Université de Paris
L3 Info/Math-Info

Machines Virtuelles
Année 2020-2021

Examen

vendredi 21 mai 2021

Tout document papier 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.

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

Exercice 1 (Compilation à la main de OCaml)**.** Donner un bytecode de OCamlrun équivalent à l'expression de OCaml suivante :

```
let x = 2 in 3 + x
```

Exercice 2 (Compilation à la main de OCaml)**.** Donner un bytecode de OCamlrun équivalent à l'expression de OCaml suivante :

```
let a = 2 in let g x = a + x in (g 2) + 1
```

Exercice 3 (Ocaml)**.** Pour chacune des listes d'instructions suivantes, deviner l'expression de OCaml qui l'a générée :

(i).

```
    const 3
    push
    const 3
    push
    acc 1
    divint
    push
    const 0
    ltint
    branchifnot L1
    const 1
    return 2
L1: const 2
    return 2
```

(ii).

```
    closure L2, 0
    push
    acc 0
    closure L1, 1
    push
    const 3
    push
    acc 1
    appterm 1, 4
L1: acc 0
    push
    envacc 1
    appterm 1, 2
L2: acc 0
    offsetint 1
    return 1
```

Exercice 4 (JVM)**.** Considérons une méthode `f` dans une classe `MaClasse` qui est de la forme

```
class MaClasse{
    public 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 :

— **donner** une trace de l'exécution de la JVM en indiquant l'évolution du PC et de la pile de calcul lorsque la méthode est appelée avec les paramètres `2` et `1`. Par exemple pour la première méthode ci-dessous cela donne (le haut de la pile est à gauche, p indique le contenu de la pile avant l'exécution de la méthode) :

```
PC:   Pile de calcul:
0      p
1      2,p
2      1,2,p
3      3,p
```

— **trouver** des instructions JAVA pour `f` qui génèrent le bytecode.

1.

```
public static int f(int, int);
    descriptor: (II)I
    Code:
      stack=2, locals=2, args_size=2
         0: iload_0
         1: iload_1
         2: iadd
         3: ireturn
```

2.

```
public static int f(int, int);
    descriptor: (II)I
    Code:
      stack=2, locals=2, args_size=2
         0: iload_0
         1: iload_1
         2: if_icmpne     7
         5: iload_0
         6: ireturn
         7: iconst_3
         8: iload_1
         9: imul
        10: ireturn
```

3.

```
public static int f(int, int);
    descriptor: (II)I
    Code:
      stack=2, locals=3, args_size=2
         0: iload_0
         1: istore_2
         2: iload_1
```

```
 3: iload_0
 4: if_icmpge     17
 7: iinc          1, 1
10: iload_2
11: iload_2
12: iadd
13: istore_2
14: goto          2
17: iload_2
18: ireturn
```

Exercice 5 (Compréhension JVM)**.**

— Expliquer brièvement pourquoi dans le bytecode dans l'exercice précédent les lignes ne sont pas toujours numérotées par des entiers successifs.

— Pour le code

```
public static int f(int);
  descriptor: (I)I
  Code:
    stack=???, locals=???, args_size=1
       0: iconst_3
       1: istore_2
       2: iload_0
       3: iconst_2
       4: iadd
       5: iload_2
       6: iload_2
       7: iadd
       8: iadd
       9: istore_1
      10: iload_1
      11: ireturn
```

indiquer la bonne valeur de `stack` et de `locals` et la valeur de retour de la méthode si on l'appelle avec `42`.

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

```
public static float f(int x){
   int y = 1;
   return (y + x) + 0.25f;
}
```

Cela correspond au bytecode :

```
public static float f(int);
  descriptor: (I)F
  Code:
    stack=2, locals=2, args_size=1
    ???
```

Compléter la partie manquante (une dizaine d'instructions).

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. 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 pc - 3, 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.

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.

modint Sets the accumulator to the modulo of the accumulator by the value popped from the stack.

divint Sets the accumulator to the division of the accumula-tor by the value popped from the stack. Raises 'zero divide' exception if the value popped from the stack is equal to 0.

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

ltint Sets the accumulator to a non-zero value or to zerowhether the accumulator is lower than the value popped fromthe 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.

Annexe JVM

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

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.

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.

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.

fadd Both value1 and value2 must be of type float. The values are popped from the operand stack. The float result is value1 + value2. 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.

i2f 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.

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

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.

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.

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