PSTL : OCaml sur la plate-forme Nand2Tetris

Pablito Bello Loïc Sylvestre

M1 Informatique, spécialité STL, Sorbonne Université

Encadrement : Emmanuel Chailloux

Mai 2020

Table des matières

1 Introduction

Ce PSTL vise à exécuter des programmes OCaml [20] sur la plate-forme `Nand2Tetris`, un environnement pédagogique mis en œuvre par Noam Nisan et Shimon Schocken dans le cadre d'un cours d'introduction à l'architecture des ordinateurs [24].

La plate-forme `Nand2Tetris` est conçue *from scratch* : à partir de la description comportementale d'une porte `Nand`, les auteurs élaborent des circuits de façon ludique et parviennent ainsi, par composition et réutilisation, à la modélisation d'un processeur et d'une mémoire. Ce processeur est programmable en langage machine, ce qui permet le développement d'une infrastructure logicielle complète : l'introduction d'un langage d'assemblage, puis d'un langage de haut niveau compilé vers du bytecode. Le bytecode étant soit interprété par une machine virtuelle implantée en Java, soit traduit en assembleur, des programmes non-triviaux peuvent être exécutés sur la plate-forme – pensez à des jeux comme `Tetris` par exemple.

Le compilateur OCaml engendre du code optimisant pour différentes architectures. Nous excluons toutefois la possibilité d'étendre son générateur de code, pour cibler spécifiquement la plate-forme `Nand2Tetris`. En effet, cette solution ne serait pas pérenne, des changements profonds pouvant intervenir dans le compilateur.

Le compilateur OCaml produit également du bytecode OCaml. La spécification de la machine virtuelle OCaml – Zinc Abstract Machine, ou ZAM [18] – a peu évolué depuis une vingtaine d'années. En cela, l'implantation de la ZAM dans différents langages constitue une solution robuste pour exécuter des programmes OCaml sur de nouvelles plate-formes, songez à `OCapic` [34] et `OmicroB` [33] pour la programmation de micro-contrôleurs en OCaml. Une dernière approche consiste à compiler le bytecode OCaml vers un autre langage. C'est la solution proposée par le compilateur `Js_of_ocaml` pour exécuter des programmes OCaml en Javascript [35].

Nous optons pour l'approche *machine virtuelle*. Nous cherchons à implanter, dans un langage exécutable sur cette plate-forme, un interprète de la ZAM accompagné d'une bibliothèque d'exécution incluant un récupérateur automatique de mémoire (GC).

Les enjeux de notre travail sont donc en premier lieu la prise en main de la plate-forme `Nand2Tetris` et le choix du langage d'implantation pour notre implémentation de la ZAM. Par ailleurs, il sera question d'adapter la spécification de la ZAM aux contraintes liées aux spécificités matérielles de la plate-forme. Le choix de la représentation des valeurs OCaml au *runtime*, à ce titre, est crucial. En effet, la plate-forme propose des entiers 16 bits et ne dispose pas des primitives de décalage nécessaires à la gestion d'un bit de marque pour distinguer valeurs immédiates et pointeurs.

Au vu des difficultés rencontrées, nous proposons une solution originale pour permettre une implantation réaliste de la ZAM sur cette plate-forme. Il s'agit du développement d'un compilateur optimisant pour le langage `Mini-ML` [9], ciblant la plate-forme `Nand2Tetris`, puis l'implantation de la ZAM en `Mini-ML`. Cette approche est bien sûr intéressante d'un point de vue pédagogique. En outres, les programmes `Mini-ML` sont faciles à écrire et à modifier, tandis que le code engendré par notre compilateur est plus sûr et en général plus efficace qu'un programme assembleur équivalent écrit à la main.

Dans la section 2 nous présentons la chaîne de Compilation `Nand2Tetris`. Dans la section 3 nous présentons de quelle manière nous avons connecté la chaîne de compilation OCaml à la plateforme `Nand2Tetris` par le biais de `Mini-ML`. La section 4 présente la compilation de `Mini-ML` vers la plateforme Nand2Tetris. La section 5 présente notre implémentation de la ZAM en `Mini-ML`. La section 6 présente d'autres travaux réalisés dans le cadre de ce projet, et la section 7 des travaux futurs. Enfin, la section 8 conclut.

Notez que l'ensemble de nos réalisations est disponible à l'adresse suivante :

https://github.com/saya-nel/OCaml2Tetris

2 La plate-forme Nand2Tetris

Cette section donne une vue d'ensemble de la chaîne de compilation Nand2Tetris.

2.1 L'architecture Hack

En premier lieu, l'*apprenant* est amené à modéliser un circuit à l'aide d'un HDL (*Hardware description Language*). Le Listing 1 donne l'exemple d'une porte Not obtenue par réutilisation d'une porte Nand à deux entrées a et b et une sortie out. Sur ce principe, l'*apprenant* modélise toutes sortes de portes logiques, des multiplexeurs, des additionneurs, et finalement une unité arithmétique et logique (ALU) manipulant des entiers 16 bits. La démarche se poursuit par la modélisation d'un banc de registres et d'une mémoire. L'ensemble est programmable par du code machine et porte le nom d'architecture Hack.

```
1 CHIP Not {
2     IN in;
3     OUT out;
4     PARTS:
5     Nand(a=in, b=in, out=out);
6 }
```

Listing 1 – Modèle de porte logique en HDL

2.2 Hack : un langage d'assemblage

Programmation ! Le mot est lâché. On peut programmer l'architecture Hack en langage machine, ou dans un langage d'assemblage qui expose le jeu d'instructions de l'architecture Hack indépendamment de sa réalisation matérielle. L'apparition de symboles et de mnémoniques en assembleur facilite l'écriture des programmes et sa lecture par un humain. Le Listing 2 donne le jeu d'instructions de l'assembleur Hack. L'assembleur Hack fournit seulement deux types d'instructions. L'instruction `@`x, où x est un entier ou un symbole traduit statiquement en un entier, écrit x dans le registre spécial **A**. L'instruction *d*=*c*;*j* calcule *c*, écrit le résultat dans *d* et réalise un branchement *j* conditionné par *c*.

```
1  ins ::= A-ins | C-ins
2
3  A-ins ::= @n | @x
4
5  C-ins ::= dst=comp;jmp | comp;jmp | dst=comp
6
7  dst ::= null | M | D | MD | A | AM | AD | AMD
8
9  comp ::= 0 | 1 | A | D | comp+comp | comp-comp
10          | -comp | !comp | comp&comp | comp|comp
11
12 jmp ::= null | JGT | JEQ | JGE | JLT | JNE | JLE | JMP
```

Listing 2 – jeu d'instructions de l'assembleur Hack

2.3 `Jack` : vers un langage de haut niveau

Avec l'introduction de l'assembleur `Hack`, la plate-forme `Nand2Tetris` devient un terrain d'expérimentation autour du logiciel. Les auteurs choisissent alors d'implanter, sur leur plate-forme, un langage de plus haut niveau appelé `Jack`. Ce langage est très reconnaissable par sa syntaxe concrète fortement inspirée de Java.

Le Listing 3 donne la syntaxe abstraite de `Jack`. Un programme `Jack` est un ensemble de définitions de classes – une classe par fichier. Le point d'entrée d'un programme `Jack` est la fonction `void Main.main()`. Une classe peut définir des variables de classes introduites par le mot clé **static**, ou des attributs privées introduits par le mot clé **field**. Une classe peut construire des objets à l'aide de fonctions spéciales introduites par le mot-clé **constructor**. Notez qu'une même classe peut définir plusieurs constructeurs. Le mot clé **this** donne l'adresse de l'instance courante. Enfin, la classe peut fournir des méthodes, des fonctions et des procédures, les procédures étant des fonctions dont le type de retour est `void`). Le corps d'une fonction, d'une procédure ou d'une méthode est formé de déclarations de variables locales (**var**), puis d'une suite d'instructions. Les instructions sont soit une assignation, soit une alternative, soit une boucle `while`, soit un appel de procédure (**`do f(x)`**), soit un `return`. Les expressions sont soit l'invocation de fonctions ou de méthodes, soit une opération primitive, soit un constante. Nous présenterons dans la section suivante les limitations de `Jack`.

```
1  π ::= dc_1 ... dc_n
2
3  dc ::= class C { a_1 ... a_n ctr df_1 ... df_n }
4
5  a ::= field τ x_1, ..., x_n; | static τ_1 x_1,...,τ_n x_n;
6
7  ctr ::= constructor τ C.x(τ_1 x_1,...,τ_n x_n){ d_1...d_n s_1...s_n }
8
9  df ::= function τ C.x(τ_1 x_1,...,τ_n x_n){ d_1...d_n s_1...s_n }
10      | method τ C.x(τ_1 x_1,...,τ_n x_n){ d_1...d_n s_1...s_n }
11
12 τ ::= int | boolean | char | void | C
13
14 d ::= var τ_1 x_1,...,τ_n x_n;
15
16 s ::= let x = e;
17     | let x[e_1] = e_2;
18     | return e;
19     | return;
20     | if (e){ s_1...s_n } else { s_1...s_n }
21     | while (e) { s_1...s_n }
22     | do C.x(e_1,...,e_n);
23     | x_1.x_2(e_1,...,e_n);
24
25 e ::= e op e | op e | ( e ) | x[e] | C.x(e_1,...,e_n)
26     | this | null | true | false | n | str
27
28 op ::= + | - | ~ | * | / | && | ||
```

2.4 `Stack!` : un langage de machine à pile

Le langage `Jack` n'est pas compilé directement en assembleur mais dans un langage intermédiaire pouvant être soit exécutée par une machine à pile, soit traduit vers l'assembleur `Hack`. Les auteurs de la plate-forme appelle ce langage « langage de la VM ». Pour autant, nous allons largement l'utiliser par la suite, et il convient donc de lui donner un nom « sonnant » : `Stack!` (prononcé simplement « *stack* »).

```
1  π ::= m_1 ... m_n
2
3  m ::= df_1 ... df_n
4
5  df ::= function M.x n ; ins_1 ; ... ; ins_n
6
7  ins ::= pop seg n | push seg n | label x | goto x | if-goto x
8         | call M.x n | return | p
9
10 p ::= add | sub | neg | and | or | not | eq | lt | gt
11
12 seg ::= constant | argument | local | static | this | that | pointer | temp
```

Listing 4 – Grammaire de `Stack!`

Le Listing 4 donne la syntaxe abstraite de `Stack!`. Un programme π est un ensemble d'unités de compilation m_i. Chaque unité de compilation est un fichier séparé `M.vm` définissant des fonctions de noms `M.`x. Une fonction `M.`x réserve n emplacement sur la pile pour stocker ces variables locales, puis exécute une série d'instructions. L'instruction `[call` $A.f$ `n]` sauvegarde le contexte courant dans la pile, puis appelle la fonction $A.f$ du fichier A en lui passant en arguments n entiers dépilés. L'instruction `[return]` restaure le contexte de la fonction appelée. Le résultat de la fonction est la valeur en sommet de pile. L'instruction `[label` x`]` pose une étiquette statique x. L'instruction `[goto` x`]` saute inconditionnellement à l'étiquette x. L'instruction `[if-goto` x`]` dépile un entier n puis saute à l'étiquette x si n n'est pas 0. Les instructions primitives, telles que [add], dépilent leurs arguments, réalisent un calcul et empile le résultat. L'instruction `[push constant` n`]` empile l'entier positif sur la pile. L'instruction `[pop local n]` dépile un entier et le stocke dans l'emplacement réservé à la n-ième variable locale du bloc d'activation courant. L'instruction `[push local n]` empile l'entier stocké dans l'emplacement réservé à la n-ième variable locale. Par analogie, on peut écrire et lire dans d'autre segments de la mémoire. Le segment `static` est accessible par toutes les fonctions d'un même fichier. Les segments this et `that` forment un tas en mémoire dans lequel le programmeur peut allouer des blocs. Enfin, le segment `temp` comporte 8 emplacements de variables temporaires pouvant être écrasées à tout moment par les fonctions appelées.

2.5 La chaîne de compilation Nand2Tetris

Cette section a présenté les différents langages de programmation coexistant sur la plate-forme `Nand2tetris`. Un ensemble d'outils permettent de traduire un programme `Jack` de haut niveau, vers du code machine exécutable sur l'architecture `Hack`. La figure 1 schématise la chaîne de compilation mise en œuvre par les auteurs. Le langage `Jack` (`J`) est traduit vers `Stack!` (`S!`) par un compilateur écrit en `Java`. Le langage `Stack!` est soit exécuté par une machine abstraite écrite en `Java`, soit traduit vers l'assembleur `Hack` (`H`). Enfin l'assembleur `Hack` est traduit en code machine (`X`), en vue d' être executé sur l'architecture `Hack` simulée par un programme Java.

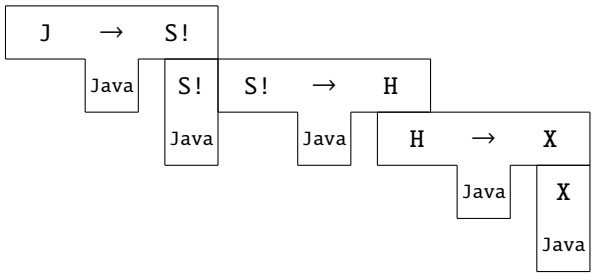

Figure 1 – La chaîne de compilation `Nand2Tetris`

3 OCaml sur la plate-forme `Nand2Tetris` : notre choix d'implantation

Il regarda au loin, au-dessus de l'Océan ; il lui sembla distinguer la Tour Eiffel, mais c'était une erreur.
Raymond Queneau

D'un point de vue pédagogique, écrire un programme en assembleur est un excellent exercice. Pour autant le code assembleur engendré par un compilateur est en général plus performant et contient moins de bugs. Le recours à un compilateur offre donc d'avantage de sûreté à l'exécution et permet au programmeur d'exprimer sa pensée algorithmique dans un langage de programmation de plus haut niveau.

Il est ainsi apparu rapidement que l'implantation de la machinerie virtuelle OCaml en assembleur `Hack` aurait été un choix peu pertinent, car difficilement maintenable. Pour s'en convaincre, le Listing 5 donne un programme simple, écrit en assembleur `Hack` : sauter à l'adresse 100 si le contenu à l'adresse 3 de la mémoire est égale à 5. Il vient donc naturellement l'idée d'écrire notre implémentation de la ZAM en `Jack`, qui est le langage de plus haut niveau dans la chaîne de production `Nand2Tetris`.

Listing 5 – un programme assembleur `Hack`

```
1    @3
2    D=M
3    @5
4    D=D-A
5    @100
6    D;JEQ
```

3.1 Le compilateur Jack et ses limitations

Le projet `Nand2Tetris` offre une approche ludique pour construire par soi même un système d'information, culminant par l'implantation de « son propre langage de programmation » : `Jack`. Le compilateur `Jack` présente cependant certaines faiblesses, d'un point de vue technique, avec en tout premiers lieu l'absence de typage. En fait, les annotations de types ne servent qu'à la résolution statique des appels de méthodes : Si une variable `x` est de type `T`, alors l'expression `x.m()` appelle sur `x` la méthode `T.m()`.

