Sorbonne Université

PSTL
Rapport

Un Langage "Pur" pour Web Assembly

Élève :
Lucas Fumard
Lauryn Pierre
Saïd Mohammad ZUHAIR

Enseignant :
Frédéric Peschanski

19 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 WebAssembly

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. Ceci rend WebAssembly intéressant pour la création d'applications multiplateformes. Les programmes pouvant être exécutées sur des ordinateurs de bureau, des appareils mobiles, et même des serveurs. WebAssembly fonctionne avec les technologies Web existants, telles que 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 domaine du 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 connus. 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. Ces types désignent respectivement les entier sur 32 bit, les entiers sur 64 bits, les nombres flottants sur 32 bits et les nombres flottants sur 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.

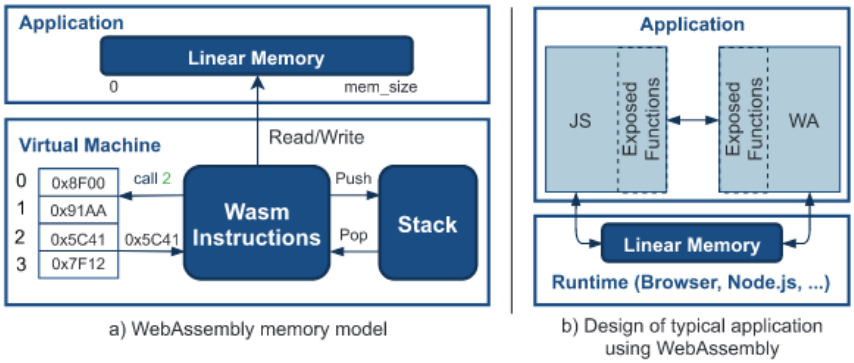

FIGURE 1 – WebAssembly Architecture [6]

3 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. 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 le langage λ_{pure} et notre compilateur, le langage λ_{RC}.

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

Comme dans l'article, nous partons du principe que le typage a déjà été vérifié et que les `let` inutiles ont été enlevés.

4 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.

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

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 expressions (`Expr`) pour ajouter la notion d'entiers à notre interpréteur, afin de tester concrètement notre interpréteur à partir de fichiers de programmes.

Chaque type dans l'AST est un produit de somme en Rust. Nous utilisons principalement les entier 32 bits, les chaînes de caractères, ainsi que les vecteurs (ou tableaux), notés respectivements `i32`, `String` et `Vec` en Rust.

Les énumérateurs feuilles dans notre AST sont les variables `Var` et les constantes `Const`, qui contiennent chacune une chaîne de caractère 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 caractère.

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 somme de produit où l'on pourrait définir chaque type proprement car c'est de cette façon que nous allons stocker les types en mémoire en WASM.

Une fois le contructeur créé, nous pouvons accéder aux champs du contructeur avec l'instrution `proj`. Il est à noter que seules les listes ont des champs, car ce sont les seuls constructeurs 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 d'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 remontant 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, 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 quelle branche du `Case` il faut exécuter, il suffit tout simplement de regarder l'identifiant du type du constructeur puis accéder à l'indice correspondant dans le vecteur des branches du `Case`. 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 table de hachage liant des constantes `Const` à des définitions de fonction `Fn`.

5 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 interprétés commes des constantes.

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, et les applications non complètes en applications partielles

Nous ajoutons par ailleur un préfixe "var_" aux variables définies par l'utilisateur.

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

6 Interpréteur

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

6.1 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.

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

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.

6.1.1 Pile et tas

La pile est une structure de type FILO liant une variable `Var` à une adresse `Loc` que nous étendons à chaque définition de variable. Nous passons la pile étendue à le suite du programme, sans la transmettre à l'appelant. Ainsi, la pile est automatiquement dépilée lorsque l'on quitte le block d'affectation de variable.

Le tas `Heap` est une structure contenant le nombre total d'allocations ainsi qu'une table de hachage d'un entier vers un `Value`, où `Value` est définie comme soit un constructeur, soit un entier. 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.

Ces structure nous permettent donc de lier une variable à sa valeur dans la mémoire. Nous ajouterons les références dans la compilation.

