Bytecode generation using ASM

A4 Preparation

Dr Jonas Lundberg

Jonas.Lundberg@lnu.se

Slides are available in Moodle

20 oktober 2024

Assignment 4

- Written exam: November 5
- Deadline A4: November 10
- Instructions
  - You should use the ASM bytecode manipulation library
  - Any valid OFP program A.ofp should be converted to a classfile A.class that can be executed using the Java Virtual Machine (JVM)
  - A.class should be a correct translation of A.ofp
    ⇒ same observable behavior for all executions
  - `void main() { .. }` in OFP should translate to
    `public static void main(Strings[] args) { .. }` in Java
  - `int max(int a, int b) { .. }` in OFP should translate to
    `private static int max(int a, int b) { .. }` in Java
  - `int, float, bool, char, string` in OFP should translate to
    `int, double, boolean, char, java.lang.String` in Java
  - `print` and `println` in OFP should translate to
    `System.out.print(), System.out.println()` in Java

More details and help available in A4. Read instructions carefully!

Introduction to ASM

- ASM is a Java bytecode manipulation and analysis framework.
- ASM can be used to modify existing bytecode or to ...
- ... *generate classes from scratch*
- ASM website: `https://asm.ow2.io`
- Latest stable version: 7 (Java 11), ASM 9.6 (Java 22) was recently released!
- We provide: `asm-all-5.0.1.jar` (Java 9)
  (which will be used in the following examples)
- User guide available as PDF for ASM 4.0. Most parts about manipulating existing bytecode ⇒ Not very useful for us!
- No good resource about generating bytecode from scratch available (as far as I know). Please inform us if you find a useful resource on the Internet.

Generate Hello bytecode (1)

Program `HW.java` generates bytecode for this program.

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("hello World!");
    }
}
```

```
public class HW extends ClassLoader implements Opcodes {
    public static void main(final String args[]) throws Exception {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
        cw.visit(V1_1, ACC_PUBLIC, "Hello", null, "java/lang/Object", null);

        // Generate code for methods (Next slide)
        ...
        cw.visitEnd();

        byte[] code = cw.toByteArray();   // Save bytecode in Hello.class
        FileOutputStream fos = new FileOutputStream("Hello.class");
        fos.write(code);
        fos.close();
    }
}
```

Generate Hello bytecode (2)

```
// Code for the (implicit) constructor. Must always be included
Method m = Method.getMethod("void <init> ()");
GeneratorAdapter mg = new GeneratorAdapter(ACC_PUBLIC, m, null, null,cw);
mg.loadThis();   // Since non-static method
mg.invokeConstructor(Type.getType(Object.class), m);
mg.returnValue();
mg.endMethod();

// Code for the 'main' method
Method main = Method.getMethod("void main (String[])");
mg = new GeneratorAdapter(ACC_PUBLIC + ACC_STATIC, main, null, null, cw);

mg.getStatic(Type.getType(System.class), "out",  // Push ref to System.out
             Type.getType(PrintStream.class));   // of type PrintStream
mg.push("Hello world!");      // Push item to be printed
mg.invokeVirtual(Type.getType(PrintStream.class),    // Code to make call
                 Method.getMethod("void println (String)"));
mg.returnValue();
mg.endMethod();
```

Generate Hello bytecode (3)

Once we have constructed the bytecode it must be saved (and optionally) verified and executed. The final part of the program `HW.java` looks like this:

```
// Save bytecode
byte[] code = cw.toByteArray();
FileOutputStream fos = new FileOutputStream("Hello.class");
fos.write(code);
fos.close();

// Bytecode diagnostics using various ASM help classes
// ==> bytecode check + printing
ClassReader cr = new ClassReader(code);
ClassVisitor tracer = new TraceClassVisitor(new PrintWriter(System.out));
ClassVisitor checker = new CheckClassAdapter(tracer, true);
cr.accept(checker,0);

