Part VIII : Code Generation II

Shortcut Evaluation Example

- Example : Translate

```
if (x > 0) || (x == 0 && y > 0)) {
   x = x + y;
}
```

Shortcut Evaluation Scheme

- Problem : A condition can now contain many jumps, which all need to be patched.
- Two classes of jumps : jump when true, jump when false.
- Idea : Maintain a set of jumps which all jump to the same label by a linked list of *Chain :*

```
class Chain {
  int jumpAdr;
  Chain next;
}
```

- When arriving at the target label of a chain, patch all jump addresses in it.

Shortcut Evaluation Scheme (2)

- Conditional items are now characterized by three elements
  - The final jump-when-false instruction.
  - A chain for all jumps which are taken because we know the condition is true.
  - A chain for all jumps which are taken because we know the condition is false.
- Here are two useful operations on chains :

```
/** patch all jumps in chain c to go to target */
void patch (Chain c, int target) { ... }
/** Form union of two chains */
Chain union (Chain c1, Chain c2) { ... }
```

CondItems for Shortcut Evaluation

```
Class CondItem extends Item {

     int jumpCode;
     Chain trueJumps;
     Chain falseJumps;
     CondItem(int jumpCode, Chain trueJumps, Chain falseJumps) {
        .. as usual ..
     }
     void load() { ... as before ... }
     void store() { ... as before ... }
     CondItem makeCondition() { return this; }

     /** Emit instruction to jump when condition is false,
       * Return chain of all jumps-when-false */
     Chain jumpIfFalse() {
       return new Chain (code.jump(jumpCode), falseJumps);
     }
```

CondItems for Shortcut Evaluation (2)

```
    /** Emit instruction to jump when condition is true,
      * Return chain of all jumps-when-true */
    Chain jumpIfTrue() {
      return negate(.)jumpIfFalse();
    }

    CondItem negate() {
        return new CondItem (((jumpCode + 1) ^ 1) - 1, falseJumps, true
    }
}
```

ShortCut Code Generation for Boolean Expressions

- We look at only some representative examples :

E = Operation LT E_1 E_2

```
genCode(E1).load(); genCode(E2).load()
return new CondItem (icmp_ge, null, null)
```

| Operation AND E_1 E_2

```
Item lcond = genCode(E1).makeCond();
Chain falseJumps = lcond.jumpIfFalse();
patch (lcond.trueJumps, code.size())
Item rcond = genCode(E2).makeCond();
return new CondItem(
   rcond.jumpCode,
   rcond.trueJumps,
   union(falseJumps, rcond.falseJumps));
```

| Operation NEG E

```
return genCode(E).makeCond().negate()
```

| Operation OR E_1 E_2

```
?
```

Shortcut Code Generation for Statements

- We look only at if-then-else; the others are similar, and bit simpler.

S = IfStatement E S_1 S_2

```
Item cond = genCode(E).makeCond();
Chain falseJumps = cond.jumpIfFalse();
patch (cond.trueJumps, code.size());
genCode(S1);
int elseJump = code.jump(goto);
patch (falseJumps, code.size());
genCode(S2);
patch(elseJump, code.size());
```

JO Code Generation Framwork : Class *ClassFile*

- See also the html documentation.

```
Class ClassFile {
/** initialize for creation of new class file.
 *  where  'name' is the name of the class to be created */
    public static void newClassFile(String name) { ... }

/** create new Code structure */
    public static Code newCode () { ... }
/** Generate a new field with given name and type signature;
 *  return constant pool entry of reference to this
 *  field as its adress. */
    public static int newField(String name, String sig) { ... }
```

JO Code Generation Framwork : Class *ClassFile* (2)

```
/** Generate a new method with given name and type signature;
 *    return constant pool entry of reference to this
 *    method as its address */
      public static int newMethod(String name, String sig) { ... }

/** complete method by generating its code */
      public static void completeMethod(Code code) { ... }

/** complete and emit class file, with given code as initialization. */
      public static void completeClassFile(Code code)
         throws IOException { ... }
}
```