Ainsi le Listing 6 présente un programme `Jack` bien formé. On déclare une variable `x1` de type statique `choux`. On assigne à `x1` un tableau ayant pour taille la chaîne de caractères `"carotte"`. On assigne alors la valeur `x1` à la case d'indice `x1` du tableau `x1`. Puis on affiche la valeur de la chaîne de caractères `x1`. Ce programme compile, et s'exécute sans erreur. Aucun affichage ne se produit. Cet exemple intentionnellement provocateur souligne le fait que `Jack` n'est pas sûr, contrairement à ce que laisse croire à premier abord sa syntaxe concrète.

Listing 6 – un programme `Jack`

```
1  class Main {
2    function void main() {
3      var choux x1;
4      let x1 = Array.new("carotte");
5      let x1[x1] = x1;
6      do Output.printString(x1);
7      return;
8    }
9  }
```

3.2 `Jack` vu comme un langage de bas niveau

Au même titre que l'écriture de programme en assembleur est un exercice formateur, il est clair que l'apprentissage de nouveaux langages de programmation – quelque soit ces caractéristiques et éventuellement faiblesses – est une activité importante du programmeur. D'ailleurs, à y regarder de plus près, la sémantique de `Jack` est bien définie : `Jack` manipule un unique type de donnée : les entiers. Des primitives permettent d'allouer des blocs, et retourne un entier, qui est l'adresse de début du bloc dans la tas. Ainsi, dans le programme donné au Listing 6, l'allocation de la chaîne de caractères ⟦`"carotte"`⟧ retourne l'entier 2050, première adresse libre dans le tas au commencement du programme. La commande ⟦`let x1 = Array.new("carotte");`⟧ alloue donc en mémoire un tableau de taille 2050. La fonction d'allocation retourne une adresse, 2058, qui est stocké dans la variable x1. La commande ⟦`let x1[x1] = x1;`⟧ écrit l'entier 2058 en mémoire à l'adresse ⟦`x1+x1`⟧ comme cela aurait été le cas en langage `C`. Enfin, la commande ⟦`do Output.printString(x1);`⟧ imprime en tant que caractères ASCII les entiers consécutifs en mémoire à partir de l'adresse 2058, tant qu'un caractère nul n'a pas été rencontré. Comme le tas est initialisé par des entiers à 0, l'adresse 2058 à ce point du programme contient 0 à ce point du programme, d'où l'absence d'affichage.

3.3 Implémentation d'un interprète en Jack : un choix décisif

L'interprète d'une machine virtuelle prend en général la forme d'une boucle dont le corps réalise le décodage de l'instruction courante, puis exécute un calcul associé. Comme la Machine Virtuelle OCaml comporte 146 instructions, cela correspondrait typiquement à un grand `switch`. Or, la commande `switch` n'existe pas en `Jack`. Il faut donc envisager de le simuler à la main. La difficulté est qu'il n'y a pas non plus d'instruction `goto` en `Jack`. Les seules structures de contrôles disponibles sont les alternatives et les boucles. On se retrouve donc amené à devoir imbriquer 146 niveaux d'alternatives, ce qui constitue un code extrêmement lourd et inefficace. Une autre approche consisterai à utiliser les objets de `Jack` : représenter le bytecode d'entrée par un tableau d'objets `code`. Chaque objet à l'indice i de tableau serait la i-ème instruction du programme. On définirait donc une classe pour chaque instruction, qui toutes implémenteraient une méthode `run`. Cela permet de réduire la phase de décodage à un simple appel de méthode. La figure 2 met en oeuvre cette solution sur un exemple simple. Malheureusement, cette solution ne fonctionne pas, car il n'y a pas de sous-typage en Jack. L'exécution du programme produit l'affichage `"InstrInstrInstrInstrInstrInstr"`, et non `"PushPopPushPushPopPop"` comme cela aurait été attendu.

```
class Instr {
  ...
  constructor Instr new(){
    return this;
  }
  method void run (){
    do Output.printString("Instr");
    return;
  }
}
```

```
class Push {
  ...
  constructor Push new(){
    return this;
  }
  method void run (){
    do Output.printString("Push");
    return;
  }
}
```

```
class Pop {
  ...
  constructor Pop new(){
    return this;
  }
  method void run (){
    do Output.printString("Pop");
    return;
  }
}
```

```
class Main {
   function void main() {
    var int pc;
    var Array code;
    var Instr current;

    // initialisation
    let pc = 0;
    let code = Memory.alloc (6);
    let code[0] = Push.new ();
    let code[1] = Pop.new ();
    let code[2] = Push.new ();
    let code[3] = Push.new ();
    let code[4] = Pop.new ();
    let code[5] = Pop.new ();

    // exécution du bytecode
    while (pc < 6){
      let current = code[pc];
      do current.run ();
      let pc = pc + 1;
    }

    return;
  }
}
```

Figure 2 – Un `switch` simulé avec des objets

Il reste enfin une dernière solution, qui est celle que nous avons définitivement adoptée. Comme chaque instruction de la machine virtuelle OCaml est un entier entre 0 et 145, nous recourons à des alternatives imbriquées de façon

dichotomique, à la manière d'un arbre binaire de recherche, afin d'améliorer sensiblement l'efficacité du décodage des instructions. Remarquez que l'écriture d'un tel programme est particulièrement inintéressante et source de bugs. Nous avons donc sérieusement envisagé l'utilisation de macros *à la* `cpp`, voire de la génération de code `Jack`, idée qui s'est avérée déterminante pour la suite de notre travail.

3.4 Génération de code `Jack`

`Jack` ne dispose pas des constructions suffisantes pour programmer à la main un interprète de machine virtuelle qui soit à la fois efficace et lisible. Nous avons donc envisagé de développer notre propre langage de programmation compilé vers `Jack`, et offrant des constructions de plus haut niveau. Nous avons opté pour un langage proche du langage `ILP2`, support du cours `Développement d'un Langage de Programmation` au premier semestre de M1 STL à Sorbonne Université. Il s'agit d'un langage d'expression avec définition de fonctions globales, liaisons locales, alternatives, boucles, affectations. On dote se langage de primitives, notamment pour manipuler des tableaux, et enfin une construction `switch` sur les entiers qui se compile en alternatives imbriquées de façon dichotomique.

Le Listing 7 donne une traduction du programme de la figure 2 dans ce mini-langage. Remarquez que le programme est beaucoup plus lisible et concis.

```
1 let main () =
2   let code = {0,1,0,0,1,1} in
3   for pc = 0 to 5 do
4     switch code[pc] case
5       0: print "push"
6       1: print "pop"
7   done
```

Listing 7 – esquisse d'interprète écrit dans un mini-langage compilé vers `Jack`

3.5 Une cible plus adaptée : `Stack!`

Nous compilons un mini-langage vers `Jack`, qui est lui même compilé vers la machine à pile `Stack!`. Le compilateur `Jack` est un exécutable java et son code source n'étant a priori pas accessible. Il n'est donc pas possible de modifier son générateur de code, ce qui empêche certaines optimisations.

Le langage `Stack!`, en revanche, expose des constructions de plus bas niveau – telle que des `goto`. La compilation de notre mini-langage vers `Stack!`, et non plus vers `Jack` est donc un choix naturel, qui facilite l'ajout de nouveaux traits et finalement l'implantation d'un langage plus réaliste à la `ML`. Ce dialecte de `mini-ML`, compilé vers la plate-forme `Nand2Tetris`, peut alors servir de langage d'implantation machine virtuelle OCaml, laquelle peut ensuite être compilée et exécutée sur la plate-forme.

3.6 Notre chaîne de compilation d'OCaml vers Nand2tetris

La figure 3 présente notre extension de la chaîne de compilation Nand2Tetris. Il s'agit en premier lieu d'un compilateur Mini-ML ciblant la plate-forme Nand2Tetris, puis de l'implantation en Mini-ML d'une machine virtuelle OCaml.

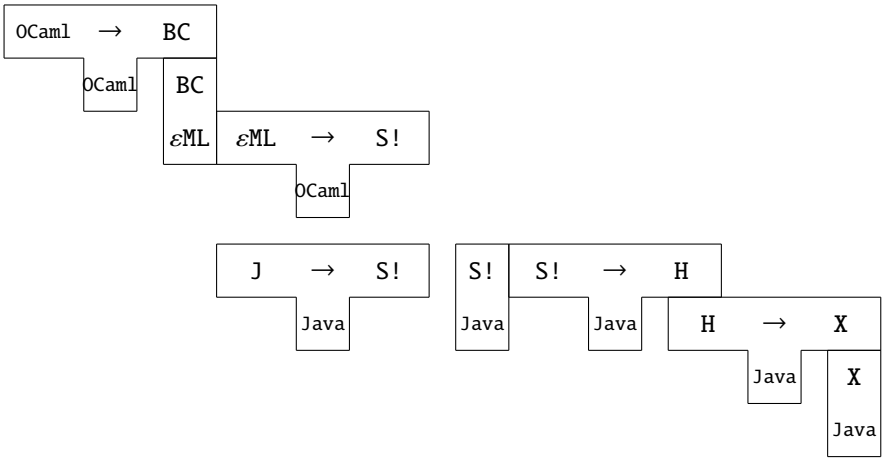

Figure 3 – La chaîne de compilation Nand2Tetris + OCaml

Le compilateur ocamlc prend un programme OCaml et engendre du bytecode caml (BC). Notre implantation de la machine virtuelle OCaml est écrite en Mini-ML (εML) est compilée vers Stack! (S!). La traduction de notre implantation de la ZAM en Stack! peut alors être soit exécuter sur la machine virtuelle Stack! écrite en Java, ou soit traduite à son tour vers l'assembleur Hack (H) puis vers du code machine (X) exécutable sur une architecture cible simulée en Java. Notez que le choix de Stack! comme langage cible du compilateur Mini-ML facilite l'interopérabilité entre mini-ML et Jack, puisque ces deux langages on ainsi la même bibliothèque d'exécution. Notre solution est ainsi compatible avec le projet Nand2Tetris existant, et s'inscrit dans la démarche des auteurs, encourageant le développement d'autres langages de programmation à vocation pédagogique sur la plate-forme. Les auteurs évoque d'ailleurs le langage Scheme [31] comme une piste possible, quitte à modifier en profondeur la machinerie virtuelle de Nand2Tetris.

Dans la section suivante, on présente la compilation de Mini-ML vers la plateforme Nand2Tetris. Notre implantation en mini-ML de la machine virtuelle OCaml sera ensuite détaillée.

4 Compilation de `Mini-ML` vers la plateforme `Nand2Tetris`

Dans cette section, on cherche à implanter un compilateur pour un langage de programmation à la `ML` ciblant la plate-forme `Nand2Tetris`. Une implémentation de la ZAM dans ce langage est discutée à la section suivante. Dans un premier temps, on s'intéresse au langage `PCF` – un λ-calcul appliqué introduit par Gordon Plotkin en 1977 [25]. On considère plus spécifiquement une extension de `PCF` avec polymorphisme paramétrique, appelée **PCF$_{+\text{poly}}$**. De plus, on limite les constructions de `PCF` au strict nécessaire, pour une mise en œuvre ludique des techniques fondamentales de compilation des langages fonctionnels, ciblant la plate-forme `Nand2Tetris`. Par la suite, on implante un langage plus réaliste, que nous appelons `mini-ML`. Il s'agit d'un sous-ensemble d'`OCaml`, extension de `PCF` avec des types inductifs, des traits impératifs, des annotations de types, et un support pour la compilation séparée. Nous détaillons l'implantation de ces différentes extensions ainsi que leur impact sur le *runtime* de `mini-ML`. On s'intéresse enfin à la mise en œuvre d'optimisation de code, visant à produire un exécutable raisonnablement efficace.

4.1 Un noyau fonctionnel : PCF$_{+\text{poly}}$

4.1.1 Syntaxe abstraite

Le Listing 8 donne la syntaxe abstraite de PCF$_{+\text{poly}}$. Un programme est réduit à une expression et calcule une valeur. La valeur de (`ifz` a_1 `then` a_2 `else` a_3) est la valeur de a_2 si la valeur de a_1 est 0, ou la valeur de a_3 sinon. Remarquez l'absence d'annotations de types dans les programmes.

Listing 8 – syntaxe abstraite de PCF$_{+\text{poly}}$

```
1  a ::= x                                  identificateur
2      | λx.a                               abstraction de fonction
3      | a1 a2                              application de fonction
4      | let x = a1 in a2                   liaison locale
5      | letrec x1 = λx2.a1 in a2           définition récursive locale
6
7      | n                                  entier positif
8      | a1 + a2                            addition
9      | a1 − a2                            soustraction
10     | ifz a1 then a2 else a3             conditionnelle
11
12 π ::= a.
```

4.1.2 Système de types

PCF$_{+\text{poly}}$ est un langage statiquement typé. Le type de chaque sous-expression du programme est inférée et les programmes incorrects d'après notre discipline de type sont rejetés à la compilation. C'est le cas du programme suivant :

```
((λx.x) + 1).
```

Algèbre de types Le Listing 9 donne l'algèbre de types PCF$_{+\text{poly}}$. On limite la quantification en tête des types – c'est le système de type de Damas-Hindley-Milner – pour palier à l'indécidabilité du typage de Système F [36], le λ-calcul polymorphe introduit indépendamment par Girard et Reynolds.

Jugement de typage La figure 4 présente les règles de typage que nous avons implémentées. Il s'agit du système de types d'Hindley-Milner dirigé par la syntaxe. La présentation des règles est telle que, pour toute expression, une seule règle s'applique. on note $\Gamma \vdash e : \tau$ le jugement de typage, qui se lit « dans l'environnement Γ, e à le type τ ». L'environnement Γ associe un type $\Gamma(x)$ à toute variable x libre dans e. On note $V(\Gamma)$ l'ensemble des types dans Γ. On note $\mathscr{L}(\tau)$ l'ensemble des variables de types libres dans τ, défini par $\mathscr{L}(\texttt{int}) = \emptyset$, $\mathscr{L}(\tau_1 \to \tau_2) = \mathscr{L}(\tau_1) \cup \mathscr{L}(\tau_2)$ et $\mathscr{L}(\forall\alpha.\tau) = \mathscr{L}(\tau) \setminus \{\alpha\}$. On note $\Gamma[x : \tau]$ l'extension de l'environnement Γ avec la liaison de l'identificateur x au type τ, définie par $\Gamma[x : \tau](x) = \tau$ et $\Gamma[x : \tau](y) = \Gamma(y)$ si $y \neq x$. La relation $\tau \leqslant \forall\alpha_1 \dots \alpha_n\tau'$ signifie que τ_1 est une instance de $\forall\alpha_1 \dots \alpha_n\tau_2$. On note $Gen(\tau, \Gamma) = \forall\alpha_1 \dots \forall\alpha_n.\tau$, où $\{\alpha_1 \dots \alpha_n\} = \mathscr{L}(\tau) \setminus \mathscr{L}(\Gamma)$.

