Université Paris Diderot
L3 Info

Machines Virtuelles
Année 2016-2017

Examen

mardi 16 mai 2017

Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints. Le temps à disposition est de **2h30**.

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

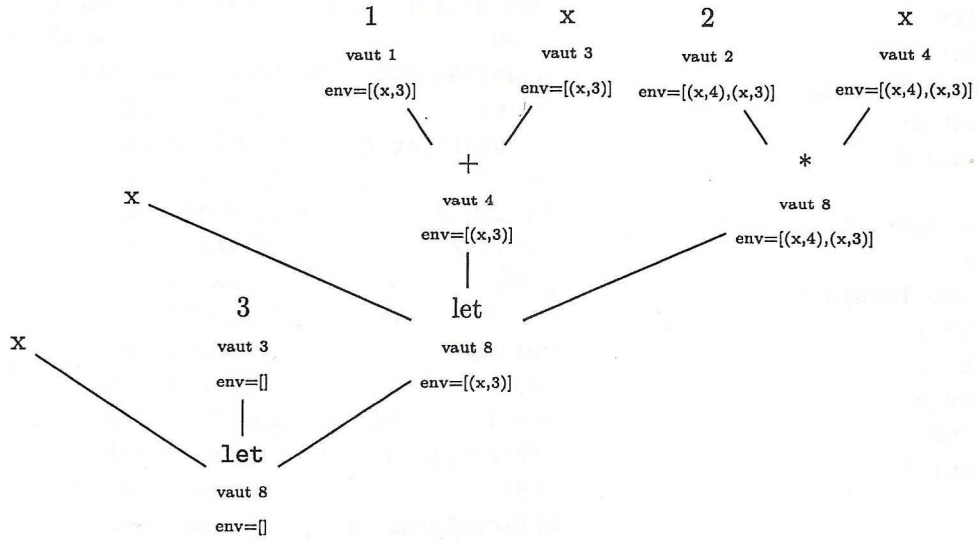
Exercice 1 (AST). Pour chacune des expressions suivantes, dessinez son arbre de syntaxe en annotant pour chaque sous-expression sa valeur et son environment.

1. `let x = 4 in 4*x`
2. `let x = 2 in if (let x = 3 in x = 2) then x+1 else 0`

Par exemple, l'expression

```
let x = 3 in let x = 1 + x in (2*x)
```

donne l'arbre de syntaxe annoté comme suit



Exercice 2 (Compilation à la main de OCaml). Pour chacune des expressions (1)-(2) du Exercice 1, donner un bytecode de OCamlrun equivalent. Par exemple, l'expression

```
let x = 3 in let x = 1 + x in (2*x)
```

corresponde au bytecode :

```
const 3   push   acc 0   push   const 1   addint
push   acc 0   push   const 2   mulint   return 3
```

Exercice 3 (Représentation données structurées). Supposons d'avoir défini un type `t` comme suit :

```
type t = A | B of t list | C of t*t | D | E of (t*t) list
```

Pour chacune des données structurées suivantes, donnez unes expression de ocaml correspondante :

1. `[0: [0: 1a [0: [1: 0a 0a] [0: 0a [0: [1: 1a 0a] 0a]]]]]`
2. `[0: 0 [0: [0: [2: 0a] [0: [2: [0: [0: 0a [1: 0a 1a]] 0a]] 0a]]]]`

Exercice 4. Pour chacune des listes d'instructions suivantes, deviner l'expression de OCaml qui l'a générée :

(i).

```
      closurerec 1, 0
      const 4
      push
      acc 1
      appterm 1, 3
L1:   acc 0
      push
      const 0
      eqint
      branchifnot L2
      const 1
      return 1
L2:   acc 0
      offsetint -1
      push
      offsetclosure 0
      apply 1
      push
      const 2
      mulint
      return 1
```

(ii).

```
      closurerec 1, 0
      const 1
      push
      const 4
      push
      acc 2
      appterm 2, 4
      restart
L1:   grab 1
      acc 0
      push
      const 0
      eqint
      branchifnot L2
      acc 1
      return 2
L2:   const 2
      push
      acc 2
      mulint
      push
      acc 1
      offsetint -1
      push
      offsetclosure 0
      appterm 2, 4
```

Exercice 5. Considerons une methode `f` dans une classe `MaClasse` qui est de la forme

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

Trouver des instructions JAVA pour `f` qui génèrent les byte-codes suivants. Décrire aussi leur exécution, lorsque les valeurs des paramètres de `f` sont respectivement `x = 4, y = 2`.

