Sorbonne Université

PSTL
Rapport

Un Langage "Pur" pour Web Assembly

Élève :
Lucas Fumard
Lauryn Pierre
Saïd Mohammad ZUHAIR

Enseignant :
Frédéric Peschanski

12 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.

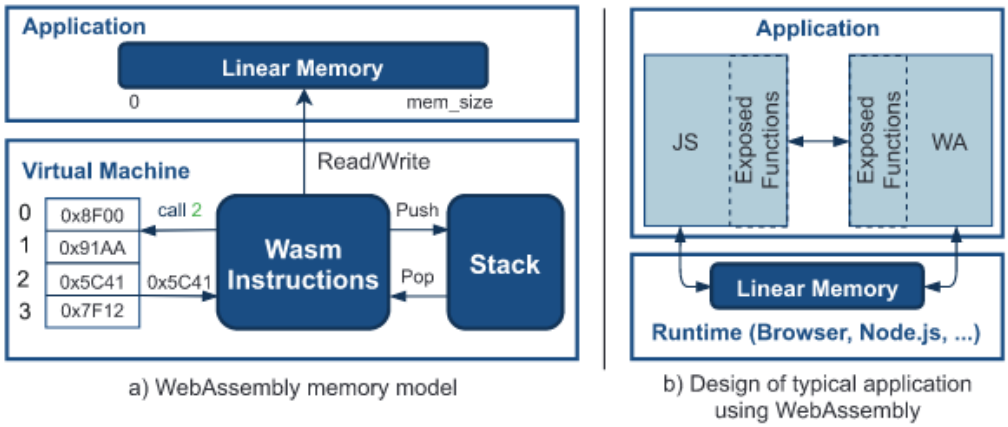

FIGURE 1 – WebAssembly Architecture [6]

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 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}.

5 AST

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

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

FIGURE 2 – Grammaire du langage source λ_{pure}

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.

$$
\begin{aligned}
CONST\_FALSE &= 0 \\
CONST\_TRUE &= 1 \\
CONST\_NIL &= 2 \\
CONST\_LIST &= 3 \\
CONST\_NUM &= 4
\end{aligned}
$$

FIGURE 3 – Type des constructeurs dans le langage $\lambda pure$

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`.

6 Parser/reader

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.

C'est aussi avec ce transformateur que l'on transforme les applications en applicaations partielles si il n'y a pas assez d'arguments pour exécuter la fonction.

Ainsi, nous obtenus un AST bien formé et correct, où une constante est réellement une constante et une application complète est aussi réellement une application complète.

7 Interpréteur

La première étape de notre projet était de créer un interpréteur du langage *λpure*, sans instruction de gestion de références.

7.1 Mémoire

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`), comme définies par les auteurs de l'article selon la Figure 4 ci-dessous.

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

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.

7.1.1 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.

7.1.2 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`. Pour la compilation, nous allons réserver la première case pour stocker le nombre d'allocations totales, ainsi, cette case est définie à 1 dans un tas vide.

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.

7.2 Interprétation

Une fois le programme parsé et transformé, nous pouvons interpréter le programme, c'est à dire exécuter une expression dans le contexte du programme. Toutes les fonctions de l'interpréteur s'exécutent avec une pile et un tas. Ainsi, nous créons une pile vierge et un tas vide que nous passons en référence à chaque fonction de l'interpréteur.