Listing 9 – algèbre de type de $\mathsf{PCF}_{+\mathsf{poly}}$

```
1 | τ ::= int              type primitif
2 |     | τ1 → τ2          type fonctionnel
3 |     | α                variable de type
4 |     | ∀α.τ             type polymorphe
```

[VAR]
$$\frac{\Gamma(x) \leqslant \tau}{\Gamma \vdash x \ : \ \tau}$$

[INT]
$$\frac{}{\Gamma \vdash n \ : \ \texttt{int}}$$

[ADD]
$$\frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \ + \ e_2 \ : \ \texttt{int}}$$

[SUB]
$$\frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 \ - \ e_2 \ : \ \texttt{int}}$$

[IFZ]
$$\frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \tau \qquad \Gamma \vdash e_3 \ : \ \tau}{\Gamma \vdash \ \textbf{ifz} \ e_1 \ \textbf{then} \ e_2 \ \textbf{else} \ e_3 \ : \ \tau}$$

[ABS]
$$\frac{\Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma : \lambda x.e \ : \ \tau_1 \to \tau_2}$$

[APP]
$$\frac{\Gamma \vdash e_1 \ : \ \tau \to \tau' \quad \Gamma \vdash e_2 \ : \ \tau}{\Gamma \vdash e_1 \ e_2 \ : \ \tau'}$$

[LET]
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \mathit{Gen}(\tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash \ \textbf{let} \ x = e_1 \ \textbf{in} \ e_2 : \tau_2}$$

[LETREC]
$$\frac{\Gamma[x : \alpha] \vdash e_1 \ : \ \tau_1 \quad \alpha \notin V(\Gamma) \quad \Gamma[x : \mathit{Gen}(\tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash \ \textbf{letrec} \ x = e_1 \ \textbf{in} \ e_2 \ : \ \tau_2}$$

Figure 4 – Typage de $\mathbf{PCF}_{+\mathbf{poly}}$

4.1.3 Schémas de compilation

En premier lieu, on se dote d'une bibliothèque d'exécution, qu'on appelle `RUNTIME`, écrite à la main en bytecode `Stack!` qui fournit quatre primitives suivantes :

— `(RUNTIME.print_int` n`)`, qui affiche à un entier sur un flux de sortie ;
— `(RUNTIME.make_block` n`)` qui rend l'adresse d'un bloc de taille n nouvellement alloué dans le tas ;
— `(RUNTIME.get_field` a n`)` qui rend l'entier stocké dans le champ d'indice n du bloc d'adresse a ;
— `(RUNTIME.set_field` a n x`)` qui assigne l'entier x au champ d'indice n du bloc d'adresse a.

Notre compilateur $\mathsf{PCF}_{+\mathsf{poly}}$ met en œuvre un environnement ρ liant chaque nom de variable à un code compilé. On maintient de plus un indice n correspondant au nombre de variables de l'environnement introduites par **let** ou **letrec**.

On utilise les notations suivantes :

— $\emptyset$ est l'environnement vide ;
— $\rho(x)$ est le code lié à la variable x dans l'environnement ρ ;
— $\rho[x \leftarrow c]$ est l'extension de ρ avec la liaison de x à c, définie telle que : $\rho[x \leftarrow c](x) = c$ et $\rho[x \leftarrow c](y) = \rho(y)$ ssi $x \neq y$;
— $|\rho|$ est le nombre de liaisons dans ρ.

Les figures 5 et 6 donnent les schémas de compilation de $\mathsf{PCF}_{+\mathsf{poly}}$ vers `Stack!`. La compilation d'un programme [PROG] est un programme `Stack!` qui calcule une expression dans un environnement initial vide, puis imprime sa valeur – un entier – sur la sortie standard. La compilation d'un entier `PCF` est l'empilement d'un entier `Stack!`. L'addition [ADD] et la soustraction [SUB] de deux expressions est la compilation des deux expressions suivies de l'instruction `add` ou `sub`. La compilation d'une alternative [IFZ] utilise les instructions **goto**, **label** et **if-goto** de `Stack!` en sortie que soit calculé la conséquence ou l'alternant suivant la valeur de la condition. La compilation d'une variable [VAR] est un « morceau de programme » liée à la variable dans l'environnement. La compilation d'une liaison locale

[LET] dans un environnement à n variables locales est la compilation de l'expression d'initialisation dans le même environnement, dépilée dans l'emplacement de variable locale $n+1$, suivi de la compilation du corps du bloc dans un environnement étendu où le nom de la variable est liée à l'instruction d'accès à cette variable. La compilation d'une définition de fonction récursive locale opère un effet de bord. L'expression d'initialisation doit être une abstraction, et se compile dans l'environnement étendu. Le code engendré doit empiler l'adresse d'une fermeture – qui est un tableau d'entiers – dont la première case de l'environnement est non-initialisé. On opère alors un effet de bord en écrivant dans ce champ l'adresse de la fermeture elle même. Le code produit dépile l'adresse de la fermeture et la place dans l'emplacement de variable locale qui lui a été réservé, puis le corps du bloc est compilé dans le nouvel environnement.

[PROG]

$\mathscr{C}[\![e.]\!] =$
function *Main.main* k
$\mathscr{C}[\![e]\!]\emptyset, 0$
call *RUNTIME.print_int* 1
return
où k est la profondeur de (e)
en nombre de **let** ou **letrec** traversés.

[INT]

$\mathscr{C}[\![n]\!]\,\rho,\kappa =$ **push constant** n

[ADD]

$\mathscr{C}[\![e_1 + e_2]\!]\,\rho,\kappa =$
$\mathscr{C}[\![e_1]\!]\,\rho,\kappa$
$\mathscr{C}[\![e_2]\!]\,\rho,\kappa$
add

[SUB]

$\mathscr{C}[\![e_1 - e_2]\!]\,\rho,\kappa =$
$\mathscr{C}[\![e_1]\!]\,\rho,\kappa$
$\mathscr{C}[\![e_2]\!]\,\rho,\kappa$
sub

[IFZ]

$\mathscr{C}[\![$**ifz** e_1 **then** e_2 **else** $e_3]\!]\,\rho,\kappa =$
$\mathscr{C}[\![e_1]\!]\,\rho,\kappa$
if-goto $\Delta_{\neq z}$
$\mathscr{C}[\![e_2]\!]\,\rho,\kappa$
goto Δ_{end}
label $\Delta_{\neq z}$
$\mathscr{C}[\![e_3]\!]\,\rho,\kappa$
label Δ_{end}

[VAR]

$\mathscr{C}[\![x]\!]\,\rho,\kappa = \rho(x)$

[LET]

$\mathscr{C}[\![$**let** $x = e_1$ **in** $e_2]\!]\,\rho,\kappa =$
$\mathscr{C}[\![e_1]\!]\,\rho,\kappa$
pop local κ
$\mathscr{C}[\![e_2]\!]\,\rho', (\kappa+1)$
où $\rho' = \rho[x \leftarrow$ **push local** $\kappa]$

[LETREC]

$\mathscr{C}[\![$**letrec** $x = \lambda y.e_1$ **in** $e_2]\!]\,\rho,\kappa =$
push constant 0
pop local κ
$\mathscr{C}[\![\lambda y.e_1]\!]\,\rho', (n+1)$
pop local κ
push local κ
push constant 1
push local κ
call *RUNTIME.set_field* 3
$\mathscr{C}[\![e_2]\!]\,\rho', (\kappa+1)$
où $\rho' = \rho[x \leftarrow$ **push local** $\kappa]$

[APP]

$\mathscr{C}[\![e_1\ e_2]\!]\,\rho,\kappa =$
$\mathscr{C}[\![e_1]\!]\,\rho,\kappa$
$\mathscr{C}[\![e_2]\!]\,\rho,\kappa$
call *Main.apply* 2

Figure 5 – Schémas de compilation de $\mathsf{PCF}_{+\mathsf{poly}}$ vers `Stack!` (partie 1)

La notion de pointeur de fonction n'existe pas en `Stack!` : les cibles des sauts sont exclusivement des étiquettes statiques ne pouvant pas être stockées en mémoire. Faute d'un mécanisme plus adapté, nous choisissons donc de représenté le *pointeur de code* par un entier ℓ stocké dans le champ 0 de la fermeture. Alors la fonction *Main.apply*, engendrée à la compilation, discrimine sur cet entier ℓ et appelle la fonction $r(\ell)$ correspondante. Cela a pour conséquence l'inefficacité du code exécutable réalisant l'application, de par l'absence de construction `switch` en `Stack!`. En effet, sa complexité temporelle est linéaire en le nombre de fermetures dans le programme. De plus, ce code de *dispatch* alourdit le programme `Stack!` généré, qui du reste peut comporter au plus 32762 instructions. Le symbole $\rightsquigarrow$ dans le schéma [ABS] indique que la fonction *Main.apply* est étendue à chaque nouvelle compilation d'une fermeture.

[ABS]

Soit $\rho = \emptyset[x_1 = c_1]...[x_m = c_m]$.

$\mathscr{C}[\![\lambda x.e]\!]\,\rho, \kappa =$

```
[ push constant (m + 1)
   call RUNTIME.make_block 1
   pop tmp

   push tmp
   push constant 0
   push constant ℓ
   call RUNTIME.set_field 3

   ------------------
   push tmp
   push constant i
   C[[x_i]] ρ, n
   call RUNTIME.set_field 3
   ------------------ ∀i = 1...m
   push tmp ]

[ function r(ℓ) k
   C[[e]] ρ', 0
   return
```

où k est la profondeur de (e) en nombre de **let** or **letrec** traversed.

et $\rho' = \emptyset[x_1 \leftarrow \omega(1)]\ ...\ [x_m \leftarrow \omega(m)][x \leftarrow$ **push argument** 1]

et $\omega(i) =$

```
push argument 0
   push constant i
   call RUNTIME.get_field 2 ]

[ function Main.apply 2
   cases
   label Δ_fin
   return ] ⇝ [ function Main.apply 2
                cases'
                label Δ_fin
                return ]
```

Figure 6 – Shémas de compilation de $\mathsf{PCF}_{+\mathsf{poly}}$ vers `Stack!` (partie 2)

4.2 Une implémentation plus réaliste du langage ML

La compilation de $\mathsf{PCF}_{+\mathsf{poly}}$ vers `Stack!` suggère qu'il n'y a pas d'obstacle majeur à l'implantation d'un langage de haut niveau exécutable sur la plate-forme `Nand2Tetris`. Rappelons simplement que certaines constructions de langage, tels les `switch` et les fermetures, pâtissent de l'absence de *goto calculé* et de pointeur de fonction en `Stack!`. Qu'à cela ne tienne, nous allons montrer comment étendre $\mathsf{PCF}_{+\mathsf{poly}}$ avec de nouvelles constructions en vue de le rendre plus réaliste.

4.2.1 Représentation des valeurs

Nous avons vu, dans le compilateur $\mathsf{PCF}_{+\mathsf{poly}}$, qu'un entier $\mathsf{PCF}_{+\mathsf{poly}}$ au *runtime* est un entier `Stack!`, tandis qu'une abstraction au *runtime* est l'adresse d'une fermeture allouée dans le tas. Il n'y a pas, en `Stack!`, de moyen de différentier un entier d'un pointeur. C'est pourquoi nous pourrions envisager l'utilisation d'un bit de marque, forcé à 1 si la valeur est un entier, ou à 0 si c'est un pointeur. Nous ne le ferons pas, pour plusieurs raisons. Tout d'abord, `Stack!` ne fournit pas d'opérateur de décalage de bit, nécessaire *a priori* pour une gestion efficace du bit de marque. Une seconde raison est la difficulté à doter `Mini-ML` d'un récupérateur automatique de mémoire (GC). Les algorithmes de GC classiques explorent un ensemble de racines, desquelles on peut déduire l'ensemble des blocs atteignables à un point donné de l'exécution du programme [4]. En `Stack!`, les racines sont essentiellement les variables locales, stockées en pile dans différents blocs d'activation. Or, dans le *runtime* de `Stack!`, tout accès à la pile en dehors du bloc d'activation courant est traité comme une erreur d'exécution. L'ajout d'un GC pour `Mini-ML` nécessiterai donc la modification de l'interprète de bytecode, ou à défaut, l'allocation explicite d'une pile d'appels, spécifique à `Mini-ML`, dans le tas de la machine virtuelle `Stack!`.

Rappelons d'ailleurs que `Nand2Tetris` est une plateforme 16 bits. Toute information stockée dans les valeurs, quel qu'en soit le codage, impacte les calculs arithmétiques et tend à réduire l'espace d'adressage. On choisit donc d'éviter, à ce sujet, tout *overhead* dans le *runtime* de `Mini-ML`. La question de la représentation des valeurs OCaml sera solutionnée au niveau du *runtime* de notre implémentation de la ZAM, implantée en `Mini-ML`. Disons simplement qu'il y a, en `Mini-ML`, qu'un seul type de donnée : les entiers, représentant tantôt des entiers `Mini-ML` tantôt des pointeurs. Heureusement, le typage statique permet d'éviter les erreurs graves qui pourraient être engendrées par cette représentation imparfaite des valeurs à l'exécution.

4.2.2 Tableaux, références et chaînes de caractères

On dote `mini-ML` d'un nouveau type de données : les tableaux. Un tableau de taille n est une adresse référençant le premier champ d'un bloc alloué dans le tas comptant $n+1$ champs. Le premier champ de ce bloc est un entête dans laquelle est stocké la taille du tableau. Les n champs suivants stockent les n éléments du tableau. L'allocation d'un tableau est réalisée par une primitive issue de la bibliothèque d'exécution de `Jack` compilée en `Stack!` et utilisée depuis un fichier de bytecode écrit à la main (`ML_internal.vm`), analogue au module `RUNTIME` de la bibliothèque d'exécution de notre compilateur `PCF`. Cela constitue la bibliothèque d'exécution de `mini-ML`. Les tableaux sont mutables : il est possible d'assigner une valeur à un champ, par appel d'une primitive. Cette primitive opère un effet de bord sur la mémoire et retourne la valeur spéciale `() : unit`, laquelle, au *runtime*, est représentée par l'entier 0.

En première approximation, les références sont identifiées à des tableaux de taille 1, bien que dans ce cas-ci, l'information de taille ne soit guère pertinente. Il se trouve que l'ajout de tableaux et de références comme fonctions primitives polymorphes paramétriques, n'est pas sûr. Le fameux programme exemple du Listing 10 est alors considéré comme bien typé, de type `bool` ; c'est le problème dit des *références polymorphes*. Pour éviter cette incorrection dans notre discipline de types, nous avons adoptée une solution simple, de nature syntaxique, qui vise à généraliser le type des seules expressions sûres, dites non-expansives – constantes et abstractions notamment. Cette technique, qui porte le nom de *value restriction* [37], est actuellement mise en œuvre en OCaml, sous-une forme relâchée [13].

Listing 10 – problème des références polymorphes

```
1 let x = ref (fun x → x) in
2 x := (fun x → x + 1); !x true ;;
```

nous ajoutons à `Mini-ML` des chaînes de caractères, qui sont en fait des tableaux d'entiers. Les chaînes sont non mutables, ce qui – hormis des questions de sûreté à l'exécution – permet une optimisation de code telle que, lors

de l'exécution du programme (`let f () = "foo" in f(); f()`), la chaîne de caractères `"foo"` ne soit allouée qu'une fois, et non à chaque appelle de la fonction `f`.