// Execute Hello.class using approach found using Google
HW loader = new HW();   // HW ==> name of this class (HW.java)
Class<?> exampleClass = loader.defineClass("Hello", code, 0, code.length);
exampleClass.getMethods()[0].invoke(null, new Object[] { null });
```

Summary – Generate Hello bytecode (4)

- We use the class `ClassWriter` to generate a class
- Class `Method` represents a method
- Class `GeneratorAdapter` is used to generate bytecode
- Skeleton for main method

```
Method main = Method.getMethod("void main (String[])");
mg = new GeneratorAdapter(ACC_PUBLIC + ACC_STATIC, main, null, null, cw);

// add bytecode using the GeneratorAdapter mg

mg.returnValue();
mg.endMethod();
```

Suggestion: Take a look at the complete `HW.java` (part of A4). Try to make it run using provided ASM .jar file. Verify functionality by executing generated `Hello.class`.

Five examples to learn ASM

We provide five examples for you to learn how to use ASM. Each program generates (and saves) executable Java bytecode for a simple program.

- **HW.java:** Simplest possible. Generates code for a simple Hello World program. See previous slides.
- **Plus.java:** Simple arithmetics + function calls
- **Sum.java:** Includes a while statement ⇒ conditional jumps
- **Float.java:** Working with decimal numbers
- **Arrays2.java:** Working with arrays

Understanding how to generate correct bytecode for a given Java program using ASM is just the start. Later on (Assignment 4) you will create a visitor that generates Java bytecode for an arbitrary .ofp program.

Example - Plus (making calls)

Java

```
public static void main( ... ) {
   int a = 25;
   int b = 25 + 3*a;
   int p = plus(a,b);
   System.out.println(p);  // 100
}

private static int plus(int a, int b) {
   return a+b;
}
```

```
private static int plus(int, int);
   0 = a, 1 = b
      0: iload_0
      1: iload_1
      2: iadd
      3: ireturn
```

javap

```
public static void main( ...);
   0 = args, 1 = a, 2 = b, 3 = p
       0: bipush 25
       2: istore_1
       3: bipush 25
       5: iconst_3
       6: iload_1
       7: imul
       8: iadd
       9: istore_2
      10: iload_1
      11: iload_2
      12: invokestatic plus:(II)I
      15: istore_3
      16: getstatic java/lang/System.
      19: iload_3
      20: invokevirtual println:(I)V
      23: return
```

Example Plus (main code)

```
mg.push(new Integer(25));
mg.storeLocal(1,Type.INT_TYPE);   // a = 25

mg.push(new Integer(25));
mg.push(new Integer(3));
mg.loadLocal(1,Type.INT_TYPE);
mg.math(GeneratorAdapter.MUL, Type.INT_TYPE);
mg.math(GeneratorAdapter.ADD, Type.INT_TYPE);
mg.storeLocal(2,Type.INT_TYPE);  // b = 25 + 3*a

mg.loadLocal(1,Type.INT_TYPE);  // push args a and b
mg.loadLocal(2,Type.INT_TYPE);
mg.invokeStatic(Type.getType("L"+"Plus"+";"),
                Method.getMethod("int plus(int,int)"));
mg.storeLocal(3,Type.INT_TYPE);  // Call Plus.plus(a,b), store in p

mg.getStatic(Type.getType(System.class),
             "out",Type.getType(PrintStream.class));  // Push System.out
mg.loadLocal(2,Type.INT_TYPE);             // print(p)
mg.invokeVirtual(Type.getType(PrintStream.class),
                 Method.getMethod("void println (int)"));
```

Example Plus (method plus code)

javap

```
private static int plus(int, int);
   0 = a, 1 = b
     0: iload_0
     1: iload_1
     2: iadd
     3: ireturn
```

ASM

