Rapport Projet S-UM

4I504 - Compilation Avancée

Sorbonne Université

Année universitaire 2017/2018

Jeudy Jordan

Sommaire

I. Description du travail produit

Machine universelle

J'ai réalisé la machine universelle en Java 8, elle fonctionne correctement et passe sur le fichier de benchmark du concours : sandmark.umz.

Lors du développement de l'interpréteur j'ai eu bug qui m'a fait coder l'interpréteur en C. Comme j'ai résolu le bug sur la version Java je me retrouve donc avec deux interpréteurs, la version principale de l'interpréteur est celle en Java.

Langage S-UM

Le compilateur est en Ocaml 4.01.0. Au niveau des instructions, seuls les procédures et le « if then else » n'ont pas été réalisé. La séquence ne marche pas comme prévu car il faut sauté une ligne après un point virgule. Un programme doit également se terminer par un saut de ligne.
Toutes les expressions ont été réalisé

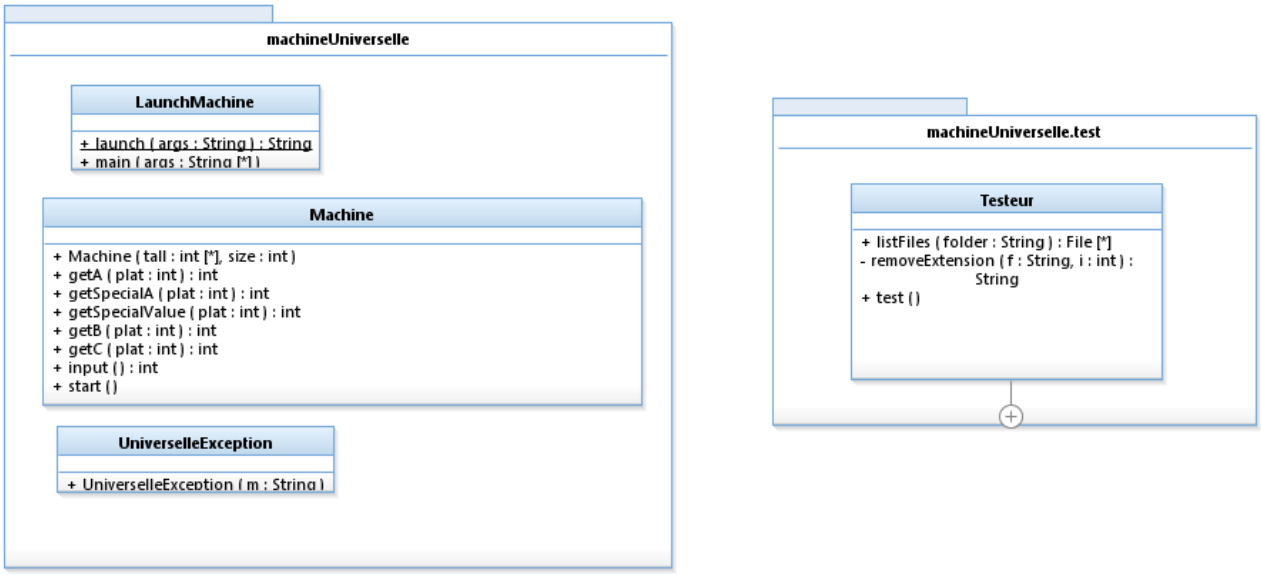
Tests

J'ai un programme qui vérifie que tout les fichiers de test produisent bien la sortie attendue.

II .Architecture logicielle

Machine universelle

Pour l'interpréteur j'ai deux packages, machineUniverselle et machineUniverselle.test



Il y a également un package machineUniverselle.command qui contient un Design Pattern Command pour simplifier l'écriture de byte code. Ce package date d'avant la sortie du projet il m'a quand même été utile car j'ai repris les deux commandes/fonctions qui permettent d'écrire un plateau pour le compilateur

Le dossier binOutput contient les fichiers .um crées par les tests.

