Emitting Scala `class` files via ASM

© Miguel Garcia, LAMP, EPFL
`http://lamp.epfl.ch/~magarcia`

April 4th, 2012

Abstract

ASM[1] is a high-performance bytecode manipulation library that automatically computes the stack map tables required in JVM class files version 50 and up. For this and other reasons (future-proofing against upcoming class file formats, more widely available developer know-how on ASM) the `GenASM` backend of the Scala compiler for JVM relies on ASM.

These notes describe the implementation, for those willing to:

- understand how it works,
- extend it (e.g., adding support for Java 7/8 goodies), or
- develop derivatives. For example, emitting Dalvik VM bytecode via ASMDEX[2], taking three-address code[3] as input.

Contents

[1] `http://asm.ow2.org/`

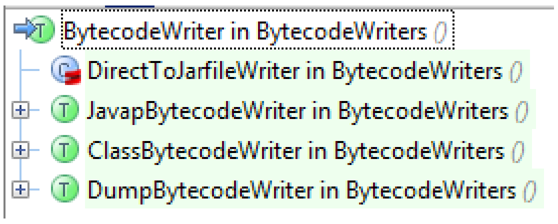
[2] `http://asm.ow2.org/asmdex-index.html`

[3] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q2/Moving3A.pdf`

1 Background

1.1 ASM

When writing classfiles, ASM can be used in either *stream-oriented* or *tree-building* modes, the latter making sense in case further passes or transformations are needed. `GenASM` uses the stream-oriented API which is both suitable and faster. The output thus produced is a byte array (the `class` file contents) whose serialization is delegated to a `BytecodeWriter`, of which there are different kinds:



BytecodeWriter in BytecodeWriters ()
DirectToJarfileWriter in BytecodeWriters ()
JavapBytecodeWriter in BytecodeWriters ()
ClassBytecodeWriter in BytecodeWriters ()
DumpBytecodeWriter in BytecodeWriters ()

Besides the core functionality of preparing a `class` file for serialization, ASM can be leveraged for other tasks. For example, a future `ScalapBytecodeWriter`[4] could take `asm.util.Textifier` as starting point. There are areas where `javap` ouput is less than ideal (e.g. when comparing different versions of the same `class` file):

- unreadable pickle (see also `https://github.com/paulp/pickler-visualizer`)
- two constant pools, while having identical contents, may be displayed differently due to physical layout.
- stack maps are displayed in encoded form by `javap`, their expansion makes more sense instead.

1.2 ASTs arriving at `GenASM`: ICode

The ICode language consists of VM-neutral stack-based instructions that operate on VM-level types (i.e., erasure has been performed and the type hierarchy has been adapted to VM capabilities, e.g. there are no Scala traits anymore just VM-level classes and interfaces). There's no significant semantic gap between ICode and JVM bytecode thus `GenASM` mostly performs a straightforward mapping, save for two minor complications (exception handlers including finalizers, and forward jumps) that require some bookkeeping on the part of `GenASM`, as described next.

An `IMethod` comprises:

- a list of locals (some of which are params),
- a list of exception handlers each covering a set of blocks and defining the handler code (consisting of blocks with one of them distinguished as the start block)
- a list of blocks that comprise all the blocks above, as well as blocks not covered by any exception handler.

[4]`http://www.scala-lang.org/docu/files/tools/scalap.html`

- the entry point to the method (one of the blocks above, aliased by `IMethod.startBlock` and `IMethod.code.startBlock`).

1.3 Control Flow Graph

Navigating the CFG is tricky because `BasicBlock.successors` includes the start block of each exception handler covering the block in question (i.e., it conflates normal and exceptional control flow). Separating normal from exceptional control flow involves:

- **normal control flow**: Only certain instructions may direct control flow to some other block ("fall-through" is represented explicitly via `JUMP`):

```
def directSuccessors: List[BasicBlock] =
  if (isEmpty) Nil else lastInstruction match {
    case JUMP(whereto)           => whereto :: Nil
    case CJUMP(succ, fail, _, _) => fail :: succ :: Nil
    case CZJUMP(succ, fail, _, _) => fail :: succ :: Nil
    case SWITCH(_, labels)       => labels
    case RETURN(_)               => Nil
    case THROW(_)                => Nil
    case _ =>
      if (closed)
        dumpClassesAndAbort("The last instruction is not a control flow instruction: " + lastInstruction)
      else Nil
  }
```

