Implémentation de la Machine Virtuelle

Katia AMICHI(3603567) nadir BELAROUCI(3704056)

Mars 2019

Table des matières

1 Introduction

Le but du projet est l'implémentation de la machine virtuelle du langage OCaml, nommée ZAM, une machine à pile à la SECD, optimisée pour les applications multi-arguments. Ce système repose sur une machine virtuelle, écrite dans le langage de notre choix (ici le Python) capable de lire les instructions des programmes et de les exécuter.

Dans un premier temps, les instructions seront abordés, puis la façon dont les problèmes ont été résolus seront vus dans une seconde partie.

2 Caractéristiques

La machine virtuelle universelle (ZAM) interprète de bytecode pour un langage ML en manipulant essentiellement des mlvalue et en gérant des mots de 32 bits, reposant sur plusieurs facteurs :

— Sept registres qui représentera l'état de la machine virtuelle à chaque exécution.
— Un pointeur de code pc (program counter).
— Instructions de branchement.
— Instruction arithmétique sur les mlvalue.
— Appels et retours de fonctions.
— Manipulation des blocs.
— Valeurs fonctionnelles (fermetures) et environnements.

3 Implémentation

3.1 Choix du langage

Le projet est implémenté avec python 3.7. Nous choisissons python car il n'est pas verbose, lisible et surtout parce qu'il est dynamique, comme nous le verrons dans la section suivante, le type des arguments de chaque instruction diffère d'une instruction à une autre et il serait difficile de les représenter en utilisant d'autres langages tels que java ou c. python a une très bonne bibliothèque compacte et c'est une autre raison pour laquelle nous l'avons utilisée, en particulier le paquetage "re" et la facilité de manipuler les liste, le surcharge des opérateurs, etc.

3.2 Arborescence du code source

(**/src**) :

— *ocamlzam.py* le ficher qui permet de charger le code à interpréter et de lancer l'exécution des différents instructions.
— **/minizam**
  — *vm.py* contient les fonctions principales de la machine virtuelle.
  — *instructions.py* contient les differentes nstructions de la machine textbfMINIZAM.
  — *mlvalue.py* correspondant à la class de type de valeur manipuler par la machine.
  — *test_instructions.py* contient les tests unitaires pour chaque instructions.

Répertoire test (**/tests**) :

— Qui contient les différents fichiers tests fourni ainsi que les jeu de tests rajouter

3.3 MiniZamVM

La machine virtuelle est un ensemble de registres, une pile et un tableau contenant le code. Le code est représenté sous la forme d'un tableau de type LineInstruction. Chaque instance de LineInstruction a une étiquette (facultatif), une commande (obligatoire) et ses arguments (facultatif).

Grâce au paquetage "re" en python, le code est chargé à l'aide d'une expression régulière, la méthode "re.findall" renvoie un tableau de 3-uplet contenant une étiquette, la commande et ses arguments (tableau de str), ce tableau est transmis à la méthode build de LineInstruction, qui produit une instance de LineInstruction et en parsant ses arguments.

```
1000            while True:
                  inst = self.prog[self.increment_pc()]
1002              self.instructions[inst.command].execute(self, inst.args)
```

Les instructions sont stockées en tant q'un champ statique de type "dict" dans la machine virtuelle, comme suit :

```
1000    instructions = {"CONST": Const(), "PRIM": Prim(),
                        "BRANCH": Branch(), "BRANCHIFNOT": BranchIfNot(), ... }
```

la machine virtuelle délègue toute les méthode liées à la pile à son instance "Stack".

3.4 Instruction

Comme la machine virtuelle a de nombreuses instructions qui changent d'état, une solution idéale serait d'utiliser le modèle de conception de commande.

La solution consiste à utiliser une classe abstraite "Instruction", qui a deux méthodes, "parse_args" et "execute".

La méthode parse_args est utilisée lors du chargement du bytecode pour analyser les arguments d'une instruction. Chaque sous-classe de la classe Instruction doit redéfinir la méthode parse_args si elle nécessite de paramètres.

La méthode execute prend comme paramètres l'état de la vm et les arguments de l'instruction en cours. Chaque sous-classe de l'instruction doit définir cette méthode comme décrit dans la spécification du projet.

L'instruction "Prim" est spéciale car elle utilise d'autres instructions primitives, la classe "Prim" a deux dictionnaires statiques, le premier contient des opérateurs binaires et le second des opératos unaires, ces opérateurs suivent la même analogie qu'une instruction normale, ils suivent le pattern "commande", en implémentant deux classes abstraites (BinaryPrim ou UnaryPrim), les deux classes ont une méthode execute, mais une avec 2 arguments et l'autre prend un seul et unique argument.

3.5 MLValue

La classe "MLValue" est un conteneur pour les types de données suivants, entiers, booléens, fermetures, blocs et unit, la classe contient les méthodes factory suivantes : from_int : crée une instance "MLValue" à partir d'un entier. from_closure : crée une instance "MLValue" à partir de "pc" et "env". from_block : crée une instance "MLValue" à partir d'un bloc.