6.2 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 à la figure 1, section 4 de l'article[3].

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 l'apparition des instructions `inc` et `dec`, nous avons strictement respecté les règles sémantiques du langage λ_{pure} défini par les auteurs de l'article.

Ainsi, pour interpréter un `let`, nous interprétons d'abord l'expression associée et nous étendons la pile avec l'adresse de la valeur de l'expression interprétée. Nous passons cette nouvelle pile dans la suite du programme.

Pour interpréter un `case`, nous récupérons le type de la variable par la pile puis le tas. Une fois le type de la variable récupéré, il s'agit d'un simple accès à un vecteur.

Pour interpréter un `proj`, nous récupérons la liste des arguments du constructeur par la pile puis le tas. Il suffit ensuite de renvoyer le bon champ, si tenté qu'il existe.

Pour l'application complète, nous fixons les arguments donnés aux paramètres de la fonctions dans une nouvelle pile et nous interprétons le corps de cette fonction avec cette pile.

Enfin, s'il s'agit d'une application partielle, si l'argument ajoutée permet d'exécuter l'appel, nous l'interprétons comme une application complète. Sinon, nous copions l'ancienne application partielle et ajoutons l'argument à la liste des arguments.

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 5 – 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 dont voici l'implémentation dans le langage.

```
1  fibo n = let m = 1;
2    let a = inf_eq n m; case a of
3    (   let m1 = 1;
4        let x = sub n m1; let f1 = fibo x;
5        let m2 = 2;
6        let y = sub n m2; let f2 = fibo y;
7        let r = add f1 f2; ret r)
8    (ret n)
```

FIGURE 6 – Implémentation d'un algorithme naïf de calcul de fibonacci

7 Compilateur

La dernière étape de notre projet est 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 est d'étendre l'AST et de créer un transformateur du langage $\lambda pure$ vers le langage λRC.

7.1 AST

Nous étendons donc l'AST du langage $\lambda pure$ en suivant l'extension faite par les auteurs de l'article[3], selon la figure 7.

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

FIGURE 7 – Extension de la grammaire du langage λ_{RC}

7.2 Insertion des instructions de références

Cette section correspond à la section 5 de l'article [3]. L'insertion des instructions `inc`, `dec`, `reset` et `reuse` se fera en trois étapes :

1. Insérer `reset` et `reuse`
2. Inférer les paramètres des fonctions
3. Insérer `inc` et `dec`

7.2.1 Insertion de reset et reuse

En utilisant ces deux instructions, on peut réduire la quantité de mémoire allouée et libérée en réutilisant les cellules mémoire lorsque les valeurs ne sont pas partagées. Cela permet d'économiser des ressources et d'optimiser les performances en évitant des opérations coûteuses d'allocation et de libération de mémoire.

Les deux instructions `reset` et `reuse` sont utilisées ensemble. Si x est une valeur partagée, alors y est initialisée avec une référence spéciale, et l'instruction reuse alloue simplement une nouvelle valeur de constructeur `ctor`. Si x n'est pas partagé, alors reset décrémente les compteurs de références des composants de x, et y est initialisée à x. Ensuite, reuse réutilise la cellule mémoire utilisée par x pour stocker la valeur d'un constructeur `ctor`. Pour savoir si x est une variable partagée, on vérifie son compteur de références.

On introduit la fonction δ_{reuse} qui se charge d'insérer les instructions `inc` et `dec` dans une fonction. Pour cela, elle fait appel à la fonction R sur le corps de la fonction. En plus de la fonction R, on utilise deux autres fonctions : D et S. Pour chaque case, R tente d'insérer des instructions `reset`\`reuse` pour la variable correspondant au case. Cela est fait en utilisant D dans chaque bras du case. La fonction D cherche les variables mortes. Elle prend en paramètre la variable z à réutiliser et l'arité n du constructeur correspondant. D recherche le premier emplacement où z où n'est pas utilisée dans le reste du corps de la fonction, puis fait appel à S en lui donnant une nouvelle variable. Dans

