Rapport Projet STL :
Ribbit sur FPGA avec Eclat

17 mai 2024

Yok Yann HUYNH, Yacine KESSAL, Marc XU

Table des matières

1 Introduction

Dans le cadre de notre première année de Master en Sciences et Technologies du Logiciel, nous avons eu l'opportunité de choisir parmi plusieurs projets de 4 mois, offrant ainsi la possibilité de mettre en pratique nos connaissances théoriques acquises et de développer nos compétences professionnelles.

Les FPGA (Field-Programmable Gate Arrays) sont des circuits numériques configurables par logiciel pour implanter matériellement des calculs sur mesure. Ces circuits sont particulièrement appréciés pour la spécialisation et la parallélisation de certaines tâches [3].

La programmation de FPGA se fait habituellement dans un langage de description de matériel (Hardware Description Language, HDL) mais il se trouve que le langage fonctionnel Eclat [4], inspiré du langage OCaml, peut être compilé vers du code HDL. Eclat est issu des développements autour du langage Macle pour l'accélération de programmes OCaml sur FPGA, qui permet d'avoir des gains significatifs par rapport à une exécution sur PC[6]. Cela permet d'implanter facilement des accélérateurs matériels sur FPGA. Par exemple, Eclat a été utilisé pour implanter la machine virtuelle OCaml et sa bibliothèque d'exécution sur FPGA [9]. Cette implantation d'OCaml est 50 fois plus rapide qu'une implantation existante ciblant un processeur softcore sur FPGA [4].

Lisp est un langage de programmation qui a engendré une famille de langages, parmi lesquels on retrouve Scheme. Scheme se distingue par son système minimaliste, se concentrant sur un petit nombre de constructions syntaxiques et un modèle de données uniforme. C'est un langage fonctionnel qui utilise une notation préfixée, où l'opérateur précède les opérandes.

Le système Ribbit [2] est une implantation légère d'un compilateur Scheme vers une machine virtuelle nommée RVM, et se caractérise par le fait d'être un système compact supportant un interprète Scheme (Read-eval-print-loop, REPL) qui est extensible et dont l'empreinte du code exécutable tient sur 4KB. Ribbit[1] est donc une machine virtuelle minimaliste offrant une gestion de la mémoire et des performances adaptées aux systèmes embarqués[1].

L'objectif principal de ce projet est d'implanter la machine virtuelle Ribbit (RVM) pour qu'elle puisse fonctionner efficacement sur FGPA, notamment en préservant sa conception minimaliste et son empreinte mémoire réduite. Plutôt que d'utiliser un langage de description matériel (HDL) tel que VHDL (Very High Speed Integrated Circuit HDL), ce qui peut être fastidieux et source d'erreurs, nous proposons d'implémenter Ribbit sous la forme d'un programme Eclat. Ce programme Eclat sera ensuite compilé en VHDL à l'aide du compilateur Eclat. Un intérêt de cette approche est la proximité entre Eclat et OCaml : nous pouvons développer et mettre au point notre implantation de Ribbit, d'abord sous la forme d'un programme OCaml, langage avec lequel nous sommes déjà familiers, que l'on pourra ensuite traduire manuellement en Eclat. On veut reproduire l'expérience de développer une VM mais pour Scheme. Au final on souhaite avoir un langage complet sur FGPA.

Pour cela, une présentation détaillée d'Eclat est d'abord nécessaire avant d'aborder le système

Ribbit et nos approches d'implémentations suivies dans la réalisation de notre machine virtuelle Ribbit.

2 Le langage Eclat

Dans cette section, nous présentons le langage Eclat, un langage fonctionnel inspiré d'OCaml, conçu spécifiquement pour la programmation des FPGA.

2.1 Caractéristiques

Eclat repose sur une sémantique synchrone conçu pour la programmation des FPGA, offrant un contrôle précis sur le comportement temporel et le parallélisme dans les applications. Basé sur un λ-calcul simplement typé avec polymorphisme `let`, Eclat permet de décrire des circuits FPGA en tant que fonctions. L'exécution des programmes Eclat est contrôlée par l'horloge globale du FPGA (ce qui évite aux programmeurs d'avoir à calculer le pire temps d'exécution (WCET, Worst Case Execution Time)), avec une fonction principale appelée à chaque front montant d'horloge pour traiter les entrées et produire des sorties instantanément.

Éclat permet aussi la création de circuits séquentiels grâce à une construction de programmation reg f last v qui permet d'initialiser un registre avec une valeur de départ v et mettre à jour à chaque exécution le registre avec la fonction f, puis retourner la nouvelle valeur. La figure 1 présente un exemple de programme Eclat implantant un compteur ainsi qu'une trace d'exécution associée. Cette trace est produite par simulation du code VHDL engendré par le compilateur Eclat, puis simulation dans le logiciel GTKWave, le résultat est donné en Hexadécimal.

```
let compteur() =
    let update(x) = x + 1 in
    reg update last 0;;
```

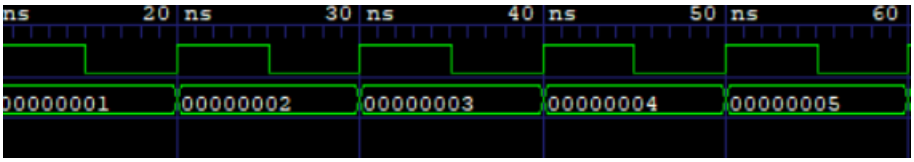

Figure 1 – programme eclat qui incrémente un compteur

Dans cet exemple, la courbe supérieure représente un signal d'horloge alternant régulièrement entre les niveaux haut et bas. Quant à la courbe inférieure, il s'agit du résulat de notre programme (la valeur présente dans le registre), la fonction compteur() étant appelée à chaque front montant de l'horloge incrémente cette valeur à chaque cycle.

Eclat permet la définition de fonctions récursives grâce à la construction let rec . Le compilateur impose que ces fonctions soient récursives terminales pour être implémentables sous la forme d'un circuit (qui ne requièrent ni pile, ni tas).

La construction exec dans le langage Éclat est utilisé pour exécuter des calculs longs c'est-à-dire des calculs qui ne peuvent pas être réalisés en une seule période d'horloge dans des systèmes réactifs synchrones, typiques des applications matérielles programmées sur FPGA.

Dans Éclat, et par extension dans les langages de programmation synchrones, chaque étape de calcul doit se terminer dans un cycle d'horloge fixe pour maintenir la réactivité du système. Cela signifie que la fonction ou le calcul doit être instantané pour respecter le modèle synchrone. Cependant, certaines fonctions, en particulier les fonctions récursives ou les calculs intensifs, ne peuvent pas se terminer instantanément.

exec permet d'intégrer ces fonctions longues dans des environnements réactifs en exécutant une étape de calcul à chaque tick d'horloge. Cela permet à la fonction de progresser graduellement, sans bloquer l'exécution des autres composants réactifs du système. Le constructeur exec rend à chaque tick une valeur formée d'un résultat (qui vaut e0 si le calcul n'est pas terminé) et un booléen rdy qui indique le moment où le calcul se termine.