Ensuite, il nous suffit d'interpréter les instructions en respectant les règles sémantiques du langage telles que définies par les auteurs, selon la Figure 5 ci-dessous.

$$\textsc{Const-App-Full}\quad \frac{\delta(c) = \lambda\ \overline{y_c}.\ F \qquad \bar{l} = \overline{\rho(y)} \qquad [\overline{y_c} \mapsto \bar{l}] \vdash \langle F, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle c\ \bar{y}, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

$$\textsc{Const-App-Part}\quad \frac{\delta(c) = \lambda\ \overline{y_c}.\ F \qquad \bar{l} = \overline{\rho(y)} \qquad |\ \bar{l}\ | < |\ \overline{y_c}\ | \qquad l' \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle \mathbf{pap}\ c\ \bar{y}, \sigma\rangle \Downarrow \langle l', \sigma[l' \mapsto (\mathbf{pap}\ c\ \bar{l}, 1)]\rangle}$$

$$\textsc{Var-App-Full}\quad \frac{\sigma(\rho(x)) = (\mathbf{pap}\ c\ \bar{l}, \_) \qquad \delta(c) = \lambda\ \overline{y_c}.\ F \qquad l_y = \rho(y) \qquad [\overline{y_c} \mapsto \bar{l}\ l_y] \vdash \langle F, \mathrm{dec}(\rho(x), \mathrm{inc}(\bar{l}, \sigma))\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle x\ y, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

$$\textsc{Var-App-Part}\quad \frac{\sigma(\rho(x)) = (\mathbf{pap}\ c\ \bar{l}, \_) \qquad \delta(c) = \lambda\ \overline{y_c}.\ F \qquad l_y = \rho(y) \qquad |\ \bar{l}\ l_y\ | < |\ \overline{y_c}\ | \qquad l' \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle x\ y, \sigma\rangle \Downarrow \langle l', \mathrm{dec}(\rho(x), \mathrm{inc}(\bar{l}, \sigma))[l' \mapsto (\mathbf{pap}\ c\ \bar{l}\ l_y, 1)]\rangle}$$

$$\textsc{Ctor-App}\quad \frac{\bar{l} = \overline{\rho(y)} \qquad l' \notin \mathrm{dom}(\sigma)}{\rho \vdash \langle \mathbf{ctor}_i\ \bar{y}, \sigma\rangle \Downarrow \langle l', \sigma[l' \mapsto (\mathbf{ctor}_i\ \bar{l}, 1)]\rangle}$$

$$\textsc{Proj}\quad \frac{\sigma(\rho(x)) = (\mathbf{ctor}_j\ \bar{l}, \_) \qquad l' = \bar{l}_i}{\rho \vdash \langle \mathbf{proj}_i\ x, \sigma\rangle \Downarrow \langle l', \sigma\rangle} \qquad \textsc{Return}\quad \frac{\rho(x) = l}{\rho \vdash \langle \mathbf{ret}\ x, \sigma\rangle \Downarrow \langle l, \sigma\rangle}$$

$$\textsc{Let}\quad \frac{\rho \vdash \langle e, \sigma\rangle \Downarrow \langle l, \sigma'\rangle \qquad \rho[x \mapsto l] \vdash \langle F, \sigma'\rangle \Downarrow \langle l', \sigma''\rangle}{\rho \vdash \langle \mathbf{let}\ x = e;\ F, \sigma\rangle \Downarrow \langle l', \sigma''\rangle}$$

$$\textsc{Case}\quad \frac{\sigma(\rho(x)) = (\mathbf{ctor}_i\ \bar{l}, \_) \qquad \rho \vdash \langle F_i, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}{\rho \vdash \langle \mathbf{case}\ x\ \mathbf{of}\ \overline{F}, \sigma\rangle \Downarrow \langle l', \sigma'\rangle}$$

FIGURE 5 – Sémantiques du langage λ_{pure}

Il est à noter que, comme indiqué précédemment, nous ne renvoyons pas le nouveau tas à chaque instruction, comme les auteurs, mais seulement l'emplacement de l'objet dans la mémoire, car nous avons préféré modifier le tas par effet de bord.

Nous n'avons pas non plus implémenté les instructions `inc`, `dec`, `reset`, et `reuse`, et donc la notion de références dans cet interpréteur car ces instructions ne font pas partie du langage λ_{pure}, mais du λ_{RC}. Nous les implémentons donc dans le compilateur, qui compile le langage λ_{RC} étendu.

