Programming Language Implementation
Part VIII: Code Generation for OO Languages

Martin Odersky

Semester 2, 1998

Code Generation Revisited

Here are some more hints that help you write your code generator.

Schematic interaction with the ClassFile and Code classes

- Here's a scheme to use `ClassFile` and `Code` for your code generation needs.
- At the start of code generation for a module, say `MyProg`, you need to initialize `ClassFile` with the module's name.

```
public Item caseModDecl(Tree tree) {
    ClassFile.newClassFile(tree.name); ...
```

- When you start generating code for a method, you declare it first to the `ClassFile` class, getting a constant pool index in return.

```
public Item caseFunDecl (FunDecl tree) {
    Symbol f = tree.sym;
    f.adr = - ClassFile.newMethod(name, f.type.signature());
    ...
```

- The next step is to create a new code structure for the method's code. Don't forget to also store the current code structure in the `Item` class, so that code generation routines in items know about it.

```
    Code code = ClassFile.newCode();
    Item.code = code; ...
```

- After processing the method's code, we conclude by calling `completeMethod` with the completed code of the method as parameter. Don't forget to make sure the code has a return instruction at the end.

```
... ClassFile.completeMethod(code);
```

- After processing the whole module, we conclude by calling `completeClassFile`.

```
    ClassFile.completeClassFile();
```


Code Generation Examples

- Here is a compile log for the HelloWorld program.

```
> java j0c.Main HelloWorld.j0
module HelloWorld {

  void main(string[] args) {
    printString("hello world");
    println();
  }
}
wrote HelloWorld.class
```

- And here is the disassembly:

```
> javap -c HelloWorld
class HelloWorld extends java.lang.Object
{
    public static void main(java.lang.String[]);
}

Method void main(java.lang.String[])
   0 ldc #46 <String "hello world">
   2 invokestatic #31 <Method void printString(java.lang.String)>
   5 invokestatic #27 <Method void println()>
   8 return
```

- Only one method: `main`.
- Constant pool indices `#i` would vary from compiler to compiler.

```
> java j0c.Main sumup.j0
module SumUp {

  int[] vector;

  void init() {
     vector[0] = 1;
     vector[1] = 2;
     vector[2] = 3;
     vector[3] = 4;
  }

  int sum(int[] v) {
    int i;
    i = 0;
    int s;
    s = 0;
    while (i < intArrayLength(v)) {
      s = s + v[i];
      i = i + 1;
    }
    return s;
  }

  void main(string[] args) {
    vector = newarray int (4);
    init();
    printString("The sum is ");
    printInt(sum(vector));
    println();
  }
}
wrote SumUp.class
```



```
> javap -c SumUp
public class SumUp extends java.lang.Object
{
    public static int vector[];
    public static void init();
    public static int sum(int[]);
    public static void main(java.lang.String[]);
}

Method void init()
   0 getstatic #44 <Field int vector[]>
   3 iconst_0
   4 iconst_1
   5 iastore
   6 getstatic #44 <Field int vector[]>
   9 iconst_1
  10 iconst_2
  11 iastore
  12 getstatic #44 <Field int vector[]>
  15 iconst_2
  16 iconst_3
  17 iastore
  18 getstatic #44 <Field int vector[]>
  21 iconst_3
  22 iconst_4
  23 iastore
  24 return

Method int sum(int[])
   0 iconst_0
   1 istore 1
   3 iconst_0
   4 istore 2
```

```
   6 goto 25
   9 iload 2
  11 aload 0
  13 iload 1
  15 iaload
  16 iadd
  17 istore 2
  19 iload 1
  21 iconst_1
  22 iadd
  23 istore 1
  25 iload 1
  27 aload 0
  29 invokestatic #15 <Method int intArrayLength(int[])>
  32 if_icmplt 9
  35 iload 2
  37 ireturn
  38 new #37 <Class java.lang.RuntimeException>
  41 dup
  42 ldc #52 <String "reaching end of function, but return type is
  44 invokespecial #40 <Method java.lang.RuntimeException(java.lang
  47 athrow

Method void main(java.lang.String[])
   0 iconst_4
   1 newarray int
   3 putstatic #44 <Field int vector[]>
   6 invokestatic #47 <Method void init()>
   9 ldc #58 <String "The sum is ">
  11 invokestatic #31 <Method void printString(java.lang.String)>
  14 getstatic #44 <Field int vector[]>
  17 invokestatic #50 <Method int sum(int[])>
  20 invokestatic #35 <Method void printInt(int)>
```



```
  23 invokestatic #27 <Method void println()>
  26 return
```

Translating object-oriented constructs

- So far, every field and method we declared existed in a single copy.
- This mode is called *static* in Java, and it is the exception rather than the rule.
- Usually, fields and methods exist once per *object*. Objects are created dynamically using `new`.
- A *class* in Java is a template which defines
  - The set of variables that an object has (its *instance variables*).
  - The set of procedures that operate on the object (its *methods*).
  - The set of operations that create an object of that class (its *constructors*).

