M1 Informatique - Projet Compilation Avancée

RAPPORT

Mini-ZAM : Interprète de bytecode fonctionnel

Étudiants :
Suxue Li
Julien Xavier

25 mars 2019

Table des matières

1 Introduction

La ZAM (ZINC Abstract Machine) est la machine virtuelle du langage Ocaml. C'est une machine à pile, qui interprète du bytecode Ocaml contenant 149 instructions différentes. Ce projet a pour but principal de réaliser une mini-ZAM, c'est à dire un tel interprète pour un langage ML restreint au noyau fonctionnel, ce qui donne environ plus d'une dizaine d'instructions de base à prendre en compte. Puis la mini-ZAM est étendue par quelques traits de fonctionnalités en plus, comme l'optimisation, la gestion des exceptions et l'usage de structures de données récursives potentiellement mutables. Nous avons donc implémenté au total une vingtaine d'instruction pour cette mini machine virtuelle.
Nous avons fait le choix d'implémenter notre mini-ZAM en Ocaml.
Ce rapport présente d'abord la structure générale du projet, puis nos choix d'implémentation, enfin quelques nouveaux jeux de test.

2 Structure générale du projet

2.1 parser.ml

Tout d'abord, pour interpréter les fichiers bytecode fournis, nous avons réalisé un petit programme de "parsing" dans le fichier *parser.ml*, ce fichier contient en outre des fonctions permettant d'afficher le programme retenu en mémoire. L'étape de "parsing" permet de lire le fichier bytecode donné en entrée, puis de stocker correctement le programme en mémoire.

2.2 minizam.ml

Le code de la mini-ZAM à proprement dit est dans le fichier *minizam.ml*. C'est dans ce fichier là que nous avons les valeurs, les registres, et les instructions de la machine. Il contient donc une partie où l'on définit le type *mlvalue*, une partie avec les définitions des différents registres, et une dernière grosse partie où toutes les instructions sont implémentées, avec quelques petites fonctions auxiliaires engendrées. La passe appterm est également contenue dans ce fichier.

2.3 main_direct.ml & main_trace.ml

Ensuite, pour exécuter un programme bytecode sur la machine, nous avons écrit deux versions dans deux fichiers distincts : *main_direct.ml*, et *main_trace.ml*. Le premier permet d'obtenir simplement une sortie standard et un valeur de retour, tandis que le deuxième affiche l'état des registres de la mini-ZAM après chaque instruction évaluée, la trace de l'exécution "corrompt" l'affichage réel du programme, par contre la valeur de retour sera toujours affichée à la fin.

2.4 makefile & readme.md

Nous avons également écrit un *makefile* contenant plusieurs cibles, permettant de compiler les différents fichiers, pour créer des exécutables. Vous trouverez plus de précision concernant l'utilisation du makefile dans le fichier *readme.md* qui vous décrit plus en détail les différentes cibles.

3 Choix d'implémentation

3.1 Valeurs

La mini-ZAM manipule des valeurs d'un type *mlvalue* que nous avons choisi de définir ainsi :

```
type mlvalue = Entier of int
        | Fermeture of int * mlvalue
        | Env of mlvalue list
        | Bloc of mlvalue array
```

Le constructeur Entier permet de construire des valeurs représentant les valeurs immédiates entières, mais également les booléens, et ().

Le constructeur Fermeture prend en argument un entier et une valeur de type *mlvalue* construite avec Env, pour représenter respectivement le pointeur de code ainsi que l'environnement d'exécution de la fermeture.

Le constructeur Env prend une liste de *mlvalue* constituant l'environnement à représenter. Env n'existait pas au début du projet, il a été introduit lors du développement. Nous n'avions pas de constructeur pour l'environnement, il était représenté par une simple liste de *mlvalue*, mais cela n'a pas fonctionné.
Un premier problème a été rencontré lors de l'implémentation de l'instruction APPLY, lorsque qu'un environnement vide devait être empilé dans la pile (*mlvalue list*). La concaténation d'une liste vide avec une autre liste faisait disparaitre la liste vide ce qui faussait l'état de la pile. Pour garder cet environnement dans la pile, nous avons donc pensé à ajouter un constructeur constant None dans la définition du type mlvalue. Une liste avec None a l'intérieur représentait donc un environnement vide. Quelques premiers tests qui ne manipulaient pas l'environnement fonctionnaient avec cette version mais un deuxième problème s'est présenté assez rapidement. Nous pensions de l'environnement comme étant un tout, c'est-à-dire une valeur, et que lorsqu'on l'empilait dans la pile parmi d'autres éléments, cet environnement serait gardé comme un tout. Or, nous n'avions pas pris en compte le fait que la concaténation aplatissait les listes, la liste représentant l'environnement n'était plus une liste, mais juste une suite de mlvalue parmi d'autres dans la pile. Il fallait donc trouver un moyen d'envelopper l'environnement, pour permettre de la stocker dans la pile. Etant donné que la pile ne prend que des mlvalue, il a été évident d'ajouter dans le type un constructeur Env pour envelopper la liste de mlvalue de l'environnement.

Enfin le constructeur Bloc a été ajouté pour l'extension sur les blocs. il prend un *mlvalue array* en argument. Nous l'avions d'abord implémenté avec une liste, mais les instructions SETFIELD, SETVECTITEM et ASSIGN nous ont très vite arrêtés et nous a contraints à changer notre implémentation pour l'array compte tenu du caractère mutable des blocs et des effets de bord que cela implique.