```
Method plus = Method.getMethod("int plus (int, int)");
mg = new GeneratorAdapter(ACC_PRIVATE + ACC_STATIC, plus, null, null, cw);
mg.loadArg(0);  // loadArg rather than loadLocal
mg.loadArg(1);
mg.math(GeneratorAdapter.ADD, Type.INT_TYPE);
mg.returnValue();
mg.endMethod();
```

Notice that ASM treats parameter access `mg.loadArg(1);` differently than local variable access `mg.loadLocal(1,Type.INT_TYPE);`

The class GeneratorAdapter

- We use class `GeneratorAdapter` to generate bytecode
- It simplifies code generations by hiding many details

- `mg.push` can be used to push many types of literals
  ⇒ replaces `iconst, bipush, ...` and various instructions to push double, boolean and string literals
- `mg.loadLocal` and `mg.storeLocal` take a type variable (e.g. `Type.INT_TYPE`) and can be used on several types.

- Take a look at API documentation for `org.objectweb.asm.commons.GeneratorAdapter` to get an idea of what methods that are available.
- Available at `https://javadoc.io/doc/org.ow2.asm/asm/5.0`

Java Bytecode Types

ASM (and JVM) error messages often refers to bytecode types:

- I ⇒ int
- C ⇒ char
- D ⇒ double
- Z ⇒ boolean
- L ClassName ; ⇒ instance of class ClassName
- [⇒ array reference

For example, a string array String[] has type

```
[Ljava/lang/String;
```

We made a call Plus.plus(int,int) in the Plus example using

```
mg.invokeStatic(Type.getType("L"+"Plus"+";"),   // Reference type Plus
                Method.getMethod("int plus(int,int)"));  // Signature
```

Live Demo

Demo tools using `Plus.java`.

- Code for simple Java program `Plus.java`.
- Compile to get `Plus.class`.
- Inspect using `javap -p -v  Plus.class`
  or `javap -p -c  Plus.class`
- Show corresponding ASM program `Plus.java`
- Run ASM program `Plus.java`

System.out.println (1)

Print a string (from HW.java): `System.out.println("Hello world!")`

```
mg.getStatic(Type.getType(System.class), "out",    // Push field "out"
             Type.getType(PrintStream.class));     // in class PrintStream
mg.push("Hello world!");
mg.invokeVirtual(Type.getType(PrintStream.class),             // Call method
                 Method.getMethod("void println (String)")); // println
```

Print an integer (from Plus.java): `System.out.println(p)`

```
mg.getStatic(Type.getType(System.class), "out",         // Same as above
             Type.getType(PrintStream.class));
mg.loadLocal(2,Type.INT_TYPE);              // print(p)
mg.invokeVirtual(Type.getType(PrintStream.class),
                 Method.getMethod("void println (int)"));  // Not as above!
```

Notice that we need to specify which version of `PrintStream.println()` we are using. One for each type of argument ⇒ we must know which data we are printing when generating code.

A visitor method for OFP `print`

```
@Override // ('print'|'println') '(' expr ')' SC
public Type visitPrintStmt(OFPParser.PrintStmtContext ctx) {
   mg.getStatic(Type.getType(System.class), "out",Type.getType(PrintStream.clas
   Type eType = visit( ctx.getChild(2) );  // Push expr, return ASM expr type

   String type = null;   // Select print type
   if (eType == Type.INT_TYPE) type = "int";
   else if (eType == Type.DOUBLE_TYPE) type = "double";
   else if (eType == Type.CHAR_TYPE)  type = "char";
   else if (eType == Type.BOOLEAN_TYPE) type = "boolean";
   else if (eType.toString().equals("java.lang.String")) type = "java.lang.Stri
   else throw new RuntimeException("Unkown print type "+eType);

   if (ctx.getChild(0).getText().equals("println"))
     mg.invokeVirtual(Type.getType(PrintStream.class),
                      Method.getMethod("void println ("+type+")"));
   else
     mg.invokeVirtual(Type.getType(PrintStream.class),
                 Method.getMethod("void print ("+type+")"));
   return null;
}
```

Example - Sum (using jumps)

Java

```
private static int sumUpTo(int n) {
   int sum = 0;
   int i = 1;
   while (i <= n) {
      sum = sum + i;
      i = i + 1;
   }
   return sum;
}
```

javap

