MV6 2015 – TP du 17 février

Benoît Valiron

Dans ce TP, vous utiliserez le fichier `mv_myrte.ml`. Avec emacs, vous pouvez soit :

- ouvrir un buffer shell en faisant « M-x shell », puis lancer `ocaml` et charger le fichier avec la commande
  `#use "mv_myrte.ml" ;;`
- ouvrir votre fichier `mv_myrte.ml`, et faire « C-c C-e » pour chaque déclaration afin de l'exécuter dans le buffer caml associé.

Exercice 1 : Code-octet

Le code `mv_myrte.ml` utilise la définition de code-octet vu en séance 1. Ce code-octet est particulièrement limité car il ne permet pas d'encoder des nombres négatifs.

Le but de cet exercice est de modifier les fonctions `assemble` et `disassemble` pour utiliser la un nouveau code octet. Avec ce nouveau formalisme, chaque instruction de la MV est codée sur 3 octets (un entier de type `char`). Le code-octet est comme suit :

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

Chaque op-code est défini comme en séance 1 (et donc comme dans le fichier `mv_myrte.ml`). 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.

1. Faites les modifications demandées.
2. Trouvez à quelle suite d'instructions de la MV ce code-octet correspond :

   ```
   00000100 00000000 00000011 00000000 00000000 00000000
   00000100 00000001 00000001 00000001 00000000 00000000
   00000101 00000000 00000000
   ```

   Vous pouvez utiliser la commande unix `xxd -b` pour afficher le contenu d'un fichier en binaire.
3. Ajouter une instruction `Subs` à la liste d'instructions de la MV, opérant la soustraction de l'accumulateur et du sommet de la pile. Choisissez-lui un op-code. Mettez à jour les fonctions `assemble`, `disassemble` et `machine`.

Exercice 2 : Machine virtuelle et compilation

1. Écrivez une fonction

   ```
   val string_of_instr : instr -> string
   ```

   qui génère une chaine de caractères à partir d'une instruction de la MV, et

   ```
   val print_instrs : instr list -> unit
   ```

   qui imprime le contenu d'une liste d'instructions, une instruction par ligne, avec leur numéro correspondant dans la liste, comme par exemple ce qui est montré dans l'exercice 2.3.

   Note : `string_of_int` permet de créer une chaine de caractères représentant un entier.

2. Évaluez le terme

   ```
   get_acc (machine (init exemple1))
   ```

   Vous noterez que la fonction `machine` est muette sur le déroulement de son fonctionnement.

   Dans cette question, ajoutez des information de déboggage. À chaque tour de boucle :
   - avant l'exécution du `match`, imprimer à l'écran la valeur du PC et imprimer l'instruction de la MV correspondante.
   - après le `match`, imprimer l'état de la machine : PC, accumulateur et pile.

   À la fin de la boucle, affichez le contenu de l'accumulateur (rappelez-vous, c'est là qu'est le résultat).

   Ré-évaluez le terme

   ```
   get_acc (machine (init exemple1))
   ```

   pour voir ce qui se passe lors de l'évaluation des instructions de la MV.

3. En utilisant `print_instrs` et `compil`, pouvez-vous trouvez à quelles expressions de Myrte correspondent chaque liste d'instructions de la MV ?

`0 - Consti 2`
`1 - Push`
`2 - Consti 3`
`3 - Addi`
`4 - Pop`
	`0 - Consti 1`
	`1 - Push`
	`2 - Consti 2`
	`3 - Addi`
	`4 - Pop`
	`5 - Push`
	`6 - Consti 3`
	`7 - Addi`
	`8 - Pop`

`0 - Consti 1`
`1 - Push`
`2 - Consti 2`
`3 - Push`
`4 - Consti 3`
`5 - Addi`
`6 - Pop`
`7 - Addi`
`8 - Pop`
	`0 - Consti 0`
	`1 - Push`
	`2 - Consti 1`
	`3 - Addi`
	`4 - Pop`
	`5 - Push`
	`6 - Consti 2`
	`7 - Push`
	`8 - Consti 3`
	`9 - Addi`
	`10 - Pop`
	`11 - Addi`
	`12 - Pop`

   Évaluez-les avec la fonction `machine` : le comportement est-il celui attendu ?

Exercice 3 : Ajout de tests

Dans cet exercice, on considère les structures de tests vu dans le cours n.3 : Les instructions supplémentaires pour la MV : `BranchIf` et `Branch`, et l'extension des expressions `expr` de Myrte avec le noeud `If of expr * expr * expr`.