1.

```
public static int f(int, int);
    descriptor: (II)I
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: iload_0
         1: iconst_2
         2: irem
         3: ifne          8
         6: iload_1
         7: ireturn
         8: iload_1
         9: iload_1
        10: imul
        11: ireturn
```

2.

```
public static int f(int, int);
    descriptor: (II)I
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=4, args_size=2
         0: iconst_1
         1: istore_2
         2: iconst_2
         3: istore_3
         4: iload_3
         5: iload_0
         6: if_icmpgt     34
         9: iload_3
        10: iload_1
        11: if_icmpgt     34
        14: iload_0
        15: iload_3
        16: irem
        17: ifne          28
        20: iload_3
        21: iload_1
        22: irem
        23: ifne          28
        26: iload_3
        27: istore_2
        28: iinc          3, 1
        31: goto          4
        34: iload_2
        35: ireturn
```

Exercice 6 (Compilation à la main de JAVA). Soit la classe

```
public class IntArray{
    public static boolean isEmpty(int [] t){
        return t == null;
    }

    public static int sumLoop(int [] t, int fromIndex, int toIndex){
        int x = 0;
        while (fromIndex <= toIndex){
            x += t[fromIndex];
            fromIndex++;
        }
        return x;
    }

    public static int sumRec(int [] t, int fromIndex, int toIndex){
        if (fromIndex == toIndex) return t[fromIndex];
        else return t[fromIndex] + sumRec(t, fromIndex+1, toIndex);
    }
}
```

Traduire en bytecode de JVM les méthodes `isEmpty`, `sumLoop` et `sumRec`, en explicitant pour chaque méthode la dimension de la pile, le nombre des variables locales et des arguments.

Annexe OCamlrun

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

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.

closurerec ofs, n as closure ofs, n and push.

offsetclosure n Sets the accumulator to the value of the n-th closure relatively to the environment.

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.

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

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.

Annexe JVM

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 long. The index must be of type int. Both arrayref and index are popped from the operand stack. The long value in the component of the array at index is retrieved and 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.

areturn The objectref must be of type reference and must refer to an object of a type that is assignment compatible with the type represented by the return descriptor of the current method. If the current method is a synchronized method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a monitorexit instruction in the current thread. If no exception is thrown, objectref is popped from the operand stack of the current frame and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The interpreter then reinstates the frame of the invoker and returns control to the invoker.

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.

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

irem Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is $value1 - (value1/value2) * value2$. The result is pushed onto the operand stack. The result of the irem instruction is such that $(a/b) * b + (a\%b)$ is equal to a.

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, ne, lt, le, gt, ge.

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.

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.

invokestatic indexbyte1 indexbyte2 The unsigned indexbyte1 and indexbyte2 are used to construct an index into the run-time constant pool of the current class, where the value of the index is (indexbyte1 << 8) | indexbyte2. The run-time constant pool item at that index must be a symbolic reference to a method, which gives the name and descriptor of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved. The resolved method must not be an instance initialization method or the class or interface initialization method. It must be static, and therefore cannot be abstract.

On successful resolution of the method, the class that declared the resolved method is initialized if that class has not already been initialized.

The operand stack must contain nargs argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved method.

....

If the method is not native, the nargs argument values are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The nargs argument values are consecutively made the values of local variables of the new frame, with arg1 in local variable 0 (or, if arg1 is of type long or double, in local variables 0 and 1) and so on. Any argument value that is of a floating-point type undergoes value set conversion prior to being stored in a local variable. The new frame is then made current, and the Java Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

ireturn cfr areturn.

iload_n cfr aload_n.