4.2.3 Gestions des erreurs à l'exécution

Nous avons ajouté à `Mini-ML` des primitives de *debug*, basées sur la bibliothèque d'exécution de `Stack!`. Il s'agit essentiellement de fonctions d'affichage – `print_int`, `print_char`, `print_string`, `print_newline` – et de fonctions d'interruption du programme – `exit` et `failwith`. On introduit de plus une construction `assert` qui teste une condition (`(`e `: bool)`). Si la condition n'est pas respectée, dans un certain contexte d'exécution, alors le programme se termine sur impression d'un message d'erreur donnant en syntaxe Caml, l'arbre de syntaxe abstraite (AST) de la condition falsifiée, accompagné d'informations précises indiquant la position de l'assertion dans le programme source `mini-ML`. Remarquez que la construction `assert` embarque à la compilation de longues chaînes de caractères, ce qui accroît considérablement la taille de l'exécutable `Stack!`, et encombre le tas. Pour éviter cela, notre compilateur `Mini-ML` désactive des assertions à la compilation. Le coût des assertions dans le code généré est alors nul. Pour les réactiver, on peut utiliser l'option `-assert`.

4.2.4 Boucles

`Mini-ML` dispose déjà de fonctions récursives pour lesquelles, nous le verrons, les appels terminaux peuvent être optimisés. L'ajout des boucles est cependant utile dans la mesure où cela contribue à améliorer la lisibilité des programmes manipulant des structures de données impératives. En outre, le schéma de compilation des boucles `while` e_1 `do` e_2 `done` est très simple, similaire à celui de l'alternative. On donne en figure 11 un schéma de compilation des boucles (`for i =` e_1 `to` e_2 `do` e_3 `done`), par réécriture d'AST.

Listing 11 – schéma de compilation des boucles `for`

$\mathscr{C}[\![$ **for** i= e_1 **in** e_2 **do** e_3 **done** $]\!]$

$\rightsquigarrow$ $\mathscr{C}[\![$ **let** z = e_2 **in**
let i = ref e_1 in
while !i < z **do**
e_3; i := !i + 1
done $]\!]$
où z est une variable fraîche.

4.2.5 Un `switch` sur les entiers

Comme évoqué à la section 3, la construction `switch` est nécessaire pour une implantation efficace d'un interprète de bytecode. On ajoute donc cette construction à `mini-ML`, en conservant la syntaxe concrète OCaml (`match` e `with` $n_1 \rightarrow e_1$ | $n_2 \rightarrow e_2$ ...). Le schéma de compilation mis en œuvre correspond à la solution proposée précédemment, laquelle consiste à imbriquer des alternatives de façon dichotomique.

4.2.6 Compilation séparée et variables globales

Le bytecode `Stack!` est organisée en unité de compilation, ce qui facilite la mise en œuvre d'un système de module élémentaire pour `mini-ML`. L'ajout des variables globales est cependant plus difficile, du fait de l'absence de construction similaire en `Stack!`. Certes, `Stack!` dispose de fonctions d'arité 0. Mais on ne peut général pas compiler directement les variables globales `mini-ML` en appels de fonctions `Stack!`, car les des éventuels traits impératifs et calcul coûteux dans l'expression d'initialisation de la variable globales ne doivent être exécute qu'une fois au cours de l'exécution du programme. Ainsi, la définition de variable globales `let x = (print_int 42; 17)` doit afficher l'entier 42 *uniquement* lors de la phase d'initialisation du module, puis lier l'entier 17 à la variable `x`. Nous avons vu que `Stack!` dispose de variable statique, qui peuvent être lues et écrites par les fonctions d'un même fichier. On en déduit un schéma de compilation pour les définition de variables globales. La définition d'une variable globale `let M.x = e` est compilé en deux fonctions : une fonction d'initialisation qui exécute le code compilé de l'expression e

et place le résultat à un emplacement n réservé du segment `static`; puis, une fonction d'accès à l'emplacement n du segment `static` de ce module. Il ne reste alors qu'à appeler la fonction d'initialisation au début du programme, puis à compiler toute occurrence de la variable globales par la fonction d'accès correspondante. Remarquez que cela permet d'utiliser une variable globale x depuis un autre module, quand bien même l'emplacement n du segment `static` de cet autre module contiendrait la valeur d'une autre variable globale.

En `Stack!`, le segment `static` de chaque module a une taille limitée à quelques dizaines de mots. Le nombre de définition de variables globales est lui aussi limité. Nous proposons cependant une optimisation consistant à compiler, par de simples fonctions `Stack!`, les définitions globales dont l'expression d'initialisation est une constante, ce qui évite l'allocation d'un emplacement dans le segment `static`.

4.2.7 Types sommes et filtrage

Un type inductif est une collection de constructeurs de type. Chaque constructeur est soit constant, soit paramétrés par des expressions de types. D'un point de vue typage, une particularité de notre implantation de `mini-ML` est qu'elle ne propose pas de n-uplets. Le typage des constructeurs de types paramétrés ne fait donc pas intervenir de types *produits*.

Nous avons choisi d'associer un type fonctionnel à chaque constructeur d'un type inductif. Par exemple, le définition d'un type α `liste` ajoute au contexte de typage les constructeurs `Nil` : α `liste` et `Cons` : $\alpha \to \alpha$ `liste`. À ce titre, nous offrons l'application partielle de constructeur, comme dans le langage `Haskell`. Du point de vue de la génération de code, l'application totale d'un constructeur de type à n arguments et un tableau à $n+1$ composantes, contenant le numéro du constructeur suivi des n arguments. L'application partielle d'un constructeur de type est η-expansé en une abstraction dont le corps réalise l'application totale du constructeur. Un constructeur sans argument – alors appelé *constructeur constant* – est représenté par un entier, et non un tableau de taille 1.

On souhaite alors disposer de filtrage de motifs, pour ainsi dé-construire les valeurs d'un type inductif. On a vu qu'une valeur d'un type inductif est soit un constructeur constant, représenté par un entier donnant un numéro de constructeur, soit l'application totale d'un constructeur paramétré, représenté par l'adresse d'un bloc allouée dont le champ 0 est le numéro du constructeur. On se retrouve donc confronté à une difficulté évoquée plus haut : nous ne pouvons distinguer les entiers des pointeurs dans le *runtime* de `mini-ML`. La solution que nous proposons part du constat suivant : le tas `Stack!` dans lequel sont allouées les blocs `mini-ML` commence à l'adresse 2050, c'est une spécificité du *runtime* de `Stack!` qui est également vérifiée lorsque `Stack!` est compilé en assembleur `Hack`. Nous imposons donc que le numéro d'un constructeur constant soit toujours inférieur à l'entier 256. Il ne peut donc pas être confondu avec un bloc. Cette solution permet d'identifier le numéro de constructeur d'une valeur v d'un type inductif quelconque à la valeur de l'expression `(if v < 256 then v else (get_field v 0))`. Nous pouvons alors étendre la construction `(match` e `with n` $\to e_1$ `| ...)` en liant à des variables locales les n paramètres effectifs d'un constructeur paramétré. Nous illustrons ce principe sur le Listing 12. Le code engendré est une imbrication d'alternative. L'imbrication est dichotomique – cela n'est pas précisé dans ce schéma à deux constructeurs. Notez que le filtrage sur les entiers naturels est maintenant limitez à l'intervalle $[0\,;\,256]$.

Listing 12 – schéma de compilation du filtrage

```
1  𝒞⟦ type α t = Empty : α t | Cons : α → α t → α t
2      match e with Empty → e1 | Cons(h,t) → e2 ⟧
3
4      ⇝ 𝒞⟦ let  z = e in
5            if z = 0 then  e1
6            else if z >= 256 && (get_field z 0) = 1
7                 then let h = get_field z 1 in
8                      let  t = get_field z 2 in
9                       e2
10                else (failwith "not exhaustive.") ⟧
11
12           where z is a fresh variable.
```

4.2.8 Bibliothèque standard

L'environnement `Mini-ML` comprends une bibliothèque standard écrite en `Mini-ML`. Quatre modules sont proposées : `Pervasives.ml` ouvert par défaut, `Array.ml, List.ml` et `String.ml`.

4.3 Structure du compilateur `Mini-ML`

4.3.1 AST décoré (PAST)

Le compilateur `Mini-ML` comprend un analyseur lexical et un analyseur syntaxique qui permette de construire un arbre de syntaxe abstraite (PAST) d'un programme, transportant des annotation de types et l'information de position de chaque nœud dans le fichier source.

4.3.2 Typeur

C'est au niveau du PAST qu'est réalisée la phase de typage. Les annotations de types sont ainsi prisent en compte. L'information de position est utilisée pour mettre en forme un messages d'erreur précis à chaque fois qu'un programme est rejeté par notre discipline de type.

4.3.3 AST

Le PAST est traduit vers une nouvelle représentation arborescente de la syntaxe abstraite (AST) allégée de toute information de types et de positions [1].

4.3.4 Réécriture d'AST et optimisation

Le *sucre syntaxique* est alors éliminé par réécriture de l'AST – les boucles `for` sont réécrites en boucles `while`, par exemple. C'est également au niveau de l'AST que sont réalisés des optimisations de code présentées plus loin.

4.3.5 AST noyau (KAST)

L'AST est traduit AST *noyau* (KAST), à la manière du cours de Compilation de L3 informatique à Sorbonne Université. Lors de cette traduction (AST → KAST), les variables sont encodés par des entiers suivant une représentation en indices de De Brijn inversée, et les abstractions sont transformées en fermetures.

4.3.6 Générateur de code

Le KAST est traduit vers une représentation structurée du bytecode `Stack!` (BC) sur laquelle est réalisée une nouvelle phase d'optimisation de code. Un *pretty-print* permet enfin de traduire la représentation structurée bytecode en un source `Stack!`.

4.4 Optimisations de code

D'après sa spécification, la machine virtuelle `Stack!` dispose d'une pile d'exécution de moins de 2 000 mots et d'un tas de environ 14 000 mots. De plus, nous avons observé que l'implantation Java de la machine virtuelle `Stack!` accepte des programmes – en code octets – de moins de 32 000 instructions. Cela correspond au ressources physiques de la plupart des micro-contrôleurs, une mémoire vive de quelques kilo-octets, et une mémoire flash de quelques mégaoctets [33]. La génération de code compact, empilant moins et allouant peu est donc une priorité. C'est pourquoi notre compilateur `Mini-ML` vers `Stack!` met en œuvre des techniques classiques de compilation optimisante pour langages fonctionnels.

1. Une exception notable est le nœud de l'AST correspondant à la construction (`assert` *e*) de Mini-ML, qui embarque les informations de positions lorsque l'option `-assert` est activée.

4.4.1 λ-lifting

Le λ-lifting [14] est une technique visant à éliminer les abstractions en les remplaçant par des fonctions globales. Les listings 13 et 14 illustre ce principe, sur une définition de fonction locales dont le corps comporte une variable libre.

Listing 13 – une fonction locale

```
1 let a = 42 in
2 let g = (fun y -> y + a) in
3   g 1
```

Listing 14 – λ-lifting

```
1 let _lambda1 a y = y + a
2 let g = _lambda1 42 in
3   g 1
```

4.4.2 Intégration d'appels de fonctions

L'intégration d'appels de fonctions (*inlining*) consiste à remplacer textuellement le site d'appel de la fonction par le corps de la fonction appelé dans lequel l'argument formel est lié à l'argument effectif. Cette phase de réécriture accroît mécaniquement la taille de l'AST, et peut d'ailleurs ne pas terminer, en cas d'intégration d'un appel récursif. Les compilateurs mettent en œuvre des heuristiques pour déterminer quel appel de fonction doit être préservé ou réécrit. L'heuristique que nous proposons consiste à calculer la profondeur de l'AST de la fonction appelée. Si cette profondeur excèdent une certaine constante n, alors l'appel est préservé. On peut indiquer au compilateur `Mini-ML` la valeur de cette constante n, via l'option `-inline=`n.

Listing 15 – deux fonctions globales

```
1 let f x = x + 1
2 let g () = 42 + f 5
```

Listing 16 – Intégration d'appels de fonctions

```
1 let f x = x + 1
2 let g () = 42 + (let x = 5 in x + 1)
```

Les listings 15 et 16 donne un exemple simple. Notez bien qu'il y a systématiquement duplication de code. Ainsi, que le site d'appel `(f 5)` est la seule occurrence d'un appel à `f` dans le module courant, `f` peut être utilisée par d'autre modules. Il serait bien sûr intéressant de répercuter cette transformation sur plusieurs modules. Nous ne l'avons pas encore implémenté car cela complexifie le processus de compilation.

4.4.3 Propagation de constantes

La propagation de constantes (*constant folding*) consiste à évaluer partiellement les programmes à la compilation. Le Listing 17 donne le résultat obtenu par propagation de constantes à partir de l'AST correspondant au Listing 16.

Listing 17 – Propagation de constantes

```
1 let f x = x + 1
2 let g () = 48
```

Dans un langage disposant de traits impératifs, des précautions doivent être prise à la propagation de constantes, pour que les effets de bord ne soient pas éliminés, ou au contraire répliqué. Par exemple, on ne propage une variable locale que si elle est liée à une constante. Une autre difficulté vient du fait de la représentation des valeurs, différentes dans le *runtime* du langage d'implantation du compilateur et le *runtime* du langage cible. En particulier, les opérations entières natives en OCaml sont réalisées modulo 31 bits, tandis qu'en `Stack!`, l'arithmétique est sur 16 bits avec erreur à l'exécution en cas de débordement. Remarquez que la propagation de constantes permet de détecter statiquement certaines divisions par zéro. Le compilateur `Mini-ML` effectue deux passes de propagation de constantes : l'une sur l'AST, l'autre sur la représentation structuré du bytecode `Stack!`.

4.4.4 Globalisation des valeurs allouées non mutables

L'exécution du code résultant de la compilation d'une chaîne de caractères alloue un bloc, le remplis, et empile son adresse. Toute chaîne de caractères dans un boucle par exemple, est alors allouée à chaque itération, ce qui pose un réel problème car `Mini-ML` n'a pas de GC. La solution consiste à extraire dans des variables globales les chaînes de caractères du programme – lesquelles, en Mini-ML, sont non-mutables – par une réécriture d'AST, comme on peut le voir sur les listings 18 et 19.

Listing 18 – création d'une chaîne dans une fonction

```
1 while true do
2   print_string "hello world"
3 done
```

Listing 19 – Globalisation

```
1 let __static1 = "hello world"
2 while true do
3   print_string __static1
4 done
```

4.4.5 Élimination des appels terminaux

L'élimination des appels terminaux est optimisation courante dans les langages fonctionnels – pensez à la spécification de `Scheme`, qui requière que tout dialecte du langage soit *tail-recursive* [1]. Pour réaliser cette optimisation, nous avons ajouter deux instructions `label` et `goto` dans l'AST noyau de `Mini-ML`, à la manière de certaines formes spéciales de la culture lispienne [26]. Les listings 20 et 21 présente l'effet de cette optimisation sur une fonction récursive terminale élémentaire.