Les Tableaux en Eclat

Eclat dispose de tableaux, qui constitue une fonctionnalité essentielle pour écrire des algorithmes généralistes. Un exemple d'utilisation de tableaux est l'implantation de la VM OCaml en Eclat [6] dans laquelle le tas et la pile OCaml sont allouées dans un grand tableau Eclat, ces tableaux sont définis au niveau global et peuvent être accédés par toutes les fonctions au sein d'un programme Éclat. Un tableau global en Éclat est stocké dans la mémoire sur puce (on-chip memory), qui est rapide et permet un accès en temps réel, ce qui est crucial pour des applications réactives et en temps réel.

La syntaxe pour définir un tableau global en Éclat commence par le mot-clé `let static`, suivi du nom du tableau, de la taille et de la valeur initiale pour tous ses éléments. Voici un exemple de déclaration d'un tableau global :

```
let static array_name = initial_value^n
```

où `n` représente la taille du tableau, et `initial_value` est la valeur initiale des éléments du tableau. L'accès en lecture s'effectue par la construction `ram[i]` et l'écriture par `ram[i] <- j`, qui s'executent en 2 cycles et 1 cycle respectivement.

Cependant, l'accès aux tableaux globaux en Éclat est séquentiel et les accès parallèles sont interdits par exemple :

```
let f(x) = ram[i] in
let a = f(0) and b = f(1)
in a + b.
```

Cela signifie que plusieurs fonctions ne peuvent pas accéder au même tableau simultanément.

En résumé, les tableaux globaux en Éclat sont essentiels pour la gestion de données complexes et pour l'interaction entre les différentes parties d'un programme qui s'exécute sur un matériel FPGA.

2.2 Exemples

Voici quelques exemples de programmes en Eclat :

```
let fibo(n) =
    let rec aux (n,a,b) =
    if n = 0 then a else aux((n-1),b,(a+b))
    in
    aux(n,0,1);;
```

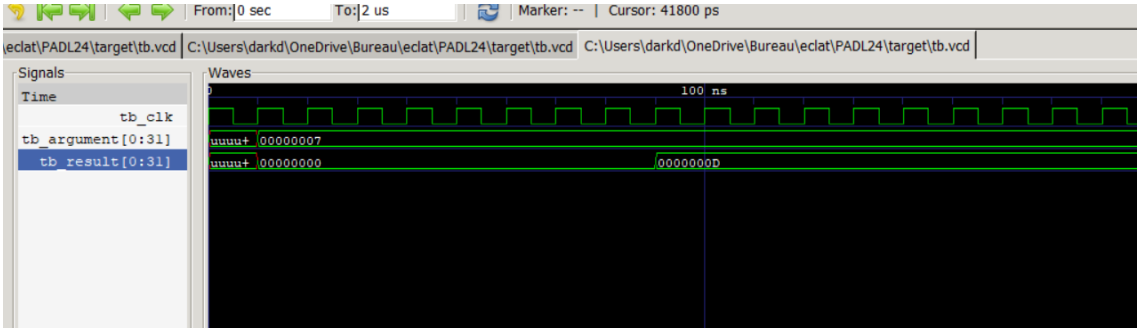

Figure 2 – La fonction Fibonacci en Eclat

```
let rec gcd(a,b) =
  if a < b then gcd(a,b-a)
  else if a > b then gcd(a-b,b)
  else a
```

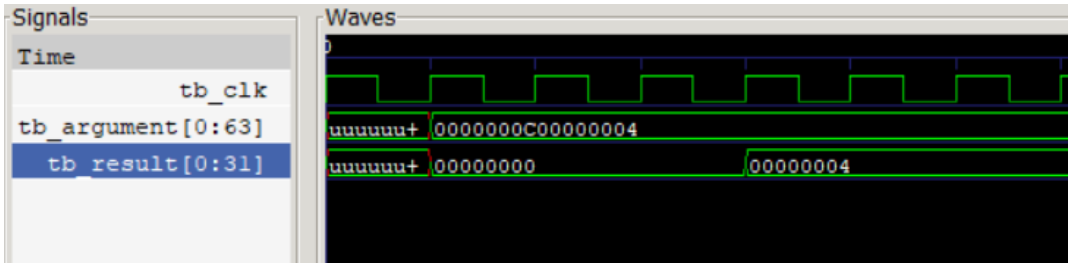

Figure 3 – Fonction GCD en Eclat et sa trace d'éxécution

La figure 2 présente deux éléments : un extrait de code pour calculer une suite de Fibonacci en Éclat, suivi d'un chronogramme montrant la trace d'exécution de cette fonction sur un FPGA.

Le code définit une fonction `fibo(n)` qui calcule le n-ième nombre de la suite de Fibonacci en utilisant une fonction auxiliaire récursive `aux`. La fonction `aux` prend deux arguments `a` et `b` qui correspondent respectivement à deux termes consécutifs de la suite. Si `n` est zéro, `a` est retourné car c'est le résultat souhaité. Si `n` est supérieur à zéro, elle s'appelle récursivement avec `n-1` et les deux prochains termes de la suite.
Le chronogramme en dessous du code représente une simulation du code avec GTKWave. Le signal `tb_clk` correspond à l'horloge qui synchronise l'exécution. Les signaux `tb_argument` et `tb_result` montrent les valeurs des arguments passés à la fonction qui est ici 7 et les résultats produits par celle-ci à chaque cycle d'horloge. On remarquera ici que `fibo(7)` prend 7 cycles de pauses, ce qui est bien égal au nombre d'appels récursifs de la fonction pour `n = 7` et le résultat est renvoyé au dernier appel qui est égal à 13.

2.3 Limites et contraintes

On choisit Eclat parce qu'il permet de compiler vers du VHDL (permet de passer ensuite sur FPGA ce qui est l'objectif final) et qu'il est proposé dans le cadre du projet. Mais le choix d'un langage si spécifique nous limite sur certains aspects. Dans cette partie, on explique les difficultés à prévoir.

Tout d'abord, le langage Eclat contraint fortement le style de programmation pour être implémentable sur circuit FPGA, ce qui peut limiter la manière dont certains algorithmes sont exprimés. De plus, des erreurs de parsing peuvent survenir facilement en raison de détails syntaxiques, tels que l'omission du "end" à la fin d'une structure "match", l'incapacité de faire correspondre deux variables simultanément dans une expression "match", ou encore l'obligation d'ajouter un double point-virgule à la fin de chaque instruction.

De plus, il n'y a pas de structures de données allouées en Eclat ni de gestionnaire d'exceptions et les fonctions récursives doivent être terminales.

