Notes on GenMSIL

© Miguel Garcia, LAMP,
École Polytechnique Fédérale de Lausanne (EPFL)
http://lamp.epfl.ch/~magarcia

April 28th, 2010

Contents

1 Following the steps of GenMSIL's elder brother: GenJVM **2**

2 From IClass to VM-level type **3**
2.1 Nested types . . . 4

3 Translating type members **5**

4 Handling static modules **5**
4.1 Module instance field, the GenMSIL story . . . 6
4.2 . . . and now the GenJVM story . . . 7
4.3 Static initializers . . . 7
4.4 dumpMirrorClass . . . 10

5 Adding constructors **10**

6 Adding fields the GenMSIL way **11**
6.1 Adding outerField and a fake local for debugging purposes . . . 11

7 Adding methods in GenMSIL **12**
7.1 Generics, exception list, and annotations (including those for parameters) 13

8 Translating code blocks **13**

9 Items not covered in previous sections **13**
9.1 invokedynamic . . . 13
9.2 Forwarders . . . 13
9.3 GenMSIL's createDelegateCaller(paramType, resType) . . . 14

10 Conclusions **14**

Abstract

I used to believe that `GenJVM` and `GenMSIL` just duplicated one-to-one fields, methods, and code blocks from ICode into bytecode, but in fact there's a fair share of non-trivial desugaring going on too. It would be interesting to identify which of those desugarings could be performed as ICode-to-ICode transformations. The motivation for this is our ongoing work on Scala.Net, where the compiler currently emits MSIL in the textual format expected by the IL Assembler. Making `GenMSIL` directly produce a .NET assembly (e.g., reusing Microsoft's *Common Compiler Framework*) amounts to rewriting large portions of `GenMSIL`. Before doing that, we explore in these notes the pros and cons of breaking apart `GenMSIL` into two phases (ICode-to-ICode desugarings followed by a more straightforward CLR-serialization phase). The resulting flexibility might also prove useful for future backends that may need the results of the first but not the second subphase (e.g., OpenCL, LLVM, program verification frameworks, etc.). As a sidenote, when translating a Scala program into a backend other than JVM or CLR, it is assumed that input programs make reference only to libraries supported by the target platform. As another advantage, an ICode-to-ICode phase would avoid unwarranted divergence across backends (for language features that should be kept consistent).

1 Following the steps of `GenMSIL`'s elder brother: `GenJVM`

The override of `ICodePhase.run` in `JvmPhase` performs dead code elimination and then applies `codeGenerator.genClass(cls)` to each IClass arriving at this phase. `genClass(IClass)` is declared in `BytecodeGenerator`, which constitutes 99% of `GenJVM`. Instead of building all JVM-level types in memory, `genClass` serializes to disk each `.class` before processing the next `IClass`.

In `genClass`, all `cls.symbol.info.parents` after the first one are taken to be interfaces. Before getting the `javaName` of their `typeSymbol`, additional supertypes are added (`SerializableClass.tpe`, `CloneableClass.tpe`, and `RemoteInterface.tpe`) depending on the `cls.symbol.annotations`. Afterwards, no modifications to the list of supertypes takes place.

`GenMSIL` instead goes first over all IClasses to find an entry point for the future assembly, the `symbol` of the method thus found is tracked in (MSIL's) `ByecodeGenerator.entryPoint`. Additionally, the first non-nested IClass during that iteration gives the string value for `ByecodeGenerator.firstSourceName`.

A single assembly file will be written to disk (not each CLR-level type individually) and it's the job of `initAssembly()` to prepare the ground for that (e.g., by setting two `java.io.File` fields: `outDir` and `srcPath`). From this point on, the assembly being generated will be represented as an `AssemblyBuilder` (whose constructor takes an `AssemblyName`, which as of now lacks public key tokens, identifying the assembly just by `assemName`):

```
massembly = AssemblyBuilderFactory.DefineDynamicAssembly(assemblyName)
```

The last thing `initAssembly()` does before returning is creating the internal representation of the main module (where all types and "global methods" will go). The only global method that gets emitted by `writeAssembly` is `globalMain` to become the assembly's entry point.

```
mmodule = massembly.DefineDynamicModule(moduleName,
                                        new File(outDir, moduleName).getAbsolutePath())
```

BTW, it's not clear why `MsilPhase` extends `GlobalPhase` and not `ICodePhase` like `GenJVM` does (an `MsilPhase` can access IClasses using `global.icodes.classes`).