notre implémentation, nous gardons un compteur qu'on incrémente à chaque fois qu'on a besoin d'une nouvelle variable. Et on passe à S une variable de la forme $w\_compteur$. Pour s'assurer qu'on utilise toujours une nouvelle variable, on interdit le fait qu'une variable puisse avoir un nom de la forme w_nombre. Si aucune instruction `ctor` correspondante ne peut être trouvée, D ne modifie pas le corps de la fonction. Sinon, on insère un `reuse`. La fonction S réalise les substitutions. S prend en argument une variable, une arité et une partie du corps de la fonction. Si S trouve un constructeur dont l'arité est n. S remplace ce constructeur par `reuse`.

7.2.2 Inférence

Dans cette partie, on veut créer une map β qui associe à chaque fonction une liste qui donne le status des paramètres : "Owned" ou "Borrowed". On va s'appuyer sur la fonction $collect\_O$ qui retourne un ensemble contenant les variables qui doivent être "owned". Un paramètre est "owned" si une de ses projections est "owned" ou s'il est passé en argument à une fonction qui prend un paramètre "owned". Si une variable x est utilisé dans un `reset`, on ne veut pas qu'elle soit "borrowed". En effet, on utilise le compteur de référence pour savoir si une variable est partagé lors de l'évaluation d'un `reset`. Or si x est "borrowed", il peut être une variable partagé et son compteur à un. De plus, une application ne doit pas avoir de paramètre borrowed. Pour résoudre ce problème, on définit une enveloppe $c_O := c$. Dans notre implémentation, le nom de c_O est de la forme $nomdelafonction_$c. Pour s'assurer que que la constante c_O n'est pas présente dans le programme, interdit au nom de fonction d'avoir pour suffixe _c. Chaque `pap` c y est remplacé par `pap` c_O y. On ajoute à β une entrée tel que $\beta(c_O) = \overline{O}$

La fonction δ_β va déduire la valeur de β pour une fonction c. Pour réaliser cette étape, on commence par une approximation $\beta(c) = \overline{B}$, puis en calculant $S = collect\_O(b)$. Ensuite, nous mettons à jour $\beta(c)_i :=$ O si $y_i \in S$, et nous répétons ce processus jusqu'à atteindre un point fixe où aucune autre mise à jour n'est effectuée sur $\beta(c)$. Pour inférer l'ensemble du programme, on appelle δ_β sur chaque fonction dans l'ordre de déclaration des fonctions. Si une fonction c utilise une autre fonction d, pour avoir avoir un résultat optimal, il faut que d soit déclarée avant c. Sinon, on considère que les paramètres de la fonction d sont "owned".

7.2.3 Insertion de inc et dec

Dans cette partie, on insère les instructions `inc` et `dec`. Lorsqu'une variable est borrowed, elle ne possède pas la responsabilité de sa gestion mémoire. D'un autre côté, une variable "owned" est responsable de sa gestion mémoire. Les compteurs des variables doivent être incrémentées avant d'être utilisées dans un contexte "owned". Si elle est utilisée dans un constructeur ou dans une application. On doit décrémenter leur compteur de référence das variables "owned" après leur dernière utilisation. On suppose que notre map β de la section précédente est bien définie. Pour chaque fonction c, on introduit une nouvelle map β_l qui associe à chaque variable dans c, son statut ('B' ou 'O'). β_l est initialisé avec les valeurs de $\beta(c)$. Par défaut, une variable sera considérée comme owned.

Nous définisson plusieurs fonctions :

— O_x^+ : la variable x n'est pas incrémentée si elle est "owned" et morte.

— C : appelée sur le corps de la fonction pour effectuer l'ajout d'instruction `inc` et `dec`.

— C_{app} : se charge des applications en effectuant des appels récursifs sur les paramètres et leur statut.
— O^- : sert à décrémenter plusieurs variables, utilisée au début d'un case et d'une fonction.
— O_x^- : décrémente la variable x si elle est "owned" et morte.

7.3 Environnement d'exécution

Notre compilateur génère du code sous le format WAT, le format texte du langage WebAssembly. Ainsi, nous avons besoin du compilateur `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 créé un fichier `runtime.js` qui regroupe toutes les fonctions nécessaires 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 à ajouter des objets dans la mémoire avant d'appeler une fonction du module WASM.