New additions to `Symbol`

The `Symbol` class now gets an additional field :

```
/** The address of this entry, needed for bytecode generation.
 *  This is used in one of two ways :
 *  negative addresses are negated constant pool indices of
 *  global variables or methods,
 *  positive addresses are slot numbers of local variables.
 *  In the first case,
 *  the true constant pool index is obtained by negating the address.
 */
int adr;
```

New additions to `Symbol` (2)

It also gets three new methods that access that field :

```
/** Return the slot number of this entry.
 *  Only defined for local variables. */
int slot () {
  if (adr < 0) throw new InternalError ("slot");
    return adr;
}

/** Return the constant pool index of this entry.
 *  Only defined for global variables and methods. */
int poolIndex() {
  if (adr >= 0) throw new InternalErro("poolIndex");
    return -adr;
}

/** Does this entry represent a global variable or method? */
public boolean isGlobal() {
    return adr < 0;
}
```

New additions to `Type`

The `Type` class gets an additional method that returns the signature of it type.
This method is defined as follows :

```
/** Convert this type into a type signature string
 *  (for bytecode generation) */
public String signature() {
   switch (tag) {
   case VOID   :
     return "V";
   case INT    :
     return "I";
   case STRING :
     return "Ljava/lang/String;";
   case ARRAY  :
     return "[" + base.signature();
   case FUN    :
     Entry p = params;
     String sig = "(";
     while (p != null) {
```

New additions to `Type` (2)

```
        sig = sig + p.type.signature();
        p = p.next;
      }
      return sig + ")" + base.signature();
    default:
      throw new internalError("signature");
    }
}
```

Predefined Operations

- The J0 language defined so far does not have enough operations to make it a useful programming language.
- In particular, I/O is missing.
- We solve that by introducing *predefined methods*
- All such methods are defined in a class **`Runtime`**, defined as follows:

```
/** Runtime support class for compiled j0 programs
 *  <p>
 *  This class offers useful methods that can be called from
 *  j0 programs */
public class Runtime {
/** print given integer */
   public static void printInt(int x) {
     System.out.print(x);
   }
/** print given string */
  public static void printString(String x) {
    System.out.print(x);
  }
/** print new line */
  public static void println() {
    System.out.println();
  }
```

Predefined Operations (2)

```
/** convert given string into an array of integers.
 *  Elements of the array
 *  contain consecutive characters of the given string. */
  public static int[] explode(String s) {
    int l = s.length();
    int[] chars = new int[l];
    for (int i = 0; i < l; i++) {
      chars[i] = s.charAt(i);
    }
    return chars;
  }
/** convert given array of integers to string by
 *  concatenating all characters in the integer array. */
  public static String implode(int[] chars) {
    char[] cs = new char[chars.length];
    for (int i = 0; i < chars.length; i++) {
      cs[i] = (char)chars[i];
    }
    return new String(cs);
  /** The length of given array of integers */
  public static int intArrayLength(int[] xs) {
    return xs.length;
  }
/** The length of given array of strings */
  public static int stringArrayLength(String[] xs) {
    return xs.length;
  }
}
```

Predefined Operation (3)

- We still need to teach the compiler about class runtime.
- This is done by entring the predefined functions into the outermost scope when the compiler starts up.

```
Public Item caseModDecl(ModDecl tree) {

  ClassFile.newClassFile(name);
  scope = new Scope(null);
  Predef.enterAll(scope);
  scope = new Scope(scope);
  ...
```

Predefined Operations (5)

- Class **Predef** is defined as follows.

```
package j0c;

/** A class that enters all predefined methods into the
 *  symbol table. The predefined methods are all methods
 *  in class <tt>Runtime</tt>.
 *  <p>
 *  If <tt>Runtime</tt> is modified, this class should be
 *  modified accordingly. */
public class Predef {

  static void enterMethod(Scope scope, int classIndex,
              String name, Type[] argtypes,
              Type restype) {
    ...
  }
/** Enter all predefined methods of class <tt>Runtime</tt> into
 *  given scope.
 *  Also create references to these methods in the currently
 *  compiled class file. */
  public static void enterAll(Scope scope) {
    int classIndex = ClassFile.pool.enterClass("Runtime");
    Type intArrayType = new Type(Type.ARRAY, Type.intType);
    Type stringArrayType = new Type(Type.ARRAY
                         Type.stringType);
```

