Université de Paris
L3 Info/Math-Info

Machines Virtuelles
Année 2022-2023

Examen

jeudi 11 mai 2023

Aucun document n'est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints. La durée de l'examen est 2 heures. L'annexe liste certaines instructions de la machine virtuelle OCaml et de la JVM.

On recommande de *bien lire* l'énoncé d'un exercice avant de commencer à le résoudre.

Exercice 1. L'expression suivante en OCaml :

```
let x = 2 in let y = 3 in x,y
```

est traduite en bytecode :

```
const 2
push
const 3
push
acc 0
push
acc 2
makeblock 2, 0
return 3
```

1. Donner l'évolution de la configuration de ocamlrun (`pc`, `accu`, `pile`, `tas`) sur ce bytecode.
2. Donner du bytecode plus court (une ligne de moins suffit) qui a le même effet.
3. En général, est-ce qu'on peut toujours remplacer deux instructions `push` et `acc 0` qui se suivent par `push` uniquement ? (justifier brièvement)
4. Donnez une expression OCaml dont la traduction en bytecode contient les instructions successives suivantes :

```
push
acc 0
push
acc 0
```

Exercice 2 (Compilation à la main de OCaml)**.** Donner le bytecode de OCamlrun produit par le compilateur OCaml pour l'expression de OCaml suivante :

```
let x = 3 in let z = 2*x in z+1
```

Exercice 3 (Compilation à la main de OCaml)**.** Donner le bytecode de OCamlrun produit par le compilateur OCAML pour l'expression de OCaml suivante :

```
let x = 42 in let y = x+1 in let f a = a + x + y in x + (f 1)
```

Exercice 4 (Ocaml)**.** Pour chacune des listes d'instructions suivantes (annotées avec des étiquettes), donner l'évolution de la configuration de ocamlrun (`pc`, `accu`, `env`, `extra_args`, `pile`, `tas` à gauche et uniquement `pc`, `accu`, `pile` à droite) **et** deviner l'expression de OCaml qui l'a générée :

(i).

```
1:      const 2
2:      push
3:      acc 0
4:      closure L1, 1
5:      push
6:      acc 1
7:      push
8:      const 3
9:      addint
10:     push
11:     const 2
12:     push
13:     const 1
14:     push
15:     acc 3
16:     appterm 3, 6
17:     restart
L1:     grab 2
L1+1:   acc 2
L1+2:   push
L1+3:   acc 2
L1+4:   push
L1+5:   acc 2
L1+6:   push
L1+7:   envacc 1
L1+8:   addint
L1+9:   addint
L1+10:  addint
L1+11:  return 3
```

(ii).

```
1:    const 2
2:    push
3:    const 13
4:    push
5:    acc 0
6:    push
7:    const 15
8:    geint
9:    branchifnot L2
10:   acc 0
11:   offsetint 2
12:   branch L1
L2:   acc 0
L1:   pop 1
L1+1: mulint
L1+2: return 1
```

Exercice 5 (JVM)**.** Considérons une méthode `f` dans une classe `MaClasse` qui est de la forme

```
class MaClasse{
    static int f(int x, int y){
        ....
        ....
    }
}
```

On rappelle que pour une méthode statique l'adresse de `this` ne se trouve pas en haut de la pile contrairement à une méthode non-statique. Pour **chacune** des portions de code octet JVM suivantes **trouver** des instructions JAVA pour `f` qui génèrent le bytecode.

(i).

```
int f(int, int);
  descriptor: (II)I
  Code:
    stack=2, locals=4, args_size=3
       0: iload_2
       1: iload_1
       2: iadd
       3: istore_3
       4: iload_3
       5: bipush        42
       7: if_icmple     16
      10: iinc          3, -1
      13: goto          19
      16: iinc          3, 1
      19: iconst_4
      20: iload_3
      21: imul
      22: ireturn
```

(ii).

```
static int f(float, int);
  descriptor: (FI)I
  flags: (0x0008) ACC_STATIC
  Code:
    stack=2, locals=4, args_size=2
       0: fload_0
       1: f2i
       2: istore_2
       3: iload_1
       4: istore_3
       5: iload_3
       6: iload_2
       7: if_icmplt     24
      10: iload_1
      11: iload_3
      12: isub
      13: istore_3
      14: iload_3
      15: iload_2
      16: imul
      17: fload_0
      18: f2i
      19: iadd
      20: istore_2
      21: goto          5
      24: iload_2
      25: ireturn
```

Exercice 6 (Compilation à la main de Java)**.** On considère la méthode :

```
static int f(int[] x){
   return x[x[x[0]]];
}
```

Cela correspond au bytecode :

```
static int f(int[]);
  descriptor: ([I)I
  flags: (0x0008) ACC_STATIC
  Code:
    stack=4, locals=1, args_size=1
```

Compléter la partie manquante (huit instructions).