Rappelez vous que si l'argument de `BranchIf` ou `Branch` est négatif, le le PC « saute » en arrière.

1. Effectuer les ajouts, et mettez à jour des fonctions correspondantes : `assemble`, `disassemble`, `machine`, `compil`, `interp`...

   Choisissez les op-codes (non-encore utilisés) que vous voulez pour les deux nouvelles instructions de la MV.

2. Encodez en expression `expr` de Myrte, compilez et faites tourner la MV sur les programmes Myrte :

   ```
   if (4=5) then (2+3) else (6+7)
   if (5=5) then (2+3) else (6+7)
   9 + (if ((if (1=2) then 3 else 4) = 5) then (6) else (7+8))
   ```

3. En utilisant `print_instrs` et `compil`, pouvez-vous trouvez à quelles expressions de Myrte correspondent chacune des listes d'instructions de la MV ?

0 - Consti 1 1 - Push 2 - Consti 2 3 - Eqi 4 - Pop 5 - BranchIf 3 6 - Consti 3 7 - Branch 2 8 - Consti 1	0 - Consti 1 1 - Push 2 - Consti 1 3 - Eqi 4 - Pop 5 - BranchIf 3 6 - Consti 3 7 - Branch 2 8 - Consti 1
0 - Consti 1 1 - Push 2 - Consti 2 3 - Eqi 4 - Pop 5 - BranchIf 3 6 - Consti 3 7 - Branch 10 8 - Consti 1 9 - Push 10 - Consti 2 11 - Eqi 12 - Pop 13 - BranchIf 3 14 - Consti 3 15 - Branch 2 16 - Consti 2	0 - Consti 0 1 - Push 2 - Consti 2 3 - Eqi 4 - Pop 5 - BranchIf 11 6 - Consti 1 7 - Push 8 - Consti 2 9 - Eqi 10 - Pop 11 - BranchIf 3 12 - Consti 3 13 - Branch 2 14 - Consti 2 15 - Branch 6 16 - Consti 1 17 - BranchIf 3 18 - Consti 3 19 - Branch 2 20 - Consti 2

   Évaluez-les avec la fonction `machine` : le comportement est-il celui attendu ?

Exercice 4 : Ajout de variables

Ici, on finit avec l'ajout de variables.

1. Étendez `expr` et `interp` comme vu en séance 3. Note : les transparents ont été mis à jour après le cours. En particulier, un environnement renvoie une exception si la variable n'est pas présente, et donc l'environnement vide a été modifié.

   Interprétez les termes

   ```
   let z = 3 in 4 + z
   let x = 1 in (let y = 3 in x + y)
   let x = 1 in (let x = 3 in x + x)
   ```

   Est-ce que vous pouvez décomposer sur papier ce qui se passe ? Quel est l'environnement pour chaque sous-terme ?

2. En cours, le type `envexpr` qui encode une table d'association était représenté comme une fonction. Changez la représentation pour utiliser une liste d'association

Et si vous avez fini le reste...

Rappelez-vous le dernier exercice du TD : il était possible de faire la multiplication par une constante.

1. Ajouter un opérateur binaire `Mult` afin de pouvoir représenter la multiplication.

   Rajouter un cas au `match` de la fonction `compil` (avant l'appel générique) :

   ```
   | Binop (Mult,Const(Int c),e2) -> ...
   ```

   et encodez la multiplication par une constante en une suite d'instructions de la MV comprenant des `Consti`, `Push`, `Pop` et des `Addi`. Essayez d'être efficace et de laisser la pile dans l'état où vous l'avez trouvé en entrant (préservation de l'invariant).

   N'hésitez pas à utiliser des `failwith` pour l'extension des fonctions auxiliaires (par exemple `op`).

2. Il n'est pas vraiment possible de réaliser la multiplication de l'accumulateur avec le premier élément de la pile uniquement avec les instructions de la MV données.

   Si vous rajoutez

   ```
   Acc n
   ```

   qui ne fait rien d'autre qu'accèder au `n`-ième élément de la pile (`Acc 0` accèdant au sommet) et le mettre dans l'accumulateur, en utilisant des branchements conditionnels (capitalisant sur le fait qu'on a des nombres négatifs) il est maintenant possible de le faire.

   Pourriez-vous trouver comment compiler la multiplication en général ? (l'efficacité n'est pas recherchée)