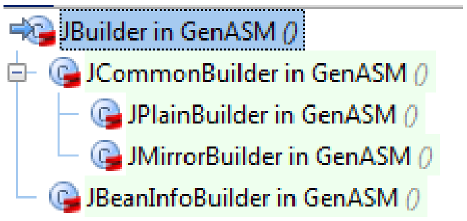
- **exceptional control flow**:

```
def exceptionSuccessorsForBlock(block: BasicBlock): List[BasicBlock] =
  method.exh collect { case x if x covers block => x.startBlock }
```

There's no `directPredecessors`, and `predecessors` conflates normal and exceptional control flow.

2 `GenASM`: Overview

The high-level structure of `GenASM` becomes evident in the `run()` override of the compiler phase it contributes to the pipeline. In a nutshell (Listing 1) for each `IClass` in the input the decision is made wether a mirror class should be generated in addition to the "plain" class (one more class, a `BeanInfo` class, may also be generated). This division of labor is reflected in subclasses that progressively add more specialized functionality:



JBuilder in GenASM ()
JCommonBuilder in GenASM ()
JPlainBuilder in GenASM ()
JMirrorBuilder in GenASM ()
JBeanInfoBuilder in GenASM ()

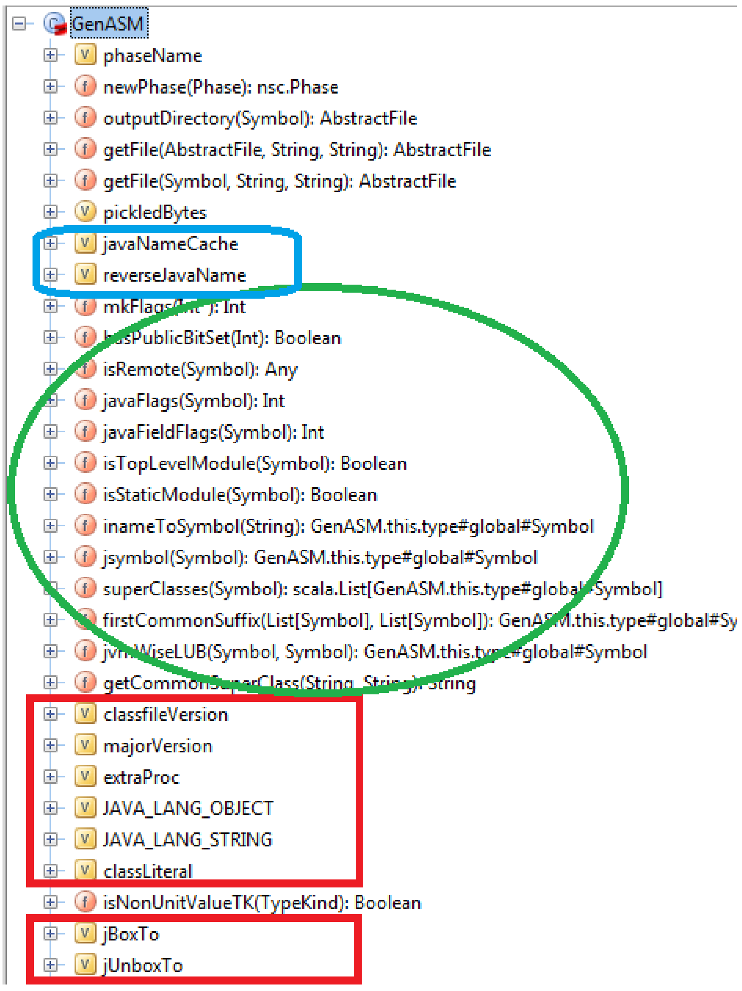
Listing 1: Sec. 2

```
override def run() {

  var sortedClasses = classes.values.toList sortBy ("" + _.symbol.fullName)

  val bytecodeWriter = initBytecodeWriter(sortedClasses filter isJavaEntryPoint)
  val plainCodeGen  = new JPlainBuilder(bytecodeWriter)
  val mirrorCodeGen = new JMirrorBuilder(bytecodeWriter)
  val beanInfoCodeGen = new JBeanInfoBuilder(bytecodeWriter)

  while(!sortedClasses.isEmpty) {
    val c = sortedClasses.head

    if (isStaticModule(c.symbol) &&
        isTopLevelModule(c.symbol) &&
        (c.symbol.companionClass == NoSymbol)) {
      mirrorCodeGen.genMirrorClass(c.symbol, c.cunit)
    }

    plainCodeGen.genClass(c)

    if (c.symbol hasAnnotation BeanInfoAttr) { beanInfoCodeGen.genBeanInfoClass(c) }

    sortedClasses = sortedClasses.tail
    classes -= c.symbol // GC opportunity
  }

  bytecodeWriter.close()
  classes.clear()
  reverseJavaName.clear()

} // end of AsmPhase.run()
```