En tenant compte de ces limitations, il est essentiel de concevoir des solutions adaptées pour surmonter ces défis lors du développement de notre machine virtuelle, dont on détaillera la réalisation dans la section suivante.

Il reste cependant une forte proximité syntaxique et sémantique entre Eclat et OCaml. il est donc raisonnable de programmer directement dans un sous-ensemble d'OCaml proche d'Eclat, en utilisant les environnements de compilation et de développement OCaml, puis basculer progressivement vers le compilateur Eclat en simulant les descriptions de circuits VHDL engendrées.

3 La machine virtuelle Ribbit (RVM) en OCaml

Dans cette partie, on décrit les élements du système Ribbit en expliquant nos approches suivies par rapport aux spécificités de la machine virtuelle.

3.1 Design de la RVM

La machine virtuelle Ribbit (RVM) est une implémentation compacte conçue pour exécuter des programmes Scheme dans des environnements à ressources limitées [3]. Elle possède une boucle REPL (Read-Eval-Print-Loop) comprenant un parseur, un évaluateur et une bibliothèque d'exécution pour les systèmes embarqués à faibles ressources comme les micro-contrôleurs pouvant tenir sur 4 Ko [3]. La RVM est une machine à pile adaptée en Scheme et est actuellement implémentée dans plusieurs langages comme C ou Python. Le système est conçu pour optimiser les performances et minimiser l'utilisation des ressources, ce qui le rend adapté aux environnements contraignants tels que les microcontrôleurs.
La RVM, conçue pour les environnements à ressources limitées, nécessite une gestion de la mémoire efficace :

— Quatre choses doivent être stockées en mémoire : les objets Scheme, les variables globales Scheme, le bytecode du programme.
— Lorsque le langage hôte ne gère pas la mémoire à l'aide d'un garbage collector (GC), comme en C, la RVM doit inclure un GC dans son code, et comme le langage Eclat n'en dispose pas, il faut donc en implémenter un.

Nous avons à disposition un compilateur AOT (Ahead-Of-Time) [2], un outil de compilation qui convertit le code source d'un programme dans un langage de haut niveau en code machine avant son exécution, offrant ainsi des performances optimales dès le démarrage du programme. C'est un composant clé de la RVM (Ribbit Virtual Machine) qui convertit le code Scheme écrit par le programmeur en une représentation textuelle compacte des instructions RVM (c'est-à-dire, le bytecode Ribit), intégrée dans le code source de la RVM et décompactée lors de l'initialisation de la machine. Il optimise le code en appliquant des techniques telles que la propagation des constantes et l'élimination de code mort pour réduire la taille du programme résultant.

Ribbit utilise une représentation singulière des objets Scheme alloués en mémoire. Les valeurs sont soit des valeurs immédiates, soit des valeurs allouées[2], les seules valeurs immédiates sont les entiers. Les valeurs allouées sont représentées par des triplets de taille fixe appelés "ribs". Tous les types de structures allouées en mémoire sont des ribs. Les ribs sont des structures des triplets qui représentent divers éléments, tels que les objets Scheme, la pile de la RVM et les instructions. Chaque rib peut contenir des valeurs immédiates ou encore des références vers d'autres ribs stockés en mémoire. Les ribs sont utilisés pour décrire la pile, des objets Scheme (voir Figure 4), et les instructions (voir Figure 5) de la RVM.

La RVM maintient deux variables principales : la pile et le PC (pointeur de code) qui fait référence au graphe d'instructions. La pile fait référence au rib situé au sommet de la pile, tandis que le PC fait référence au rib contenant l'instruction en cours d'exécution. Ces deux variables utilisent un champ du rib pour chaîner explicitement les ribs, établissant ainsi une structure similaire à une liste chaînée où le dernier champ n'est pas réellement utilisé. Ces deux variables servent de racines pour le garbage collector de la machine virtuelle (VM) ; ainsi, tout objet qui n'est pas accessible à partir de la pile ou du flux d'instructions peut être libéré par le GC.

Une table des symboles fait partie de l'implémentation de la machine. Cette table contient les noms de tous les symboles et variables globales utilisés dans un certain ordre, formant ainsi la table des symboles du programme. Elle est gérée par la bibliothèque d'exécution. Les symboles ont leur propre structure distincte qui les différencie des autres éléments (ribs).

objet (Rib)	type
(car, cdr, 0)	pair
(code, env, 1)	procédure (primitive ou fermeture)
(valeur, nom, 2)	symbole, valeur globale
(chars, length, 3)	chaîne de caractères
(elems, length, 4)	vecteur
(- , - , 5)	valeur spéciale : #f, #t, ()

Figure 4 – La représentation des objets Scheme avec des ribs[2].

La figure4 ci-dessus présente une représentation des objets Scheme sous forme de ribs. Le dernier champ du rib représente le type de l'objet. Par exemple, les paires `(car, cdr, 0)` sont décrites par deux champs `car` et `cdr` représentant les éléments de la paire.

Les procédures `(code, env, 1)` sont représentées par un champ `code` qui contient soit un entier, soit un autre rib indiquant s'il s'agit d'une procédure primitive ou d'une fermeture. S'il s'agit d'une primitive, le champ `code` correspond simplement à l'entier qui est associé à une primitive en particulier (voir Figure6). Cette procédure est expliquée plus en détails dans la sous-section "Primitives".

Les fermetures se distinguent des procédures primitives par leur structure interne. Le premier champ de `code` est un entier qui indique le nombre d'arguments attendus par la procédure représentée par la fermeture. Le troisième champ du rib est une référence à un autre rib qui correspond à l'instruction RVM située au point d'entrée de la procédure. Le champ `env` des fermetures permet d'accéder à leurs variables libres. Lorsqu'une procédure est définie dans la portée globale (hors de toute fonction ou bloc interne), elle n'a pas de variables libres. Par conséquent, le champ `env` n'est pas utilisé dans ce cas. Le compilateur AOT (Ahead-Of-Time) gère cette situation en créant une procédure constante avec une liste vide dans le champ `env`. En revanche, pour les procédures définies à l'intérieur de blocs imbriqués, le compilateur AOT utilise la primitive `close` pour créer la fermeture correspondante. Cette primitive `close` prend comme seul argument un modèle de

procédure constante[1] dont le champ `code` sera copié dans le champ `code` de la fermeture finale. La primitive `close` enregistre également la variable de pile à ce moment dans le champ `env` de la fermeture (voir Figure 6). Lorsque la procédure est appelée ultérieurement, le champ `env` est utilisé pour accéder aux variables qui étaient accessibles au moment de la création de la fermeture. Cette fonctionnalité est rendue possible grâce à l'ajout d'une référence à la procédure courante sur la pile lors de l'appel de la fonction.

Pour les symboles, le deuxième champ est le rib correspondant au nom du symbole en tant que chaîne de caractères Scheme, et le premier champ est utilisé pour stocker la valeur de la variable globale portant ce nom.