Ainsi, hormis le tas et les l'apparition des instructions `inc` et `dec`, nous avons strictement respecté les règles sémantiques du langage λ_{pure}.

Ainsi, pour interpréter un `let`, nous interprétons d'abord l'expression associée, qui renvoie un emplacement dans le tas de la valeur interprétée, et nous lions cet emplacement à la variable du `let` en étendant la pile, qui nous passons dans la suite du programme.

Pour un `case`, il suffit de récupérer le type de la valeur dans le tas de dont l'emplacement a été lié dans la pile à la variable donnée, puis interpréter la suite de l'AST stocké dans un vecteur selon le type du constructeur (voir Figure 3). Une fois le type de la variable récupéré, il s'agit donc d'un simple accès à un vecteur.

Pour un `proj`, nous récupérons la valeur à travers la pile puis le tas, puis, si l'index du champ demandé ne dépasse pas la taille du vecteur des champs du constructeur, nous renvoyons l'emplacement stocké.

Pour l'interprétation des applications de constantes, il faut d'abord savoir si il s'agit d'une primitive ou d'une fonction définie par l'utilisateur, puis nous fixons les arguments aux noms des paramètres de la fonctions puis nous exécutons le corps de la fonction. Il n'y a pas besoin de vérifier que le nombre d'arguments soit exact, car le transformateur s'est occupé de transformer celles-ci en définitions d'applications partielles sur des constantes.

Enfin, s'il s'agit d'une application partielle, nous vérifions le nombre d'arguments puis nous exécutons une routine similaire, en interprétant le corps de la fonction demandée avec les arguments fixés si le nombre d'arguments correspond, ou nous renvoyons une nouvelle application partielle étendue du nouvel argument appliqué si il n'y en a pas assez.

Nous avons par ailleurs défini un certain nombre de primitives arithmétiques et booléennes qui nous permettent d'écrire et de tester des programmes dans le langage λ_{pure}.

— Fonctions arithmétiques (`add`, `sub`, `mul`, `div`, `mod`)
— Fonctions booléennes sur booléens (`and`, `or`, `not`)
— Fonctions booléennes sur nombres (`eq`, `sup`, `inf`, `sup_eq`, `inf_eq`)

FIGURE 6 – Liste des primitives implémentées

Nous avons testé cet interpréteur en créant plusieurs tests sur les sémantiques, mais aussi quelques programmes simples, tels que le calcul de fibonacci.

```
1  fibo n = let m1 = 1; let m2 = 2;
2      let a = inf_eq n m2; case a of
3      (let x = sub n m1; let y = sub n m2;
4          let m = fibo x; let n = fibo y;
5          let r = add m n; ret r)
6      (ret m1)
```

FIGURE 7 – Implémentation d'un algorithme naïf de calcul de fibonacci

8 Compilateur

La dernière étape de notre projet était de créer un compilateur du langage λRC. Pour cela, il faut insérer les instructions `inc`, `dec`, `reset` et `reuse` dans l'AST. Ainsi, la première chose à faire était d'étendre l'AST et de créer un transformateur du langage $\lambda pure$ vers le langage λRC.

8.1 AST

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

FIGURE 8 – Grammaire du langage λ_{RC}

8.2 Insertion des instructions de références

8.3 Environnement d'exécution

Notre compilateur génère du code sous le format WAT, le format texte du langage WebAssembly. Ainsi, nous avons besoin du ompilateur `wat2wasm` pour créer du code WASM compilé.

Pour être exécuté, un module WASM a besoin d'être appelé par un script JavaScript. Ainsi, nous avons créer un fichier `runtime.js` qui regroupe toutes les fonctions necéssaires à l'exécution d'un module WASM ainsi que l'interprétation du résultat obtenu. Nous avons aussi ajouté des fonctions de création de constructeurs de façon à créer et ajouter des objets dans la mémoire avant d'appeler une fonction du module WASM.

8.4 Structure de la mémoire en WASM

Pour implémenter le langage en WASM, nous avons besoin de deux structures essentielles : le tas et la pile.