Listing 20 – une fonction locale

```
1 let rec fact acc n =
2   if n = 0 then acc
3   else fact (acc * n) (n - 1)
```

Listing 21 – Propagation de constantes

```
1 let rec fact acc n =
2   label fact;
3   if n = 0 then acc
4   else goto fact (acc * n) (n - 1)
```

Les Listing 14, 16, 17, 19 et 21 ont été produit automatiquement par impression de l'AST post-optimisation, en syntaxe Caml, grâce à l'option `-printast` du compilateur `Mini-ML`.

4.5 Discussion

Dans cette section, nous avons présenté la compilation d'un langage *jouet*, variante de `PCF` avec typage polymorphe, vers la plate-forme `Nand2Tetris`. Ce compilateur pour `PCF`, implanté essentiellement à des fins pédagogiques, pose déjà la question de la représentation des valeurs au *runtime*, et en particulier la représentation des valeurs fonctionnelles par des fermetures, implanté par des blocs alloué en mémoire transportant l'équivalent d'un pointeur de code est l'environnement de la valeurs fonctionnelles. Cette représentation plate des fermetures est analogue à celle mise en œuvre dans le compilateur `ML` de Cardelli [5]. Une autre représentation possible consisterait à chaîner les environnements, comme c'est le cas dans la Machine Abstraite Catégorique [10].

À partir du compilateur `PCF`, nous avons développé un langage plus réaliste `Mini-ML`. Nous avons vu que l'ajout de traits impératifs dans un langage polymorphe impose certaines précautions, et nous avons à ce titre implémenté *value restriction* de Wright [37]. Nous avons ajouté également des types sommes à `Mini-ML`, afin de pouvoir représenter la syntaxe abstraite d'un langage dans un programme `mini-ML`, et ainsi pouvoir écrire des interprètes et compilateur en `Mini-ML`. Enfin, nous avons entamé la traduction du compilateur `Mini-ML` en `Mini-ML` et avons esquissé son auto-amorçage. Le compilateur `Mini-ML` met en œuvre des techniques d'optimisation de code, qui permettent d'engendrer un exécutable `Stack!` de meilleure qualité, au sens de la taille de l'exécutable ainsi que la pression du programme sur le tas et la pile lors de son exécution. Le typage statique en `Mini-ML` permet un développement fiable et prodiguant d'avantage de confort au programmeur. En ce sens, `Mini-ML` apparaît comme un *bon* langage d'implantation de la ZAM pour exécuter des programmes OCaml sur la plate-forme `Nand2Tetris`.

5 Implantation de la ZAM en `Mini-ML`

Afin d'interpréter du bytecode `OCaml` sur la plate-forme `Nand2Tetris`, nous proposons une implémentation de la ZAM en `mini-ML`. Comme `mini-ML` est un sous-ensemble d'`OCaml`, au moins pour ce qui est de la syntaxe concrète, notre implantation de la ZAM en `mini-ML` est également exécutable en `OCaml`, ce qui facilite le développement de celle-ci.

5.1 Spécification de la ZAM

La ZAM est une machine à pile décrit dans [18], accompagnée d'une bibliothèque d'exécution comportant un récupérateur automatique de mémoire (*garbage collector* ou GC).

Le jeux d'instructions du bytecode OCaml est formé de 146 instructions décrites en détails par Xavier Clerc dans `Caml Virtual Machine - Instruction set`[2].

Le *runtime* de la ZAM comprend une pile et six registres :
- Un accumulateur ;
- Un environnement local ;
- Un registre de pile (`sp`) ;
- Un pointeur de code (`pc`), référençant la position de l'instruction courante dans le bytecode ;
- Un registre `extra_args` pour les applications partielles ;
- Un registre `trap_sp` pour la gestion des exceptions.

Enfin, la ZAM peut lire et écrire dans une mémoire réparti en plusieurs segments :
- Le segment *data*, comprenant des définitions de constantes manipulées par le programme ;
- Le segment *global*, qui s'apparente à un large bloc alloué à l'extérieur du tas ;
- Le tas.

On peut regrouper les instructions en différentes catégories :
- opération sur la pile ;
- allocation et manipulation de blocs en mémoire, dont flottant, tableau de char et objets ;
- opérations arithmétiques et logiques ;
- création et application de valeurs fonctionnelles : notament les instructions CLOSURE, APPLY, CLOSUREREC ... ;
- appels de fonctions extérieures ;
- gestion des exceptions.

La majeure partie des 146 instructions de ZAM sont des raccourci. Par exemple, L'instruction `ACC0` est l'abréviation de `(ACC 0)`, tandis que `(PUSHACC` n`)` est l'abréviation de `(PUSH; ACC` n`)`.

2. http://cadmium.x9c.fr/distrib/caml-instructions.pdf

5.2 Représentation des valeurs OCaml sur la plate-forme `Nand2Tetris`

Avant d'écrire la machine virtuelle, il y a une question importante à se poser : comment les valeurs vont-elles être représentées ?

Le seul type de donnée disponible sur la plate-forme `Nand2Tetris` est le type entier 16 bits signés en représentation complément à 2. Un bit est réservé au signe. On a donc des entiers allant de -32768 à 32767.

Or, l'interprète de bytecode OCaml et la bibliothèque d'exécution manipulent à la fois des valeurs immédiates et des pointeurs. Il faut donc être en mesure de différentier un immédiat d'un pointeur. Une technique classique consiste à réserver le bit de poids faibles – ou bit de marque – positionner à 0 si c'est une adresse, ou à 1 si c'est un immédiat. Notez que cette représentation est très efficaces sur les architectures classiques, car les adresses sont en général alignées sur des multiples de 4 et ont donc naturellement leur premier bit à 0. En outres, l'instruction de décalage de bit s'exécute très rapidement sur la plupart des processeurs. Il se trouve que le jeu d'instruction de l'assembleur `Hack` ne fournit pas d'instruction de décalage de bits. Il pourrait certes être simulé à l'aide de multiplications ou de divisions par 2, elles mêmes simulées par des additions et soustractions, mais cela serait inefficace et pose certaines difficultés lorsque les entiers sont négatifs.

Une autre technique, dite d'*unboxing*, consiste à allouer toutes les valeurs, y compris les entiers immédiats. Cette technique est utilisé dans le *runtime* de Java notamment. Il faut toutefois reconnaître que cette approche est peu efficace. Sur la plateforme `Nand2Tetris`, cela s'avère même rédhibitoire, du fait d'un espace d'adressage réduit, comparable à certain micro-contrôleurs [33].

Nous avons donc opté pour une approche différente, que l'on pourrait qualifier de *positiviste* : comme les adresses des blocs alloués en mémoires sont comprises entre 0 et 16384, on constate qu'il reste à disposition les entiers supérieurs à 16384, ainsi que les entiers négatifs, pour représenter les valeurs immédiates. Notre choix – totalement arbitraire et que l'on estime toutefois raisonnable – est de considérer que les programmes manipule en général d'avantage les entiers positifs que les entiers négatifs. On offre donc aux valeurs immédiates l'espace [-16385 ; 32767] et l'on décale l'espace des pointeurs dans l'intervalle [-32768 ; -16384]. Cette solution est très efficace puisque les valeurs immédiates sont alors encodées par elles même, tandis qu'une adresse n est encodée par l'entier $((-n) - 16384)$.

En résumé, une valeur dans le *runtime* OCaml (ou *mlvalue*) est un entier représentant soit une valeur immédiate [-16385,32767], soit un pointeur [-32768, -16384]. Les *mlvalues* sont utilisées partout dans la machine virtuelle : la pile et le tas sont des tableaux de *mlvalues*, les registres quant à eux référencent une *mlvalue*.

Un bloc est une zone mémoire allouée dans le tas et contient un en-tête sur 1 bit indiquant la taille du bloc (nombre de champs) et son tag (le type du bloc : fermeture, tag infixe pour les fonctions mutuellement récursives, forward pointer pour le GC...).

En plus de son entête le bloc possède des champs, dont le nombre est donc fixé à la création de celui-ci. Les champs étant des *mlvalues*, ils peuvent être des valeurs immédiates ou des pointeurs.

Remarquez que la spécification de la ZAM nécessite un support pour les flottants en représentation IEE754. Comme la plate-forme `Nand2Tetris` n'en dispose pas, et que le codage des flottants avec des entiers s'éloigne assez largement de notre sujet de recherche, nous avons choisis de ne pas traiter les instructions de la ZAM, manipulant les flottant.

En ce qui concerne les chaînes de caractères, l'approche classique consiste à coder plusieurs caractères dans un même entier, à l'aide de masques et d'opérateurs de décalage de bits. Cette technique ne peut cependant pas être mise en œuvre sur la plate-forme `Nand2Tetris`, pour les raisons évoquez plus haut. Nous avons donc choisies de représenter les chaînes de caractères par des entiers 16 bits – un entier par caractère – consécutives en mémoire.

La représentation des valeurs étant déterminée, la machine virtuelle peut être implémentée. Dans les prochaines sections nous mettrons en avant les parties que l'ont estime plus compliqués à mettre en oeuvre et plus intéressantes à présenter.

Le point d'entrer de la machinerie est le fichier main.ml, qui va appeler la boucle d'exécution de l'interprète de la VM, qui se trouve dans **interp.ml**.

La boucle principale exécute les instructions les unes après les autres, celles-ci modifiant la pile, le tas et les différents registres/champs.

Lors de l'exécution d'une instruction qui alloue de la mémoire dans le tas, la fonction d'allocation d'un bloc de **alloc.ml** est appelée. Celle-ci lancera la récupération de mémoire quand le tas n'a plus assez de place pour faire l'allocation demandée.

Il est important de noter qu'un mode debug est disponible dans l'interprète, mais également dans le garbage collector.

5.3 Choix d'implantation

5.3.1 Deux chaînes de compilation

Notre implémentation de la ZAM est en `mini-ML`, qui est un sous-ensemble d'OCaml. Cette spécificité intéressante permet à la fois une exécution sur la plate-forme `Nand2Tetris`, et une exécution sur la chaîne de compilation OCaml classique, par exemple via `ocamlc` et `ocamlrun`.

5.3.2 Chargement du bytecode OCaml

La plate-forme `Nand2Tetris` ne dispose pas de primitive de lecture de fichiers. Il convient donc d'incorporer à notre implémentation de la ZAM une représentation du bytecode OCaml à exécuter, dans un fichier `input.ml`. L'ensemble du code source de notre implémentation de la ZAM sera donc recompilé à chaque chargement d'un nouveau programme exécutable.

5.3.3 Décodage du bytecode OCaml

Le bytecode est préalablement analysé par un utilitaire présenté plus loin, puis imprimer dans le fichier `input.ml`, sous la forme d'un tableau d'entier `mini-ML`. Chaque entier est soit le numéro de l'une des 146 instructions que comporte le jeu d'instructions de la ZAM, soit l'opérande d'une instruction. Par la suite, nous raffinerons ce modèle afin d'incorporer un segment DATA et des directives d'appels de fonctions extérieures.

5.3.4 Organisation des sources

- **input.ml** : fichier auto-généré définissant un tableau d'entier lié à une variable `code`, contenant l'ensemble du bytecode à executer ;
- **main.ml** : point d'entrée du programme, lance l'interprète de bytecode ;
- **interp.ml** : l'interprète de bytecode OCaml ;
- **domain.ml** : définition des différents segments mémoires, avec leurs taille respectives ;
- **mlvalues.ml** : représentation des valeurs, Notez qu'il existe deux version de ce fichier, l'une destinée à être compilé sur plate-forme `Nand2Tetris`, l'autre visant la chaîne de compilation OCaml classique ;
- **block.ml** : manipulation de blocs mémoires ;
- **alloc.ml** : allocation de blocs mémoires ;
- **prims.ml** : opérations arithmétiques et logiques utilisé par l'interprète de Bytecode ;
- **data.ml** : gestion des segments globales et data ;
- **call.ml** : appels de fonctions primitives ;
- **gc.ml** : récupérateur automatique de mémoire.

5.3.5 Fonctionnement

On appelle la commande make en lui passant en argument les fichiers `OCaml` qu'on souhaite exécuter. Dès lors, ces fichiers sont compilés par le compilateur `ocamlc` afin d'obtenir un fichier de bytecode exécutable sur diverses plate-formes disposant d'une implémentation de la ZAM, et donc en particulier la plate-forme `Nand2Tetris`, suite à notre travail.

Afin d'engendrer un fichier **input.ml** tenant lieu d'exécutable pour notre implémentation de la ZAM, nous avons développé un utilitaire `obytelibParser` utilisant la bibliothèque `OByteLib`[3]. de Benoît Vaugon afin d'extraire les différents segment d'un exécutable OCaml produit par le compilateur `ocamlc`. Notez que nous utilisons au préalable l'outil `ocamlclean` pour éliminer le code mort dans l'exécutable OCaml.

3. https://github.com/bvaugon/obytelib

5.4 Développement d'un outil d'analyse de bytecode OCaml

Nous avons vu que après compilation d'un programme OCaml en fichier de bytecode, la machine virtuelle a besoin d'accéder à l'ensemble d'instructions contenu dans le bytecode afin de l'exécuté.

Nous utilisons `OByteLib` qui est une librairie permettant d'extraire les informations utiles à partir de bytecode OCaml, dont l'ensemble d'instructions.

Néanmoins, notre programme ne peut pas accéder aux informations données par `OByteLib`, car le simulateur Nand2tetris ne dispose pas de primitive pour lire dans un fichier. On doit les transmettre à notre programme d'une autre façon : pour cela on crée un programme `OCaml` qui va récupérer la sortie `OByteLib`, la transformer en un tableau d'entiers correspondant au codage des instructions à exécuter et à leurs arguments puis écrire ce tableau dans un fichier **input.ml** qui pourra être compilé par `Mini-ML` avec le reste de notre machine virtuelle.

Quelques modifications doivent être effectuées sur le retour qu'on obtient à partir d'`OByteLib`, pour s'en rendre compte prenons un exemple très simple de programme `OCaml`. Ici nous avons une déclaration de fonction f qui renvoie toujours 42, puis la fonction f est appelée.

```
1 | let f () = 42 in f ()
```

A partir de ce programme, une fois compilé et donner à `OByteLib`, on obtient la sortie suivante :

```
1  |  pc=0              BRANCH L2
2  |
3  |  pc=1    L1:       CONSTINT 42
4  |  pc=2              RETURN 1
5  |
6  |  pc=3    L2:       CLOSURE 0 L1
7  |  pc=4              PUSHCONST0
8  |  pc=5              PUSHACC1
9  |  pc=6              APPLY1
10 |  pc=7              POP 1
11 |  pc=8              STOP
```

On peut voir qu'à chaque valeur de pc correspond une instruction et ses arguments si elle en a. Nous voulons obtenir un tableau d'entiers. Il faut donc qu'à chaque case de ce tableau on ait un unique entier correspondant soit à une instruction, soit à un argument d'une instruction.

Nous allons donc modifier la sortie d'`OByteLib` pour obtenir l'effet escompté.