7.4 Structure de la mémoire en WASM

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

La pile n'a pas besoin que nous implémentons une de structure particulière, car nous utilisons la pile de WASM lors de l'appel d'une fonction (le cadre de la 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. La case mémoire 0 est réservée pour indiquer le prochain espace mémoire libre. Par défaut, 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 défini dans la figure 3 suit le format suivant : `<type> <nb_ref> <args>`.

Les argument des types prennent une case mémoire chacun, c'est à dire la place d'un entier signé 32 bits. Ainsi, l'alignement de la mémoire sera toujours de 32 bits, soit 4 octets. Les adresses de nos variables seront toujours multiples de 4.

Nous ordonnons donc les objets dans la mémoire selon la figure 8 suivante.

— False : 0 <#refs>
— True : 1 <#refs>
— Nil : 2 <#refs>
— List : 3 <#refs> <@arg1> <@arg2>
— Num : 4 <#refs> <entier>

FIGURE 8 – Schéma mémoire des constructeurs

Ainsi, un constructeur `FALSE` référencé deux fois est représenté sous la forme `"0 2"` dans la mémoire et un entier 10 référencé 3 fois sous la forme `"4 3 10"`.

Nous avons aussi décidé de stocker les applications partielles 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 caractères, 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. Nous ajoutons un nouveau type pour les applications partielles à notre liste des types de la figure 3.

$$CONST\_PAP = 5$$

FIGURE 9 – Type des PAP en WASM

Ainsi, dans la mémoire, nos applications partielles sont stockés sous la forme de la figure 10 suivante.

— Pap : 5 <#refs> <id fonction> <#arguments fixés> <@arg1> <@arg2> ...

FIGURE 10 – Schéma mémoire pour les applications partielles

Par exemple, une application partielle est sous la forme `5 1 7 2 12 14 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 `14` sont les arguments fixés et `0` est l'emplacement des arguments non fixés que la fonction peut accepter.

7.5 Compilation des instructions en WAT

Il nous faut compiler les instructions du langage *λpure* ainsi que les nouvelles instructions `inc`, `dec`, `reset` et `reuse`. Pour ces nouvelles instructions, nous suivons les sémantiques définies par les auteurs, définies à la figure 2 dans la section 5.1 de l'article[3].

WebAssembly est un langage possédant des fonctions où il est possible d'écrire dans un style fonctionnel, avec les instructions imbriquées les unes dans les autres, entourées de parenthèses, ou impératif, une instruction par ligne. Nous avons choisi d'adopter un style impératif dans la compilation des instructions de WebAssembly car ce style nous permet de regrouper des instructions récurentes en fonctions, ce qui simplifie la lecture du code du compilateur. Cependant, nous utilisons quelques fois dans la compilation et dans ce rapport le style fonctionnel. Ces deux styles sont strictement égaux et valides en WebAssembly.

7.5.1 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`, modifient la mémoire pour y former un objet du type demandé. Puis, ces fonctions modifient l'emplacement 0 de la mémoire, pour y écrire le prochain emplacement libre, aligné sur 4 octets.

Chaque fonction implémente dans la mémoire un des quatre stockages différents définis dans le schéma de la mémoire à la figure 8.

Nous avons aussi défini la fonction inter `$__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 – Exemple d'utilisation de block et br_table en WebAssembly

Des `block`s 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 embranchement selon la valeur en haut de la pile. La valeur 0 embranchera sur la branche d'index 0, la valeur 1 embranchera sur la branche d'index 1, etc. Il est à noter qu'une valeur supérieure à la taille de la table embranchera 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 embranchera 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`.

Ces deux instructions de WebAssembly seront aussi utiles pour la compilation de notre instruction `case`.

7.5.2 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 variable 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 avons également ajouté "var_" à toutes les variables définies grâce au transformateur. Cela nous permet de définir des variables internes telles que `__intern_var` qui ne peuvent pas être modifiées par l'utilisateur.