Pour les chaînes de caractères et les vecteurs Scheme, le deuxième champ est un entier indiquant la longueur de la chaîne, et le premier champ est respectivement la liste de caractères et d'éléments.

Enfin, les valeurs spéciales telles que #f, #t et () sont répertoriées avec un indicateur de type (tag) commun, représentant respectivement le false, le true et le nil, et dont les deux premiers champs ne sont pas utilisés et sont initialisés à une valeur pratique dépendante de l'implémentation.

3.2 Graphe d'instructions

Le graphe d'instructions constitue une représentation intermédiaire essentielle à l'interprétation des programmes exécutés par la machine virtuelle Ribbit. Comprendre la structure et la manipulation de ce graphe est fondamental pour optimiser l'exécution des programmes dans notre environnement de machine virtuelle RVM.

Le flux d'instructions est stocké dans un rib, où le dernier champ fait référence au rib de l'instruction suivante (à l'exception de l'instruction `jump` qui n'a pas d'instruction suivante). Les instructions sont exécutées séquentiellement, à partir de l'adresse de départ du programme. Le programme forme un graphe d'instructions. La RVM ne comporte que six types d'instructions, à l'exception de lambda qui est traitée différemment (Figure 5). Comme indiqué dans la figure, le premier champ du rib contient un entier indiquant le type d'instruction (0 = call ou jump, 1 = set, 2 = get, 3 = const, 4 = if). Pour l'instruction `if`, qui dépile le sommet de la pile (Top of stack, TOS) et le compare au rib correspondant à "false" : #f, le deuxième champ est une référence au code vers lequel on saute lorsque TOS $\neq$ #f et le dernier champ est la référence au code vers lequel on saute lorsque TOS = #f. Pour l'instruction `const`, le deuxième champ est une référence à l'objet à empiler sur la pile. Pour les instructions de call, jump, de set and get, le deuxième champ est l'opérande de l'instruction, qui peut être soit un entier non négatif, soit un symbole. Un entier indique un emplacement de la pile par rapport au sommet (c'est-à-dire que 0 est le TOS) et un symbole indique la variable globale avec ce nom.

Instructions (Rib)	action
(0, slot de pile/variable globale, 0)	jump : appel terminal
(0, slot de pile/variable globale, next))	call : appel non terminal
(1, slot de pile/variable global , next)	set : slot de pile/variable globale ← pop()
(2, slot/global , next)	get : push(slot/global)
(3, valeur , next)	const : push(valeur)
(4, then, else)	if : if pop() ≠ #f goto then

Figure 5 – La représentation des six instructions de la RVM avec des ribs[2].

3.3 Primitives

Les primitives représentent les fonctions de base fournies par le système Ribbit et qui font partie de l'environnement d'exécution. Comprendre la nature et les fonctionnalités de ces primitives, ainsi que leur implémentation dans le cadre de notre projet RVM, est crucial pour assurer une intégration fluide et sécurisée avec le reste du système Ribbit.

Primitive	Code	Action
rib	0	z ← pop() ; y ← pop() ; x ← pop() ; r ← rib(x, y, z)
id	1	x ← pop() ; r ← x
arg1	2	y ← pop() ; x ← pop() ; r ← x
arg2	3	y ← pop() ; x ← pop() ; r ← y
close	4	x ← pop() ; r ← rib(x[0], stack, 1)
rib ?	5	x ← pop() ; r ← bool(x is a rib)
field0	6	x ← pop() ; r ← x[0]
field1	7	x ← pop() ; r ← x[1]
field2	8	x ← pop() ; r ← x[2]
field0-set !	9	y ← pop() ; x ← pop() ; x[0] ← y ; r ← y
field1-set !	10	y ← pop() ; x ← pop() ; x[1] ← y ; r ← y
field2-set !	11	y ← pop() ; x ← pop() ; x[2] ← y ; r ← y
eqv ?	12	y ← pop() ; x ← pop() ; r ← bool(x is identical to y)
<	13	y ← pop() ;x ← pop() ;r ← bool(x<y)
+	14	y ← pop() ;x ← pop() ; r ← x + y
-	15	y ← pop() ;x ← pop() ; r ← x - y
*	16	y ← pop() ;x ← pop() ; r ← x * y
quotient	17	y ← pop() ;x ← pop() ; r ← x // y
getchar	18	x ← getchar() ; r ← x
putchar	19	x ← pop() ; putchar(x) ; r ← x
	bool(x)	= #t si x, sinon #f

Figure 6 – Les procédures primitives définies par le RVM. Le résultat résultat de la primitive est r[2].

Une primitive est aussi un rib (Figure 6) dont le premier champ (code) est un entier compris entre 0 et 19, le deuxième champ (env) qui est inutilisé par la primitive et le troisième champ contient l'entier 1, l'indicateur de type des procédures. Lorsqu'une primitive est appelée par une

instruction d'appel, elle retire un certain nombre d'arguments de la pile (*pop*()) et y remet ensuite le résultat de l'opération. Le nombre d'arguments est fixe pour une primitive donnée et la RVM ne vérifie pas s'il y a suffisamment d'arguments sur la pile [2]. Les mêmes opérations sont exécutées lorsqu'une primitive est appelée à travers une instruction jump, mais avant que le résultat ne soit poussé sur la pile, les variables de pile et pc de la RVM sont mises à jour en fonction de la continuation dans le cadre de pile actuel qui contient l'état de ces variables lorsque l'appel a été exécuté. L'instruction jump correspond naturellement à un appel terminale, c'est-à-dire qu'elle appelle une autre fonction comme dernière action qu'elle effectue, sans aucune opération supplémentaire à réaliser après le retour de cette fonction. et toutes les activations de procédure se terminent par un saut. Une procédure qui se termine par un résultat qui n'est un tail-call dans le code source peut renvoyer la valeur en la poussant sur la pile puis en exécutant un saut vers la primitive id.

3.4 Approche suivie

Dans un dépôt GitHub public, plusieurs implémentations de Ribbit sont disponibles, notamment en OCaml. Cependant, la version OCaml existante ne respecte pas strictement les règles des langages fonctionnels, car elle utilise des boucles, ce qui n'est pas typique des paradigmes fonctionnels qui peuvent rendre difficile la traduction en Eclat, un langage fonctionnel pur.

Pour résoudre ce problème, nous avons entrepris de développer une nouvelle version en OCaml qui respecte mieux les contraintes d'Eclat. Cette nouvelle implémentation s'inspire des versions existantes en Scheme et en C du langage Ribbit. En adaptant notre implémentation pour correspondre aux spécificités d'Eclat, nous nous assurons que le langage est compatible avec ses limites et son paradigme fonctionnel pur.

