JVM Code Generation

Outline

Introduction

Once the AST has been fully analyzed, all variables and expressions have been typed, any necessary tree rewriting has been done, and a certain amount of setup needed for code generation has been accomplished

The compiler is now ready to traverse the AST one more time to generate the Java Virtual Machine (JVM) code, ie, build the class file for the program

For example, consider the following very simple program

```
public class Square {
    public int square(int x) {
        return x * x;
    }
}
```

Compiling the program with our *j--* compiler

```
$ $j/j--/bin/j-- Square.java
```

produces a class file `Square.class`

Running the `javap` program on the class file

```
$ javap -verbose Square
```

produces the symbolic representation of the file shown in the next slide

Introduction

```
public class Square extends java.lang.Object
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Asciz    Square;
const #2 = class    #1; //  Square
const #3 = Asciz    java/lang/Object;
const #4 = class    #3; //  java/lang/Object
const #5 = Asciz    <init>;
const #6 = Asciz    ()V;
const #7 = NameAndType  #5:#6;//  "<init>":()V
const #8 = Method   #4.#7;  //  java/lang/Object."<init>":()V
const #9 = Asciz    Code;
const #10 = Asciz   square;
const #11 = Asciz   (I)I;

{
public Square();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial #8; //Method java/lang/Object."<init>":()V
   4: return

public int square(int);
  Code:
   Stack=2, Locals=2, Args_size=2
   0: iload_1
   1: iload_1
   2: imul
   3: ireturn

}
```

Introduction

To emit JVM instructions, we firstly create a `CLEmitter` instance, which is an abstraction of the class file we wish to build, and then call upon `CLEmitter`'s methods for generating the necessary headers and instructions

For example, to generate the class header

```
public class Square extends java.lang.Object
```

we would invoke the `addClass()` method on `output`, an instance of `CLEmitter`

```
output.addClass(mods, "Square", "java/lang/Object", null, false);
```

As another example, the no-argument instruction `aload_1` may be generated by

```
output.addNoArgInstruction(ALOAD_1);
```

Introduction

For a more involved example of code generation, consider the `Factorial` program from before

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field

    public static int factorial(int n) {
        // position 1:
        if (n <= 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int x = n;

        // position 2:
        System.out.println(n + "! = " + factorial(x));
    }

    static int n = 5;
}
```

Introduction

Running `javap` on `Factorial.class` produced by the *j--* compiler gives us

```
public class pass.Factorial extends java.lang.Object
  minor version: 0
  major version: 49
  Constant pool:
  ...
{
static int n;

public pass.Factorial();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial #8; //Method java/lang/Object."<init>":()V
   4: return

public static int factorial(int);
  Code:
   Stack=3, Locals=1, Args_size=1
   0: iload_0
   1: iconst_0
   2: if_icmpgt 10
   5: iconst_1
   6: ireturn
   7: goto  19
   10: iload_0
   11: iload_0
   12: iconst_1
   13: isub
   14: invokestatic #13; //Method factorial:(I)I
   17: imul
   18: ireturn
   19: nop
```

Introduction

```
public static void main(java.lang.String[]);
  Code:
   Stack=3, Locals=2, Args_size=1
   0: getstatic #19; //Field n:I
   3: istore_1
   4: getstatic #25; //Field java/lang/System.out:Ljava/io/PrintStream;
   7: new   #27; //class java/lang/StringBuilder
   10: dup
   11: invokespecial    #28; //Method java/lang/StringBuilder."<init>":()V
   14: getstatic    #19; //Field n:I
   17: invokevirtual    #32; //Method java/lang/StringBuilder.append:
                             (I)Ljava/lang/StringBuilder;
   20: ldc  #34; //String ! =
   22: invokevirtual    #37; //Method java/lang/StringBuilder.append:
                             (Ljava/lang/String;)Ljava/lang/StringBuilder;
   25: iload_1
   26: invokestatic #13; //Method factorial:(I)I
   29: invokevirtual    #32; //Method java/lang/StringBuilder.append:
                             (I)Ljava/lang/StringBuilder;
   32: invokevirtual    #41; //Method java/lang/StringBuilder.toString:
                             ()Ljava/lang/String;
   35: invokevirtual    #47; //Method java/io/PrintStream.println:
                             (Ljava/lang/String;)V
   38: return

public static {};
  Code:
   Stack=2, Locals=0, Args_size=0
   0: iconst_5
   1: putstatic #19; //Field n:I
   4: return
}
```

Generating Code for Classes and their Members

`JCompilationUnit.codegen()` drives the generation of code for classes; for each type (ie, class) declaration, it

- invokes `codegen()` on the `JClassDeclaration` for generating the code for that class,
- writes out the class to a class file in the destination directory, and
- adds the in-memory representation of the class to a list that stores such representations for all the classes within a compilation unit; this list is used in translating JVM byte code to native (SPIM) code