Pour la pile, nous n'implémentons pas de structure particulière, nous utilisons la pile de WASM lors de l'appel d'une fonction.

Afin d'implémenter le tas en WebAssembly, nous utilisons une structure `Memory` qui nous permet de définir une mémoire que nous pouvons manipuler de WebAssembly ainsi que de JavaScript. C'est dans cette mémoire que nous stockons les objets que nous créons L'accès à cette mémoire 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. Par défault, nous mettons la valeur de cette case à l'adresse de la seconde case, soit 4, car la mémoire est alignée sur des entiers naturels de 32 bit ou 4 octets. 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".

8.5 Applications partielles

Pour les applications partielles, nous avons décider de les stocker dans le tas de la même façon que toutes les autres valeurs. Il nous faut cependant trouver une alternative au stockage du nom de la fonction en une chaîne de charactères, car WebAssembly ne supporte pas ce type. Ainsi, nous avons décidé de donner à chaque fonction, primitive ou définie par l'utilisateur, un identifiant sous forme d'un entier, qui nous servira à choisir quelle fonction appeler lors de l'exécution de l'application partielle. Il nous faut aussi définir un nouveau nombre définissant le type des applications partielles, étendant les types déjà définis par à la Figure 3. Bien que nous définissions les applications partielles de la même façon que les constructeurs, les applications partielles n'en sont pas.

Ainsi, nous définissons ces types selon la figure suivante.

$$CONST\_FALSE = 0$$
$$CONST\_TRUE = 1$$
$$CONST\_NIL = 2$$
$$CONST\_LIST = 3$$
$$CONST\_NUM = 4$$
$$CONST\_PAP = 5$$

FIGURE 9 – Type des constructeurs dans le langage λRC en WASM

Dans la mémoire, nos applications partielles sont stockés sous la forme `<type> <nombre de références> <identifiant de la fonction> <nombre d'arguments fixés> <arguments>`. Par exemple, une application partielle est sous la forme `5 1 7 2 12 13 0` où 5 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. 0 est l'emplacement des arguments non fixés que la fonction peut accepter.

8.6 Schéma mémoire

Voici donc le schéma de stockage des objets dans la mémoire.

— False : 0 <#refs>
— True : 1 <#refs>
— Nil : 2 <#refs>
— List : 3 <#refs> <@arg1> <@arg2>
— Num : 4 <#refs> <entier>
— Pap : 5 <#refs> <id fonction> <#arguments fixés> <@arg1> <@arg2> ...

FIGURE 10 – Schéma mémoire

8.7 Compilation des instructions en WAT

Nous avons choisi d'adopter un style impératif dans la compilation des instructions de WebAssembly. En effet, ce style nous permet de regrouper des instructions récurentes en fonctions, ce qui simplifie la lecture du code du compilateur.

8.7.1 Formation des objets dans la mémoire

Pour créer les objets dans la mémoire, nous avons écrit quelques fonctions en WebAssembly. Ces fonctions, `__make_num`, `__make_no_arg`, `__make_list` et `__make_pap`, modifie la mémoire pour y former un objet du type demandé. Puis, ces fonctions modifie l'emplacement 0 de la mémoire, pour y écrire le prochain emplacement libre.

Chaque fonction implémentent dans la mémoire un des quatre stockages différents définis dans le schéma de la mémoire à la figure 10.

La fonction la plus simple est `__make_no_arg`. C'est avec cette fonction que nous créons des objets sans paramètres, soit les booléens `False` et `True` et le mot de fin de liste `Nil`. Pour utiliser cette fonction, nous chargeons d'abord l'identifiant du type demandé, soit `CONST_FALSE` (0), `CONST_TRUE` (1) ou `CONST_NIL` (2), puis `__make_no_arg` met cet identifiant à l'adresse désignée par la case 0, initialise le nombre de références à 1 dans la case d'adresse suivante, et incrémente de 8 la valeur de la case 0 pour l'aligner à la prochaine case vide. Enfin, l'adresse de l'objet créé est renvoyée.