Design de la RVM

À, l'exception des tableaux globaux, Eclat ne possède pas de variables globales modifiables en place tel qu'on le trouve dans d'autres langages de programmation. De plus, Eclat n'a pas de type sommes récursifs. Or une manière naturelle de définir les valeurs Ribbit est un type somme : Une valeur Ribbit serait soit un triplet de valeurs, soit un entier :

```
type t = Rib of t * t * t | Integer of int
```

Nous avons donc développé notre propre structure de mémoire dans Eclat, qui agit essentiellement comme un tableau de triplets. Chaque rib (bloc de mémoire) est représenté comme un élément dans un tableau global, ce qui reflète la mémoire utilisée par notre implémentation de la RVM. Pour modifier un rib, il suffit maintenant de connaître son index dans ce tableau.

Dans la définition des types `word` et `rib`, le type `Triplet` est utilisé pour stocker un entier qui représente l'indice dans le tableau mémoire faisant référence à un autre rib. En d'autres termes,

un rib peut contenir des valeurs immédiates (Constructeur Nil ou Int of int) et des références à d'autres ribs en utilisant le constructeur `Triplet` pour stocker leurs adresses (c'est-à-dire, leur position dans le tableau).

```
type word =
    Nil
  | Int of int
  | Triplet of int
type rib = word * word * word

type ram = rib array

let pc = ref (-1)
let sp = ref size_ram
let hp = ref (-1)
```

Figure 7 – Représentation des valeurs Ribbit dans notre implémentation en OCaml

La clé de cette approche réside dans l'utilisation des indices pour accéder et modifier les ribs stockés dans la mémoire. Nous utilisons deux indices principaux : le sommet de la pile (stack pointer - sp) et le pointeur de code (program counter - pc). L'indice `sp` dans notre implémentation est un pointeur (de type `ref int`) qui représente l'indice vers la pile. Ce pointeur `sp` fait référence à un triplet dans notre structure de données, où le premier champ du triplet représente le sommet de la pile (c'est-à-dire la valeur en haut de la pile), et le deuxième champ du triplet est une référence au reste de la pile, formant ainsi une structure de pile chaînée. tandis que le pointeur de code pointe vers le rib contenant l'instruction en cours d'exécution. En plus de `sp` et `pc`, nous utilisons également un pointeur `hp` qui réfère au pointeur de tas (heap pointer). Ce pointeur `hp` indique la dernière case mémoire occupée dans notre structure de données. Il permet de gérer efficacement l'allocation de mémoire dynamique en marquant la position actuelle dans le tas. Ainsi, lorsqu'une nouvelle allocation de mémoire est nécessaire, elle se fait à partir de la position indiquée par `hp`, puis `hp` est mis à jour pour pointer vers la nouvelle position allouée. Ainsi, pour accéder à un rib spécifique dans la mémoire, il suffit d'utiliser son indice correspondant dans le tableau.

Pour prévoir l'implémentation de la RVM en Eclat, nous avons aussi décidé de rendre toutes nos fonctions récursives terminales. Nous développons un premier prototype de la RVM en OCaml, avec cette representation de la mémoire sous forme de tableau et n'employant uniquement des fonctions récursives terminales, ce qui permettra de porter ce protoype en Eclat dans la suite de notre travail.

Nous avons à disposition un compilateur AOT permettant de générer du bytecode Ribbit à partir de programmes Scheme, pour ensuite l'intégrer dans le code source de la RVM. Illustrons cela par un exemple simple. En compilant le programme Scheme suivant :

```
(putchar (if (pair? (cons 1 2)) 65 66))
```

```
(putchar 10)
```

Nous obtenons le bytecode suivant, que nous embarquons dans le code source de notre implémentation OCaml de Ribbit :

```
bytecode = #epyt-riap,?bir,?ecnatsni,di,?vqe,!tes-1dleif,2dleif,esolc,snoc,?riap,
1gra,rahctup,,,,bir;'lu?m>lvR7'lu?m>lvR6~@lAmml?mDmki#!*#nk`^}'!)Glk!0+l/li$.mbCl
^~Hl^{{!2k!/:nlkl!(:nlkm!+:nlko!1:nlkp!,:nlks!-:nlku!.:nlkv/!':nlkv6{
```

Ce bytecode est formé de deux parties. La première partie (du caractère "#" au caractère ";") correspond aux noms de tous les symboles et variables globales utilisés par le graphe d'instructions, séparés par des virgules. En d'autres termes la table des symboles du programme. La table des symboles garantit que chaque symbole n'est alloué qu'une seule fois, ce qui permet d'économiser de l'espace dans la mémoire du programme.

Table des Symboles

Pour faire fonctionner la récupération de la table des symboles nous avons besoin d'un pointeur `pos` qui fait référence au caractère courant que l'on souhaite étudié dans le bytecode. De plus, nous avons également un pointeur vers la table des symboles `stbl`, similaire à celui utilisé pour la pile (`sp` pour le sommet de pile). La table des symboles est également structurée sous forme de chaîne, où chaque symbole est chaîné de manière similaire aux éléments de la pile. Dans le processus de construction de la table des symboles, nous parcourons le bytecode du programme source, et chaque fois que nous rencontrons une virgule (","), cela signifie que le symbole est terminé. À ce moment-là, nous créons un rib pour stocker le nom du symbole. Si nous rencontrons un point-virgule (";"), cela indique la fin de la liste des symboles. Pour chaque symbole rencontré, deux opérations sont effectuées. Tout d'abord, un nouveau rib de type string est créé pour stocker le nom du symbole. Ce rib contient généralement des informations telles que le nom du symbole lui-même ainsi que sa taille. Ensuite, un rib distinct est créé pour représenter le symbole lui-même. Ce rib contient a comme premier champ #f et en deuxième champ le rib précédemment créé. Une fois ces ribs créés, le symbole est ajouté à la table des symboles de la machine virtuelle. Grâce à cette construction dynamique de la table des symboles, la machine virtuelle peut efficacement représenter et manipuler les symboles du programme en cours d'exécution. La table des symboles est le premier élément alloué, et elle n'est pas modifiée au cours de l'exécution. Elle est placée au début de la mémoire. Une fois la table créée, le graphe d'instructions du programme sera stocké à la suite immédiatement dans la mémoire.

Graphe d'instructions

La deuxième partie du bytecode correspond au code RVM encodé. Nous devons traduire les instructions du format compact utilisé par la machine virtuelle en instructions interprétables. Pour

simplifier la reconstruction de la représentation en rib du graphe d'instructions, la représentation en chaîne contient les instructions codées dans l'ordre inverse. Au fur et à mesure que chaque instruction est décodée, elle est ajoutée au début de la liste accumulée des ribs. Par exemple lorsqu'une instruction `jump` est décodée, le graphe d'instructions actuel est empilé sur une pile et un nouveau graphe d'instructions contenant uniquement l'instruction `jump` est démarré. Lorsqu'une instruction `if` ou `closure const` est décodée, la pile est dépilée pour obtenir l'opérande à mettre dans l'instruction.