```
public void codegen(CLEmitter output) {
    for (JAST typeDeclaration : typeDeclarations) {
         typeDeclaration.codegen(output);
         output.write();
         clFiles.add(output.clFile());
    }
}
```

Generating Code for Classes and their Members

`JClassDeclaration.codegen()` does the following

- It computes the fully-qualified name for the class, taking any package name into account
- It invokes an `addClass()` on the `CLEmitter` for adding the class header to the start of the class file
- If there is no explicit constructor with no arguments defined for the class, it invokes the private method `codegenImplicitConstructor()` to generate code for the implicit constructor as required by the language
- It generates code for its members, by sending the `codegen()` message to each of them.
- If there are any static field initializations in the class declaration, then it invokes the private method `codegenClassInit()` to generate the code necessary for defining a static block, a block of code that is executed after a class is loaded

Generating Code for Classes and their Members

JMethodDeclaration.codegen()

```
public void codegen(CLEmitter output) {
    output.addMethod(mods, name, descriptor, null, false);
    if (body != null) {
        body.codegen(output);
    }

    // Add implicit RETURN
    if (returnType == Type.VOID) {
        output.addNoArgInstruction(RETURN);
    }
}
```

JConstructorDeclaration.codegen()

```
public void codegen(CLEmitter output) {
    output.addMethod(mods, "<init>", descriptor, null, false);
    if (!invokesConstructor) {
        output.addNoArgInstruction(ALOAD_0);
        output.addMemberAccessInstruction(INVOKESPECIAL,
                ((JTypeDecl) context.classContext().definition())
                        .superType().jvmName(), "<init>", "()V");
    }
    // Field initializations
    for (JFieldDeclaration field :
             definingClass.instanceFieldInitializations()) {
        field.codegenInitializations(output);
    }
    // And then the body
    body.codegen(output);
    output.addNoArgInstruction(RETURN);
}
```

Generating Code for Classes and their Members

Since the analysis phase has moved initializations, `codegen()` for `JFieldDeclaration` need only generate code for the field declaration itself

JFieldDeclaration.codegen()

```java
public void codegen(CLEmitter output) {
    for (JVariableDeclarator decl : decls) {
        // Add field to class
        output.addField(mods, decl.name(), decl.type()
            .toDescriptor(), false);
    }
}
```

Generating Code for Control and Logical Expressions

Almost all control statements in *j--* are controlled by some Boolean expression

For example, consider the if-then-else statement below

```
if (a > b) { c = a; } else { c = b; }
```

The JVM code produced for the statement is as follows

```
0: iload_1
1: iload_2
2: if_icmple 10
5: iload_1
6: istore_3
7: goto  12
10: iload_2
11: istore_3
12: ...
```

Notice a couple of things

1. We don't compute a Boolean value onto the stack and then branch on its value, but make use of the underlying JVM instruction set, which makes for more compact code
2. We branch to the else-part if the condition is `false`

```
branch to elseLabel if <condition> is false
    <code for thenPart>
    branch to endLabel
elseLabel:
    <code for elsePart>
endLabel:
```

Generating Code for Control and Logical Expressions

Suppose we wish implement the Java do-while statement in *j--*; for example

```
do {
    a++;
}
while (a < b);
```

The code we generate might have the form

```
topLabel:
    <code for body>
    branch to topLabel if <condition> is true
```

Note that we branch when the condition is `true`

In generating code for a condition, one needs a method specifying three arguments

1. The `CLEmitter` instance
2. The target label for the branch
3. A `boolean` flag `onTrue`; if `onTrue` is `true` then the branch should be made on the condition, and if `false`, the branch should be made on the condition's complement

Thus, every boolean expression must support a version of `codegen()` with these three arguments; for example, here is that overloaded `codegen()` method for `JGreaterThanOp`

```
public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
    lhs.codegen(output);
    rhs.codegen(output);
    output.addBranchInstruction(onTrue ? IF_ICMPGT : IF_ICMPLE, targetLabel);
}
```

Generating Code for Control and Logical Expressions

The three-argument `codegen()` method is invoked on the condition controlling execution

For example, the `codegen()` method in `JIfStatement` makes use of the three-argument `codegen()` method in producing code for the if-then-else statement