Example: Some classes for geometric objects:

```
abstract class GeoObject {
  abstract double circumference();
  abstract double area();
}

class Point extends GeoObject {
  int x;
  int y;
  Point (int x, int y) {
    this.x = x;
    this.y = y;
  }
  double circumference() { return 0; }
  double area() { return 0; }
}

class Rectangle extends GeoObject {

  Point ll;
  Point ur;

  Rectangle(int llx, int lly, int width, int height) {
    ll = new Point(llx, lly);
    ur = new Point(llx + width, lly + height);
  }
  double circumference() {
    return ((ur.x - ll.x) + (ur.y - ll.y)) * 2;
  }
  double area() {
    return (ur.x - ll.x) * (ur.y - ll.y);
  }
```

```
}

class Test {

  static GeoObject[] objs = new GeoObject[2];
  static {
    objs[0] = new Point(1, 2);
    objs[1] = new Rectangle(0, 0, 1, 2);
  }

  public static void main(String[] args) {
    double totalArea = 0.0;
    int i = 0;
    while (i < objs.length) {
      totalArea = totalArea + objs[i].area();
      i = i + 1;
    }
    System.out.print("totalArea = ");
    System.out.println(totalArea);
  }
}
```


Disassembly of class Point

```
ares>javap -c Point
Compiled from geom.java
class Point extends GeoObject
    /* ACC_SUPER bit set */
{
    double x;
    double y;
    Point(double,double);
    double circumference();
    double area();

Method Point(double,double)

!! super()
   0 aload_0
   1 invokespecial #1 <Method GeoObject.<init>()V>

!! this.x = x;
   4 aload_0
   5 dload_1
   6 putfield #2 <Field Point.x D>

!! this.y = y;
   9 aload_0
  10 dload_3
  11 putfield #3 <Field Point.y D>
  14 return
```

```
Method double circumference()
!! return 0.0
   0 dconst_0
   1 dreturn

Method double area()
!! return 0.0
   0 dconst_0
   1 dreturn

}
```


Handling "this"

- There's one set of instance fields per object.
- The standard identifier `this` refers to the current object.
- How do we find out about the value of `this`?
- When an object is first created, it is represented by an address in the heap. This address is returned from the call to `new`..
- Each instance method gets passed the object address as an implicit first parameter.
- To access a field `x`, use `this.x`. In bytecodes:

```
 4 aload_0
 6 putfield #2 <Field Point.x D>
```

- Analogous for `put`:

```
 9 aload_0
10 dload_3
11 putfield #3 <Field Point.y D>
```

Disassembly of class Rectangle

```
ares>javap -c Rectangle
Compiled from geom.java
class Rectangle extends GeoObject
    /* ACC_SUPER bit set */
{
    Point ll;
    Point ur;
    Rectangle(double,double,double,double);
    double circumference();
    double area();

Method Rectangle(double,double,double,double)

!! super()
   0 aload_0
   1 invokespecial #1 <Method GeoObject.<init>()V>

!! ll = new Point(llx, lly);
   4 aload_0
   5 new #2 <Class Point>
   8 dup
   9 dload_1
  10 dload_3
  11 invokespecial #3 <Method Point.<init>(DD)V>
  14 putfield #4 <Field Rectangle.ll LPoint;>

!! ur = new Point(llx + width, lly + height);
  17 aload_0
```



```
  18 new #2 <Class Point>
  21 dup
  22 dload_1
  23 dload 5
  25 dadd
  26 dload_3
  27 dload 7
  29 dadd
  30 invokespecial #3 <Method Point.<init>(DD)V>
  33 putfield #5 <Field Rectangle.ur LPoint;>
  36 return

Method double circumference()
!! (ur.x - ll.x)
   0 aload_0
   1 getfield #5 <Field Rectangle.ur LPoint;>
   4 getfield #6 <Field Point.x D>
   7 aload_0
   8 getfield #4 <Field Rectangle.ll LPoint;>
  11 getfield #6 <Field Point.x D>
  14 dsub
!! (ur.y - ll.y)
  15 aload_0
  16 getfield #5 <Field Rectangle.ur LPoint;>
  19 getfield #7 <Field Point.y D>
  22 aload_0
  23 getfield #4 <Field Rectangle.ll LPoint;>
  26 getfield #7 <Field Point.y D>
  29 dsub
!! +
  30 dadd
!! * 2
  31 ldc2_w #8 <Double 2.000000>
```

```
  34 dmul
!! return
  35 dreturn

Method double area()
   0 aload_0
   1 getfield #5 <Field Rectangle.ur LPoint;>
   4 getfield #6 <Field Point.x D>
   7 aload_0
   8 getfield #4 <Field Rectangle.ll LPoint;>
  11 getfield #6 <Field Point.x D>
  14 dsub
  15 aload_0
  16 getfield #5 <Field Rectangle.ur LPoint;>
  19 getfield #7 <Field Point.y D>
  22 aload_0
  23 getfield #4 <Field Rectangle.ll LPoint;>
  26 getfield #7 <Field Point.y D>
  29 dsub
  30 dmul
  31 dreturn

}
```


