Université Paris Diderot — Paris 7
Informatique — Licence 3

Machines Virtuelles MV6
Année 2012–2013

Examen

16/05/2013

Durée: 3h. Tous documents autorisés.

Exercice 1 (Thème). Compiler en bytecode OCaml les expressions suivantes[1]:

1. `(2 + 2) = (1 + 3)`
2. `1 :: 2 :: []`
3. `let f x = x * x in f 2`
4. `let x = (true, false) in snd x && fst x`
   (*indice*: `fst` et `snd` sont traduits directement en primitives sur les blocs, et la conjonction en un saut conditionnel)
5. `match [1;2] with [] -> 0 | x :: xs -> x`
6. `match N (N (L, L), L) with L -> L | N(L, y) -> y | N(x, y) -> x`
   (avec la définition `type tree = N of tree * tree | L`)
7. `let x = {contents = None} in x.contents <- Some (); x`
   (on rappelle la définition `type 'a ref = {mutable contents: 'a}`)
8. `let x = 2+2 in fun y -> x :: y`

Exercice 2 (Version). Soit le bytecode OCaml suivant:

```
        closurerec 1, 0
        const 10
        push
        acc 1
        appterm 1, 3
L1:     acc 0
        push
        const 1
        eqint
        branchif L3
        acc 0
        push
        const 2
        eqint
        branchifnot L2
L3:     const 1
        return 1
L2:     acc 0
        offsetint -2
        push
ICI:    offsetclosure 0
        apply 1
        push
        acc 1
        offsetint -1
        push
        offsetclosure 0
        apply 1
        addint
        return 1
```

1. On exécute ce bytecode jusqu'au premier passage par l'instruction d'étiquette `ICI`. Décrire alors l'état de la machine virtuelle OCaml après cette instruction: contenu de l'accumulateur et de la pile.

[1] Les valeurs allouées seront construite explicitement avec des instructions; on n'utilisera pas la notation abrégée vue en TD.

2. Retrouver une expression OCaml dont la compilation produit ce bytecode. Associer à chaque sous-expression la partie de bytecode qui correspond.
3. Quelle est la valeur de cette expression?

Exercice 3 (Bytecode Java). Étant donné le bytecode Java suivant de la fonction `func`[2]:

```
public static int[] func(int[]);
  Code:
   0:   aload_0
   1:   arraylength
   2:   newarray int
   4:   astore_1
   5:   iconst_0
   6:   istore_2
   7:   iload_2
   8:   aload_0
   9:   arraylength
   10:  if_icmpge       33
   13:  aload_1
   14:  iload_2
   15:  aload_0
   16:  iload_2
   17:  iaload
   18:  aload_1
   19:  iload_2
   20:  iconst_1
   21:  isub
   22:  iaload
   23:  iadd
   24:  iconst_2
   25:  imul
   26:  iastore
   27:  iinc    2, 1
   30:  goto    7
   33:  aload_1
   34:  areturn
```

1. On appelle la fonction `func` avec comme argument `{1, 2, 3}`. Décrire l'état de la machine après l'exécution des instructions 4, 8, et après le premier passage à l'instruction 20.
2. Retrouver la fonction Java dont la compilation produit ce bytecode.
3. On échange les instructions 13 et 14, et on assemble le résultat en un fichier `prog.class`. Que produit l'exécution de `java prog`? Décrire précisément où se situe alors le problème.

Problème 1 (Analyse statique de consommation mémoire). On considère le langage Myrte, sa sémantique et sa machine virtuelle à a-pile tels que défini en cours. On rappelle que ses expressions sont formées des constantes 0, 1, 2, ..., **true**, **false**, et des opérations binaires $+$, $=$ et $\wedge$. Son bytecode est composé des instructions $\texttt{consti}(\bar{n})$, `push`, `addi` et `andi`, et la fonction de compilation $[\![e]\!]$ est définie récursivement de façon à calculer la valeur de e de gauche à droite:

$$[\![n]\!] = \texttt{consti}(\bar{n}) \tag{1}$$
$$[\![e_1 + e_2]\!] = [\![e_1]\!]; \texttt{push}; [\![e_2]\!]; \texttt{addi} \tag{2}$$
$$[\![e_1 = e_2]\!] = [\![e_1]\!]; \texttt{push}; [\![e_2]\!]; \texttt{eqi} \tag{3}$$
$$[\![e_1 \wedge e_2]\!] = [\![e_1]\!]; \texttt{push}; [\![e_2]\!]; \texttt{andi} \tag{4}$$

[2] On rappelle la sémantique de certaines de ces instructions:
`if_icmpge` n dépile successivement deux valeurs entières i_1 et i_2, et saute à l'instruction n si $i_2 \geq i_1$
`iaload` dépile successivement i (entier) et a (tableau) et empile la valeur $a[i]$
`iastore` dépile successivement n (entier), i (entier) et a (tableau), et stocke n à la i-ème case de a
`iinc` n, i incrémente la variable locale entière n de i

1. Soit e l'expression $((2 + 3) = 5) \wedge ((8 + 1) = 9)$. Dessiner son arbre de syntaxe abstraite.
2. Proposer une liste d'instructions permettant de calculer e. Associer à chaque sous-arbre de syntaxe la séquence d'instructions contiguës calculant sa valeur.
3. Annoter chaque instruction avec la taille de la pile *après* son exécution, sachant qu'on part du début du programme avec une pile vide. De combien de cases mémoire a-t-on besoin pour calculer e?
4. Reporter cette annotation sur chaque noeud de l'arbre de syntaxe abstraite. Quelle est la relation entre l'annotation d'un noeud et celles de ses fils directs?
5. Proposer une fonction définie par récursion sur une expression qui calcule la taille de pile nécessaire à son calcul.
6. On change maintenant la façon de compiler les expressions Myrte: au lieu de les calculer de gauche à droite, on les calcule de droite à gauche. Autrement dit, la fonction de compilation est changée en:

$$[\![n]\!] = \texttt{consti}(\bar{n}) \tag{5}$$
$$[\![e_1 + e_2]\!] = [\![e_2]\!]; \texttt{push}; [\![e_1]\!]; \texttt{addi} \tag{6}$$
$$[\![e_1 = e_2]\!] = [\![e_2]\!]; \texttt{push}; [\![e_1]\!]; \texttt{eqi} \tag{7}$$
$$[\![e_1 \wedge e_2]\!] = [\![e_2]\!]; \texttt{push}; [\![e_1]\!]; \texttt{andi} \tag{8}$$

   Notez que le résultat de l'exécution d'une expression compilée est le même que précédemment. Répondre à nouveau aux questions 2.–5. avec ce nouveau schéma de compilation?
7. Supposons que l'on étende Myrte avec la soustraction entière $-$. Décrire l'extension correspondante de la machine virtuelle et de la fonction de compilation de gauche à droite. À quel problème se heurte-t-on pour définir la compilation de droite à gauche, et comment le résoudre?
8. Proposer une fonction de compilation optimisante, qui minimise l'espace de mémoire nécessaire à l'exécution du code en résultant.