Languages & Translators

LINGI2132

Generating JVM Bytecode with ASM

Nicolas LAURENT
Université catholique de Louvain

JVM Bytecode Libraries

- Use the ASM library
    - "Low-level", comprehensive
- There exists a few other JVM bytecode libraries
    - BCEL: similar but ASM is better (maintained, faster, better design, ...)
    - Javassist: higher-level, but some limitations
    - ByteBuddy, CGLIB: focused on higher-level patterns,
      in particular dynamic proxys
- Use ASM for compilers, ASM or Javassist for pimping existing applications

Example (1/3)

```
final String className = "Test";
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
// public class <className> extends Object {}
cw.visit(V1_8, ACC_PUBLIC, className, null, "java/lang/Object", null);
// public static void main (String[] args) {}
MethodVisitor mainMethodWriter = cw.visitMethod(ACC_PUBLIC | ACC_STATIC, "main",
    "([Ljava/lang/String;)V", null, null);
```

Example (2/3)

```
mainMethodWriter.visitCode();
// System.out
mainMethodWriter.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
    "Ljava/io/PrintStream;");
// "hello"
mainMethodWriter.visitLdcInsn("hello");
// System.out.println("hello");
mainMethodWriter.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println",
    "(Ljava/lang/String;)V", false);
// return;
mainMethodWriter.visitInsn(RETURN);
mainMethodWriter.visitEnd();
mainMethodWriter.visitMaxs(-1, -1);
cw.visitEnd();
```

Example (3/3)

```
byte[] bytecode = cw.toByteArray();
ByteArrayClassLoader loader = new ByteArrayClassLoader();
Class<?> test = loader.defineClass(className, bytecode);
try {
    test.getMethod("main", String[].class).invoke(null, (Object) new String[0]);
} catch (NoSuchMethodException | IllegalAccessException | InvocationTargetException e) {
    e.printStackTrace();
}
```

ASM

- Manages the constant pool for you
    - Just emit a xCONST_x / LDC instruction, and ASM will make sure the constant is added to the pool.
- Takes care of sizing the stack and the space for locals.
- Manages jump targets using *labels*.
- Otherwise, it's pretty much as straightforward as emitting a sequence of instruction for each method.

ASM - Development Process

- What bytecode should I emit?
    - Use `javap` / https://javap.yawk.at
    - Lookup the instructions
        - https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
        - https://docs.oracle.com/javase/specs/jvms/se16/html/jvms-6.html
- How to do emit the instruction?
    - https://asm.ow2.io/javadoc/
    - Jump to definition in IDE, e.g. `MethodVisitor` and do a text search for the instruction.
- Search in the manual.

ASM - My Own Experience

- I implemented a bytecode compiler for my Sigh demo language
    - https://github.com/norswap/sigh/blob/master/src/norswap/sigh/bytecode/BytecodeCompiler.java
- I wrote some generic utilities you can reuse
    - working with Class objects can sometimes be a bit nicer, in particular to call Java method
      e.g. `invokeStatic(method, SighRuntime.class, "concat", String.class, String.class);`
    - Unified interface for loading constants (`loadConstant(methodVisitor, 1)` vs `methodVisitor.visitInsn(ICONST_1)`)
- I recommend writing your own language specific type utilities.
    - Those will help convert from your semantic-level types into the type descriptors for the JVM types used to represent the data.
- It's ok to have multiple method visitors open at the same time.
- Adding debugging information: use `visitLocalVariable`, `visitLineNumber`

Notations

- Field Descriptors
    - see next slides
- Method Descriptors
    - **(**<param1 field desc><param2 field desc>....**)**<return field desc>
- Slash-separated binary class names
    - https://docs.oracle.com/javase/specs/jls/se8/html/jls-13.html#jls-13.1
    - e.g. `java/util/String`
    - unique: `class A { static class X {} }` `class B extends A {}`
        - `pkg/A$X` not `pkg/B$X`

Field Descriptors

- https://docs.oracle.com/javase/specs/jvms/se16/html/jvms-4.html#jvms-4.3

FieldType term	Type	Interpretation
`B`	`byte`	signed byte
`C`	`char`	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
`D`	`double`	double-precision floating-point value
`F`	`float`	single-precision floating-point value
`I`	`int`	integer
`J`	`long`	long integer
`L` *ClassName* `;`	`reference`	an instance of class *ClassName*
`S`	`short`	signed short
`Z`	`boolean`	`true` or `false`
`[`	`reference`	one array dimension

Next Time

Optimization