Although that design helps with code comprehension, there's still shared mutable state reused by each kind of generator (for plain classes, for mirror classes, etc.) with the goal of avoiding instantiating a new instance when emitting a new `IClass`. This decision can be revised in the future.

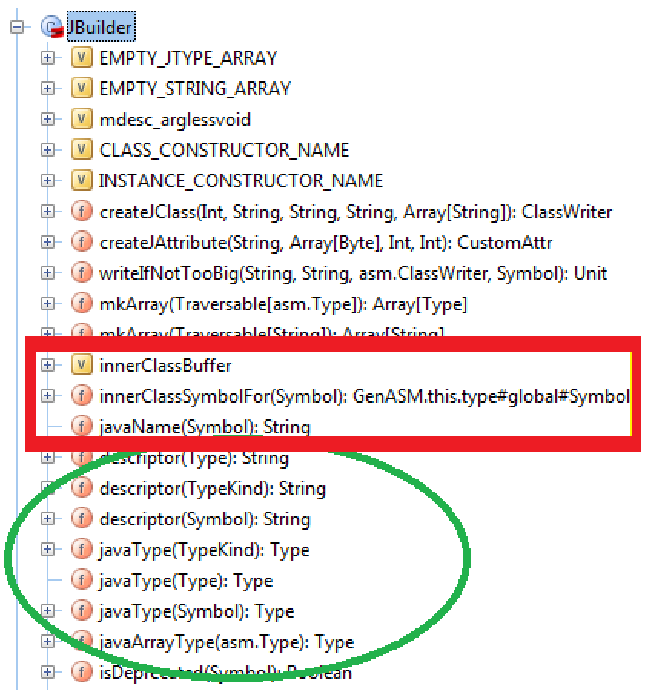
Similary, some members that would logically belong in one of the class builders are owned instead by `GenASM` (of which there's only one instance) again for the purpose of avoiding repeated invocation of their constructors. Except for the readability impact no harm is done because most such members are either immutable (red squares in Figure 1) or non-side-effecting query methods (tagged with an oval in Figure 1).

The only members of `GenASM` accessing mutable data structures (that come to mind!) are `javaNameCache` and `reverseJavaName` (blue rounded-corners square in Figure 1) which map between symbols and bytecode-level names during code generation and also when computing least-upper-bounds (as part of computing stack map tables). In more detail, `javaNameCache` contains entries for classes as well as members, while `reverseJavaName` only for class symbols (which are thus mapped to bytecode-level *internal names*).



```
GenASM
    phaseName
    newPhase(Phase): nsc.Phase
    outputDirectory(Symbol): AbstractFile
    getFile(AbstractFile, String, String): AbstractFile
    getFile(Symbol, String, String): AbstractFile
    pickledBytes
    javaNameCache
    reverseJavaName
    mkFlags(Int*): Int
    hasPublicBitSet(Int): Boolean
    isRemote(Symbol): Any
    javaFlags(Symbol): Int
    javaFieldFlags(Symbol): Int
    isTopLevelModule(Symbol): Boolean
    isStaticModule(Symbol): Boolean
    inameToSymbol(String): GenASM.this.type#global#Symbol
    jsymbol(Symbol): GenASM.this.type#global#Symbol
    superClasses(Symbol): scala.List[GenASM.this.type#global#Symbol]
    firstCommonSuffix(List[Symbol], List[Symbol]): GenASM.this.type#global#Sy
    jvmWiseLUB(Symbol, Symbol): GenASM.this.type#global#Symbol
    getCommonSuperClass(String, String): String
    classfileVersion
    majorVersion
    extraProc
    JAVA_LANG_OBJECT
    JAVA_LANG_STRING
    classLiteral
    isNonUnitValueTK(TypeKind): Boolean
    jBoxTo
    jUnboxTo
```