Exercice 7. On considère l'instruction `x = (a*b)+(c*d)` en Java (toutes les variables sont des entiers). Le bytecode Java correspondant a besoin d'une pile de taille 3 (Le résultat de `a*b` est stocké dans la pile pendant le calcul de `c*d`). Pourtant, en utilisant d'autres variables on pourrait transformer cette instruction en plusieurs instructions de sorte que le byte-code correspondant ait besoin d'une pile de taille 2 uniquement. Expliquez comment pour l'exemple. Comment faire cela en général pour des méthodes qui n'appellent pas d'autre méthodes et qui manipulent uniquement des entiers (int) ?

Annexe OCamlrun

acc n Peeks the n+1-th element of the stack and puts it into the accumulator.

envacc n Sets the accumulator to the field of index n of the environment.

apply n Sets extraArgs to n-1. Sets pc to the code value of the accumulator. Then sets the environment to the value of the accumulator.

appterm n, s Slides the n top elements from the stack towards bottom of s - n positions (In other words the top n elements of the stack are preserved and the following s - n elements are removed). Then sets pc to the code value of the accumulator, the environment to the accumulator, and increases extraArgs by n-1.

return n Pops n elements from the stack. If extraArgs is strictly positive then it is decremented, pc is set to the code value of the accumulator, and the environment is set to the value of the accumulator. Otherwise, three values are popped from the stack and assigned to pc, environment and extraArgs.

restart Computes n, the number of arguments, as the size of the environment minus 2. Then pushes elements of the environment from index n - 1 to 2 onto the stack. Environment is set to the element of index 1 of the environment and extraArgs is increased by n.

grab n If extraArgs is greater than or equal to n, then extraArgs is decreased by n. Otherwise, creates a closure of extraArgs+3 elements in the accumulator. Code of this closure is set to the preceeding restart, element of index 1 is set to the environment and other elements are set to values popped from the stack. Then pc, environment, and extraArgs are popped from the stack.

closure ofs, n If n is greater than zero then the accumulator is pushed onto the stack. A closure of n + 1 elements is created into the accumulator. The code value of the closure is set to pc + ofs. Then, the other elements of the closure are set to values popped from the stack.

makeblock n, t Creates a block of n elements, with tag t. The element of index 0 of the block is set to the value of the accumulator, the n-1 other elements are popped from the stack. Then the accumulator is set to the created block.

getfield n Sets the accumulator to the value of the field of index n of the accumulator.

setfield n Sets the field of index n of the block in the accumulator to the value popped from the stack. Then sets the accumulator to the unit value.

branchifnot ofs Performs an conditional jump by adding ofs to pc if the accumulator is zero.

eqint Sets the accumulator to a non-zero value or to zero whether the accumulator is equal to the value popped from the stack or not.

const n Sets the accumulator to n.

addint Sets the accumulator to the sum of the accumulator and the value popped from the stack.

mulint Sets the accumulator to the product of the accumulator by the value popped from the stack.

geint Sets the accumulator to a non-zero value or to zero whether the accumulator is greater than or equal to the value popped from the stack or not.

ltint Sets the accumulator to a non-zero value or to zero whether the accumulator is lower than the value popped from the stack or not.

offsetint ofs Adds ofs to the accumulator.

pop n Pops n elements from the stack.

push Pushes the accumulator onto the stack. The accumulator is unchanged.

Annexe JVM

On rappelle que les instruction de la JVM sont typées. Certaines des instructions suivantes existent donc aussi pour d'autre types (par exemple iconst existe aussi pour les long (lconst), iadd pour les floats (fadd), etc. Attention : Quand on fait référence à value1 et value2 dans une instruction, value2 est en haut de la pile et value1 juste en dessous.

iconst_n Push the int constant n (-1, 0, 1, 2, 3, 4 or 5) onto the operand stack.

bipush Push a byte onto the stack as an integer value

iload_n The n must be an index into the local variable array of the current frame. The local variable at n must contain an int. The value of the local variable at n is pushed onto the operand stack.

istore_n The n must be an index into the local variable array of the current frame. The value on the top of the operand stack must be of type int. It is popped from the operand stack, and the value of the local variable at n is set to value.

aload_n The n must be an index into the local variable array of the current frame. The local variable at n must contain a reference. The objectref in the local variable at n is pushed onto the operand stack.