After initializing the `AssemblyBuilder`, `GenMSIL` goes on to create contents in a top-down manner:

```
classes.values foreach codeGenerator.createTypeBuilder
classes.values foreach codeGenerator.createClassMembers
```

Afterwards, it decorates some more the created types: as shown below, `genClass` adds cloning methods, dumps a mirror class for top-level modules without a companion class, adds the *symtab attribute*, and other type attributes dictated by the `iclass.sym.annotations`. Oh, and one more thing: `genMethod` is also invoked on each `iclass.methods`. In terms of code, this third and last iteration over `classes.values` looks as follows:

```
try {
  classes.values foreach codeGenerator.genClass
} finally {
  codeGenerator.writeAssembly
}
```

2 From IClass to VM-level type

With the information collected so far, `GenJVM` creates a JVM-level type (a `ch.epfl.lamp.fjbg.JClass`) for the IClass being processed.

The correspondence `IClass` ↔ `JClass` need not be tracked outside the `genClass` method. In contrat, `GenMSIL` tracks this correspondence with the `types` field in the `SymbolLoaders.clrTypes` object. That field allows finding a `msil.Type` for an `IClass.symbol`.

During `createTypeBuilder(IClass)`, `GenMSIL` catches up with `GenJVM` by computing the supertypes. Please beware that inside file `GenMSIL.scala` the unqualified `Type` refers to `Type` from trait `scala.tools.nsc.symtab.Types`, not `msil.Type`, because of the following import:

```
import ch.epfl.lamp.compiler.msil.{Type => MsilType, _}
```

After running

```
val interfaces: Array[MsilType] =
  parents.tail.map(p => msilTypeFromSym(p.typeSymbol)).toArray
```

`interfaces` always contains `MsilTypes` for which a `TypeBuilder` has already been created, because of the way `msilTypeFromSym` works: it creates a builder in case the `types` map does not yet map to it. Alternatively, all type representatives could be created first to add the supertype topology later (I guess this idiom is more frequent).

> TODO add `definitions.ObjectClass.tpe` as first supertype (to IClasses that are not interfaces) whenever it's not in that position, not only when the `parents` list is empty

At this point, the type representatives created by the JVM and MSIL backends are in both cases connected over the supertype topology. `GenMSIL` went one step further in connecting nested types to their declaring types (where "nesting" equates `iclass.symbol.isNestedClass`) while `GenJVM` will do that only at `emitClass` time (see Sec. 2.1), using as "nesting" criteria the following:

```
def addOwnInnerClasses(cls: Symbol) {
  for (sym <- cls.info.decls.iterator if sym.isClass)
    innerClasses = innerClasses + sym;
}

// add inner classes which might not have been referenced yet
atPhase(currentRun.erasurePhase.next) {
  addOwnInnerClasses(clasz.symbol)
  addOwnInnerClasses(clasz.symbol.linkedClassOfClass)
}
```

Method `addOwnInnerClasses` is not the only place where the `innerClasses: Set[Symbol]` is assigned, invoking the getter-like `javaName(sym: Symbol): String` has that as side-effect:

```
if (sym.isClass && !sym.rawowner.isPackageClass && !sym.isModuleClass) {
  innerClasses = innerClasses + sym;
}
```

TODO I guess the above behavior has to be preserved in `GenMSIL`

2.1 Nested types

Nothing in IClass points to nested or owning classes, however the following may be true: `IClass.symbol.isNestedClass`. The CLR-level type will have a different owner in each case (where `sym` is `iclass.symbol`):

```
if (sym.isNestedClass) {
  val ownerT = msilTypeFromSym(sym.owner).asInstanceOf[TypeBuilder]
  val tBuilder =
    ownerT.DefineNestedType(msilName(sym), msilTypeFlags(sym), superType, interfaces)
  mapType(sym, tBuilder)
} else {
  val tBuilder =
        mmodule.DefineType(msilName(sym), msilTypeFlags(sym), superType, interfaces)
  mapType(sym, tBuilder)
}
```

TODO factor `mapType(sym, tBuilder)` out to appear after the `if-then-else`

In the `GenJVM` world, there's something going on about nested classes as late as `emitClass`. Look:

```
def emitClass(jclass: JClass, sym: Symbol) {
  addInnerClasses(jclass) /*- <-- this must have sthg to do with nested classes */
  val outfile = getFile(sym, jclass, ".class")
  val outstream = new DataOutputStream(outfile.bufferedOutput)
  jclass.writeTo(outstream)
  outstream.close()
  informProgress("wrote " + outfile)
}
```

3 Translating type members

To give some context, in GenMSIL we have explored how TypeBuilders come to being, and are ready to tackle createClassMembers:

```
codeGenerator.initAssembly

classes.values foreach codeGenerator.createTypeBuilder
classes.values foreach codeGenerator.createClassMembers /*- <-- we're about to enter here */

try {
  classes.values foreach codeGenerator.genClass
} finally {
  codeGenerator.writeAssembly
}
```

In GenJVM, the first type members to be added to the current JClass are for static initialization, where we meet isStaticModule as discussed in the following subsection.