```
private static int sumUpTo(int);
      n = 0, sum = 1, i = 2
         0: iconst_0
         1: istore_1
         2: iconst_1
         3: istore_2
         4: goto 14
         7: iload_1
         8: iload_2
         9: iadd
        10: istore_1
        11: iinc 2, 1
        14: iload_2
        15: iload_0
        16: if_icmple 7
        19: iload_1
        20: ireturn
```

Notice the two jumps `4: goto 14` and `16: if_icmple 7`, and the increase instruction `iinc 2, 1`

`iinc 2, 1` should be interpreted as: integer increase, variable 2, step 1 ⇒ `i = i + 1`

Sum using ASM

```
mg.push(new Integer(0));
mg.storeLocal(1,Type.INT_TYPE);  // sum = 0
mg.push(new Integer(1));
mg.storeLocal(2,Type.INT_TYPE);  // i = 1
   Label exitWhile = new Label();   // jump to condition
   mg.goTo(exitWhile);
        Label enterWhile = mg.mark();   // Loop body
        mg.loadLocal(1,Type.INT_TYPE);
        mg.loadLocal(2,Type.INT_TYPE);
        mg.math(GeneratorAdapter.ADD, Type.INT_TYPE);
        mg.storeLocal(1,Type.INT_TYPE);
        mg.loadLocal(2,Type.INT_TYPE);   // start of i = i + 1
        mg.push(new Integer(1));
        mg.math(GeneratorAdapter.ADD, Type.INT_TYPE);
        mg.storeLocal(2,Type.INT_TYPE);
   mg.mark(exitWhile);             // backpatching
   mg.loadLocal(2,Type.INT_TYPE);  // condition i<n
   mg.loadArg(0);                  // Read n
   mg.ifICmp(GeneratorAdapter.LE, enterWhile);  // Jump to loop body
mg.loadLocal(1,Type.INT_TYPE);     // Push result
mg.returnValue();
```

Parameter and Variable Indices (1)

- ASM instructions like `loadLocal`, `storeLocal`, and `loadArg` refer to parameter and local variable indices
- Example: Method below use indices: n = 0, sum = 1, i = 2

```
private static int sumUpTo(int n) {
    int sum = 0;
    int i = 1;
    ...
}
```

- In general:
  - Indices start at 0
  - Parameters first, ordered left-to-right
  - Local variabels, top-to-bottom
  - Doubles requires special treatment due to their size (coming soon!)
- Q: How to treat indices in your code generation?
- A: Extend the function symbol with additional features

Parameter and Variable Indices (2)

The function symbol class is in charge of keeping track of parameter and local variable indices.

```
public class FunctionSymbol extends Symbol {
   ...

   // Prepare for code generation ==> collect var/par indices
   private int varCount = 0;
   private Map<Symbol,Integer> indices = new LinkedHashMap<Symbol,
                                                          Integer>();
   public void addVariable(Symbol varSym) { ... }
   public void addParameter(ParamSymbol parSym) { ... }
   public int indexOf(Symbol sym) { ... }  // lookup used in code generation
}
```

- We add information in the SymtabListener
- We resolve indices during code generation using `indexOf`
- `LinkedHashMap` is fast and maintains insertion order ⇒ Excellent!

Example - Floats (Index Problems)

Java

```
... void main( ... ) {
   double f = 2.34;
   double ff = 2.0;
   double fff = mult(f,ff);
   System.out.println(fff);  // 4.68
}

... double mult(double a, double b) {
   return a * b;
}
```

javap

```
 ... double mult(double, double);
   #0 = a, #2 = b
     0: dload_0
     1: dload_2
     2: dmul
     3: dreturn
```

javap

```
 public static void main( ... );
 #0 = args, #1 = f, #3 = ff, #5 = fff
        0: ldc2_w 2.34d
        3: dstore_1
        4: ldc2_w 2.0d
        7: dstore_3
        8: dload_1
        9: dload_3
       10: invokestatic  mult:(DD)D
       13: dstore 5
       15: getstatic java/lang/System.
       18: dload 5
       20: invokevirtual java/io/Print
       23: return
```

Notice indices 1,3,5 for local variables f,ff,fff!

Parameter and Variable Indices (3)

Generating indices for double variables requires a bit of extra work.

- Double local variables requires one extra space ⇒ skip one index after each such variable
- Example: Indices for `main` in example Floats