iaload The arrayref must be of type reference and must refer to an array whose components are of type int. The index must be of type int. Both arrayref and index (which is on top) are popped from the operand stack. The int value in the component of the array at index is retrieved and pushed onto the operand stack.

goto branchbyte1 branchbyte2 The unsigned bytes branchbyte1 and branchbyte2 are used to construct a signed 16-bit branchoffset, where branchoffset is (branchbyte1 << 8) | branchbyte2. Execution proceeds at that offset from the address of the opcode of this goto instruction. The target address must be that of an opcode of an instruction within the method that contains this goto instruction. In the disassembled bytecode the target address is explicitly given.

iadd Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 + value2. The result is pushed onto the operand stack. The result is pushed onto the operand stack.

imul Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 * value2. The result is pushed onto the operand stack.

idiv Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is the value of the Java programming language expression value1 / value2. The result is pushed onto the operand stack.

isub Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 - value2. The result is pushed onto the operand stack.

i2l Convert int to long. The value on the top of the operand stack must be of type int. It is popped from the operand stack and sign-extended to a long result. That result is pushed onto the operand stack.

l2i Convert long to int. The value on the top of the operand stack must be of type long. It is popped from the operand stack and converted to an int result by taking the low-order 32 bits of the long value and discarding the high-order 32 bits. The result is pushed onto the operand stack.

i2f Convert int to float. The value on the top of the operand stack must be of type int. It is popped from the operand stack and converted to the float result using IEEE 754 round to nearest mode. The result is pushed onto the operand stack.

f2i Convert float to int. The value on the top of the operand stack must be of type float. It is popped from the operand stack and converted to an int result. The result is pushed onto the operand stack.

ldc Push item from runtime constant pool. To simplify, write for example `ldc 0.5f`.

ldc2_w Push long or double from run-time constant pool (wide index). To simplify, write for example `ldc2_w 2L`

lcmp Compare long. Both value1 and value2 must be of type long. They are both popped from the operand stack, and a signed integer comparison is performed. If value1 is greater than value2, the int value 1 is pushed onto the operand stack. If value1 is equal to value2, the int value 0 is pushed onto the operand stack. If value1 is less than value2, the int value -1 is pushed onto the operand stack.

if<cond> branchtype1 branchtype2 The value must be of type int. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows : ifeq succeeds if and only if value $= 0$. ifne succeeds if and only if value $\neq 0$. iflt succeeds if and only if value < 0. ifle succeeds if and only if value ≤ 0. ifgt succeeds if and only if value > 0. ifge succeeds if and only if value ≥ 0. If the comparison succeeds, the unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be (branchbyte1 << 8) — branchbyte2. Execution then proceeds at that offset from the address of the opcode of this if<cond> instruction. The target address must be that of an opcode of an instruction within the method that contains this if<cond> instruction. Otherwise, execution proceeds at the address of the instruction following this if<cond> instruction. In the disassembled bytecode the target address is explicitly given.

if_icmp<cond> branchbyte1 branchbyte2 Both value1 and value2 must be of type int. They are both popped from the operand stack and compared. All comparisons are signed. The possible comparisons are : eq (equal), ne (not equal), lt (less than), le (less or equal), gt (greater than), ge (greater or equal).

If the comparison succeeds, the unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be (branchbyte1 << 8) | branchbyte2. Execution then proceeds at that offset from the address of the opcode of this if_icmp<cond> instruction. The target address must be that of an opcode of an instruction within the method that contains this if_icmp<cond> instruction.

Otherwise, execution proceeds at the address of the instruction following this if_icmp<cond> instruction.

In the disassembled bytecode the target address is explicitly given.

if<cond> branchbyte1 branchbyte2 The value must be of type int. It is popped from the operand stack and compared against zero. Cfr if_icmp<cond>.

ifnonnull branchtype1 branchtype2 The value must be of type reference. It is popped from the operand stack. If value is not null, the unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be (branchbyte1 << 8) | branchbyte2. Execution then proceeds at that offset from the address of the opcode of this ifnonnull instruction. The target address must be that of an opcode of an instruction within the method that contains this ifnonnull instruction.

Otherwise, execution proceeds at the address of the instruction following this ifnonnull instruction.

In the disassembled bytecode the target address is explicitly given.

iinc index const The index is an unsigned byte that must be an index into the local variable array of the current frame. The const is an immediate signed byte. The local variable at index must contain an int. The value const is first sign-extended to an int, and then the local variable at index is incremented by that amount.

ireturn Return int from method