Une ligne contenant une instruction instr avec n arguments, ayant la position 0 à pour valeur de pc :

```
1 |  pc=x          instr arg1 arg2  ... argn
```

formera, après modification $1 + n$ lignes :

```
1  |  pc=x'          instr
2  |  pc=x' + 1      arg1
3  |  pc=x' + 2      arg2
4  |          ⋮
5  |  pc=x' + n      argn
```

Dans le tableau final, l'instruction sera bien sûr remplacée par sa représentation entière.

Nous avons également fait le choix d'avoir des références absolues vers des labels. Ici l'instruction :

```
1    pc=3     L2:      CLOSURE 0 L1
```

possède une référence vers le label L1, qui se trouve originellement à la ligne 3, ayant pour pc 1. L1 serait donc remplacé par 1 dans le tableau final.

Nous avons modifié les valeurs du pc des instructions en incrémentant le pc à chaque argument d'instruction en plus de l'incrémenter à chaque instruction, on doit donc en tenir compte. Voici l'exemple complet, avant et après modification :

Listing 22 – bytecode donné par `OByteLib`

```
1    pc=0              BRANCH L2
2
3    pc=1    L1:       CONSTINT 42
4    pc=2              RETURN 1
5
6    pc=3    L2:       CLOSURE 0 L1
7    pc=4              PUSHCONST0
8    pc=5              PUSHACC1
9    pc=6              APPLY1
10   pc=7              POP 1
11   pc=8              STOP
```

Listing 23 – bytecode final

```
1    pc=0              BRANCH
2    pc=1              L2
3
4    pc=2    L1:       CONSTINT
5    pc=3              42
6    pc=4              RETURN
7    pc=5              1
8
9    pc=6    L2:       CLOSURE
10   pc=7              0
11   pc=8              L1
12   pc=9              PUSHCONST0
13   pc=10             PUSHACC1
14   pc=11             APPLY1
15   pc=12             POP
16   pc=13             1
17   pc=14             STOP
```

L'instruction précédente, pour référencer L1 dans le tableau final, remplacera L1 par son nouveau pc, soit 2. Voici le tableau d'entiers généré :

```
1    let code = [|84;6;103;42;40;1;43;0;2;104;11;19;1;143|]
```

On retiendra donc qu'à chaque instruction/argument d'instruction correspond un unique entier dans le tableau de code et que les références aux labels se font de manière absolue.

5.5 Présentation des principaux points techniques

Durant le développement de notre machine virtuelle, certains éléments ont demandés une réflexion plus poussée. C'est notamment le cas des fermetures récursives et mutuellement récursive, des appels de fonctions extérieures, des variables globales et segment data, mais aussi de l'implémentation du garbage collector.

5.5.1 Fermetures récursives et mutuellement récursives

Les fermetures récursives (particulièrement mutuellement récursive) ont nécessité beaucoup de travail. Il s'agit en effet d'une optimisation du compilateur `ocamlc` visant à représenter les fonctions mutuellement récursives par un seul bloc en mémoire contenant à la fois les pointeurs de codes de chacune des fonctions et leur environnement partagé. La technique met en œuvre un subtile jeu de décalage pour sauter d'une fermeture à l'autre dans ce m ême bloque. Nous détaillons ici un exemple complet, donné en Listing 24 comportant 4 fonctions mutuellement récursives partagent un environnement de 5 variables.

Listing 24 – fonctions mutuellement récursives dans un programme OCaml

```
1  let a = 42 in
2  let b = 17 in
3  let c = 25 in
4  let d = 8 in
5  let e = 3 in
6  let rec f1 x = a + (f4 x)
7  and f2 x = b + (f3 x)
8  and f3 x = if x = 0 then c else a + (f1 (x-1))
9  and f4 x = d + e + (f2 x)
10 in
11   (f1 10)
```

La représentation en mémoire des valeurs allouées par le programme précédent sont des fermetures mutuellement récursives, une optimisation consistant à placer dans un même bloc alloué chaque pointeur de code et un environnement partagé :

Listing 25 – représentation mémoire de quatre fermetures mutuellement récursives

```
1  f = 4 ; v = 5 ; size = (2×f)−1+v ; tag = closure_tag
2
3  [ header ; f1; InfixTag; f2; InfixTag; f3; InfixTag; f4; 42; 17; 25; 8; 3 ]
```

Une fermeture de taille 12 est créer en mémoire, comme tout bloc elle a bien sûr un en-tête (header). L'entête du bloc est suivi d'un pointeur vers la première fonction : f1.

Ensuite vont s'alterner tags infixe et pointeur vers une fonction pour toutes les autres fonctions mutuellement récursives, on a donc un tag infixe pour f2 suivi d'un pointeur vers f2, puis pareil pour f3 et f4. On trouve finalement l'environnement de la fermeture, qui contient ici les valeurs a, b, c, d et e.

Voici le bytecode généré, correspondant à l'exemple, une fois traité par OByteLib et notre parser OByteLib.

Les arguments des instructions sont sur la même ligne que l'instruction associée pour avoir une représentation plus compacte, mais on peut voir sur la figure 7 que le pc est bien incrémenté à chaque instruction/argument.

Au début du bytecode, on fait un saut vers L6 : on met dans la pile a, b, c, d et e.

On peut voir à la ligne 50 la création de la fermeture récursive avec les quatre fonctions désignées ici par leur labels : L1 pour f1, L2 pour f2, L3 pour f3, et L5 pour f4. f1 est ensuite appelée avec 10 pour paramètre.

Dans chaque fonction on peut constater la façon dont sont appelées les autres fonctions mutuellement récursives, cela étant décrit directement en commentaire.

5.5.2 Interopérabilité : appel de fonctions extérieures

Le bytecode OCaml comporte fréquemment des appels de fonctions extérieures, par le biais d'instructions telles `C-CALL`. Les fonctions extérieures ne sont pas définies directement dans le programme à exécuter, mais accessibles dans la bibliothèque d'exécution fournie avec la machine virtuelle, ou par un mécanisme d'interopérabilité.

Le tableau d'entiers interprété par notre implémentation de la ZAM peut contenir des appels à de telles primitives, représenté par une instruction C-CALL, suivi du numéro de primitive à appeler et de ses éventuels arguments. Tous les numéros des primitives implémentées par notre VM se trouvent dans le fichier `call.ml`.

Il y a plusieurs instructions C-CALL, allant de C-CALL1 à C-CALL5, au-delà on utilise C-CALLN. Le chiffre désigne l'arité de la primitive (son nombre d'arguments).

```
1  pc=0              BRANCH L6
2
3  pc=2     L1 :     ACC0
4  pc=3              PUSHOFFSETCLOSURE 6
5  pc=5              APPLY1          // f[1] appelle f[4] : (env + 2 × (4 − 1)) = (env + 6)
6  pc=6              PUSHENVACC 7
7  pc=8              ADDINT
8  pc=9              RETURN 1
9
10 pc=11     L2 :     ACC0
11 pc=12             PUSHOFFSETCLOSURE2
12 pc=13             APPLY1          // f[2] appelle f[3] : (env + 2 × (3 − 2)) = (env + 2)
13 pc=14             PUSHENVACC 6
14 pc=16             ADDINT
15 pc=17             RETURN 1
16
17 pc=19    L3 :     ACC0
18 pc=20             BNEQ 0 L4
19 pc=23             ENVACC 5
20 pc=25             RETURN 1
21 pc=27    L4 :     ACC0
22 pc=28             OFFSETINT 1
23 pc=30             PUSHOFFSETCLOSURE 4
24 pc=32             APPLY1          // f[3] appelle f[1] : ((env − 1) + (2 × (1 − 3))) = (env − 1 − 4)
25 pc=33             PUSHENVACC3
26 pc=34             ADDINT
27 pc=35             RETURN 1
28
29 pc=37    L5 :     ACC0
30 pc=38             PUSHOFFSETCLOSURE 4
31 pc=40             APPLY1          // f[4] appelle f[2] : (env + (2 × (2 − 4))) = (env − 4)
32 pc=41             PUSHENVACC 5
33 pc=43             PUSHENVACC 4
34 pc=45             ADDINT
35 pc=46             ADDINT
36 pc=47             RETURN 1
37
38 pc=49    L6 :     CONSTINT 42
39 pc=51             PUSHCONSTINT 17
40 pc=53             PUSHCONSTINT 25
41 pc=55             PUSHCONSTINT 8
42 pc=57             PUSHCONST3
43
44 pc=58             PUSHACC0
45 pc=59             PUSHACC2
46 pc=60             PUSHACC4
47 pc=61             PUSHACC6
48 pc=62             PUSHACC 8
49
50 pc=64             CLOSUREREC 4 5 L1 [ L2; L3; L5 ]
51 pc=71             CONSTINT 10
52 pc=73             PUSHACC4
53 pc=74             APPLY1
54 pc=75             POP 9
55 pc=77             ATOM0
56 pc=78             SETGLOBAL {Rec}
```

Figure 7 – Le bytecode associé au programme donné en Listing 24

Prenons par exemple un appel à la primitive `caml_greaterequal`, qui permet de tester à partir de deux entiers v1 et v2, si v1 est supérieur ou égal à v2. Un morceau du tableau de code contenant l'appel à cette primitive serait :

```
1 [|...;94;Call.caml_greaterequal;...|]
```

Ici, on a d'abord 94 qui correspond à l'instruction C-CALL2 indiquant un appel de primitive, suivi du numéro de primitive : Call.caml_greaterequal, qui correspond en réalité à l'entier 2. Les arguments n'apparaissent pas sur l'exemple, en effet pour cet appel v2 doit être au sommet de pile et v1 dans l'accumulateur.

Listing 26 – code de l'instruction C-CALL2

```
1  | 94  C-CALL2  ->
2    let p = take_argument code in
3    let v = pop_stack () in
4    push_stack !Domain.env;
5    Domain.acc := (match p with
6                   | 0 -> Call.caml_make_vect_code  !Domain.acc v
7                   | 1 -> Call.caml_array_get_addr_code !Domain.acc v
8                   | 2 -> Call.caml_greaterequal_code !Domain.acc v
9                                          ⋮
10                  | _ -> Call.not_available ());
11   pop_stack_ignore 1
```

On peut donc voir que c'est la fonction Call.caml_greaterequal_code qui est finalement appelée et sa valeur de retour sera stockée dans l'accumulateur.

5.5.3 Variables globales et segment data

Le segment data (extrait par OByteLib) et les variables globales sont stockés en mémoire dans deux tableaux, global et data. En effet, on ne veut pas que le garbage collector puisse libérer ses blocs. Ils doivent rester accessibles tout au long de l'exécution du programme : il n'est donc pas une bonne idée de les stocker dans le tas.

Les variables globales permettent entre autre de découper son programme en plusieurs modules, chaque module pouvant appelée ce qui a été déclaré global au sein des autres modules.

Le segment data contient quant à lui les chaînes de caractères et plus globalement toutes les constantes du programme, ces données sont chargées par `OByteLib` dans le fichier **input.ml**.

5.5.4 Implantation d'un GC `Stop&Copy`

Afin d'éviter de ne plus avoir de mémoire au cours de l'exécution d'un programme par notre machine virtuelle, nous devons implémenter un récupérateur de mémoire, celui-ci sera écrit en `Mini-ML` et sera compilé en même temps que la VM.

Plusieurs algorithmes connus permettent de remplir cette tâche [38]. Nous avons implanté l'algorithme de Cheney [8], de type *Stop&Copy*, présenté dans le cours de Compilation Avancée au second semestre de M1 STL à Sorbonne Université.

Cet algorithme utilise deux tas au lieu d'un seul, cela à pour effet d'utiliser davantage de mémoire, mais il a l'avantage d'être simple mettre en oeuvre.

Notre algorithme, lorsqu'il est déclenché, ajustera aussi la taille des tas en fonction de l'occupation du tas dans lequel les blocs sont alloués.

Toutes les allocations mémoire ont lieux dans un unique tas, appelé `from_space`. Si lors d'une demande d'allocation le `from_space` ne possède plus assez de mémoire pour l'honorer, le GC est déclenché.

L'algorithme parcourt alors les racines : la pile, les globales et les registres afin de récupérer les blocs encore utilisés par le programme.

Ces blocs sont copiés vers le second tas, appelé `to_space`. Lors de la copie, on marque la version du bloc comme copié vers l'autre tas en changeant le tag du bloc et on met le pointeur vers sa nouvelle adresse dans son premier champ, ce pointeur est appelé `forward_pointer`.

Si lors du parcours des racines on tombe sur un bloc qui a déjà été copié dans le `to_space`, on modifie l'adresse qu'on parcourt afin quelle pointe vers la zone mémoire ou le bloc a été copié (on suit le `forward_pointer`).

Une fois tous les pointeurs de chaque racine traités, on doit parcourir tous les blocs qu'on a copié dans `to_space` afin d'ajuster leurs champs qui contiennent des pointeurs vers des zones mémoires appartenant au `from_space`. On ajuste ces pointeurs afin qu'ils pointent vers les blocs correspondants fraîchement copiés dans le `to_space`.

Enfin, on peut échanger `from_space` et `to_space`, les allocations seront faites dans le nouveau `from_space`.

5.6 Démonstration

Le dossier benchs contient un ensemble de tests ayant été utilisés durant le développement de la machine virtuelle. Ceux-ci peuvent bien entendu être exécutés via OCaml ou Nand2tetris. Nous allons exécuté le premier avec le simulateur Nand2tetris. Prenons un premier exemple de fonctions mutuellement récursives, donné en Listing 27.

Listing 27 – une deux fonctions paire/impaire mutuellement récursives

```
1 let rec even = function 0 -> true | n -> odd (n-1)
2 and odd = function 0 -> false | n -> even (n-1) in
3
4 if even 42
5 then print_int 1
6 else print_int 0
```

Ici, on affiche 1 si le nombre est pair, 0 sinon. On peut compiler ce programme avec `ocamlc` puis executer le bytecode généré sur notre implémentation `Mini-ML` de la ZAM, via la commande suivante :

```
make zam-miniML-run MLFILES=benchs/oddeven.ml
```

La fenêtre de l'émulateur `Nand2Tetris` se lance. Afin que l'exécution soit rapide, on va sélectionner "no animation" dans le menu déroulant "animate", puis aller chercher notre fichier Main.tst généré en cliquant sur file > load script. Le fichier se trouve dans le dossier `vm/zam-miniML/bin`. Une fois la sélection du fichier validée, on appuie sur "run", modéliser par une double flèche de couleur bleu, puis on indiquera "oui" à la fenêtre pop-up. Le programme s'exécute dès lors, et on obtient le résultat escompté sur l'écran à droite : 1.

