Review: Printing Trees into Bytecodes

To evaluate $e_1 * e_2$ interpreter

- evaluates e_1
- evaluates e_2
- combines the result using *

Compiler for $e_1 * e_2$ emits:

- code for e_1 that leaves result on the stack, followed by
- code for e_2 that leaves result on the stack, followed by
- arithmetic instruction that takes values from the stack and leaves the result on the stack

```
def compile(e : Expr) : List[Bytecode] = e match { // ~ postfix printer
  case Var(id) => List(ILoad(slotFor(id)))
  case Plus(e1,e2)  => compile(e1) ::: compile(e2) ::: List(IAdd())
  case Times(e1,e2) => compile(e1) ::: compile(e2) ::: List(IMul())
  … }
```

Shorthand Notation for Translation

[e_1 + e_2] =
       [e_1]
       [e_2]
       iadd

[e_1 * e_2] =
       [e_1]
       [e_2]
       imul

Code Generation for Control Structures

Sequential Composition

How to compile statement sequence?

s1; s2; ... ; sN

- Concatenate byte codes for each statement!

```
def compileStmt(e : Stmt) : List[Bytecode] = e match {
  ...
  case Sequence(sts) =>
    for { st <- sts;  bcode <- compileStmt(st) }
      yield bcode
}
```

i.e.　　　　　　　sts **flatMap** compileStmt

semantically:　　(sts **map** compileStmt) **flatten**

Compiling Control: Example

static void count(**int** from,
                       int to,
                       int step) {

  int counter = from;

  while (counter < to) {

    counter = counter + step;

  }

}

We need to see how to:

- translate boolean expressions

- generate jumps for control

0: **iload**_0

1: **istore**_3

2: **iload**_3

3: **iload**_1

4: **if_icmpge**   14

7: **iload**_3

8: **iload**_2

9: **iadd**

10: **istore**_3

11: **goto**   2

14: **return**

Representing Booleans

Java bytecode verifier does not make hard distinction between booleans and ints

– can pass one as another in some cases if we hack .class files

As when compiling to assembly, we need to choose how to represent truth values

We adopt a **convention** in our code generation for JVM:

The generated code uses 'int' to represent boolean values in: **local variables**, **parameters**, and intermediate **stack values**.

In such cases, the code ensures that these int variables always either

– **0**, representing false

– **1**, representing true

Truth Values for Relations: Example

static boolean test(int x, int y){
  return (x < y);
}

0: **iload**_0

1: **iload**_1

2: **if_icmpge**   9

5: **iconst**_1

6: **goto**   10

9: **iconst**_0

10: **ireturn**

if_icmpge instruction from spec

if_icmp<cond>

Branch if int comparison succeeds

format: if_icmp<cond>

            branchbyte1

            branchbyte2

if_icmpeq = 159 (0x9f)

if_icmpne = 160 (0xa0)

if_icmplt = 161 (0xa1)

if_icmpge = 162 (0xa2)

if_icmpgt = 163 (0xa3)

if_icmple = 164 (0xa4)

Operand Stack:

        ..., value1, value2 → ...

Both value1 and value2 must be of type int. They are both popped from the operand stack and compared. All comparisons are signed.

The results of the comparison are as follows:

   if_icmpeq succeeds if and only if value1 = value2

   if_icmpne succeeds if and only if value1 ≠ value2

   if_icmplt succeeds if and only if value1 < value2

   if_icmple succeeds if and only if value1 ≤ value2

   if_icmpgt succeeds if and only if value1 > value2

   if_icmpge succeeds if and only if value1 ≥ value2

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.

Compiling Relational Expressions

```
def compile(e : Expr) : List[Bytecode] = e match { …
  case Times(e1,e2) => compile(e1) ::: compile(e2) ::: List(IMul())

  case Comparison(e1, op, e2) => {
    val nFalse = getFreshLabel(); val nAfter =  getFreshLabel()
      compile(e1)
    :::compile(e2)
    :::List(     if_icmp_instruction(converse(op), nFalse),
                 IConst1,

                 goto_instruction(nAfter),
label(nFalse), IConst0,

label(nAfter))       // result: 0 or 1 added to stack
  }
}
```

is there a **dual** translation?

A separate pass resolves labels before emitting class file

ifeq instruction from spec

if<cond>

Branch if int comparison with zero succeeds

    if<cond>

      branchbyte1

      branchbyte2

ifeq = 153 (0x99)

ifne = 154 (0x9a)

iflt = 155 (0x9b)

ifge = 156 (0x9c)

ifgt = 157 (0x9d)

ifle = 158 (0x9e)

Operand Stack

    ..., value →...

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 ≠ 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.

