SWEN430 - Compiler Engineering

Lecture 13 - Bytecode Generation II

Erin Greenwood-Thessman

with thanks to Linsday Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Generating Invoke Bytecodes

`invokevirtual`	[1 byte op][2 bytes index] *Invoke method on a receiver of class type. The method and receiver types are located in the constant pool at the given index.*
`invokeinterface`	[1 byte op][2 bytes index] *Invoke method on a receiver of interface type. The method and receiver types are located in the constant pool at the given index.*
`invokestatic`	[1 byte op][2 bytes index] *Invoke static method. The method and receiver types are located in the constant pool at the given index.*
`invokespecial`	[1 byte op][2 bytes index] *Invoke special method (e.g. constructor). The method and receiver types are located in the constant pool at the given index.*

Generating Invoke Bytecodes (Cont'd)

```
class Test {
 Test(int x) { }
 int f(String s, int i) {
   return 1;
 }

 static void m(String[] s){
   Test t = new Test(123);
   t.f(s[0],2);
 }
}
```

```
static void m(String[] s):
Code:
  0:   new Test
  3:   dup
  4:   bipush  123
  6:   invokespecial Test.<init>:(I)V
  9:   astore_1
  10:  aload_1
  11:  aload_0
  12:  iconst_0
  13:  aaload
  14:  iconst_2
  15:  invokevirtual Test.f:(L...;I)I
  18:  pop
  19:  return
```

- Receiver pushed on stack first (line 10)
- Parameters pushed on stack next in order (lines 13-14)
- Return value is popped afterwards since its not used (line 18)

Determining Maximum Stack Height

- Must determine **maximum stack height** of each method
- Java Compiler can calculate the **stack difference** for each bytecode:
- Examples:

Bytecode	Stack Difference	Bytecode	Stack Difference
`bipush`	+1	`pop`	-1
`iload X`	+1	`lload X`	+2
`iadd`	-1	`dadd`	-2
`iaload`	-1	`daload`	0
`ineg`	0	`d2f`	-1
`invokevirtual`	???		

- Then, it traverses bytecode sequence determining the max height
- How should we deal with branching?

Determining Maximum Stack Height (cont'd)

```
int f(java.lang.String[]);
Code:                              # diff # height
 0:   aload_1                      #  +1  #  1  #
 1:   ifnull  12                   #  -1  #  0  #
 4:   getstatic System.out         #  +1  #  1  #
 7:   ldc "Hello World"            #  +1  #  2  #
 9:   invokevirtual println:(Ljava/lang/String;)V
                                   #  -2  #  0  #
 12:  iconst_0                     #  +1  #  1  #
 13:  istore_2                     #  -1  #  0  #
 14:  iload_2                      #  +1  #  1  #
 15:  aload_1                      #  +1  #  2  #
 16:  arraylength                  #  0   #  2  #
 17:  iadd                         #  -1  #  1  #
 18:  istore_2                     #  -1  #  0  #
 19:  iload_2                      #  +1  #  1  #
 20:  ireturn                      #  -1  #  0  #
```

- Hence, the maximum stack height for this method is 2

Line Number Information

```
int f(int x, int y) {
  int r = x / y;
  return r;
}
```

```
int f(int, int);
   0:   iload_1
   1:   iload_2
   2:   idiv
   3:   istore_3
   4:   iload_3
   5:   ireturn
 LineNumberTable:
   line 3: 0
   line 4: 4
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.f(Test.java:3)
        at Test.main(Test.java:8)
```

- Can associate **line number information** with Bytecode
- `LineNumberTable` attribute is for this (see JVM Spec §4.7.8)
- Then can see **where** exceptions occur in original source file

