Compilation Avancée

Rapport de fin de projet

Mini-Zam : Interprète de byte code fonctionnel

BAH Thierno

ZHENG Pascal

Introduction

ZAM est la machine virtuelle du langage OCaml, c'est une machine à pile qui peut être vu comme une machine de Krivine avec une stratégie d'évaluation par appel par valeur.

Pour exécuter un programme OCaml, la ZAM a à sa disposition 149 instructions différentes qui modifie son état interne qui de modification en modification conduit à l'exécution du programme.

Dans ce projet notre objectif a été de réaliser une machine semblable mais avec beaucoup moins d'opérations que la ZAM.

Cette "Mini-Zam" aura moins de 20 instructions et va être capable d'interpréter un bytecode pour un langage ML restreint, on va donc pouvoir interpréter de petit programme fonctionnel qui réalise des opérations comme les calculs arithmétiques simples, les conditions, les branchements, les appels de fonctions à plusieurs arguments ou encore les appels de fonctions récursives.

Les valeurs manipulées par notre machine seront d'un type que nous nommerons "*MLValue*".

Le bytecode représentant notre programme est sous la forme de fichier texte dans lequel chaque ligne représente une instruction à exécuter. Les instructions peuvent avoir des arguments et il y a certaines instructions identifiées par des "*labels*".

Nous avons choisi comme langage de programmation le Python car c'est le langage qui nous attirait le plus et que nous avions déjà une idée de comment nous voulions réaliser notre machine, nous voulions utiliser la façon de gérer les classes, la facilité de lire les fichiers ainsi que le typage dynamique de python.

Dans ce projet nous avons deux types de classes une qui va représenter les *MLvalue* et une autre qui va représenter notre machine avec ses registres et toutes ses instructions (les opérations de la classe machine).

Dans ce rapport nous allons d'abord voir dans un premier temps comment nous avons décidé de représenter les valeurs que l'on manipule (les *MLvalue*) puis nous allons voir l'implémentation de la machine virtuelle et enfin des exemples d'exécution sur des jeux d'essai.

Les MLValues

Les *MLValue* qui vont être manipulés par notre machine sont représentés par des classes Python.

Nous avons une classe *MLvalue* abstraite qui contient la représentation de *MLTrue*, *MLFalse* et *MLUnit*.

Nous avons ensuite des classes qui héritent de cette dernière et qui vont être les valeurs que nous allons manipuler : *MLInt* qui représente les valeur entières, *MLClosure* qui représente une fermeture (un couple pc, environnement) et *MLBlock* qui va représenter un type bloc qui nous permettra de manipuler des collections comme les listes ou encore les tableaux.

Nous avons choisi de faire de cette façon car il est plus facile pour plus tard d'étendre MLValue pour représenter d'autres types, il suffit de créer une classe qui hérite de MLValue et d'ajouter ensuite dans la machine les instructions capables de les manipuler.

Structure de la MiniZam

1 - Etat interne de la machine virtuelle

La classe "MaxiZam" est la classe qui représente la machine, grâce à cette classe nous allons pouvoir lancer une machine sur un fichier texte qui contient le bytecode.

Comme attributs de cette classe il y a tout les registres dont est composée la machine virtuelle, on y retrouve :

- **prog** : une liste qui va nous servir à stocker le programme sous forme de liste de liste de *string*, chaque élément *i* de prog correspond à la *i-eme* ligne du programme. Donc prog[0] nous donne une liste de string qui représente les mots de la ligne 0 séparés par des espaces, on pourra ainsi reconnaître les labels, les instructions ou encore les arguments et cela très facilement.
- **stack** : une liste qui va représenter la pile de notre machine, on va pouvoir y mettre les paramètres des fonctions, des pointeurs de code, des fermetures etc… Cette **stack** est une FIFO et est vide en début et à la toute fin d'un programme.
- **env** : une liste qui va représenter l'environnement de la fermeture courante, **env** est vide à la toute fin d'un programme.
- **pc** : un entier qui représente le pointeur de code.

- **accu** : une variable qui va nous permettre de stocker des valeurs intermédiaires.
- **extra_args** : un entier qui représente le nombre d'argument restants à appliquer à une fonction, ce registre est utile lors de l'application des fonctions *n-aire*.
- **trap_sp** : un entier qui nous sera utile lors de la gestion des exceptions, cet entier est chargé de représenter la position du rattrapeur d'exceptions situé plus haut dans la **stack**. On initialise **trap_sp** à une valeur comme -9999 pour indiquer qu'il n'y a pas encore de rattrapeur qui est mis en place.

2 - Initialisation et instructions de la machine virtuelle

Lors de l'initialisation de notre machine nous allons remplir la liste **prog** avec le programme que nous voulons interpréter, pour effectuer une opération nous utilisons une opération *treat* qui lit une ligne et lance l'opération *MySwitch* qui selon le nom de l'instruction contenu dans la ligne va lancer la bonne opération pour exécuter la bonne instruction et modifier les registres notamment le pc pour pouvoir *treat* la prochaine instruction.

Il y a aussi une opération *position* utilitaire qui permet de renvoyer le **pc** qui correspond à un *label* donné.