```
public void codegen(CLEmitter output) {
    String elseLabel = output.createLabel();
    String endLabel = output.createLabel();
    condition.codegen(output, elseLabel, false);
    thenPart.codegen(output);
    if (elsePart != null) {
        output.addBranchInstruction(GOTO, endLabel);
    }
    output.addLabel(elseLabel);
    if (elsePart != null) {
        elsePart.codegen(output);
        output.addLabel(endLabel);
    }
}
```

Generating Code for Control and Logical Expressions

The semantics of Java, and so of *j--*, requires that the evaluation of expressions such as `arg1 && arg2` be short-circuited, ie, if `arg1` is `false`, then `arg2` is not evaluated

The code to be generated depends of whether the branch for the entire expression is to be made on `true`, or on `false`

```
Branch to target when           Branch to target when
    arg1 && arg2 is true:                arg1 && arg2 is false:

    branch to skip if               branch to target if
        arg1 is false                    arg1 is false
    branch to target when           branch to target if
        arg2 is true                     arg2 is false
skip: ...
```

Generating Code for Control and Logical Expressions

For example, the code generated for

```
if (a > b && b > c) { c = a; } else { c = b; }
```

would be

```
0: iload_1
1: iload_2
2: if_icmple 15
5: iload_2
6: iload_3
7: if_icmple 15
10: iload_1
11: istore_3
12: goto 17
15: iload_2
16: istore_3
17: ...
```

The `codegen()` method in `JLogicalAndOp`

```
public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
    if (onTrue) {
        String falseLabel = output.createLabel();
        lhs.codegen(output, falseLabel, false);
        rhs.codegen(output, targetLabel, true);
        output.addLabel(falseLabel);
    } else {
        lhs.codegen(output, targetLabel, false);
        rhs.codegen(output, targetLabel, false);
    }
}
```

Generating Code for Control and Logical Expressions

Notice that our method prevents unnecessary branches to branches; for example, consider the slightly more complicated condition in

```
if (a > b && b > c && c > 5) { c = a; } else { c = b; }
```

The JVM code produced for this targets the same exit on `false`, for each of the `&&` operations

```
0:   iload_1
1:   iload_2
2:   if_icmple       18
5:   iload_2
6:   iload_3
7:   if_icmple       18
10:  iload_3
11:  iconst_5
12:  if_icmple       18
15:  iinc    1, -1
18:  ...
```

The `codegen()` method in `JLogicalNotOp`

```
public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
    arg.codegen(output, targetLabel, !onTrue);
}
```

Generating Code for Message Expressions, Field Selection, and Array Expressions

The `codegen()` method in `JMessageExpression` proceeds as follows

1. If the message expression involves an instance message, `codegen()` generates code for the target
2. The message invocation instruction is determined: `invokevirtual` for instance messages and `invokestatic` for static messages
3. The `addMemberAccessInstruction()` method is invoked to generate the message invocation instruction; this method takes the following arguments
   - (a) The instruction (`invokevirtual` or `invokestatic`)
   - (b) The JVM name for the target's type
   - (c) The message name
   - (d) The descriptor of the invoked method, which was determined in analysis.
4. If the message expression is being used as a statement expression and the return type of the method is non-void, then the method `addNoArgInstruction()` is invoked for generating a `pop` instruction; this is necessary because executing the message expression will produce a result on top of the stack, and this result is to be thrown away

Generating Code for Message Expressions, Field Selection, and Array Expressions

For example, the code generated for

```
... = s.square(6);
```

would be

```
aload s' # s' denotes offset of s
bipush  6
invokevirtual   #6; //Method square:(I)I
```

whereas the code generated for

```
s.square(6);
```

would be

```
aload s'
bipush  6
invokevirtual   #6; //Method square:(I)I
pop
```

We invoke static methods using the `invokestatic` instruction; for example the following *j--* code

```
... = Square.square(5);
```

where `int square(int)` is a static method in `Square`, would generate the following JVM code

```
iconst_5
invokestatic    #5; //Method square:(I)I
```

Generating Code for Message Expressions, Field Selection, and Array Expressions

The `codegen()` method in `JFieldSelection` works as follows

1. It generates code for its target; if the target is a class, no code is generated
2. The compiler must again treat the special case, `a.length` where `a` is an array; the code generated makes use of the special instruction, `arraylength`
3. Otherwise, it is treated as a proper field selection; the field selection instruction is determined: `getfield` for instance fields and `getstatic` for static fields
4. The `addMemberAccessInstruction()` method is invoked with the following arguments
   - (a) The instruction (`getfield` or `getstatic`)
   - (b) The JVM name for the target's type
   - (c) The field name
   - (d) The JVM descriptor for the type of the field, and so the type of the result

Generating Code for Message Expressions, Field Selection, and Array Expressions

For example, the following code