```
public static void main( ... );
#0 = args, #1 = f, #3 = ff, #5 = fff
    ...
```

- Example: Indices for `mult` in example Floats

```
private static double mult(double, double);
  #0 = a, #2 = b
    0: dload_0
    1: dload_2
    2: dmul
    3: dreturn
```

- Handling double indices requires that we treat double variables a bit different in the `addX` methods in class FunctionSymbol

Example Floats (ASM)

Using ASM to generate `main` method code

```
m = Method.getMethod("void main (String[])");
mg = new GeneratorAdapter(ACC_PUBLIC + ACC_STATIC, m, null, null, cw);
mg.push(new Double(2.34));
mg.storeLocal(1,Type.DOUBLE_TYPE);
mg.push(new Double(2.0));
mg.storeLocal(3,Type.DOUBLE_TYPE);  // 3 since previous double requires two slo
  ...
```

Using ASM to generate `mult` method code

```
m = Method.getMethod("double mult (double,double)");
mg = new GeneratorAdapter(ACC_PRIVATE + ACC_STATIC, m, null, null, cw);
mg.loadArg(0);
mg.loadArg(1);
mg.math(GeneratorAdapter.MUL, Type.DOUBLE_TYPE);
mg.returnValue();
mg.endMethod();
```

- We use indices 1,3 for assigning `f,ff` values in `main`
- We use indices 0,1 to load arguments `a,b` in `mult` in spite of indices 0,2 used in the corresponding javap output!

Example - Arrays2.java (1)

A simple Java program initializing an array with two element, accessing and printing one of them.

```
public static void main(String[] args) {
    int[] arr = {6,7};
    int a = arr[0];
    System.out.println(a);        // Prints 6

    // Above Java is in bytecode handled as
    // int[] arr = new int[2];
    // arr[0] = 6;
    // arr[1] = 7;
    // int a = arr[0];
    // System.out.println(a);
}
```

Notice: Array initialization `int[] arr = {6,7};` is in the byte code handled as creating an empty array of size two, followed by adding elements 6 and 7 individually.

```
public static void main(java.lang.Stri
   args = 0, arr = 1, a = 2
        0: iconst_2
        1: newarray       int
        3: dup
        4: iconst_0
        5: bipush        6
        7: iastore
        8: dup
        9: iconst_1
       10: bipush        7
       12: iastore
       13: astore_1
       14: aload_1
       15: iconst_0
       16: iaload
       17: istore_2
       18: getstatic     #2
       21: iload_2
       22: invokevirtual #3
       25: return
```

Example - Arrays2.java (2)

ASM instructions for `int[] arr = {6,7};`

```
mg.push(Integer.valueOf(2));
mg.newArray(Type.INT_TYPE);   // pop size and push array ref

mg.dup();   // push array ref again
mg.push(0); // push index
mg.push(Integer.valueOf(6));  // Push element value 6
mg.arrayStore(Type.INT_TYPE);  // pop, pop, pop and store in array

mg.dup();   // push array ref again
mg.push(1); // push index
mg.push(Integer.valueOf(7));  // Push element value 7
mg.arrayStore(Type.INT_TYPE);  // store in array
```

Notice: We doesn't initialize the array right away. Instead we create an array of size 2 and add each element separately. Similar to

```
int[] arr = new int[2];
arr[0] = 6;
arr[1] = 7;
```

Example - Arrays2.java (2)

Remaining array related code in code `Arrays2.java` ⇒ assign created array to variable arr, read value at position 0 and print it.