Ainsi, l'objet est créé et la prochaine adresse disponible est définie comme étant celle de la case suivant l'objet.

Les fonctions `__make_num` et `__make_list` ont un fonctionnement similaire, mais prennent en paramètres non pas le l'objet mais les arguments de l'objet à créer, soit l'entier à créer pour `__make_num`, et les adresses des objets composant la tête et la queue de la liste pour `__make_list`. La valeur de la case 0 est incrémentée de 12 et de 16 respectivement.

Enfin, la fonction `__make_pap` prend seulement l'identifiant de la fonction appliquée. Elle écrit ensuite le type de l'objet, soit `CONST_PAP` (5), le nombre de références (1), l'identifiant de la fonction donné en paramètre et le nombre d'arguments fixés, initialisé à 0. La fonction libère ensuite la place nécessaire à l'écriture de l'ensemble des arguments fixés. Elle incrémente donc la valeur de la case 0 de $16 + N * 4$ où N est le nombre d'arguments que la fonction de l'identifiant attend.

Ainsi, pour obtenir ce nombre d'arguments par fonction, il nous faut définir une fonction interne, `$__nb_args` qui est compilée de sorte à associer chaque identifiant de fonction au nombre de paramètres, que nous récoltons au préalable à la compilation, avec l'utilisation des instructions `block` et `br_table`.

L'utilisation de ces deux instructions est la suivante.

```
1  (block $b1
2    (block $b2
3      local.get $x
4      (br_table $b2 $b1)
5    )
6    i32.const 0
7    return
8  )
9  i32.const 1
10 return
```

FIGURE 11 – Example d'utilisation de block et br_table en WebAssembly

Des `blocks` avec un label `$l` peuvent être définis. L'on peut sortir du `block` en enbranchant au label du `block` avec une instruction du type `br $l`.

La `br_table` est une table permettant d'effectuer un enbranchement selon la valeur en haut de la pile. La valeur 0 enbranchera sur la branche d'index 0, la valeur 1 enbranchera sur la branche d'index 1, etc...Il est à noter qu'une valeur supérieure à la taille de la table enbranchera sur la dernière branche de la table.

Ainsi, en associant les deux, et avec la figure 11 comme exemple, si la valeur x (ligne 3) est `0`, la table enbranchera à la fin du block `$b2` pour exécuter la suite du programme (ligne 5) et retourner la valeur `0`. Si la valeur de x (ligne 3) est `1` ou supérieure, la table enbranchera à la fin du block `$b1` pour exécuter la suite du programme (ligne 8) et retourner la valeur `1`.

8.7.2 Compilation des fonctions

Pour compiler une fonction, il faut d'abord récolter le nom de toutes les variables utilisées afin de les déclarer et on y ajoute une variale utile à certains scenarios de compilation, `$__intern_var`. Ensuite, nous changeons le nom interne de la fonction pour ajouter "fun_" devant pour éviter les redéfinitions de fonctions internes. Nous ajoutons également "var_" à toutes les variables définies. Il suffit enfin d'écrire la signature de la fonction, en n'oubliant pas d'exporter au nom original.

Puis, on compile le corps de la fonction, et on ajoute derrière une parenthèse fermante pour fermer la fonction.

Ainsi, la structure de la fonction `fibo` de la figure 7 est la suivante :

```
(func $fun_fibo (export "fibo") (param $n i32) (result i32)
  (local $__intern_var i32)
  (local $var_m1 i32)
  (local $var_m2 i32)
  (local $var_a i32)
  (local $var_r i32)
  (local $var_y i32)
  (local $var_x i32)
  (local $var_m i32)
  ;; corps de la fonction
)
```

FIGURE 12 – Compilation de la signature de la fonction fibo

8.7.3 Let

Le `let` est très simple à compiler, en effet, il faut compiler l'expression en premier pour mettre la valeur de retour dans la variable, et compiler le reste de la fonction. Pour une variable `x` définie par l'utilisateur, le code est donc le suivant

```
;; expression
local.set var_x
;; corps de fonction suivant
```

FIGURE 13 – Compilation d'un let

8.7.4 Case

Avec `block` et `br_table` qui permet de choisi un embranchement selon la valeur en haut de la pile

8.7.5 Proj

On ne peut exécuter un proj que sur un objet de type `CONST_LIST`, car c'est le seul constructeur qui contient des champs qui pointent vers d'autres objets.