```
... = s.instanceField;
```

would be translated as

```
aload s'
getfield instanceField
```

whereas the following code

```
... = Square.staticField;
```

would be translated as

```
getstatic staticField
```

Code generation for array access expressions is straightforward; for example, if the variable `a` references an array object, and `i` is an integer, then the following code

```
... = a[i];
```

is translated to

```
aload a'
iload i'
iaload
```

Generating Code for Assignment and Similar Operations

Consider the simple assignment statement

```
x = y;
```

which asks that the value of the variable y be stored in variable x

We want the l-value (address or location) for x and the r-value (content or value) for y

All expressions have r-values, but many have no l-values; for example, if a is an array of ten integers, and o is an object with field f, C is a class with static field sf, and x is a local variable, the following have both l-values and r-values

```
a[3]
o.f
C.sf
x
```

while the following have r-values, but not l-values

```
5
x+5
Factorial.factorial(5)
```

Generating Code for Assignment and Similar Operations

The right-hand-side expression is compiled to produce code for computing its r-value and leaving it on the stack

For the left-hand-side, sometimes no code needs to be generated, as in the following example

```
x = y;
```

produces

```
iload y'
istore x'
```

On the other hand, compiling

```
a[x] = y;
```

produces

```
aload a'
iload x'
iload y'
iastore
```

An assignment may act as a statement, as shown below

```
x = y;
```

or as an expression, as shown below

```
z = x = y;
```

Generating Code for Assignment and Similar Operations

In the first case, no value is left on the stack

In the second case, `x = y` must assign the value of `y` to `x` but also leave a value (the r-value for `y`) on the stack so that it may be popped off and assigned to `z`, ie, the code might look something like

```
iload y'
dup
istore x'
istore z'
```

In parsing, when an expression is used as a statement, `Parser`'s `statementExpression()` method sets a flag `isStatementExpression` in the expression node to `true`, and the code generation phase makes use of this flag in deciding when code must be produced for duplicating r-values on the run-time stack

The most important property of the assignment is its side effect; one uses the assignment operation for its side effect: overwriting a variable's r-value with another

There are several Java (and *j*--) operations having the same sort of side effect; for example,

```
x--
++x
x += 6
```

Generating Code for Assignment and Similar Operations

The table below compares the various operations (labeled down the left), with an assortment of left-hand sides (labeled across the top)

	x	a[i]	o.f	C.sf
lhs = y	iload y' [dup] istore x'	aload a' iload i' iload y' [dup_x2] iastore	aload o' iload y [dup_x1] putfield f	iload y' [dup] putstatic sf
lhs += y	iload x' iload y' iadd [dup] istore x'	aload a' iload i' dup2 iaload iload y' iadd [dup_x2] iastore	aload o' dup getfield f iload y' iadd [dup_x1] putfield f	getstatic sf iload y' iadd [dup] putstatic sf
++lhs	iinc x',1 [iload x']	aload a' iload i' dup2 iaload iconst_1 iadd [dup_x2] iastore	aload o' dup getfield f iconst_1 iadd [dup_x1] putfield f	getstatic sf iconst_1 iadd [dup] putstatic sf
lhs--	[iload x'] iinc x',-1	aload a' iload i' dup2 iaload [dup_x2] iconst_1 isub iastore	aload o' dup getfield f [dup_x1] iconst_1 isub putfield f	getstatic sf [dup] iconst_1 isub putstatic sf

The instructions in brackets [...] must be generated if and only if the operation is a sub-expression of some other expression, ie, if the operation is not a statement expression

Generating Code for Assignment and Similar Operations

The table above suggests four sub-operations common to most of the assignment-like operations in *j--*

1. `codegenLoadLhsLvalue()` - this generates code to load any up-front data for the left-hand side of an assignment needed for an eventual store, ie, its l-value
2. `codegenLoadLhsRvalue()` - this generates code to load the r-value of the left-hand side, needed for implementing, for example the `+=` operator
3. `codegenDuplicateRvalue()` - this generates code to duplicate an r-value on the stack and put it in a place where it will be on top of the stack once the store is executed
4. `codegenStore()` - this generates the code necessary to perform the actual store

The code needed for each of these differs for each potential left-hand side of an assignment: a simple local variable `x`, an indexed array element `a[i]`, an instance field `o.f`, and a static field `C.sf`

The code necessary for each of the four operations, and for each left-hand-side form, is illustrated in the table below

	`x`	`a[i]`	`o.f`	`C.sf`

`codegenLoadLhsLvalue()`	`[none]`	`aload a'`
		`iload i'`
			`aload o'`	`[none]`

`codegenLoadLhsRvalue()`	`iload x'`	`dup2`
		`iaload`
			`dup`
			`getfield f`
				`getstatic sf`

`codegenDuplicateRvalue()`	`dup`	`dup_x2`	`dup_x1`	`dup`
`codegenStore()`	`istore x'`	`iastore`	`putfield f`	`putstatic sf`

Generating Code for Assignment and Similar Operations

Our compiler defines an interface `JLhs`, which declares four abstract methods for these four sub-operations; each of `JVariable`, `JArrayExpression` and `JFieldSelection` implements `JLhs`

Of course, one must also be able to generate code for the right-hand side expression, but `codegen()` is sufficient for that

For example, `JPlusAssignOp`'s `codegen()` is shown below