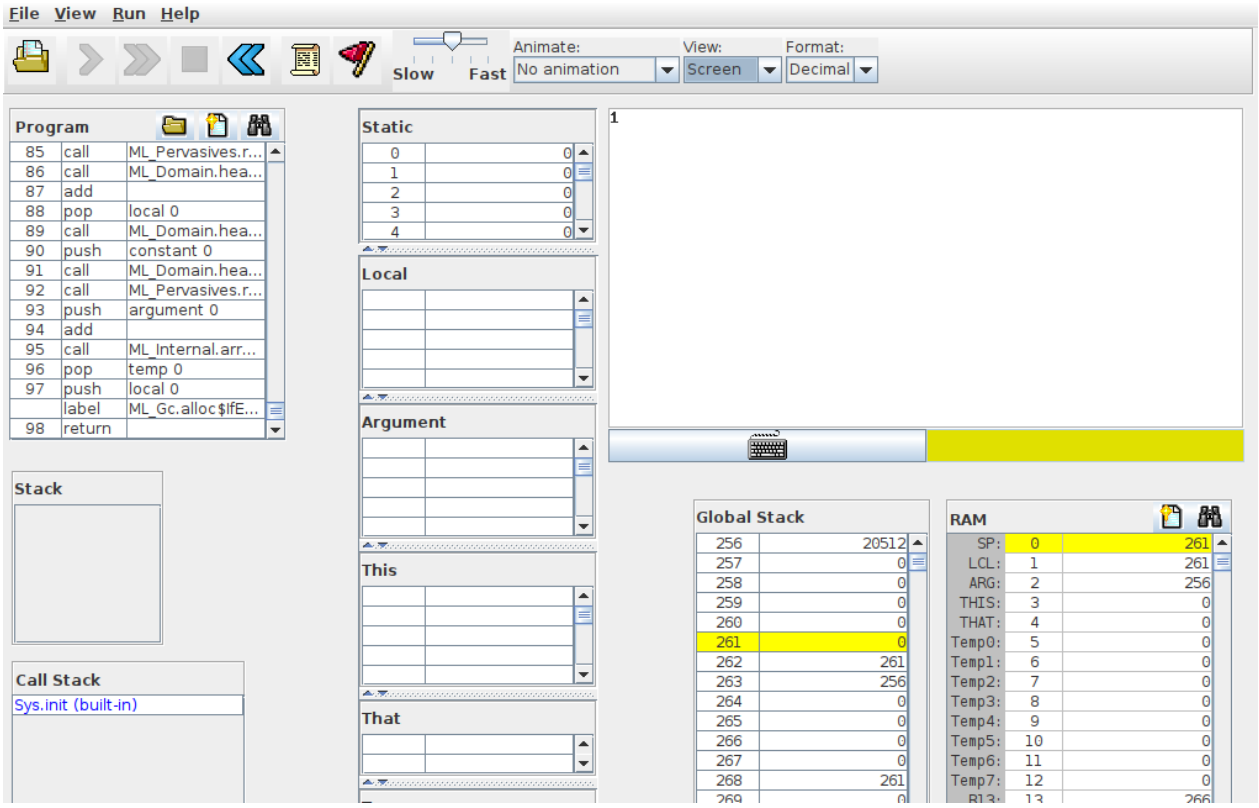

Figure 8 – fenêtre de l'émulateur `Nand2Tetris`

Nous allons maintenant exécuter un exemple sur les listes, donné en Listing 28 :

Listing 28 – un tri par insertion en OCaml

```
let rec iter f = function
| [] -> ()
| h::t -> f h; iter f t ;;

let rec sort = function
| [] -> []
| x :: l -> insert x (sort l)

and insert elem = function
| [] -> [elem]
| x :: l -> if elem < x then elem :: x :: l
            else x :: insert elem l ;;

iter print_int (sort [6;1;3;7;2;5;9;4]);;
```

Ici, on va simplement trier une liste et afficher le résultat. Tout d'abord, on crée une fonction `iter` qui prend une fonction f et une liste l en paramètres puis applique f sur chaque élément de l. Elle nous servira pour afficher le résultat. On crée ensuite deux fonctions mutuellement récursives : `sort` et `insert`, qui appliquent simplement un algorithme de trie à une liste. On appelle finalement `sort` avec une liste.

```
make zam-ocaml-run MLFILES=benchs/list_sort.ml
```

On peut voir via l'affichage des éléments de la liste résultat que la liste a bien été triée.

5.7 Tests

Listing 29 – la fonction d'Ackermann

```
let rec ack m n =
  if m = 0 then n + 1 else
  if n = 0 then ack (m-1) 1 else
  ack (m-1) (ack m (n-1))
in
  print_int (ack 3 9)
```

Le Listing 29 donne un programme relativement coûteux. Sur notre implémentation de la ZAM tournant avec `ocamlrun`, Ce programme affiche l'entier 4093 en 13,9 secondes. Exécuter `ack 3 9` directement avec `ocamlrun` produit le même affichage en 0.14s. On observe donc un facteur 100 sur cette exemple, qui est compréhensible dans la mesure où nous faisons tourner une implémentation de machine virtuelle sur une implémentation de machine virtuelle.

Le tabeau suivant confirme cette tendance :

	ocaml2tetris (en s.)	ocamlrun (en s.)
`oDivE.ml`	3.712	033
`ack.ml`	13.9	0.14
`oSieve.ml`	49.5	0.34
`loooop.ml`	54.7	0.52

Le fichier `oDivE.ml` est une code OCaml extrait de `Coq` que l'on nous a fournit. Le fichier `oSive.ml` calcule est surtout affiche l'ensemble des nombres premiers inférieurs à cent mille, en construisant explicitement une liste de taille cent mille, et en la filtrant par l'algorithme dit du *crible d'Ératosthène*. On constate que notre implantation de la ZAM tournant sur `ocamlrun` est alors 150 fois plus lente qu'`ocamlrun`. Nous pensons que cet écart par rapport aux autres tests s'explique par la profusion d'appels à la fonction `print_int`. Le fichier `loooop.ml` comporte une boucle imbriquée réalisant 10000×10000 itérations.

À travers le développement de notre machine virtuelle, nous avons pu nous rendre compte que ce n'est pas un exercice trivial. En effet notre machine virtuelle n'est pas parfaite, mais permet de mettre en évidence le fait qu'on peut porter`OCaml` sur la plateforme `Nand2Tetris`. De nombreuses améliorations restent possible que ce soit sur l'implantation de la ZAM, du GC ou du compilateur `Mini-ML`.

Nous allons présenter dans les prochaines sections d'autres travaux que nous avons réalisés durant le développement du projet, mais aussi de possibles travaux futures.

6 D'autres travaux réalisés

Parallèlement à l'implantation de la ZAM en `Mini-ML`, nous avons expérimenté des problématiques connexes. La première est l'écriture du compilateur `Mini-ML` en `Mini-ML`, qui pourrait donner lieu par la suite à l'auto-amorçage du compilateur – et par là même de toute la chaîne de compilation OCaml – sur la plate-forme `Nand2Tetris`. La seconde est l'implantation d'une machine virtuelle `Stack!` en OCaml, qui pourrait ainsi être étendue par des primitives supplémentaires, pour permettre en particulier la lecture de fichiers, point essentiel en vue de d'exécuter le compilateur `Mini-ML` sur la plate-forme `Nand2Tetris`. Rappelons qu'il s'exécute actuellement avec `ocamlrun`, et engendre du bytecode `Stack!` exécutable sur la plate-forme. C'est cette dépendance à `ocamlrun` que l'on souhaiterait couper.

6.1 Réécriture du compilateur `Mini-ML` en `Mini-ML`

L'ajout des types sommes à `Mini-ML` rend possible la définition et la manipulation de la syntaxe abstraite d'un programme, en `Mini-ML`. Nous pouvons alors implanter des interprètes et des compilateurs en `Mini-ML`. Se pose bien sûr la question de l'implémentation en `Mini-ML` d'un compilateur `Mini-ML` ciblant la plate-forme `Nand2Tetris`.

Actuellement, une très large partie du compilateur `Mini-ML` est implanté dans un sous-ensemble d'OCaml compatible `Mini-ML`. De réelles difficultés subsistent dans le *front-end* du compilateur, qui repose sur les outils `ocamllex` et `ocamlyacc`. À l'avenir, une solution consisterait à doter `mini-ML` d'une syntaxe bien-parenthésée à la `Scheme`, facilitant l'analyse syntaxique des programmes [2]. Une autre difficulté est l'absence de primitives de manipulation de fichiers, sur la plate-forme `Nand2Tetris`.

6.2 Implantation de la machine virtuelle `Stack!` en OCaml

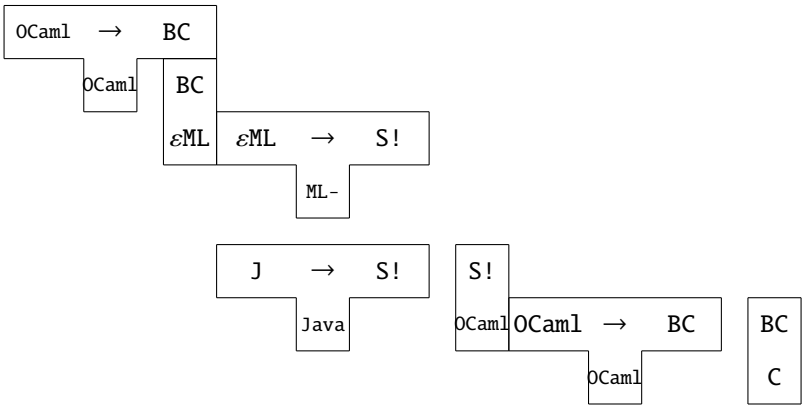

Figure 9 – Chaîne de compilation `Nand2Tetris` vers `OCaml`

7 Travaux futurs

7.1 Un GC plus réaliste

Notre GC est de type `Stop&Copy`, on a vu que ce type de GC est relativement simple à mettre en place, mais consomme beaucoup de mémoire, car il utilise deux tas au lieu d'un seul, en revanche il offre des allocations mémoire très rapide.

7.1.1 Un GC `Mark&Sweep`

Une première amélioration possible serait l'implémentation d'un garbage collector Mark and Sweep, qui n'utilise qu'un tas et donc consomme moins de mémoire, mais alloue la mémoire moins rapidement que le `Stop&Copy`.

Le Mark&Sweep parcourt toutes les racines (pile, registres...) et marque tous les blocs encore vivants (utilisés par le programme). La mémoire est ensuite parcourue dans son intégralité et chaque bloc non marqué est libéré.

7.1.2 Un GC générationnel : combinaison d'un Mark&Sweep et d'un Stop&Copy

Une deuxième amélioration possible serait l'implémentation d'un GC générationnel. Le tas est séparé en au moins deux générations : les objets jeunes, qui ont été alloués mais n'ont pas encore subi de passe de GC, et les objets vieux, qui ont survécut à au moins deux passes de GC. Le Stop&Copy serait alors utilisé sur la jeune génération et le Mark&Sweep sur la vieille génération. Cela permet d'avoir des allocations rapides tout en limitant l'espace mémoire utilisé par rapport à un GC Stop&Copy. Notez que cette technique à été largement étudié dans le cours de Compilation avancée, en M1 STL à Sorbonne université, et a donner lieu à un projet qui a largement contribué à améliorer notre compréhension de la représentation des valeurs OCaml.

7.2 Auto-amorçage de la chaîne de compilation Nand2Tetris + OCaml

Le compilateur Mini-ML vers Stack! est écrit dans un sous-ensemble d'OCaml. Actuellement, nous utilisons donc le compilateur ocamlc pour compiler le compilateur Mini-ML. Or, ocamlc ne s'exécute pas sur la plate-forme Nand2Tetris. L'auto-amorçage du compilateur Mini-ML consisterais à interpréter le compilateur Mini-ML sur la plate-forme Nand2Tetris, par le biais de notre implémentation de la ZAM, en vue de compiler le compilateur et ainsi obtenir un programme exécutable sur la plate-forme Nand2Tetris.

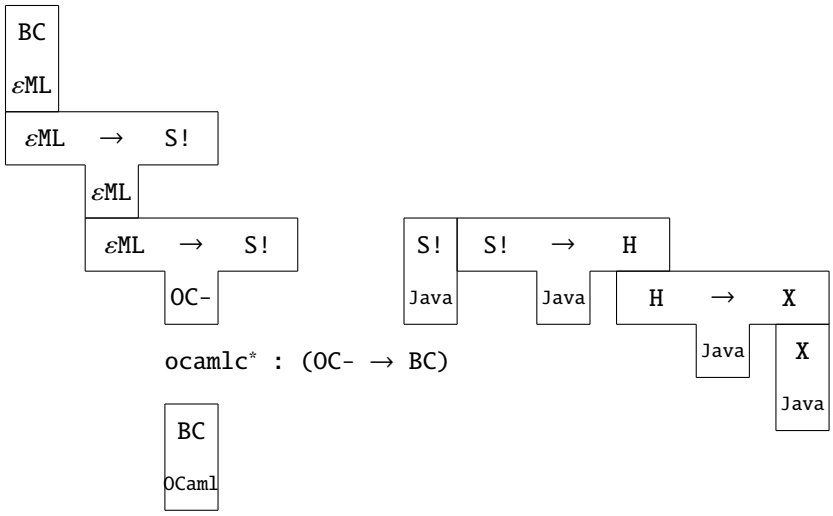

Figure 10 – Auto-amorçage de la chaîne de compilation Nand2Tetris + OCaml

La figure 10 présente une proposition d'auto-armorçage de la chaîne de compilation Nand2Tetris + OCaml. Notre implémentation de la ZAM est écrite en Mini-ML (εML), qui est un sous-ensemble d'OCaml (OC-). On peut donc exécuter du bytecode OCaml (BC) sur la chaîne de compilation OCaml. Si l'on dispose d'un compilateur ocamlc* produisant BC à partir de OC-, alors on serait en mesure de compiler en BC notre compilateur εML écrit εML $\subset$ OC-, puis à l'exécuter sur notre interprète de bytecode OCaml. Le compilateur εML, interprété par notre implémentation de la ZAM, pourrait alors se compiler lui même vers Stack!. Nous disposerions alors du source Stack! du compilateur εML, lequel, s'exécutant sur la plate-forme Nand2Tetris, permettrait de compiler en Stack! notre implémentation de la ZAM écrite en εML. Nous pourrions alors pleinement exécuter du bytecode OCaml sur la plate-forme Nand2Tetris.

Notez qu'ocamlc* peut être potentiellement implanté en BC, puisque le compilateur ocamlc de la distribution OCaml est auto-amorcé.

8 Conclusion

L'objectif de ce projet est l'exécution de programmes OCaml sur la plate-forme `Nand2Tetris`. Nous proposons pour cela un ensemble de solutions expérimentales, suivant une approche *machine virtuelle*.

Nous prenons comme point de départ les fichiers de bytecode OCaml exécutables engendrés par le compilateur `ocamlc`. Ce bytecode peut être transformé, en particulier par l'outil `ocamlclean` et la bibliothèque `ObyteLib`, developpés par Benoît Vaugond, et éprouvés dans le cadre de projets universitaires et industriels, tels `OCapic` [34] et `OmicroB` [33]. L'outil `ocamlclean` permet de détecter et éliminer du code mort dans un bytecode OCaml exécutable. La bibliothèque `ObyteLib` permet d'extraire les différentes sections du bytecode OCaml, et permet l'évaluation partielle d'un bytecode OCaml exécutable. Nous pouvons ainsi reconstruire un programme en bytecode OCaml dans le but de le rendre manipulable sur la plate-forme `Nand2Tetris`.