III .Choix technique machine universelle

La collection de tableau de plateaux est représenté par une List<List<Integer>> et donc un plateau est représenté par un entier. Un tableau de 8 entiers représente les 8 registres. Je met à jour une variable « PC » qui contient le numéro du plateau à executer. Charger le tableau 0 ne fait modifier le PC ça permet d'éviter de recharger le plateau.

IV .Description du compilateur du langage S-UM

Le compilateur est composé de lexer.mll parser.mly ast.ml eval.ml umc.ml.
J'ai deux types, un type expression et un type statement. Ce n'était pas dans le sujet mais les variables commencent pas une lettre. Pour le booléen je me suis inspiré du langage C. Pour une expression x on a : x= false → x= 0 , x = true → x !=0.
J'utilise deux tableaux globaux (à l'exécution), le tableau 1 crée avec une taille raisonnable qui va contenir les variables que j'appelle le *Heap*. Le tableau 2 que je manipule comme une pile pour les arguments des opérateurs que j'appelle l'*ALU*.
collection[1] = Heap
collection[2] = ALU
Une variable globale aluInd représente le sommet de la pile des opérandes.
A l'éxecution en plus de crée des tableaux, j'ai 3 registres dont le contenu est invariant qui peut simplifier des calculs.
Reg[0] = 0 ; Reg[1]=1 ; Reg[2] = 2
De plus, j'ai des registres qui ont une certaine signification
Reg[4] = indice sommet pile.
Reg[6] = operande1
Reg[3] = operande2
Reg[5] = valeur a affiché

Enfin, pour la lisibilité et pour éviter de réécrire plusieurs fois la même chose j'ai une fonction writeToFile qui écrit un plateau dans un fichier. Et une fonction writeToFileSpecial qui écrit un plateau avec l'opérateur 13.
Au départ j'étais partit pour faire des schéma de compilation dont l'évaluation des expressions auraient pour destination Reg[5]. Mais j'ai changer pour une pile avec le tableau ALU. Donc pour les schémas la destination d'une évaluation est void car une expression est push dans la pile.

Schémas de compilation

Entiers :

→void

entier

---

writeToFileSpecial 13 d entier file; (reg[d] = entier)

writeToFileSpecial 13 4 !aluInd file; (reg[4] = aluInd)

writeToFile 2 regIndALU 4 d file; (ALU[reg[4]] =d)

push() (aluInd++)

Print expr :

(∀ expr ≠ String) →void

print expr

→void

expr ;

pop(); (aluInd--)

writeToFileSpecial 13 regIndCptALU (!aluInd) file; (reg[4]=aluInd)

writeToFile 1 6 regIndALU 4 file; (reg[6]=ALU[reg[4]])

writeToFile 10 0 0 6 file; (print reg[6])

Print string :

→void

print string

Pour chaque ascii ∈ string

writeToFileSpecial 13 6 ascii file; (reg[6]=ascii)

writeToFile 10 0 0 6 file (print reg[6)

Variable :

→void

variable

---

writeToFile 1 regPrint regIndHeap id file (reg[regPrint]=Heap[id])

writeToFileSpecial 13 regIndCptALU !aluInd file;

writeToFile 2 regIndALU regIndCptALU regPrint file;

push()

Addition :

→void

expr1 + expr2

---

fonction getOperands eval expr1 expr2 : on compile les opérandes et on les récupères dans la pile ALU.

fonction setResult 2 : on push le résultat dans la pile ALU, c'est un peu particulier comme je ne fait pas les deux pop dans getOperands, je fais un pop a la fin qui correspond aux deux pop suivi d'un push.

```
→void
expr1 ;
→void
expr2 ;
writeToFileSpecial  13 regIndCptALU (!aluInd-2) file;
writeToFile 1 regOperand1 regIndALU regIndCptALU file;

writeToFileSpecial  13 regIndCptALU (!aluInd-1) file;
writeToFile 1 regOperand2 regIndALU regIndCptALU file

writeToFile 3 regPrint regOperand1 regOperand2 file;

writeToFileSpecial  13 regIndCptALU (!aluInd-2) file;
writeToFile 2 regIndALU regIndCptALU regPrint file ;
pop()
```

Multiplication :

→void

expr1 * expr2

```
getOperands eval x y;
writeToFile 4 regPrint regOperand1 regOperand2 file;
setResult 2 ;
pop()
```