Grâce à ses opérations nous allons faire progresser le **pc** et lancer les instructions du fichier dans le bon ordre pour exécuter le programme que le bytecode décrit.

Les instructions de la machine virtuelle sont pour nous les opérations de la classe "MaxiZam" qui modifient l'état des registres et les laissent dans un état cohérent en respectant la spécification donnée.

3 - Exemple d'exécution

le fichier test : "tests/unary_funs/fun2.txt"

```
bytecode :
        CONST 3
        PUSH
        CONST 2
        PUSH
        ACC 0
        PUSH
        ACC 2
        PRIM +
        POP
        POP
        STOP
code ml :
        (fun x -> (fun y -> x + y) 2) 3
```

```
affichage de l'execution :
        DEBUT DE L'EXECUTION
        pc : 0 accu :  env : [] stack : [] extra_args : 0
        --------['CONST', '3']----------
        pc : 1 accu : 3 env : [] stack : [] extra_args : 0
        --------['PUSH']----------
        pc : 2 accu : 3 env : [] stack : [3] extra_args : 0
        --------['CONST', '2']----------
        pc : 3 accu : 2 env : [] stack : [3] extra_args : 0
        --------['PUSH']----------
        pc : 4 accu : 2 env : [] stack : [2, 3] extra_args : 0
        --------['ACC', '0']----------
        pc : 5 accu : 2 env : [] stack : [2, 3] extra_args : 0
        --------['PUSH']----------
        pc : 6 accu : 2 env : [] stack : [2, 2, 3] extra_args : 0
        --------['ACC', '2']----------
        pc : 7 accu : 3 env : [] stack : [2, 2, 3] extra_args : 0
        --------['PRIM', '+']----------
        pc : 8 accu : 5 env : [] stack : [2, 3] extra_args : 0
        --------['POP']----------
        pc : 9 accu : 5 env : [] stack : [3] extra_args : 0
        --------['POP']----------
        pc : 10 accu : 5 env : [] stack : [] extra_args : 0
        --------['STOP']----------
        FIN D'EXECUTION
        VALEUR CALCULEE : 5
```

Optimisations et nouvelles fonctionnalités

Nous avons implémenter les trois optimisations suivantes :

1 - Appels terminaux

Cette optimisation ajoute une instruction **APPTERM n,m** qui va remplacer tous les couples **APPLY n + RETURN m-n** .

Cette optimisation est là car il arrive souvent qu'un appel de fonction soit la dernière chose qui est réalisé dans la fonction appelante, nous avons donc souvent un **APPLY** suivi d'un **RETURN** et il n'est pas nécessaire et coûteux de sauvegarder

le contexte de l'appelant dans la pile et le restaurer après juste pour sortir de la fonction après.

Avec cette optimisation nous avons ajouté une autre fonction utilitaire *apptermOpti* qui va transformer le bytecode en parcourant le registre **prog** et en transformant tous ces couples **APPLY n + RETURN m-n** en **APPTERM n,m.**

APPTERM n,m change l'état interne de la pile et applique les effets des instructions **APPLY** et **RETURN** consécutive.

2 - Blocs de valeurs

Dans cette optimisation nous avons ajouté un nouveau type de valeur : les **bloc.** Les blocs représentent des collections de *MLvalue* ici il va s'agir de collections d'entiers, grâce aux blocs nous pouvons représenter les listes et les tableaux dans notre machine.

Pour manipuler ce nouveau type de valeur nous avons ajouté plusieurs instructions dans notre machine.

3 - Gestion des exceptions

Avec cette optimisation nous avons ajouter le contrôle d'exception à notre machine. Nous avons ajouté 3 nouvelles opérations qui vont nous servir afin d'implémenter un *try catch*.

Jeux d’essais

Nous voulions faire un test dans lequel nous utilisions les blocs et les exceptions mais nous n'avons malheureusement pas réussi à traduire nos idées dans le bytecode pour le faire interpréter par notre machine.

Test voulu version Ocaml :

```
exception OutExp
let L = 1::2::[];;

let safe_acces n =
    if n > 2 the raise OutExp
    else List.nth L n

let _= safe_acces 2
```

Nous pouvions ainsi tester les accès au bloc que nous avons crée en attrapant des exceptions.

Notre machine fonctionne tout de même sur toute la batterie de test que nous avons dans le dossier tests.

Conclusion

Pour implémenter une machine virtuelle il faut comprendre son fonctionnement et les modifications des registres, comprendre les opérations et faire en sorte que ses opérations assurent la cohérence interne de la machine.

Pour l'implémentation de cette machine un langage de haut niveau comme Python nous a permis de l'implémenter relativement “facilement” en utilisant des structures comme les listes et les classes.

Les listes en Python sont assez gourmandes en mémoire pour ce que nous faisons il serait peut-être intéressant d'utiliser un langage de bas niveau pour avoir un meilleur contrôle des opérations et de la mémoire sur laquelle cells-ci agissent.

La machine virtuelle en Python est fonctionnelle mais pourrait facilement être amélioré en utilisant un langage plus proche de la machine.