```
Type intArray = Type.getType(int[].class);
mg.storeLocal(1, intArray);  // assign array to variable arr

mg.loadLocal(1, intArray);  // push array ref
mg.push(0);                 // push index 0
mg.arrayLoad(Type.INT_TYPE);   // push value for arr[0]

mg.storeLocal(2, Type.INT_TYPE); // a = ...

mg.getStatic(Type.getType(System.class), "out",Type.getType(PrintStream.class))
mg.loadLocal(2,Type.INT_TYPE);              // print(a)
mg.invokeVirtual(Type.getType(PrintStream.class), Method.
```

Generated instructions

```
13: astore_1
14: aload_1
15: iconst_0
16: iaload
17: istore_2
    ...
```

Learn ASM

The best way to learn ASM is by writing your own programs like `Plus.java` or `Arrays2.java` for Java features you are interested in.

Approach for Java feature X

1. Write a very simple Java program `X.java` using feature X
2. Compile into `X.class`
3. Use `javap -p -c` to see generated bytecode instructions
4. Try to write an ASM program generating instructions in 3.

We provide five such programs:
`HW.java, Plus.java, Sum.java, Floats.java, Arrays2.java`.

Suggestion: Learn if statements and strings using short programs like:

```
int a = 5;
int b = 7;
int max;
if (a>b)
    max = a;
else
    max = b;
System.out.println(max);
```

```
String str = "Hello";
int i = 0;
while (i < str.length) {
    char c = str[i];
    System.out.println(c);
}
```

OFP to Bytecode using ASM

We suggest

- Use a small step approach, start with an OFP program like

```
void main() {
   int a = 7;
   int b = a + 8;
}
```

- Add one feature at the time
- For each feature X
  1. Write small Java program using feature X
  2. Generate bytecode view using `javap -p -v`
  3. Add corresponding bytecode generating code in your `BytecodeVisitor`
- It doesn't work? Write a separate Java/ASM program trying to generate byte code for the small Java program using the ASM bytecode library. Time consuming ⇒ use it with care!

Bytecode part of `Main.java`

```
System.out.println("\nBytecode  generation started\n");
BytecodeGenerator  bcGen = new BytecodeGenerator(table,progName);
bcGen.visit(root);

System.out.println("\nVerify and Print bytecode\n");
byte[] code = bcGen.getByteArray();
ClassReader cr = new ClassReader(code);
ClassVisitor tracer = new TraceClassVisitor(new PrintWriter(System.out));
ClassVisitor checker = new CheckClassAdapter(tracer, true);
cr.accept(checker,0);

File javaOutFile = new File("test_class_files/"+progName+".class");
FileOutputStream fos = new FileOutputStream(javaOutFile);
fos.write(code); fos.close();
System.out.println("Bytecode saved in "+javaOutFile.getAbsolutePath());

System.out.println("\nExecuting bytecode");
Main loader = new Main();
Class<?> exampleClass = loader.defineClass(progName, code, 0, code.length);
exampleClass.getMethods()[0].invoke(null, new Object[] { null });
```

We provide ...

- ASM as a jar: `asm-all-5.0.1.jar`
- Four Java programs showing how to use ASM to generate byte code
  1. `HW.java`: Hello World!
  2. `Plus.java`: Integer arithmetics, method calls
  3. `Floats.java`: Float arithmetics, indices
  4. `Sum.java`: A while statement with jumps
  5. `Arrays2.java`: Uses arrays
- An updated set of test programs
  1. A few error fixed in previous programs
  2. A new set of very small programs using bool and float

Internet resources

- ASM 5 API
  `https://javadoc.io/doc/org.ow2.asm/asm/5.2`
- ASM Official website contains
  1. A tutorial for version 4 (Useless!)
  2. A developers guide for version 6 (Not much help)
  3. API for version > 5
- In short, information for ASM 5 is sparse
- In general, information for ASM bytecode generation is sparse
- Feel free to use ASM > 5 (but do not ask for help)
- Please let me know if you find any Internet resource that you found useful for Assignment 4

This is it!

- This was the final lecture!
- Remaining tutoring sessions: October 27 and November 3
- Good Luck with Assignment 4
- Good Luck with the written exam