Predefined Operations (6)

```
    enterMethod(scope, classIndex,
            "printInt",
            new Type[]{Type.intType}, Type.voidType);
    enterMethod(scope, classIndex,
            "printString",
            new Type[]{Type.stringType}, Type.voidType);
    enterMethod(scope, classIndex,
            "println",
            new Type[]{}, Type.voidType);
    enterMethod(scope, classIndex,
            "explode",
            new Type[]{Type.stringType}, intArrayType);
    enterMethod(scope, classIndex,
            "implode",
            new Type[]{intArrayType}, Type.stringType);
    enterMethod(scope, classIndex,
            "intArrayLength",
            new Type[]{intArrayType}, Type.intType);
    enterMethod(scope, classIndex,
            "stringArrayLength",
            new Type[]{stringArrayType}, Type.intType);
    }
}
```

- Feel free to add other useful methods into **`Runtime`** and **`Predef`**.

The Main Method

- To actually run a generated class file, it needs to have a main method.
- The method should have the following signature:

```
void main(String[] args) { ... }
```

- The parameter name is irrelevant, but its type has to be just as given above.
- The parameter to main will point at run-time to a vector of program arguments.
- You can determine the length of this vector using the predefined **`stringArrayLength`** method.

Group Assignment

- Extend your JO compiler with a code generator.
- Due date: End of term.
- Optional bits:
  - Implement conditional operators `&&` and `||` for short-circuit evaluation of boolean expressions.
  - Implement constant folding in your compiler.
  - Implement bytecode optimizations in your compiler.

Tool Support

- When you use the code generation framework, you will generate classfiles with extension **.class**.
- Let's assume you have a **`module MyProg`**, which generates a class file **`MyProg.class`**.
- There are several things you can do with this file:
  - Dump it at low-level using

    ```
    java DumpClassFile MyProg.class
    ```

    This should work even for corrupt class files.
  - Dump it at a higher level using

    ```
    javap -c MyProg
    ```

    This will reject corrupt class files.
  - Run it using

    ```
    java MyProg
    ```

    This is what you will eventually have to do.

Good luck!

Appendix : javap Usage

```
>javap
```

Usage: `javap [-v] [-c] [-p] [-h] [-verify] [-verify-verbose] files...`

where options include:

```
-c          Disassemble the code
-classpath <directories separated by colons>
            List directories in which to look for classes
-h          Create info that can be put into a C header file
-l          Print local variable tables
-p          Include private fields and methods
-v          Print verbosely
-s          Print field and method descriptors in internal form
-verify     Run the verifier
            Verify, printing out debugging info
-version    Print the javap version string
```

Appendix : java Usage

```
>java
```

Usage: `java [-options] class`

where options include:

```
-help                print out this message
-version             print out the build version
-v -verbose          turn on verbose mode
-debug               enable remote JAVA debugging
-noasyncgc           don't allow asynchronous garbage collection
-verbosegc           print a message when garbage collection
                occurs
-noclassgc           disable class garbage collection
-cs -checksource     check if source is newer when loading classes
-ss<number>          set the maximum native stack size for any
                     thread
-oss<number>         set the maximum Java stack size for any
                thread
-ms<number>          set the initial Java heap size
-mx<number>          set the maximum Java heap size
-D<name>=<value>     set a system property
```

Appendix : java Usage (2)

```
-classpath <directories separated by colons>
                  list directories in which to look for classes
-prof[:<file>]    output profiling data to ./java.prof or
                  ./<file>
-verify            verify all classes when read in
-verifyremote      verify classes read in over the network
           [default]
-noverify          do not verify any class
```