Compiling **If Statement**
using compilation of 0/1 for condition

```
def compileStmt(e : Stmt) : List[Bytecode] = e match {  …
    case If(cond,tStmt,fStmt) => {
        val nElse = getFreshLabel(); val nAfter = getFreshLabel()
        compile(cond)
      :::List(Ifeq(nElse))
      :::compileStmt(tStmt)
      :::List(goto(nAfter))
      :::List(label(nElse))
      :::compileStmt(eStmt)
      :::List(label(nAfter))
    }
}
```

Compiling If Statement
using compilation of 0/1 for condition

Shorthand math notation for the previous function:

[if (cond**)** tStmt **else** eStmt **] =**

       [cond **]**
       Ifeq(nElse)
       [tStmt **]**
       goto(**nAfter**)

nElse: **[** eStmt **]**
nAfter:

Compiling While Statement
using compilation of 0/1 for condition

[**while** (cond) stmt] =

nStart: [cond]
      Ifeq(**nExit**)
      [stmt]
      goto(**nStart**)

nExit:

give a translation with only one jump during loop

LOOP with EXIT IF

Oberon-2 has a statement

LOOP

  code1

  EXIT IF cond

  code2

END

which executes a loop and exits when the condition is met. This generalizes 'while' and 'do ... while'

Give a translation scheme for the LOOP construct.

Apply the translation to

$j = i$

LOOP

  $j = j + 1$

  EXIT IF $j > 10$

  $s = s + j$

END

$z = s + j - i$

solution

```
[ LOOP
  code1
    EXIT IF cond
  code2
END ] =
```

while loop: example

static boolean condition(int n)
{ ... }

static void work(int n) { ... }

static void test() {

  int n = 100;

  while (condition(n)) {

   n = n - 11;

   work(n);

  }

}

```
0:   bipush  100
2:   istore_0
3:   iload_0
4:   invokestatic #4;// condition:(I)Z
7:   ifeq   22
10:  iload_0
11:  bipush  11
13:  isub
14:  istore_0
15:  iload_0
16:  invokestatic    #5; work:(I)V
19:  goto   3
22:  return
```

Bitwise Operations

```
  10110
& 11011
= 10010


  10110
| 11011
= 11111
```

These operations always evalute both arguments.

- In contast, **&&** **||** operations only evaluate their second operand if necessary!

- We must compile this correctly. It is not acceptable to emit code that always evaluates both operands of &&,||

What does this program do?

```
static boolean bigFraction(int x, int y) {
  return ((y==0) | (x/y > 100));
}
public static void main(String[] args) {
  boolean is = bigFraction(10,0);
}
```

should be **||**

Exception in thread "main" java.lang.ArithmeticException: **/ by zero**
    at Test.bigFraction(Test.java:4)
    at Test.main(Test.java:19)

What does this function do?

```
static int iterate() {
        int[] a = new int[10];
        int i = 0;
        int res = 0;
        while ((i < a.length) & (a[i] >= 0)) {
            i = i + 1;
            res = res + 1;
        }
        return res;
}
```

should be **&&**

Exception in thread "main" java.lang.**ArrayIndexOutOfBoundsException**: 10
        at Test.iterate(Test.java:16)
        at Test.main(Test.java:25)

Compiling Bitwise Operations - Easy

[e_1 **&** e_2] =
    [e_1]
    [e_2]
    iand

[e_1 **|** e_2] =
    [e_1]
    [e_2]
    ior

[e_1 **&&** e_2] =
    [e_1]
    [e_2] ← not allowed to evaluate e_2 if e_1 is **false**!
    ...

Also for (e1 || e2): if e1 **true**, e2 not evaluated

Conditional Expression

Scala:

    if (c) t **else** e

Java, C:

    c ? t : e

Meaning:

- c is evaluated
- if c is true, then t is evaluated and returned
- if c is false, then e is evaluated and returned

- Meaning of **&&**, **||**:

(p && q) ==
    if (p) q **else false**

(p **||** q) ==
    if (p) **true else** q

- To compile ||,&& transform them into 'if' **expression**

Compiling **If Expression**

- Same as for if statement, even though code for branches will leave values on the stack:

[if (cond**)** t **else** e **]** =

        [cond **]**
        Ifeq(nElse)
        [t **]**
        goto(**nAfter**)

nElse: **[** e **]**
nAfter:

Java Example

int f(**boolean** c, **int** x, **int** y) {
  return (c ? x : y);
}

```
0:   iload_1
1:   ifeq   8
4:   iload_2
5:   goto   9
8:   iload_3
9:   ireturn
```

Compiling **&&**

[if (cond**)** t **else** e **]** =

      [cond **]**

      Ifeq(nElse)

      [t **]**

      goto(**nAfter**)

nElse: **[** e **]**

nAfter:

[p **&&** q **]** =

[if (p) q **else false]** =

      [p **]**

      Ifeq(nElse)

      [q **]**

      goto(**nAfter**)

nElse: **iconst**_0

nAfter:

Compiling ||

[**if (**cond**)** t **else** e] =

    [cond]
    Ifeq(nElse)
    [t]
    goto(**nAfter**)

nElse:[e]
nAfter:

[p **||** q] =
[**if** (p) **true else** q] =

    [p]
    Ifeq(nElse)
    iconst_1
    goto(**nAfter**)

nElse:[q]
nAfter:

true, false, variables

[true] =
    iconst_1

[false] =
    iconst_0

for boolean variable b, for which n = slot(b)

[b **]** =
    iload_n

[b = e **]** =   (assignment)
    [e **]**
    istore_n

Example: triple **&&**

Let x,y,z be in slots 1,2,3

Show code for assignment

y = (x && y) && z

Does the sequence differ for assignment

y = x && (y && z)

	iload_1
	ifeq n1
	iload_2
	goto n2
n1:	**iconst**_0
n2:	**ifeq** n3
	iload_3
	goto n4
n3:	**iconst**_0
n4:	

Cool Alternative:
Compiling by Tree Rewriting

Code Compiled with javac

```
static int k = 0;
static boolean action(int si,
                boolean ob,
                int sm, int pr) {
    if (sm + 2*pr > 10 &&
    !(si <= 5 && ob)) {
        k++; return true;
    } else {
        return false;
    } }
```

Compared to our current translation:

if 'sm+2*pr > 10' false, immediately ireturns

if 'si > 5' is true, immediately goes to 'then' part

no intermediate result for if condition - do branches directly

negation sign eliminated and pushed through

only one iconst_0 and one iconst_1

```
0:  iload_2
1:  iconst_2
2:  iload_3
3:  imul
4:  iadd
5:  bipush  10
7:  if_icmple     29
10: iload_0
11: iconst_5
12: if_icmpgt     19
15: iload_1
16: ifne   29
19: getstatic     #2; //Field k
22: iconst_1
23: iadd
24: putstatic     #2; //Field k
27: iconst_1
28: ireturn
29: iconst_0
30: ireturn
```

Solution: macro 'branch' instruction

Introduce an imaginary big instruction

branch(c,nThen,nElse)

Here

    c is a potentially complex Java boolean expression

    nThen is label to jump to when c evaluates to true

    nFalse is label to jump to when c evaluates to false

We show how to:

- use **branch** to compile if, while, etc.

- expand **branch** recursively into concrete bytecodes

Using **branch** in Compilation

[**if** (c) t **else** e] =
    branch(c,nThen,nElse)
nThen: [t]
        goto nAfter
nElse: [e]
nAfter:

[**while** (c) s] =
lBegin: branch(c,start,lExit)
start: [s]
          goto lBegin
lExit:

Decomposing **branch**

branch(**!**c,nThen,nElse) =
    branch(c,nElse,nThen)

branch(c1 **&&** c2,nThen,nElse) =
    branch(c1,nNext,nElse)
nNext: branch(c2,nThen,nElse)

branch(c1 **||** c2,nThen,nElse) =
    branch(c1,nThen,nNext)
nNext: branch(c2,nThen,nElse)

branch(**true**,nThen,nElse) =
    goto nThen

branch(**false**,nThen,nElse) =
    goto nElse

boolean var b with slot N
branch(b,nThen,nElse) =
  iload_N
  ifeq nElse
  goto nThen

Compiling Relations

branch(e1 **R** e2,nThen,nElse) =

    [e1 **]**

    [e2 **]**

    if_cmpR nThen

    goto nElse

Putting boolean variable on the stack

Consider storing   x = c

where x,c are boolean and c has &&,||

How to put result of **branch** on stack to allow istore?

[c **]** =

        branch(c,nThen,nElse)

nThen: **iconst**_1

        goto nAfter

nElse: **iconst**_0

nAfter:

Compare Two [...] on This While Loop

```
static void count(int from,
                  int to,
                  int step) {
  int counter = from;
  while (counter < to) {
    counter = counter + step;
  }
}
```

Exercise: break statement

A common way to exit from a loop is to use a 'break' statement e.g.

```
while (true) {
  code1
  if (cond) break
  cond2
}
```

Consider a language that has expressions, assignments, the {...} blocks, 'if' statements, while, and a 'break' statement. The 'break' exits the innermost loop and can appear inside arbitrarily complex blocks and if conditions. How would translation scheme for such construct look like?

Destination Parameters in Compilation

- To compilation functions **[…]** pass the label to which instructions should jump when they finish

[x = e **]** dest =   // new parameter dest

   [e **]**

   istore_slot(x)

   goto dest   // at the end jump to it

[s1 ; s2 **]** dest brk =

           [s1 **]** freshL

   freshL: **[** s2 **]** dest

we could have any junk in between
because [s1] freshL will do the jump

More Control, More Destination Paramameters

[s1 ; s2 **]** dest brk =

             [s1 **]** freshL brk

  freshL: **[** s2 **]** dest brk

[x = e **]** dest brk =

  [e **]**

  istore_slot(x)

  goto dest

[break] dest brk =

  goto brk

[while (c) s **]** dest brk =

test: **branch**(c,body,dest)
body: **[** s **]** dest dest

this is where the second parameter gets bound to the exit of the loop