À la fin de la construction du graphe d'instruction, notre pointeur pc fait référence à la première instruction à décoder. Il est utilisé pour suivre l'exécution du programme en pointant vers l'instruction à exécuter.

Lorsque le programme est en cours d'exécution, le pointeur de code est mis à jour pour pointer vers l'instruction suivante selon la logique définie par la spécification de la machine virtuelle. Cela permet à la machine virtuelle de parcourir séquentiellement les instructions du bytecode. Le programme se termine lorsqu'il atteint une instruction de sortie ou lorsqu'une condition d'erreur est rencontrée.

Primitives

Le tableau des primitives est constitué de fonctions qui représentent chaque opération primitive du langage. Chaque primitive est conçue pour prendre en charge un nombre spécifique d'arguments qui sont récupérés en extrayant des valeurs de la pile de la machine virtuelle Ribbit à l'aide de l'opération `pop()`.

Pour simplifier la gestion des différents nombres d'arguments attendus par chaque primitive, des fonctions utilitaires (telles que `prim0`, `prim1`, `prim2`, etc), sont définies. Ces fonctions utilitaires prennent en argument la primitive correspondante et s'occupent de dépiler le nombre approprié d'arguments. Par exemple, `prim2` est conçu pour gérer les primitives nécessitant deux arguments : elle extrait deux valeurs de la pile, les passe à la primitive correspondante (par exemple, +), puis pousse le résultat de cette opération sur la pile.

Il convient de noter que l'ordre de dépilement des arguments de la pile est inverse par rapport à l'ordre des arguments attendus par la primitive. Par exemple, dans `prim2`, le premier `pop()` correspond au deuxième argument de la primitive, et le deuxième `pop()` correspond au premier argument.

Chaque entrée du tableau des primitives utilise ces fonctions utilitaires pour gérer efficacement les arguments et simplifier ainsi l'implémentation des différentes primitives du langage Ribbit au sein de l'interpréteur. Cette approche permet d'encapsuler la logique de gestion des arguments et de rendre le code plus clair et modulaire.

Nous avons désormais tous les outils nécessaire pour décoder et exécuter le programme Scheme. Nous avons une fonction qui permet de le faire, elle est responsable de traiter chaque instruction

pas à pas jusqu'à ce que le programme se termine.

Instructions

En fonction de la valeur du premier champ du rib de l'instruction, la fonction effectue une action spécifique.

L'instruction `call` est un peut plus complexe, effectuant plusieurs actions. Tout d'abord, il alloue un rib de continuation à remplir ultérieurement. Ensuite, il extrait de la pile autant de valeurs que le nombre de paramètres attendus par la procédure, et accumule ces valeurs dans une liste dont la dernier élément est le rib de continuation. Par conséquent, la liste contient les valeurs dans l'ordre inverse de celui dans lequel elles étaient sur la pile, la pile étant elle-même dans l'ordre inverse de celui du code source. Le rib de continuation est ensuite initialisé comme suit : la variable de pile de la RVM est stockée dans le premier champ, une référence à la procédure appelée est placée dans le deuxième champ, et la prochaine de l'instruction d'appel est placé dans le troisième champ. Enfin, la liste construite est assignée à la variable de pile de la RVM et le rib de point d'entrée de la procédure est assigné à la variable pc de la RVM. Maintenant que le contrôle a été transféré à la procédure appelée, toutes les informations nécessaires pour accéder aux variables locales et libres, ainsi que pour retourner le contrôle à l'appelant, sont disponibles via la variable de pile de la RVM. Lorsqu'une instruction de `jump` est exécutée, le rib de continuation est trouvé en parcourant la pile jusqu'à ce qu'un rib avec un troisième champ non nul soit trouvé. Une variable locale est accédée en utilisant une instruction `get` dont l'indice entier indique un emplacement avant le rib de continuation. Une variable libre est également accédée en utilisant une instruction `get` qui contient un indice indiquant un emplacement après le rib de continuation. Cela fonctionne parce que le chaînage des ribs de la pile se trouve dans son deuxième champ, et dans le deuxième champ du rib de continuation se trouve une référence à la procédure actuelle, et le deuxième champ de l'objet procédure est le champ `env` contenant une référence à la pile lors de la création de la fermeture.

Une nouvelle instruction spécifique a été ajoutée au répertoire des instructions existantes pour la machine virtuelle Ribbit. Cette instruction, représentée par `(5, 0, 0)`, est conçue pour signaler explicitement la fin du programme. Avant de lancer l'interprétation du graphe d'instructions, cette instruction d'arrêt est préalablement ajoutée à la pile. L'ajout de cette instruction permet de définir de manière claire et explicite le point de terminaison du programme. Lorsque cette instruction est rencontrée pendant l'exécution du programme, elle déclenche la fin de l'exécution et indique à la machine virtuelle Ribbit d'arrêter le traitement du graphe d'instructions.

3.5 Tests unitaires / fonctionnels

L'objectif principal de ces tests est de comparer les résultats produits par notre machine virtuelle Ribbit en OCaml avec ceux de la version en C. Nous comparons les résultats produits par notre implémentation en OCaml avec la trace de la machine en C exécutant le même programme. Cette comparaison nous permet de valider le comportement de notre implémentation. Pour tester notre première implémentation en OCaml, on utilise la base de tests fournie par le dépôt github Ribbit, et en rajoutant quelques tests écris par nos soins.

La méthodologie adoptée consistait à utiliser le conmpilateur AOT de Ribbit [10] pour générer du bytecode à partir des programmes source. Le processus de compilation a été réalisé en dupliquant le programme de la machine virtuelle Ribbit en OCaml, en remplaçant la représentation par défaut du bytecode par celle correspondant au programme compilé. Ensuite, le fichier résultant a été compilé en un exécutable utilisant le compilateur OCaml.

Un script de test a été développé pour évaluer la machine virtuelle Ribbit, elle automatise le processus de compilation, d'exécution et d'analyse des résultats des tests. Ce script parcourt un répertoire spécifié contenant des fichiers de test au format `00-nom.scm`, où `00` représente le numéro de test et `nom` est le nom du test. Pour chaque fichier de test trouvé, le script utilise le compilateur Ribbit (`rsc`) pour générer un fichier OCaml correspondant. Il renomme ensuite ce fichier OCaml pour éviter les avertissements liés aux noms de module incorrects. Ensuite, le script compile le fichier OCaml généré en un exécutable. L'exécutable est ensuite lancé, et la sortie est redirigée vers un fichier de trace dans un répertoire dédié. Les fichiers de trace ainsi générés peuvent être analysés pour comparer les résultats obtenus avec ceux attendus, permettant ainsi de valider le comportement de la machine virtuelle Ribbit. Cette automatisation du processus de test contribue à assurer la fiabilité et la cohérence de l'interprétation des programmes Ribbit, facilitant ainsi le développement et la maintenance du système.