Figure 1: Sec. 2



```
JBuilder
  EMPTY_JTYPE_ARRAY
  EMPTY_STRING_ARRAY
  mdesc_arglessvoid
  CLASS_CONSTRUCTOR_NAME
  INSTANCE_CONSTRUCTOR_NAME
  createJClass(Int, String, String, String, Array[String]): ClassWriter
  createJAttribute(String, Array[Byte], Int, Int): CustomAttr
  writeIfNotTooBig(String, String, asm.ClassWriter, Symbol): Unit
  mkArray(Traversable[asm.Type]): Array[Type]
  mkArray(Traversable[String]): Array[String]
  innerClassBuffer
  innerClassSymbolFor(Symbol): GenASM.this.type#global#Symbol
  javaName(Symbol): String
  descriptor(Type): String
  descriptor(TypeKind): String
  descriptor(Symbol): String
  javaType(TypeKind): Type
  javaType(Type): Type
  javaType(Symbol): Type
  javaArrayType(asm.Type): Type
  isDeprecated(Symbol): Boolean
```

Figure 2: Sec. 2.1

2.1 Functionality shared by all class builders (`JBuilder`)

1. keeping track of JVM-level inner classes referred somewhere in the class being emitted (red box in Figure 2).

2. mapping symbols (for classes and members) to JVM-level names, in particular mapping of class symbols to internal names (as above, red box in Figure 2).

3. mapping from

   - the type representation used by ICode (`TypeKind` instances, which in turn rely on symbols and types used throughout the compiler); to
   - the type representation used by ASM `asm.Type`, which includes *method descriptors* (green oval in Figure 2).

The first item is taken care of by the methods that map symbols (second item). In detail, the JVM requires including an `InnerClasses` attribute listing those defined or referred to from the class in question. `JBuilder.javaName()` tracks that behind the scenes (adding to the running list in `JBuilder.innerClassBuffer`, which starts empty just before emitting a class). Beware of situations where an internal name is needed but the potential side-effect on `innerClassBuffer` is not desired (e.g. `getCommonSuperClass()`). In those cases, invoke `javaBinaryName()` on the class symbol instead.