Pour compiler un $proj_i$, il suffit de renvoyer le champ désigné par i, soit ajouter $(i+1)*4$ à l'adresse de la variable, car d'après notre schéma de la mémoire, le seul constructeur possédant des champs est la liste, dont les champs commencent deux cases après le début de l'objet. Il ne faut pas oublier que nous commençons l'indexation des champs à 1, c'est à dire que $proj_1$ donne le premier champ, $proj_2$ le deuxième, etc...

Ainsi, pour une variable `x` de type liste définie par l'utilisateur, la compilation de $proj_1$ est la suivante.

```
local.get $var_x
;; calcul de l'offset en ajoutant la case des references et sur
   alignement des entier 32 bits
;; sur liste : 3 4 123 456, proj1 => 123 (offset de 8) et proj2
   => 456 (offset de 12)
i32.const 8 ;; (i + 1) * 4
i32.add     ;; calcul de l'adresse a recuperer
i32.load    ;; chargement du champ (adresse d'un objet)
```

FIGURE 14 – Compilation d'un $proj_1$

8.7.6 Application complète

8.7.7 Application partielle

8.7.8 Inc et dec

La compilation d'un `inc` est très simple. En effet, nous ajoutons `1` aux nombre de références de l'objet. Ainsi, pour une variable `v` définie par l'utilisateur, le code de sortie est le suivant.

```
;; chargement de l'adresse
local.get $var_x
i32.const 8     ;; 8 car #ref est deuxieme, donc 2*4
i32.add

;; chargement des references
local.get $var_x
i32.const 8
i32.add
i32.load

;; calcul des nouvelles references
i32.const 1
i32.sub

;; stockage
i32.store

;; corps de fonction suivant
```

FIGURE 15 – Compilation d'un inc

Le `dec` est un peu plus compliqué, car il ne suffit pas seulement de décrémenter le nombre de références, il faut aussi appliquer un `dec` sur les arguments des `ctor` et `pap` lorsque ceux-ci se retrouvent à 0. Ainsi, nous définissons une fonction `$__dec`, qui décrémente l'objet et les arguments de l'objet si besoin.

La fonction `__dec` décrémente donc les références de l'objet de façon similaire à `inc`, en utilisant `i32.sub` au lieu de `i32.add`, puis si ce nombre de références est nul, teste le

type de l'objet. Si ce type est celui de `CONST_LIST`, on charge le premier champ de l'objet, pour y appliquer la fonction `$__dec` et de même sur le deuxième champ. Si le type est ceui de `CONST_PAP`, on utilise une boucle (`loop $nom_boucle` en WebAssembly) pour charger et chacun des objets du `pap` et y appliquer la fonction `$__dec`.

8.7.9 Reset et reuse

9 Comparaisons

En testant notre interpréteur et notre compilateur sur une implémentation de la fonction de fibonacci (voir Figure 7), l'on s'apperçoit que le code compilé est à beaucoup plus rapide que le code interprété, mais surtout qu'il alloue moins d'espace mémoire pour s'exécuter.

TODO : Tableau de Comparaisons

10 Améliorations possibles

Memory dans le module WAT et exportée, et non l'inverse.

Applications partielles sont remplies puis exécutées. Tester le nombre d'arguments avant de les copier rendrait le tout plus performant.

Enlever la variable `$__intern_var` et remplacer par des fonctions.

11 Conclusion

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).