Dans cet exemple de code (Figure 8), nous avons une fonction `fact` qui calcule la factorielle d'un nombre `n` en utilisant une définition récursive.

```
(define fact
  (lambda (n)
    (if (< n 2)
        1
        (* n (fact (- n 1))))))

(write (fact 10))
(putchar 10)

;;;expected:
;;;3628800
```

Figure 8 – Fonction factoriel en Scheme

La fonction `fact` est définie comme une fonction lambda prenant un argument `n`. Elle utilise une expression conditionnelle (`if`) pour vérifier si `n` est inférieur à 2. Si c'est le cas, la fonction retourne `1` (le cas de base pour le factoriel). Sinon, elle retourne le produit de `n` par `fact(n - 1)`, ce qui équivaut à calculer le factoriel récursivement. Le code appelle la fonction `fact` avec l'argument `10` pour calculer le factoriel de 10. Cette expression affiche le résultat du calcul du factoriel de 10 à l'aide de la primitive `(write)`. Après avoir affiché le résultat du calcul, le code utilise `(putchar 10)` pour ajouter un saut de ligne. Ce programme en Scheme démontre l'utilisation des fonctions anonymes (`lambda`), des expressions conditionnelles (`if`), des opérations arithmétiques (`*`), ainsi que l'affichage de résultats à l'aide de primitives. Elle permet aussi de tester toutes les instructions de la machine virtuelle Ribbit, ainsi qu'un nombre important de primitives. La fonction `fact` met en évidence la capacité de la machine virtuelle à gérer les appels récursifs. Elle teste les manipulations de la pile d'appels et la gestion des contextes dans notre implémentation de la RVM. La trace d'exécution correspond à ce que l'on attendait.

```
(define (f a b)
    (let ((x (+ a b)))
        (lambda (z) (* x z))))

(define h (f 3 4))

(putchar (h 8))

;;;expected:
;;;"8"
```

Figure 9 – Fonction factoriel en Scheme

Ce code en Scheme définit une fonction `f`(Figure 9) qui prend deux paramètres `a` et `b`. À l'intérieur de `f`, une variable `x` est calculée comme la somme de `a` et `b`. Ensuite, `f` retourne une fonction anonyme (une fermeture) qui utilise cette valeur `x` pour calculer le produit avec un autre paramètre `z`. En appelant `f` avec les arguments `3` et `4`, on crée une fonction `h` qui multiplie `7` (résultat de `3 + 4`) avec `z`. Lorsqu'on appelle `h` avec `8` (`(h 8)`), le résultat est `56`, qui correspond au caractère ASCII `8`. Ainsi, l'affichage de `(putchar (h 8))` produit le caractère `8`. Cette démonstration met en évidence l'utilisation des fermetures pour encapsuler des variables locales et les rendre accessibles même après la fin de l'exécution de la fonction externe.

Comme exemple réaliste d'application Scheme supportée par notre implémentation de la RVM, nous avons testé un programme Scheme appelant la boucle REPL, embarquant ainsi le compilateur Scheme entier. Le programme `(repl)` en Scheme est un exemple simple qui représente l'appel à la boucle REPL (Read-Eval-Print Loop). Lorsque ce programme est exécuté, il lance la boucle REPL, ce qui permet à l'utilisateur d'entrer des expressions Scheme à évaluer de manière interactive. La boucle REPL lit les expressions, les évalue, affiche les résultats et attend ensuite une

nouvelle entrée de l'utilisateur. Ce type de boucle REPL est couramment utilisé dans les langages de programmation interactifs pour permettre aux développeurs d'explorer dynamiquement le comportement des expressions et des fonctions du langage. La boucle REPL facilite également le prototypage rapide et le débogage interactif en offrant un moyen immédiat d'interagir avec le système de programmation. En rendant la boucle REPL fonctionnelle sur la machine Ribbit en OCaml, nous nous rapprochons de l'objectif final d'implémenter Ribbit sur FPGA en utilisant Eclat pour décrire la machine virtuelle RVM, sa boucle REPL et son compilateur.

Malgré les progrès réalisés, un défi demeure dans l'achèvement du ramasse-miettes (GC) en OCaml, qui n'a pas encore été pleinement implémenté. Notre objectif était d'implémenter un garbage collector (GC) efficace pour gérer la mémoire de manière optimale dans l'environnement Ribbit, notamment en raison des contraintes de mémoire limitée des FPGA. Dans notre implémentation, nous avons mis de ce problème en allouant un zone mémoire relativement grande pour éviter les problèmes potentiels liés à la gestion de la mémoire.

4 La machine virtuelle Ribbit en Eclat

Pour mettre en œuvre la machine virtuelle (VM) en Eclat, on a adopté une méthode systématique consistant à traduire progressivement chaque fonction, une par une. À chaque fois qu'une fonction était traduite, on procédait à une compilation pour vérifier l'exactitude de la traduction. Cette compilation, agissant comme un contrôle de qualité, confirmait que la fonction était correctement implémentée avant de passer à l'instruction suivante. Ce processus itératif fût une tache lente mais garantissait non seulement la fidélité de la traduction mais servait également à détecter toute rupture avec les fonctionnalités déjà implémentées.

En OCaml, la gestion de la RAM est effectuée via des références pour pc, sp, hp (comme dans la figure 7). En contrast, Eclat exige une approche différente pour simuler ces pointeurs, en utilisant des tableaux avec des variables spécifiques comme pc pour le compteur de programme, sp pour le sommet de pile, et hp pour le tas. Cette adaptation à Eclat nécessite une modification profonde de la gestion des pointeurs pour maintenir la logique de la machine virtuelle, garantissant ainsi que chaque opération de base reste intacte et efficace, comme il n'y a pas de référence en Eclat, nous les avons encodés sous la forme de tableaux de une case comme le montre la figure 10 ci-dessous :

2

Version OCaml

```
let pc = ref (-1)
let sp = ref size_ram
let hp = ref (-1)
let stbl = ref (-1)
let pos = ref 0
```

Version Eclat

```
let size_ram = 9000;;
let alloc_limit = size_ram / 2 -1;;

let ram = array_create size_ram;;
let sp = array_create 1;;
let pc = array_create 1;;
let stbl = array_create 1;;
let pos = array_create 1;;
let hp = array_create 1;;

pc.(0) <- -1;;
sp.(0) <- size_ram;;
stbl.(0) <- -1;;
pos.(0) <- 0;;
hp.(0) <- -1;;
```

Figure 10 – Comparaison des définitions de pointeurs en OCaml et Eclat.