Même raisonnement pour la division

AND :

→void

expr1 AND expr2

---

(0) getOperands eval x y;

(1) writeToFile 0 regOperand2 regIndHeap regOperand2 file;

(2) writeToFile 0 regOperand1 regIndHeap regOperand1 file;

(3) writeToFile 6 regOperand1 regOperand1 regOperand2 file;

(4) writeToFile 6 regPrint regOperand1 regOperand1 file;

(5) writeToFile 0 regPrint regIndHeap regPrint file;

(6) setResult 2;

pop()

Pour compiler le AND après avoir compiler expr1 et expr2 dans reg[regOperand1] et reg[regOperand2] (0). Je met 1 (true) dans les registres opérande si ils contiennent une valeur différente de 0 (false), ça me permet de simplifier les tests.

Donc, $\forall x \neq 0, \forall y \neq 0$ *x* AND *y* $\Leftrightarrow$ 1 AND 1. (1)(2)

Pour faire un AND j'utilise le fait que le NAND est une fonction universelle on a donc

(a NAND b) NAND (a NAND b) = ¬(a NAND b) = a AND b. (3)(4)

En (5) si expr1 AND expr2 est différent de 0 le résultat vaut 1. En (6) je met le résultat dans la pile.

OR :

→void

expr1 OR expr2

---

getOperands eval x y;

writeToFile 0 regOperand2 regIndHeap regOperand2 file;

writeToFile 0 regOperand1 regIndHeap regOperand1 file;

(1) writeToFile 6 regOperand1 regOperand1 regOperand1 file;

(2) writeToFile 6 regOperand2 regOperand2 regOperand2 file;

(3) writeToFile 6 regPrint regOperand1 regOperand2 file;

writeToFile 0 regPrint regIndHeap regPrint file;

setResult 2;

pop()

(a NAND a) NAND (b NAND b) = ¬a NAND ¬b = a OR b (1)(2)(3)

Égalité :

→void

expr1 = expr2

---

getOperands eval x y;

(1) writeToFile 6 regOperand1 regOperand1 regOperand1 file;

(2) writeToFile 3 regOperand1 regOperand1 regIndHeap file;

(3) writeToFile 3 regOperand1 regOperand1 regOperand2 file;

(4) writeToFileSpecial 13 regPrint _TRUE file;

(5) writeToFile 0 regPrint0 regOperand1 file;

writeToFileSpecial 13 regIndCptALU (!aluInd-2) file;

writeToFile 2 regIndALU regIndCptALU regPrint file;

pop()

expr1 = expr2 ⇒ expr2 - expr1 = 0

Comme il n'y a pas de soustraction, je fais le complément à deux de expr1 :

- J'inverse tout les bits de expr1 avec ¬expr1 (1)
- J'ajoute 1 à ¬expr1 le résultat (-expr1) va dans reg[regOperand1] (2)
- expr2 + (-expr1) (3)

Je met _TRUE (la valeur 1) dans le registre regPrint (4)

expr2-expr1 ≠ 0 ⇒ reg[regPrint] = reg[0] sinon reg[regPrint] vaut toujours _TRUE (5)

Supérieur :

→void

expr1 > expr2

---

getOperands eval x y;

(1) writeToFile 6 regOperand1 regOperand1 regOperand1 file;

(2) writeToFile 3 regOperand1 regOperand1 regIndHeap file;

(3) writeToFile 3 regOperand1 regOperand1 regOperand2 file;

(4) writeToFile 3 regOperand2 regOperand1 0 file;

(5) writeToFileSpecial 13 regPrint32768 file;

(6) writeToFile 5 regOperand1 regOperand1 regPrint file;

(7) writeToFile 5 regOperand1 regOperand1 regPrint file;

(8) writeToFile 5 regOperand1 regOperand1 regIndALU file;

(9) writeToFileSpecial 13 regPrint _FALSE file;

(10) writeToFile 0 regPrint regIndHeap regOperand2 file;

(11) writeToFileSpecial 13 regOperand2 _FALSE file;

(12) writeToFile 0 regOperand2 regIndHeap regOperand1 file;