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 6 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

7.5.3 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

7.5.4 Case

Pour compiler les `case`, nous avons besoin d'une structure nous permettant d'exécuter une branche du code suivant la valeur du type de la variable donnée. Ainsi, nous utilisons la combinaison de `block` et `br_table` aussi utilisée pour l'exécution d'une application partielle, comme montré sur la figure 11.

Le code d'une instruction `case` à deux embranchements sur la variable `x` définie par l'utilisateur se compile ainsi de la manière suivante

```
(block $__case0
  (block $__case1
    local.get $var_x
    i32.load
    (br_table $_case1 $_case0)
  )
  ;; suite si type False
)
;; suite si type True
```

FIGURE 14 – Compilation d'un case

7.5.5 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 donc 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.

```
1  local.get $var_x
2  ;; calcul de l'offset en ajoutant la case des references et sur
       alignement des entier 32 bits
3  i32.const 8 ;; (i + 1) * 4
4  i32.add     ;; calcul de l'adresse a recuperer
5  i32.load    ;; chargement du champ (adresse d'un objet)
```

FIGURE 15 – Compilation d'un $proj_1$

7.5.6 Application complète

Pour l'application complète d'une fonction définie par l'utilisateur, il faut charger tous les arguments donnés puis appeler l'équivalent de la fonction en compilé. Ainsi, l'application des variables `x` et `y` définiées par l'utilisateur sur la fonction `f` définie par l'utilisateur se compile selon la figure 16.

```
1  local.get $var_x
2  local.get $var_y
3  call $fun_f
```

FIGURE 16 – Compilation d'une application complète

Si la fonction est une primitive, nous appliquons les arguments sur la définition de la primitive en WebAssembly. Voici par exemple dans la figure 17 la compilation de la primitive `add` appliquée aux variables `x` et `y` définies par l'utilisateur.

```
1  ;; decallage de deux entiers, la place de la valeur du nombre
2  (i32.load (i32.add (local.get $var_x) (i32.const 8)))
3  (i32.load (i32.add (local.get $var_y) (i32.const 8)))
4  i32.add
5  call $__make_num
6
7  (call $__dec (local.get $var_x))
8  (call $__dec (local.get $var_y))
```

FIGURE 17 – Compilation d'une application sur une primitive

Il ne faut pas oublier de décrémenter les références des objet, car leur des `inc` ont été ajoutés, comme spécifié dans la section **??** lors de l'application sur `add` dans l'expression `add x y`.

7.5.7 Application partielle

L'application d'une variable sur une application partielle est délicat. En effet, nous avons choisi d'implémenter les applications partielles en stockant toutes les variables appliquées dans la mémoire. Cependant, cela signifie que nous devons copier l'application partielle avant d'y fixer la nouvelle variable et incrémenter le nombre d'arguments. Nous appelons donc la fonction `__copy_pap` qui crée une nouvelle application partielle du même type et copie les arguments un à un. Il ne faut pas oublier de decrémenter les références de l'application partielle copiée.

Puis, s'il y a assez de variables fixées d'après `__nb_args`, nous appelons la fonction `__exec_pap`, qui charge les variables une à une sur la pile d'exécution de WebAssembly avec une `loop` puis nous appelons la fonction ou nous compilons la primitive selon l'identifiant de la fonction de l'application partielle. Enfin, nous déréférençons l'application partielle.

De la même manière qu'avec la compilation de `case`, nous savons quelle branche appeler grâce à la combinaison des instructions `block` et `br_table`. Une fois le code compilé, cette fonction est assez longue, car elle contient toutes les primitives définies ainsi que tous les appels des fonctions définies par l'utilisateur.

7.5.8 Inc et dec

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

```
;; chargement de l'adresse pour le stockage
local.get $var_x

;; chargement des references
(i32.add (local.get $var_x) (i32.const 4))
i32.load

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

;; stockage
call $__set_ref

;; corps de fonction suivant
```

FIGURE 18 – Compilation d'un inc

Avec la fonction interne `__set_ref` permettant de fixer le nombre de références.

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`.