En outre, l'adaptation du tableau des primitives a également nécessité une attention particulière. Dans la version OCaml, le tableau de primitives est un tableau direct de fonctions, ce qui facilite l'accès direct aux opérations primitives par un index, exécutant des actions spécifiques sur la machine virtuelle. Cependant, en Eclat, les tableaux de fonctions ne sont pas supportés il a donc fallu encoder les primitives par des entiers, et définir une fonction *apply_prim* qui pour chaque entier (numéro de primitive) appele la bonne primitive.

Cette transition des primitives et la simulation des références par des tableaux en Eclat démontrent bien les défis et les stratégies adoptés pour adapter une structure de machine virtuelle complexe d'un langage à l'autre, chaque étape étant cruciale pour assurer la fidélité fonctionnelle tout en optimisant les performances et la maintenabilité du système dans son nouvel environnement.

une différence importante entres la version OCaml et la version Eclat vient de la representation du bytecode. En OCaml, le bytecode est typiquement manipulé en tant que chaînes de caractères,

ce qui facilite son interprétation et son exécution directe grâce aux fonctions intégrées du langage pour la manipulation des chaînes, en Eclat ne possède pas les mêmes capacités intégrées pour la manipulation de chaînes, ce qui a nécessité une adaptation significative. Pour contourner cette limitation, nous avons représenté le bytecode par un tableau Eclat. Nous devons donc pour cela développer un programme OCaml qui réalise cette étape de traduction du bytecode Ribbit en tableau Eclat.

La transition de notre machine virtuelle d'OCaml vers Eclat a révélé des variations syntaxiques nécessitant des ajustements précis. En particulier, Eclat requiert des séparateurs (`;;`) à la fin de chaque définition de fonctions.

Comme il n'y a pas d'exceptions en Eclat, nous les avons simulées en définissant une fonction `fatal_error`, de type (string -> 'a) qui boucle infiniment lorsqu'elle est appelée.

Ces modifications, bien que mineures, sont importantes pour pouvoir compiler la machine virtuelle en Eclat.

Pour conclure, la transition de la machine virtuelle d'OCaml à Eclat a été achevée avec succès du point de vue de la traduction du code et de l'adaptation syntaxique. Tous les ajustements nécessaires à l'intégration du code dans l'environnement d'Eclat ont été réalisés sans rencontrer d'erreurs de syntaxe ou de compilation, confirmant l'efficacité de notre démarche méthodique. Cependant, il est important de souligner que, malgré ces avancées et nos efforts pour respecter scrupuleusement la syntaxe d'Eclat, ces adaptations ne sont pas suffisantes pour assurer le fonctionnement de la machine virtuelle en Eclat, alors qu'elle fonctionne parfaitement en OCaml ce n'est pas le cas de la version Eclat qui n'arrive pas à s'exécuter. Ce constat met en lumière que la simple traduction du code vers un nouveau langage ne garantit pas nécessairement la performance opérationnelle de la machine virtuelle dans son nouvel environnement.

5 Conclusion

Ce projet de développement des machines virtuelles Ribbit en OCaml et en Eclat a été à la fois exigeant et enrichissant, marqué par une série de défis significatifs qui ont testé notre compréhension et notre capacité d'adaptation. Tout d'abord, l'apprentissage d'Eclat, un langage que nous ne connaissions pas au début de ce projet, a représenté un premier défi notable.

De plus, la nécessité de concevoir une version de la machine virtuelle Ribbit en OCaml qui puisse être traduite efficacement en Eclat a imposé des contraintes strictes sur notre implémentation. Nous avons dû plusieurs fois revoir notre approche de la VM en OCaml, afin qu'elle se conforme aux caractéristiques spécifiques d'Eclat, notamment en ce qui concerne la gestion de la mémoire et la manipulation des types de données.

Ce projet nous a permis de mettre en pratique des concepts théoriques avancés de programmation. Notre travail sur la machine virtuelle Ribbit en OCaml a abouti à une version fonctionnelle qui consolide notre compréhension des machines virtuelles. Le développement de notre machine virtuelle Ribbit en Eclat progresse bien, malgré quelques défis à cause des limites et contraintes du langage. Jusqu'à présent, nous avons réussi à mettre en œuvre une version fonctionnelle en OCaml et avons fait d'importants progrès sur la version en Eclat, bien que cette dernière n'ait pas encore été testée sur FPGA, ni sur GHDL (G Hardware Design Language).

Les efforts déployés pour respecter les contraintes de la programmation en Eclat, notamment en termes de gestion de la mémoire et d'intégration des primitives nécessaires à la manipulation des données et des instructions, ont été particulièrement instructifs. Notre implémentation de la machine virtuelle Ribbit en OCaml permet l'exécution de programmes Scheme réalistes. Parmi les exemples concrets testés figurent le REPL, mais également des fonctionnalités avancées telles que les fermetures dans un environnement non vide, les fonctions avec des paramètres optionnels et les fonctions variadiques.

Pour les phases suivantes éventuelles du projet, il faudrait finaliser la traduction et l'optimisation du code Eclat, avant de procéder aux tests sur FPGA. Ces tests sont cruciaux pour évaluer la performance de notre système dans un environnement réel et pour effectuer les ajustements nécessaires à l'amélioration de la fiabilité et de l'efficacité du système. De plus, il est impératif d'implémenter un gestionnaire de mémoire afin de résoudre le problème actuel qui survient lorsque le tas est plein, entraînant ainsi l'interruption de l'exécution avec une erreur. Pour ce faire, une approche envisageable serait de nous inspirer du Garbage Collector (GC) stop&copy utilisé dans la machine virtuelle OCaml, développée en Eclat [9].

Références

[1] Samuel Yvon, Marc Feeley : A Small Scheme VM, Compiler, and REPL in 4K.

[2] Leonard Oest O'Leary, Marc Feeley : A Compact and Extensible Portable Scheme VM. In MoreVMs Workshop (MOREVMS@PROGRAMMING'23), 2023

[3] https ://fr.wikipedia.org/wiki/Circuit_logique_programmable

[4] Loïc Sylvestre, Emmanuel Chailloux, Jocelyn Sérot : "Work-in-Progress : mixing computation and interaction on FPGA", International Conference on Embedded Software (EMSOFT 2023), 2023.

[5] John Hennessy, David Patterson : "Computer Architecture : A Quantitative Approach"

[6] Loïc Sylvestre, Emmanuel Chailloux, Jocelyn Sérot : Accelerating OCaml programs on FPGA. International Journal of Parallel Programming, 51(2), 186-207.

[7] https ://wiki.c2.com/ ?LispSchemeDifferences

[8] https ://github.com/udem-dlteam/ribbit/tree/main

[9] Loïc Sylvestre, Jocelyn Sérot, Emmanuel Chailloux : Hardware implementation of OCaml using a synchronous functional language. In. 26th International Symposium on Practical Aspects of Declarative Languages (PADL'24), 2024.

[10] `https://github.com/udem-dlteam/ribbit/tree/main`