Université Paris Cité
L3 Informatique

Machines Virtuelles
Année 2024-2025

TP n°2

Implémentation d'une machine à a-pile

Dans ce TP, vous utiliserez les fichiers `mv.ml` et `assembleur.ml` qui vous trouvez sur la page Web du cours et qui ont été expliqués en cours.

1 Premiers pas avec mv.ml

Dans un shell on lance `ocaml`. Ensuite, avec `#use "mv.ml";;` on charge la machine virtuelle.

Exercice 1. Définissez les valeurs de type `instr array` associées aux programmes suivants :
ex1 : Consti 3; Push; Consti 2; Addi
ex2 : Push; Addi; Push; Addi; Push; Consti 3; Addi
ex3 : Consti 3; Pop; Push
Mettez ces valeurs dans des variables pour utilisation antérieure.

Exercice 2. Exécutez à la main la machine virtuelle sur les états initials associés aux trois exemples du Exercice 1. Ensuite, vérifiez vos résultats en lançant la function `exec_big` avec les bons paramètres. Comprenez vous le résultat du calcul pour chaque exemple ?

Exercice 3. Vous noterez que la fonction `exec_big` est muette sur le déroulement de son fonctionnement. Dans cette question, ajoutez des informations de déboggage. À chaque appel récursif (ou tour de boucle, selon la version que vous utilisez) imprimez l'état de la machine (PC, A et pile (définir une fonction auxiliaire `string_of_state`)), comme aussi l'instruction pointée par PC (définir une fonction auxiliaire `string_of_instr`).

2 Extension du jeu d'instructions

Le jeu d'instructions de la machine virtuelle présenté en cours est insuffisant pour implémenter des fonctions simples comme la multiplication. Le but de cette section est d'ajouter les instructions suivantes :

- `Acc n :` copie dans l'accumulateur A la valeur du n-ème élément de la pile S sans dépiler l'élément (la tête de la pile étant le 0-ème element) ;
- `Assign n :` écrit la valeur de l'accumulateur comme n-ème élément de la pile S (en écrasant la valeur précédente de ce n-ème élément et en supposant que la pile contient au moins $n + 1$ éléments). L'accumulateur prend la valeur 0 ;
- `Popi n :` dépile les premiers n éléments du sommet de la pile S (c'est tout) ;
- `Branch n :` saute n instructions en avant (ou en arrière si n est négatif), c'est-à-dire $PC = PC+n$;
- `Branchif n :` saute n instructions si l'accumulateur A est différent de 0, sinon il avance à l'instruction suivante ;
- `Multi :` dépile un mot n de S, remplace A par $A \times n$;
- `Leqi :` dépile un mot n de S, remplace A par 1 si $n \leq A$, 0 sinon.

Qu'est-ce qu'il devrait se passer si l'argument `n` dépasse les limites de la pile dans `Acc n`, `Assign n` ou `Popi n` ? et s'il dépasse le tableau d'instructions dans `Branch n` ou `Branchif n` ?

Exercice 4. Ajoutez les nouveaux constructeurs au type `instr`. Définissez les valeurs associées aux programmes suivants :

ex4 : Consti 3; Push; Branchif 5; Consti −1; Addi; Push; Branch −4; Popi 1
ex5 : Consti 2; Push; Consti 1; Push; Acc 0; Push; Acc 2; Assign 1; Acc 0; Assign 1
ex6 : Branch 0
ex7 : Popi −3

Exercice 5. Modifiez la fonction `exec_small` pour lui permettre de traiter les nouvelles instructions introduites. Testez votre implémentation en appliquant `exec_big` sur les examples de l'exercice 4, ainsi que sur les exercices du TD 1. Vérifiez que le résultat calculé corresponde bien à la sémantique opérationnelle donnée ci-dessus. N'hésitez pas à faire d'autres tests si vous avez des doutes.

3 Code-octet

Le code `mv.ml` utilise la définition du code-octet vue en cours. Ce code-octet est particulièrement limité car il ne permet pas d'encoder des nombres négatifs ni a suffisamment d'espace pour associer des op-code aux nouvelles instructions.

Le but de cette section est de modifier les fonctions `assemble` et `disassemble` pour utiliser un nouveau code-octet. Avec ce nouveau formalisme, chaque instruction de la MV est codée sur 3 octets. Le code-octet est comme suit :

Instruction 1			Instruction 2			Instruction 3			...
opcode	signe	valeur	opcode	signe	valeur	opcode	signe	valeur	...

Les op-codes sont définis comme suit (aucun changement sur les codes des instructions vues en cours) :

`Push` ↦ 00000000	`Addi` ↦ 00000001	`Eqi` ↦ 00000010	`Ori` ↦ 00000011
`Consti` ↦ 00000100	`Pop` ↦ 00000101	`Acc` ↦ 00000110	`Popi` ↦ 00000111
`Branch` ↦ 00001000	`Branchif` ↦ 00001001	`Assign` ↦ 00001010	`Multi` ↦ 00001011
`Leqi` ↦ 00001100			

Les deux octets « signe » et « valeur » ne sont utilisés que pour les instructions de la MV avec arguments : le signe est 0 si l'argument est positif ou nul, 1 sinon. La valeur est l'argument, en valeur absolue.

Exercice 6. Trouvez à quelle suite d'instructions de la MV ce code-octet correspond :

```
00000100 00000001 00000001 00000000 00000000 00000000
00000100 00000000 00000110 00000001 00000000 00000000
```

Exercice 7. Adaptez `assemble` et `disassemble` au nouveau code-octet. Testez bien vos fonctions sur les exemples des Exercices 4 et 6.