3.2 Registres

La mini-ZAM contient au total 7 registres, ils sont définis comme variables globales dans *minizam.ml*.

```
let prog : triplet list ref = ref (parse Sys.argv.(1))
let stack : mlvalue list ref = ref []
let env : mlvalue ref = ref (Env([]))
let pc = ref 0
let accu = ref (Entier 0)
let extra_args = ref 0
let trap_sp = ref (-1)
```

En Ocaml, il n'y a pas de déclaration de variable, lorsqu'on introduit une variable, on doit lui affecter directement une valeur pour que le compilateur infère le type. Pour que la compilation se passe correctement, nous devons donc lui préciser le type des références. Pour le registre stack par exemple, on l'initialisait par une ref sur une liste vide, le compilateur inférait le type *'_weak1 list ref*, ce qui n'était pas bon. C'est pourquoi nous avons indiqué clairement le type des variables telles que prog, stack et env.

Le registre prog permet de stocker le programme à exécuter. Nous avons choisi de représenter chaque instruction par un enregistrement. Cela prend moins de place en mémoire par rapport à un n-uplet. De plus on peut manipuler les champs directement par leur nom, ce qui est rapide et plus facile. Nous avons donc déclaré le type de cet enregistrement dans le fichier *parser.ml* sous le nom de *triplet*. Prog est une liste de triplets. Un triplet est composé d'un champs *label*, *instr* et *args*. Le label est de type *string option* car il peut ne pas y avoir de label. L'instruction est de type *string*, juste pour contenir le nom de celui-ci. Et enfin, les arguments sont stockés dans une liste de string, une liste vide signifie qu'il n'y a pas d'argument dans l'instruction. Un programme est "parsé", puis stocké dans prog, on peut ensuite passer la passe dessus, qui a la possibilité de modifier des instructions, c'est pourquoi prog est une référence.

Le registre stack est une référence sur une liste de mlvalue, la pile doit être modifiable. Il existe un module Stack en Ocaml, mais nous avons préféré utiliser le module List. Les deux modules présentent chacun ses avantages et inconvénients, mais les fonctions de List répondaient mieux à nos besoins, et le seul inconvénient qu'il présentait était le manque de la fonction pop. Nous l'avons donc implémenté nous même dans avec la fonction *depile n* qui dépile directement de la stack n éléments, et retourne ces n mlvalue dans une liste.

Grâce à l'introduction du constructeur Env, le registre env est donc simplement une référence vers une mlvalue, ce qui nous facilite beaucoup de chose, mais cela induit une utilisation intense du motif de filtrage, pour récupérer la valeur d'un environnement.
Le registre accu est une référence sur une valeur de type mlvalue, il est initialisé au début à Entier(0).
Enfin, les registres restant : pc, extra_args, et trap_sp sont de simples références sur un entier.

3.3 Instructions

Au niveau de l'implémentation des instructions, nous n'avons eu spécialement de soucis. Comme vous pourrez le constater dans le code source, nous nous sommes beaucoup appuyés sur le motif de filtrage étant donné les différents constructeurs pour le type mlvalue. L'implémentation des instructions nous a permit de détecter nos erreurs concernant les choix pour le type de mlvalue et des registres.

4 Optimisation APPTERM : comparaison de la pile

Pour les fichiers contenant la suite d'instructions APPLY n ; RETURN m-n, nous avons comparé la taille maximum de la pile sans et avec la passe qui remplace cette suite par l'instruction APPTERM n,m. On a notamment fait une version du test *fun_ appterm.txt* en remettant la suite APPLY et RETURN au lieu des APPTERM, le fichier se trouve dans le dossier jeu_de_test. Voici le tableau qui présente les différentes tailles maximales de la pile :

Fichier test	**Taille de pile SANS passe**	**Taille de pile AVEC passe**
appterm/facto_tailrec.txt	112	8
jeu_de_test/fun_appterm.txt	948	8
block_values/insertion_sort.txt	34	30
block_values/liste_iter	50	19
n-ary_funs/grab2.txt	12	8
n-ary_funs/grab4.txt	11	8
unary_funs/fun4-nooptim.txt	13	7

TABLE 1 – Taille maximale de la pile sans et avec la passe appterm selon différents tests.

On remarque effectivement une différence, particulièrement pour les tests dans appterm, cette différence est flagrante. En fait, plus il y a d'appels récursives avec APPLY RETURN, plus la taille de la pile augmente si on n'utilise pas l'instruction APPTERM à la place.

5 Nouveaux jeux d'essai

Dans le dossier jeu_de_test, nous avons écrit 3 nouveaux tests (sans compter le fun_appterm.txt) pour la mini-ZAM. Vous y trouvez un fichier *.ml* ainsi qu'un fichier *.txt* pour chaque test.
Le test *op_ arithm* teste l'enchainement des différentes opérations arithmétiques. *exception* teste une exception non levée. Et enfin le dernier, *map* teste les blocs avec de la récursivité.

6 Conclusion

L'implémentation de la mini-ZAM nous a permit de comprendre le fonctionnement d'un interprète bytecode fonctionnel, notamment au niveau du fonctionnement interne de la machine. Les sessions de débuggage ont été enrichissantes, dans le sens où l'on devait simuler son évolution lorsque nous avions des erreurs. De plus, faire ce projet en Ocaml nous avons apprit à mieux connaître ce langage, tout le coté fonctionnel typé static a été très convaincant au niveau de l'efficacité.