Disassembly of class Test

```
ares>javap -c Test
Compiled from geom.java
class Test extends java.lang.Object
    /* ACC_SUPER bit set */
{
    static GeoObject objs[];
    Test();
    public static void main(java.lang.String []);
    static void <clinit>();

Method Test()  !! internal name: <init>
!! super();
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object.<init>()V>
   4 return

Method void main(java.lang.String [])
!! double totalArea = 0.0;
   0 dconst_0
   1 dstore_1

!! int i = 0;
   2 iconst_0
   3 istore_3

!! while (i < objs.length) {
   4 iload_3
   5 getstatic #2 <Field Test.objs [LGeoObject;>
```

```
  8 arraylength
  9 if_icmpge 30

!!   totalArea = totalArea + objs[i].area();
 12 dload_1
 13 getstatic #2 <Field Test.objs [LGeoObject;>
 16 iload_3
 17 aaload
 18 invokevirtual #3 <Method GeoObject.area()D>
 21 dadd
 22 dstore_1

!!  i = i + 1;
 23 iload_3
 24 iconst_1
 25 iadd
 26 istore_3

!! }
 27 goto 4

!! System.out.print("totalArea = ");
 30 getstatic #4 <Field java.lang.System.out Ljava/io/PrintStream;>
 33 ldc #5 <String "totalArea = ">
 35 invokevirtual #6 <Method java.io.PrintStream.print(Ljava/lang/Str

!! System.out.println(totalArea);
 38 getstatic #4 <Field java.lang.System.out Ljava/io/PrintStream;>
 41 dload_1
 42 invokevirtual #7 <Method java.io.PrintStream.println(D)V>

 45 return
```



```
Method void <clinit>()

!! static GeoObject[] objs = new GeoObject[2];
   0 iconst_2
   1 anewarray class #8 <Class GeoObject>
   4 putstatic #2 <Field Test.objs [LGeoObject;>

!! objs[0] = new Point(1, 2);
   7 getstatic #2 <Field Test.objs [LGeoObject;>
  10 iconst_0
  11 new #9 <Class Point>
  14 dup
  15 dconst_1
  16 ldc2_w #10 <Double 2.000000>
  19 invokespecial #12 <Method Point.<init>(DD)V>
  22 aastore

!! objs[1] = new Rectangle(0, 0, 1, 2);
  23 getstatic #2 <Field Test.objs [LGeoObject;>
  26 iconst_1
  27 new #13 <Class Rectangle>
  30 dup
  31 dconst_0
  32 dconst_0
  33 dconst_1
  34 ldc2_w #10 <Double 2.000000>
  37 invokespecial #14 <Method Rectangle.<init>(DDDD)V>
  40 aastore

  41 return
}
```

- `<clinit>` is the method that contains the initializer (empty in this

Virtual methods

- When calling `objs[i].area` in class test, this is translated to a call of method `GeoObject.area()`:

```
invokevirtual #3 <Method GeoObject.area()D>
```

- But this method is abstract; it does not have a body!
- What *should* happen is a call to either `Point.area` or `Rectangle.area()`, depending on the object's type at run-time.
- This is achieved by looking up the correct target of the call in a *virtual method table* (VMT).
- In Java byte-codes, the lookup is part of the semantics of the `invokevirtual` instruction.
- When compiling to native code, it has to be programmed explicitly.

Handling Subtyping

- Java has the Subtyping principle: A subclass can be used in every context its superclass can be used.
- How can we support this in code generation?
- Idea: Keep the layout of fields and methods in a superclass in the subclass (see blackboard).
- This works well for single inheritance. Waht about multiple inheritance? interfaces?
- What can we do in these cases?

Summary: Code Generation for Object-Oriented Languages

- Objects are represented as references to records on the heap.
- Instance variables are represented as record fields.
- Methods are referred to by VMT's, one per class.
- Methods are passed the address of the receiver object as an additional hidden parameter.
- In a single inheritance language, the field and method layout of a subclass keeps the one of the superclass, but adds more fields and methods at the end.