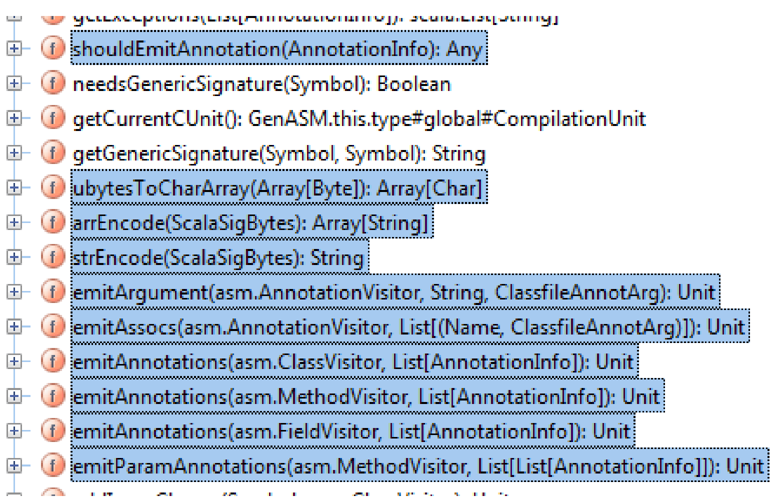

shouldEmitAnnotation(AnnotationInfo): Any
needsGenericSignature(Symbol): Boolean
getCurrentCUnit(): GenASM.this.type#global#CompilationUnit
getGenericSignature(Symbol, Symbol): String
ubytesToCharArray(Array[Byte]): Array[Char]
arrEncode(ScalaSigBytes): Array[String]
strEncode(ScalaSigBytes): String
emitArgument(asm.AnnotationVisitor, String, ClassfileAnnotArg): Unit
emitAssocs(asm.AnnotationVisitor, List[(Name, ClassfileAnnotArg)]): Unit
emitAnnotations(asm.ClassVisitor, List[AnnotationInfo]): Unit
emitAnnotations(asm.MethodVisitor, List[AnnotationInfo]): Unit
emitAnnotations(asm.FieldVisitor, List[AnnotationInfo]): Unit
emitParamAnnotations(asm.MethodVisitor, List[List[AnnotationInfo]]): Unit

Figure 3: Sec. 2.2

2.2 Functionality to build plain and mirror classes (`JCommonBuilder`)

- **pickles**: `pickleMarkerLocal`, `pickleMarkerForeign`, `getAnnotPickle()`. Because a pickle is emitted as a Java 5.0 annotation, the functionality to emit annotations is also relevant for pickling.
- **Java 5.0 annotations**: as shown in Figure 3.
- **Java generic signature**
- **Forwarders**

2.3 Building plain classes (`JPlainBuilder`)

The behavior of `JPlainBuilder` is in large part determined by ASM requirements on invocation order of methods in the sream-oriented API. These typestate or API protocol requirements are documented in the *ASM User Guide* and thus not reproduced here. To give an idea, a `MethodWriter` has the protocol:

```
visitAnnotationDefault?
( visitAnnotation | visitParameterAnnotation | visitAttribute )*
( visitCode
   ( visitTryCatchBlock | visitLabel | visitFrame | visitXxxInsn | visitLocalVariable | visitLineNumber )*
  visitMaxs )?
visitEnd
```

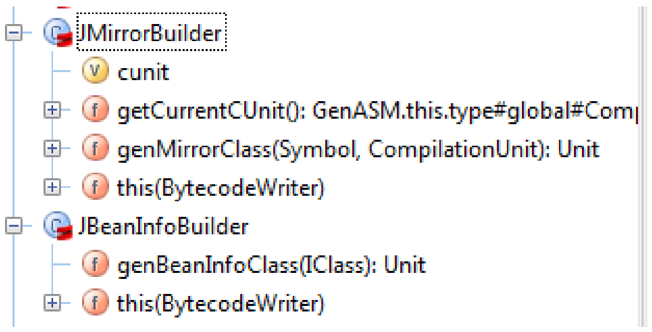
The bulk of code generation is performed in `JPlainBuilder.genCode()` (that's almost 1KLOC). To aid readability, it contains local methods and can be considered sub-divided into (as indicated by source comments):

1. Setting up one-to-one correspondence between ASM Labels and `BasicBlocks` `linearization`

2. demarcating exception handler boundaries (`visitTryCatchBlock()` must be invoked before `visitLabel()` in `genBlock()`)
3. Utilities to later emit debug info for local variables and method params (`LocalVariablesTable` bytecode attribute).
4. Bookkeeping (to later emit debug info) of association between line-number and instruction position.
5. Utilities to emit code proper (most prominently: genBlock()).

2.4 The rest: `JMirrorBuilder` and `JBeanInfoBuilder`

These builders taken together are about 1/10th the size of that for plain classes:



```
JMirrorBuilder
    cunit
    getCurrentCUnit(): GenASM.this.type#global#Com
    genMirrorClass(Symbol, CompilationUnit): Unit
    this(BytecodeWriter)
JBeanInfoBuilder
    genBeanInfoClass(IClass): Unit
    this(BytecodeWriter)
```