```java
public void codegen(CLEmitter output) {
    ((JLhs) lhs).codegenLoadLhsLvalue(output);
    if (lhs.type().equals(Type.STRING)) {
        rhs.codegen(output);
    } else {
        ((JLhs) lhs).codegenLoadLhsRvalue(output);
        rhs.codegen(output);
        output.addNoArgInstruction(IADD);
    }
    if (!isStatementExpression) {
        // Generate code to leave the r-value atop stack
        ((JLhs) lhs).codegenDuplicateRvalue(output);
    }
    ((JLhs) lhs).codegenStore(output);
}
```

Generating Code for String Concatenation

In *j--*, as in Java, the binary `+` operator is overloaded; if both of its operands are integers, it denotes addition, but if either operand is a string then the operator denotes string concatenation and the result is a string

The compiler's analysis phase determines whether or not string concatenation is implied, and when it is, the concatenation is made explicit, ie, the operation's AST is rewritten, replacing `JPlusOp` with a `JStringConcatenationOp`

Also, when `x` is a string, analysis replaces

```
x += <expression>
```

by

```
x = x + <expression>
```

Generating Code for String Concatenation

To implement string concatenation, the compiler generates code to do the following

1. Create an empty string buffer, ie, a `StringBuffer` object, and initialize it
2. Append any operands to that buffer; that `StringBuffer`'s `append()` method is overloaded to deal with any type makes handling operands of mixed types easy
3. Invoke the `toString()` method on the string buffer to produce a `String`

`JStringConcatenationOp`'s `codegen()` makes use of a helper method, `nestedCodegen()` for performing only step 2 for any nested string concatenation operations, which eliminates the instantiation of unnecessary string buffers

For example, given the *j--* expression

```
x + true + "cat" + 0
```

the compiler generates the following JVM code

```
new java/lang/StringBuilder
dup
invokespecial StringBuilder."<init>":()V
aload x'
invokevirtual append:(Ljava/lang/String;)StringBuilder;
iconst_1
invokevirtual append:(Z)Ljava/lang/StringBuilder;
ldc "cat"
invokevirtual append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
iconst_0
invokevirtual append:(I)Ljava/lang/StringBuilder;
invokevirtual StringBuilder.toString:()Ljava/lang/String;
```

Generating Code for Casts

Analysis determines both the validity of a cast and the necessary `Converter`, which encapsulates the code generated for the particular cast

Each `Converter` implements a method `codegen()`, which generates any code necessary to the cast

For example, consider the converter for casting a reference type to one of its sub-types (narrowing cast) which requires that a `checkcast` instruction be generated

```java
class NarrowReference implements Converter {
    private Type target;

    public NarrowReference(Type target) {
        this.target = target;
    }

    public void codegen(CLEmitter output) {
        output.addReferenceInstruction(CHECKCAST, target.jvmName());
    }
}
```

On the other hand, when any type is cast to itself (the identity cast), or when a reference type is cast to one of its super types (called widening), no code need be generated

Generating Code for Casts

Casting an `int` to an `Integer` is called boxing and requires an invocation of the `Integer.valueOf()` method

```
invokestatic java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
```

Casting an `Integer` to an `int` is called unboxing and requires an invocation of the `Integer.intValue()` method

```
invokevirtual java/lang/Integer.intValue:()I
```

Certain casts, from one primitive type to another require that a special instruction be executed; for example, the `i2c` instruction converts an `int` to a `char`

There is a `Converter` defined for each valid conversion in *j--*