Type Information

```
int f(int, int);
   0:   iload_1
   1:   iload_2
   2:   idiv
   3:   istore_3
   4:   iload_3
   5:   ireturn
   ...
  LocalVariableTable:
     Start  Length  Slot  Name   Signature
         0       6     0  this   Ltest;
         0       6     1     x   I
         0       6     2     y   I
         4       2     3     r   I
```

```
int f(int x, int y) {
  int r = x / y;
  return r;
}
```

- `LocalVariableTable` identifies **type** of each slot at each point

Exception Handlers

```
String f(Integer i) {
 try {
  return i.toString();
 } catch(Exception e){
  return "";
}}
```

```
String f(Integer);
  Code:
   Stack=1, Locals=3
   0:   aload_1
   1:   invokevirtual Integer.toString()
   4:   areturn
   5:   astore_2
   6:   ldc     ""
   8:   areturn
  Exception table:
   from   to  target type
     0     4     5   Class Exception
```

- Exception handlers implemented as table rows:
  - » Range of bytecodes, destination and class
  - » Range cannot include handler itself

Exception Handlers Table

- Java Compiler generates exception handlers such that:
  - » Either exception handler ranges are **disjoint**, or one is a **subrange** of the other
  - » Exception handler code is never **within its own range**
  - » Entry for exception handler only via exception (not via e.g. `goto`)
- Surprisingly, these restrictions **not enforced** by bytecode verifier
  - » Because not considered a threat to integrity of JVM
  - » Still require that e.g. every nonexceptional path to handler has a single object on the operand stack, etc
  - » See JVM Specification, §3.10

Another Example

```
String f(Integer i) {
 try {
  return i.toString();
 } catch(NullPointerException e){
  return "null";
 } catch(Exception e){
  return "";
}}
```

```
String f(Integer);
  Code:
   Stack=1, Locals=3, Args_size=2
   0:   aload_1
   1:   invokevirtual Integer.toString:()
   4:   areturn
   5:   astore_2
   6:   ldc  "null"
   8:   areturn
   9:   astore_2
   10:  ldc     ""
   12:  areturn
  Exception table:
   from   to  target type
     0     4     5   Class NullPointerException

     0     4     9   Class Exception
```

- Multiple Exception handlers are triggered **in order of appearance**

Translating While: Type Representation

WHILE	JVM	Notes
`bool`	`JvmTypes.Bool`	Compiles to `int` in some cases.
`int`	`JvmTypes.Int`	
`T[]`	`List<Object>`	Boxing / unboxing
`{T f, ... }`	`Map<String,Object>`	Boxing / unboxing

- **Suggested mapping** from WHILE to JVM types (above)
- You will encounter an **impedance mismatch** though!
- For example, how to **translate** "`x = [1,2,3]`" **...** ?

Translating While: Boxing / Unboxing

WHILE	JAVA

`type Rec is {int g}`
 	
`int y = 1;`
`Rec xs = { f: y+1 };`
	`int y = 1;`
	`Map<String,Object> xs = ...;`
	`xs.put("f",y+1);`

- **Compiles** in JAVA ...

  ... but on JVM is more **complicated** ...

  Why?

Translating While: Boxing / Unboxing

```
 0: iconst_1
 1: istore_1                        // y = 1
 2: new   java/util/HashMap
 5: dup
 6: invokespecial java/util/HashMap."<init>":()V
 9: astore_2
10: aload_2
11: ldc   "f"
13: iload_1
14: iconst_1
15: iadd                           // y + 1
16: invokestatic  java/lang/Integer.valueOf:(I);
19: invokeinterface java/util/Map.put:(Object;Object;);
24: pop
25: return
```

Translating While: Cloning

WHILE	JAVA

`int[] xs = [1,2];`
`int[] ys = xs;`
`ys[0] = 3;`
	`List xs = new ArrayList();`
	`xs.add(1);`
	`xs.add(2);`
	`List ys = xs;`
	`ys.set(0,3);`

- Above is **broken** ... *why?*
- **Remember**: WHILE has **value semantics** for arrays ...

  ... but JVM does **not**!