 12/02/2023).

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

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