7.5.9 Reset et reuse

La fonction `__reset` décrémente le nombre de références de la variable donnée et renvoie `0` si la variable est effectivement détruite ou l'adresse de la variable si elle est encore en vie. Voici le code WebAssembly de cette fonction.

```
(func $__reset (param $var_var i32) (result i32)
  (call $__dec (local.get $var_var))
  (i32.add (local.get $var_var) (i32.const 4))
  i32.load
  i32.eqz
  if
    i32.const 0
    return
  end
  local.get $var_var
)
```

FIGURE 19 – La fonction reset

Il suffit ensuite de charger la variable concernée et d'appeler cette fonction `__reset`.

L'instruction `reset` est toujours suivie d'une instruction `reuse`. Celle-ci teste si l'adresse de la variable donnée, soit le résultat de l'instruction `reset`, nul puis s'il y a assez de place dans l'ancienne variable. Si le résultat de ces tests est `1`, on peut copier les arguments donnés après `reuse` dans l'emplacement mémoire de l'ancienne variable.

Sinon, on doit créer un nouvel objet selon le `ctor` donné.

8 Résultats

Noous avons testé notre interpréteur et notre compilateur sur la fonction de fibonacci naïf telle que définie sur la figure 6.

n	fibo(n)	Temps exécution interprété	Temps exécution compilé	Total allocations mémoire (interprété et compilé)	Objets en vie compilé
5	5	0 ms	0.015 ms	66	1
10	55	10 ms	0.095 ms	795	1
15	610	153 ms	0.562 ms	8 877	1
20	6 765	2.18 s	3.77 ms	98 508	1
25	75 025	30.2 s	34.17 ms	1 092 531	1
30	832 040	6.5 mins	366.94 ms	12 116 415	1

TABLE 1 – Résultats de l'exécution de fibo sur l'interprète et en compilé

En testant notre interpréteur et notre compilateur sur une implémentation de la fonction de fibonacci (voir Figure 6), nous obtenons les bons résultats de ficonacci, avec un temps d'exécution et un nombre total d'allocations dans la mémoire qui augmentent ded façon exponentielle.

Nous remarquons que les instructions `reset`, `reuse` n'ont pas été insérées dans ce code, car il y a autant d'allocations pour le code interprété que pour le code compilé. Nous remarquons également que nous obtenons à chaque fois un seul objet en vie à la fin de l'exécution du code compilé, c'est à dire un seul objet dont le nombre de références n'est pas nul.

Ainsi, les instructions `inc` mais surtout `dec` ont bien été insérées avec succès dans le code pour modifier le nombre de références des objets au fur et à mesure de l'exécution.

9 Améliorations possibles

Comme précisé à la fin de la section 3, nous avons implémenté un compilateur en partant du principe que le typage est bon et que les `let` inutiles ont été enlevés. Le langage implémenté n'est donc pas typé statiquement.

De plus, nous avons implémenté les applications partielles de la façon la plus simple à laquelle nous avons pensé. Cette implémentation n'est pas optimale car nous créons une nouvelle application partielle avant de l'exécuter et de la détruire.

Nous aurions pu utiliser un unique objet global pour des constructeurs sans arguments tels que $False$, $True$ ou Nil. Nous avons en effet choisi de créer des objets à chaque fois pour avoir plus d'éléments sur lesquels agir. Dans une version plus complète du compilateur, il serait plus intelligent de rendre ces constructeurs sans arguments uniques. L'on pourrait par ailleur introduire une meilleure gestion de la mémoire semblable à un `malloc` en C. De plus, on peut utiliser un i32 pour représenter les types et le nombre de références, avec les instructions `i32.store16_u` et `i32.load16_u`, qui stockent et chargent des entiers sur 16 bits non signés.

Quand on infére les status des paramètres d'une fonction, nous parcourons les fonctions dans l'ordre de leur déclarations. Si jamais une fonction c utilise une autre fonction d avant qu'elle soit déclaré on considère ses paramètres comme étant "owned". On peut changer notre implémentation, pour inférer d au lieu de lui supposer que ses paramètres sont "owned". Bien sûr nous devrions mettre en place un mécanisme pour les fonctions mutuellement récursives.

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).