La plate-forme `Nand2Tetris` donne accès à un ensemble réduit de primitives. à l'aide de ces primitives, nous représentons le bytecode OCaml de façon *plate*, par un bloc alloué en mémoire et dont chaque champ correspond à l'opcode d'une instruction de bytecode ou à l'opérande d'une instruction. Ce choix de représentation est similaire à celui adopté par `OmicroB`. La plate-forme `Nand2Tetris` propose trois langages aptes à manipuler un tel bloc. L'assembleur `Hack`, Le bytecode `Stack!` et le langage objet `Jack`. Tout l'enjeu consiste à implanter dans l'un de ces trois langages un interprète de bytecode OCaml, et une bibliothèque d'exécution comprenant en récupérateur automatique de mémoire.

Nous sommes alors confronté à une première difficulté. Un interprète de bytecode décode en général les instructions une à une, en suivant le flot de contrôle du programme exécutable. Or, la spécification de la machine virtuelle OCaml (ZAM) définie 146 instructions. Ce décodage d'une instruction parmi 146 se programmerait donc naturellement par un `switch`, mais cette construction n'est offerte dans aucun des trois langages envisagés. La raison profonde à cela est l'absence de *goto calculé* en assembleur `Hack`, chaque cible d'un saut ne pouvant autre qu'une étiquette statique. Nous pouvons alors chercher à encoder un `switch` par un tableau d'objets `Jack`, vu comme implémentation explicite d'une table de sauts. Pour autant, cette solution ne fonctionne pas, car l'appel de méthode est lui aussi statique. Une solution par défaut, et malgré tout efficace, consiste à réécrire les `switch` sous-forme d'alternatives imbriquées de manière dichotomique. Comme il n'est pas raisonnable d'écrire un tel code à la main, que cela soit en assembleur ou dans un langage objet, on peut envisager de l'engendrer, soit à l'aide de macros à la `cpp`, soit par l'intermédiaire d'un programme, réalisant la transcription en `Jack` d'un autre programme écrit potentiellement dans un autre langage. Nous avons donc implémenté une esquisse de compilateur pour un langage analogue à ILP2, présenté dans le cours *Développement d'un langage de programmation* en M1 STL à Sorbonne Université. Dans un second temps, il apparaît que le langage `Stack!`, cible du compilateur `Jack` exécutable sur la plate-forme `Nand2Tetris` permet l'implantation de schéma de compilation plus fins, car il dispose des instructions de contrôle `label`,`goto` et `goto-if` qui sont plus malléables que les seuls `if` et `while` de `Jack`. Cette expérimentation à aboutit à l'implantation de l'implantation de deux langage de programmation. Le premier, PCF, est un noyau fonctionnelle permettant d'illustrer de façon ludique des techniques de compilation de langages fonctionnels vers `Stack!`, et en particulier la représentation des valeurs fonctionnelles au *runtime*. Le second, `Mini-ML`, est une extension de PCF avec des traits impératifs. Un soin particulier à été apporté à la mise en œuvre d'optimisations de code, afin que l'exécutable `Stack!` engendré à partir d'un source `Mini-ML` soit raisonnablement efficace, du point de vue de la taille du code, de la pression sur la pile et sur la mémoire, et plus globalement du point de vue de la vitesse d'exécution.

Ayant doté `Mini-ML` d'une construction `match` sur constantes, s'apparentant à un `switch`, et qui se compile en alternatives imbriquées dichotomiques suivant la solution évoquée plus haut, nous disposons d'un nouveau langage exécutable sur la plate-forme `Nand2Tetris` apte à l'implantation d'un interprète de bytecode. Nous proposons donc une solution originale : L'implantation en `Mini-ML` de la machine virtuelle OCaml.

Se pose alors la question de la représentation de valeurs OCaml. En effet, `Nand2Tetris` est une plate-forme 16 bits, qui de plus ne dispose pas d'opérateur de décalage de bits. Cela complique l'implémentation d'un bit de marque différenciant entier et pointeur dans le *runtime* de la ZAM. Nous optons pour une approche positiviste : l'espace d'adressage sur la plate-forme `Nand2Tetris` étant réduit à la première moitié de l'espace des entiers positifs représentables sur 16 bits, on choisis, par un codage, de *repousser* les pointeurs dans l'espaces des entiers négatifs inférieurs à -16384. Les valeurs sont alors représentées par elles mêmes, en prenant garde que les opérations arithmétiques n'engendrent pas un pointeur lors d'un débordement !

Nous pouvons finalement envisager sereinement l'implantation de l'interprète de bytecode OCaml et de la bibliothèque d'exécution comprenant un récupérateur automatique de mémoire. L'ensemble est programmé entièrement en `Mini-ML`, compilé en `Stack!`, et donc exécutable sur la plate-forme `Nand2Tetris`.

Cette solution originale que nous avons proposé amène toutes sortes de développements, qui sont en partie réalisée dans le cadre de ce PSTL. Un premier développement consiste à implanter le compilateur `Mini-ML` en `Mini-ML`. Ce programme étant trop gros pour être exécuté sur l'implémentation Java de la machine virtuelle `Stack!` proposé par les auteurs [24], un second développement consiste à programmer notre propre implémentation de la machine virtuelle `Stack!` en vue de lui apporter des améliorations : ajouter des primitives, une arithmétique 32 bits, et accroître la taille de la pile d'appels et du tas. Ces deux développements sont en parties achevés. Par la suite nous pourrions éprouver la stratégie d'auto-amorçage que nous avons proposée en section 7. Cela consisterait à exécuter le compilateur `Mini-ML` sur notre implémentation de la ZAM, pour ainsi compiler le compilateur `Mini-ML`.

Notre travail, faut-il le reconnaître a été très largement centré sur OCaml. D'autres développements bien différents peuvent toutefois être envisagés autour de la plate-forme `Nand2Tetris`. D'une part, on peut modifier l'architecture `Hack`, voire tenter une expérience similaire sur un processeur plus réaliste comme le MIPS32. D'autres part, on peut chercher à implanter d'autres langages de programmation sur la plate-forme `Nand2Tetris`, d'autres bibliothèques, dans d'autres paradigmes, un support pour des Fair Threads [4] par exemple.

4. https://www-sop.inria.fr/mimosa/rp/FairThreads/FTC

Références

[1] Abelson, H., Dybvig, R. K., Haynes, C. T., Rozas, G. J., Adams, N., Friedman, D. P., Kohlbecker, E., Steele, G., Bartley, D. H., Halstead, R., et al. Revised 5 report on the algorithmic language scheme. *Higher-order and symbolic computation 11*, 1 (1998), 7–105.

[2] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. "compilers–principles, techniques and tools", pearson education, 2007, 2016.

[3] Appel, A. W., and MacQueen, D. B. Separate Compilation for Standard ML. *SIGPLAN Not. 29*, 6 (June 1994), 13–23.

[4] Boehm, H.-J., and Weiser, M. Garbage collection in an uncooperative environment. *Software : Practice and Experience 18*, 9 (1988), 807–820.

[5] Cardelli, L. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984), pp. 208–217.

[6] Cardelli, L. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (New York, NY, USA, 1984), LFP '84, Association for Computing Machinery, p. 208–217.

[7] Chailloux, E. *Compilation des Langages Fonctionnels : CeML un Traducteur ML vers C*. PhD thesis, Université de Paris VII, Nov. 1991.

[8] Cheney, C. J. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (1970), 677–678.

[9] Clément, D., Despeyroux, T., Kahn, G., and Despeyroux, J. A simple applicative language : Mini-ml. In *Proceedings of the 1986 ACM conference on LISP and functional programming* (1986), pp. 13–27.

[10] Cousineau, G., Curien, P.-L., and Mauny, M. The categorical abstract machine. In *Conference on Functional Programming Languages and Computer Architecture* (1985), Springer, pp. 50–64.

[11] Doligez, D., and Leroy, X. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1993), POPL '93, Association for Computing Machinery, p. 113–123.

[12] Frenkel, K. A. An interview with Robin Milner. *Communications of the ACM 36*, 1 (1993), 90–97.

[13] Garrigue, J. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming* (2004), Springer, pp. 196–213.

[14] Johnsson, T. Lambda lifting : Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture* (1985), Springer, pp. 190–203.

[15] Krivine, J.-L. A call-by-name lambda-calculus machine. *Higher-order and symbolic computation 20*, 3 (2007), 199–207.

[16] Landin, P. J. The mechanical evaluation of expressions. *The computer journal 6*, 4 (1964), 308–320.

[17] Landin, P. J. The next 700 programming languages. *Commun. ACM 9*, 3 (Mar. 1966), 157–166.

[18] Leroy, X. The ZINC experiment : an economical implementation of the ML language.

[19] Leroy, X. Le système Caml Special Light : modules et compilation efficace en Caml. Research Report RR-2721, INRIA, 1995.

[20] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. The ocaml system release 4.07 : Documentation and user's manual.

[21] Mauny, M., and De Rauglaudre, D. Parsers in ml. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (New York, NY, USA, 1992), LFP '92, Association for Computing Machinery, p. 76–85.

[22] McCarthy, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM 3*, 4 (Apr. 1960), 184–195.

[23] Milner, R. A Proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (New York, NY, USA, 1984), LFP '84, Association for Computing Machinery, p. 184–197.

[24] Nisan, N., and Schocken, S. *The elements of computing systems : building a modern computer from first principles*. MIT press, 2005.

[25] Plotkin, G. D. LCF considered as a programming language. *Theor. Comput. Sci. 5*, 3 (1977), 223–255.

[26] Queinnec, C. *Lisp in small pieces*. Cambridge University Press, 2003.

[27] Schocken, S. Virtual machines : abstraction and implementation. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education* (2009), pp. 203–207.

[28] Schocken, S. Taming complexity in large-scale system projects. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (2012), pp. 409–414.

[29] Schocken, S., Nisan, N., and Armoni, M. A synthesis course in hardware architecture, compilers, and software engineering. *ACM SIGCSE Bulletin 41*, 1 (2009), 443–447.

[30] Serrano, M. *Vers un Compilation Portable* et *Performante des Langages Fonctionnels*. PhD thesis, Université Paris 6, Dec. 1994.

[31] Sussman, G. J., and Steele, G. L. Scheme : An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation 11*, 4 (1998), 405–439.

[32] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *J. of Math 58*, 345-363 (1936), 5.

[33] Varoumas, S. *Modèles de programmation de haut niveau pour microcontrôleurs à faibles ressources*. PhD thesis, Sorbonne Université, 2019.

[34] Vaugon, B., Wang, P., and Chailloux, E. Programming microcontrollers in ocaml : the ocapic project. In *International Symposium on Practical Aspects of Declarative Languages* (2015), Springer, pp. 132–148.

[35] Vouillon, J., and Balat, V. From bytecode to JavaScript : the Js_of_ocaml compiler. *Software : Practice and Experience 44*, 8 (2014), 951–972.

[36] Wells, J. B. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic 98*, 1-3 (1999), 111–156.

[37] Wright, A. Polymorphism for imperative languages without imperative types. Tech. rep., 1993.

[38] Zorn, B. Comparing Mark-and Sweep and Stop-and-Copy Garbage Collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (New York, NY, USA, 1990), LFP '90, Association for Computing Machinery, p. 87–98.

A Tutoriel d'utilisation

A.1 Installation

Opam doit être installé sur votre machine.

Voici les commandes à exécuter dans un terminal :

```
1 opam switch create ocaml-base-compiler
      .4.07.1
2 opam install dune
3 opam install obytelib
4 opam install ocamlclean
```

La version de dune doit être dune 1.11, vous pouvez-aussi essayer de modifier le fichier dune-project avec votre version.

Obytelib 1.5 nécessite ocaml >= 4.07, mais manipule du bytecode ocaml <= 4.07.1

A.2 Compilation de la ZAM

Dans tous les exemples, nom_du_fichier.ml est à remplacer par le nom d'un fichier source OCaml.

- `make zam-miniML MLFILES=`*`nom_du_fichier`*`.ml` : Compile en `MiniML` notre implémentation de la ZAM. Les exécutable sont dans zam/bin, accompagné d'un script Main.tst permettant de configurer le simulateur Nand2Tetris. Pour lancer le programme depuis le simulateur, on ouvrira Main.tst avec file load script, on pourra enlever les animations pour rendre la VM plus rapide : animate > no animation, puis on cliquera sur run (bouton double flèche bleue).
- `make zam-ocaml MLFILES=`*`nom_du_fichier`*`.ml` : compile en OCaml notre implémentation de la ZAM. L'executable est zam/src/ocaml/zam.exe
- `make zam-miniML-run MLFILES=`*`nom_du_fichier`*`.ml` : compile en `Mini-ML` notre implémentation de la ZAM, puis lance le simulateur
- `make zam-ocaml-run MLFILES=`*`nom_du_fichier`*`.ml` : compile en OCaml notre implémentation de la ZAM, puis lance une implémetation Java de la VM `Nand2Tetris` (`VMemulator`).

Pour lancer un programme séparé en plusieurs fichiers, toutes les commandes précédentes peuvent contenir plusieurs noms de fichiers.

Listing 30 – Exemple avec plusieurs fichiers

```
1 make zam-ocaml-run MLFILES=nom_du_fichier1.ml nom_du_fichier2.ml
```

Exemple concret d'utilisation sur le fichier `bench/fact.ml` :

```
1 $ make zam-miniML-run MLFILES=benchs/fact.ml
2 720
```

A.3 Tests unitaires

Il est possible de tester notre implémentation de la ZAM, compilée par `ocamlc`, sur un ensemble de fichiers de tests. Pour cela, il suffit de saisir :

```
1 $ make test
```

A.4 Options de compilation de notre implémentation de la ZAM en `Mini-ML`

Les options du compilateur `Mini-ML` peuvent être ajouté en redéfinissant, dans le Makefile prinicipal, la constante `MINIML-FLAGS`.

Exemple :

```
1 $ make zam-miniML-run MLFILES="benchs/fact.ml" MINIML-FLAGS="-printast"
```

- `-printpast` : Affiche l'AST en syntaxe Caml
- `-printast` : Affiche l'AST simplifié en syntaxe Caml après typage et optimisation
- `-typecheck` : type le programme est abandonne si celui-ci est mal typé
- `-inline` : profondeur d'inlining
- `-noglobalize` : désactive la globalisation des valeurs immutables allouées.
- `-nofolding` : désactive la propagation des constantes
- `-nosmpvar` : désactive la réécriture des variables globales de la forme [let x = constante] en fonction d'arité 0
- `-src` : spécifie où chercher les fichiers sources à compiler
- `-dst` : spécifie le dossier où seront placés les fichiers compilés
- `-stdlib` : chemin vers la bibliothèque d'exécution de `mini-ML`
- `-assert` : embarque les assertions dans le code.

Exemple :

```
1 make zam-miniML-run MLFILES="benchs/fact.ml" MINIML-FLAGS="-printast"
```

A.5 Étapes intermédiaires

- `make miniML` : compile (en OCaml) le compilateur `mini-ml`.
- `make link MLFILES="f1.ml f2.ml ..."` : compile les sources avec `ocamlc`, avec un support à l'édition de lien en cas d'appels de fonctions extérieures.
- `make ocamlclean` : produit l'exécutable `vm/link/byte.out` à partir de l'exécutable `vm/link/a.out`
- `make obytelib` : construit le tableau code `zam/input.ml` à partir de l'exécutable `vm/link/byte.out`.