(13) writeToFile 6 regPrint regOperand2 regPrint file;

(14) writeToFile 6 regPrint regPrint regPrint file;

setResult 2;

pop()

Pour compilé le supérieur j'utilise : $x>y \iff 0>y-x$

Les étapes (1)(2)(3) correspondent au calcul de y-x, je copie ce résultat dans reg[regOperand2] (4). Pour tester si 0>y-x il faut regarder le MSB, pour ça je décale de 31 bits vers la droite y-x. Je met 2^15 dans reg[regPrint] (5), je divise y-x par 2^15 deux fois (6)(7) puis je divise par 2 (8). A ce moment reg[regOperand1] contient 0 si c'est un nombre positif, 1 si c'est un nombre négatif.

En (9) reg[regPrint] = 0, (10) $y-x \neq 0 \Rightarrow$ reg[regPrint] = 1, (11) reg[regOperand1]=0, (12) $MSB(y-x) \neq 0 \Rightarrow$ reg[regOperand2] =1. Enfin, (13)(14) $y-x \neq 0 \ \wedge \ MSB(y-x) \neq 0 \Rightarrow x>y$

Même raisonnement pour inférieur. $x < y \iff x-y<0$

Not :

→void

NOT expr

→void

expr

writeToFileSpecial 13 regIndCptALU (!aluInd-1) file;

writeToFile 1 regOperand1 regIndALU regIndCptALU file;

(1) writeToFileSpecial 13 regPrint _TRUE file;

(2) writeToFile 0 regPrint 0 regOperand1 file;

writeToFileSpecial 13 regIndCptALU (!aluInd-1) file;

writeToFile 2 regIndALU regIndCptALU regPrint file;

(1) reg[regPrint] = 1

(2) expr = 0 ⇒ reg[regPrint] =1

Affectation :

→void

let var = expr

→void

expr

pop();

(1) writeToFileSpecial 13 regIndCptALU (!aluInd) file;

(2) writeToFile 1 regPrint regIndALU regIndCptALU file;

writeToFile 2 regIndHeap numV regPrint file

Les variables sont stockées dans une case mémoire du Heap.

En (1)(2) reg[regPrint]=expr, (3) Heap[numV] = reg[regPrint].

Les variables sont donc numérotés lors du parsing, une HashMap : varTab, qui permet de garder les couples <nom_var ,numéro_var> pour savoir si une variable a déjà été définis et lever une exception explicite.

Scan :

→void
scan var

---

(1) writeToFile 11 0 0 regPrint file;
(2) writeToFile 2 regIndHeap id regPrint file

(1) Lecture de la valeur saisie par l'utilisateur dans reg[regPrint]
(2) Heap[id] = reg[regPrint]
« id » est trouvé lors du parsing dans la HashMap, varTab.

Sequence :

→void
expr1;expr2

→void
expr1
→void
expr2

Difficulté :

En dehors du codage sur Ocaml, la première difficulté a été de trouver le système de pile avec « l'ALU ». En début de projet je pensais pouvoir m'en sortir uniquement avec les registres opérande et le registre servant a print. C'est à dire que le résultat d'une expression était l'opérande 1 de l'expression suivante mais rapidement sur des expression plus intéressante ça ne fonctionnait plus. Concernant l'ALU je n'ai pas réussi à déterminer à la compilation sa taille maximal pour pouvoir allouer juste ce qu'il fallait. Même chose pour le Heap dont la taille est connu lors du parsing mais je n'ai pas réussi à récupérer le compteur de variable du parser dans l'évaluateur.
Enfin, le nombre de test est proportionnelle au nombre de bug que j'ai eu sur une expression, les expressions logiques (ou relationnelles) ayant le plus de test ça a donc été ma principale difficulté. Les schémas de compilation sont assez long et me donne l'impression d'avoir fait plus compliquer que prévu.

V .Architecture des tests

Il y a au moins un test par expressions/instructions, je n'ai pas fait des tests très poussés c'était uniquement pour debug mon compilateur et essayé de révéler des fautes.
La classe Testeur, lit tout les fichiers du dossier tests et compare la sortie de la machine avec le fichier .output.