La classe contient trois champs finaux statiques, _TRUE, _FALSE, _UNIT, pour accéder à ces champs, nous utilisons les méthodes "true", "false", "unit" qui les instancient si nécessaire. ces champs sont des instances "MLValue" qui enveloppent 1 ou 0 comme décrit dans la spécification du projet. Puisque l'instruction Prim applique des opérations aréthmitiques et logiques sur des instances "MLValue", il est obligatoire que cette classe surcharge tous les opérateurs aréthmitiques(__add__, __mul__,__sub__,__truediv__) et logiques (__bool__).

4 Etude expérimental

Le code est fortement testé, plus de 90% de sa couverture en utilisant le test unitaire. Dans le fichier test_instructions.py, il existe un "TestCase" pour chaque instruction ; ces tests sont indépendants et ne dépendent pas les uns des autres. chaque "TestCase" prépare un scénario approprié pour les instructions testées, il prépare la machine virtuelle, la pile le code si nécessaire, puis la méthode execute est appelée. après l'exécution, l'état vm, stack, acc est vérifié comme dans la spécification du projet.

- **Optimisation** :Nous exécutons le fichier facto_tailrec.txt avec le code optimisé et le code non optimisé, voici le résultat :
  - load_file_optimized temps = 0.000368000000000003 sec
  - load_file = temps = 0.0004289999999999988 sec
- **Historique d'éxécution : n-ary_funs/grab2.txt**
- BRANCH pc = 0
  pc = 14
  accu = ()
  stack= []
- CLOSURE pc = 14
  pc = 15
  accu = MLValue(Value : (8, []))
  stack= []

— PUSH pc = 15
pc = 16
accu = MLValue(Value : (8, []))
stack= [MLValue(Value : (8, []))]
— ACC pc = 16
pc = 17
accu = MLValue(Value : (8, []))
stack= [MLValue(Value : (8, []))]
— CLOSURE pc = 17
pc = 18
accu = MLValue(Value : (1, [MLValue(Value : (8, []))]))
stack= [MLValue(Value : (8, []))]
— PUSH pc = 18
pc = 19
accu = MLValue(Value : (1, [MLValue(Value : (8, []))]))
stack= [MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— CONST pc = 19
pc = 20
accu = MLValue(Value : 5)
stack= [MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— PUSH pc = 20
pc = 21
accu = MLValue(Value : 5)
stack= [MLValue(Value : 5), MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— ACC pc = 21
pc = 22
accu = MLValue(Value : (1, [MLValue(Value : (8, []))]))
stack= [MLValue(Value : 5), MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— APPLY pc = 22
pc = 1
accu = MLValue(Value : (1, [MLValue(Value : (8, []))]))
stack= [MLValue(Value : 5), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— CONST pc = 1
pc = 2
accu = MLValue(Value : 2)
stack= [MLValue(Value : 5), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— PUSH pc = 2
pc = 3
accu = MLValue(Value : 2)
stack= [MLValue(Value : 2), MLValue(Value : 5), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— ACC pc = 3
pc = 4
accu = MLValue(Value : 5)
stack= [MLValue(Value : 2), MLValue(Value : 5), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— PUSH pc = 4
pc = 5
accu = MLValue(Value : 5)
stack= [MLValue(Value : 5), MLValue(Value : 2), MLValue(Value : 5), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— ENVACC pc = 5
pc = 6
accu = MLValue(Value : (8, []))
stack= [MLValue(Value : 5), MLValue(Value : 2), MLValue(Value : 5), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]

— APPTERM pc = 6
pc = 8
accu = MLValue(Value : (8, []))
stack= [MLValue(Value : 5), MLValue(Value : 2), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— GRAB pc = 8
pc = 9
accu = MLValue(Value : (8, []))
stack= [MLValue(Value : 5), MLValue(Value : 2), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— ACC pc = 9
pc = 10
accu = MLValue(Value : 2)
stack= [MLValue(Value : 5), MLValue(Value : 2), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— PUSH pc = 10
pc = 11
accu = MLValue(Value : 2)
stack= [MLValue(Value : 2), MLValue(Value : 5), MLValue(Value : 2), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— ACC pc = 11
pc = 12
accu = MLValue(Value : 5)
stack= [MLValue(Value : 2), MLValue(Value : 5), MLValue(Value : 2), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— PRIM pc = 12
pc = 13
accu = MLValue(Value : 3)
stack= [MLValue(Value : 5), MLValue(Value : 2), 23, [], 0, MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— RETURN pc = 13
pc = 23
accu = MLValue(Value : 3)
stack= [MLValue(Value : (1, [MLValue(Value : (8, []))])), MLValue(Value : (8, []))]
— POP pc = 23
pc = 24
accu = MLValue(Value : 3)
stack= [MLValue(Value : (8, []))]
— POP pc = 24
pc = 25
accu = MLValue(Value : 3)
stack= []
— STOP pc = 25 acc = MLValue(Value : 3)

5 Guide d'utilisation

pour éxécuter les test : python -m unittest src/minizam/vm/test_instructions.py

pour éxécuter un fichier bytecode : python ocamlzam.py file pour éxécuter un fichier bytecode : python ocamlzam.py - file ; ;

6 Conclusion

L'implementation de cette machine virtuelle (MINIZAM) demande donc de comprendre son fonctionnement interne, de comprendre la manière dont les registres sont modifier et de pouvoir identifier les difficultés inhérentes particulier le fonctionnement des differentes instructions.

On choisissant le languge Python, cela nous a facilité l'utilisation des structures comme les listes et les classes.