In GenMSIL, createClassMembers follows a different order: it adds fields first, then methods (unless the IClass is definitions.ArrayClass which also has special handling in GenJVM), and finally adds static initializers as parf of handling static module as discussed next.

4 Handling static modules

Depending on whom you ask, a static module is:

- According to GenJVM.BytecodeGenerator:

```
def isStaticModule(sym: Symbol): Boolean = {
  sym.isModuleClass && !sym.isImplClass && !sym.hasFlag(Flags.LIFTED)
}
```

- According to GenMSIL.BytecodeGenerator:

```
// if the module is lifted it does not need to be initialized in
// its static constructor, and the MODULE$ field is not required.
// the outer class will care about it.
private def isStaticModule(sym: Symbol): Boolean = {
  // .net inner classes: removed '!sym.hasFlag(Flags.LIFTED)', added
  // 'sym.isStatic'. -> no longer compatible without skipping flatten!
  sym.isModuleClass && sym.isStatic && !sym.isImplClass
}
```

- According to Symbol:

```
final def isStaticModule = isModule && isStatic && !isMethod
```

Firing *Find usages* for the GenJVM and the GenMSIL versions of the above allows pinpointing where the emitted code has to abide by "static module semantics". For example (there are others), it influences how a SuperCall is executed:

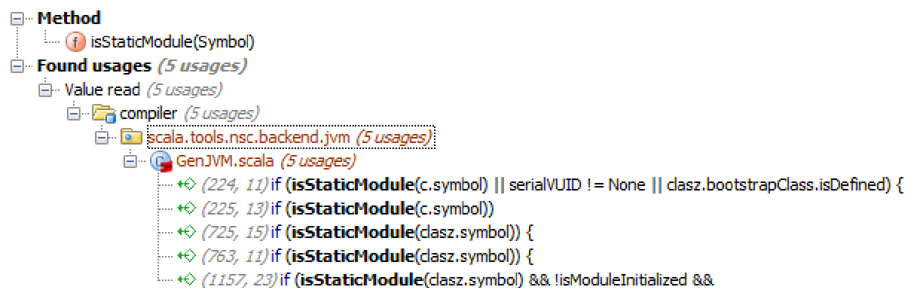

Figure 1: Places in `GenJVM` where code emission depends on `isStaticModule`

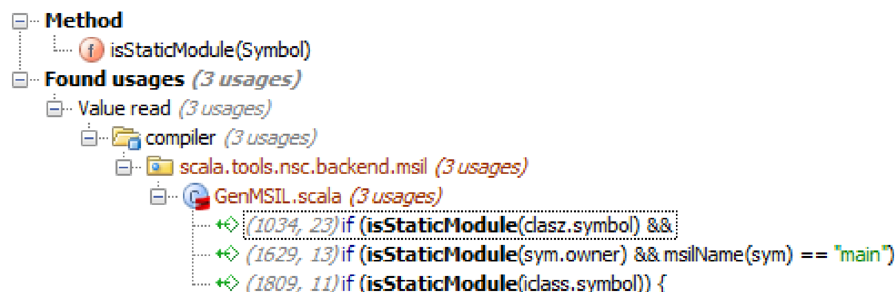

Figure 2: Places in `GenMSIL` where code emission depends on `isStaticModule`

```
case SuperCall(_) =>         /*- snippet from GenMSIL */
  mcode.Emit(OpCodes.Call, constructorInfo)
  if (isStaticModule(clasz.symbol) &&
     notInitializedModules.contains(clasz.symbol))
    {
      notInitializedModules -= clasz.symbol
      mcode.Emit(OpCodes.Ldarg_0)
      mcode.Emit(OpCodes.Stsfld, getModuleInstanceField(clasz.symbol))
    }
```

Instead of cataloging here all those places where code emission depends on `isStaticModule`, we'll cover them in the context of the construct being emitted. Figure 1 shows those places in `GenJVM` where code emission depends on `isStaticModule`, and Figure 2 does the same for `GenMSIL`.

4.1 Module instance field, the `GenMSIL` story . . .

At the end of `createClassMembers(IClass)` in `GenMSIL` we find:

```
if (isStaticModule(iclass.symbol)) {
  addModuleInstanceField(iclass.symbol)
  notInitializedModules += iclass.symbol
  addStaticInit(iclass.symbol)
}
```

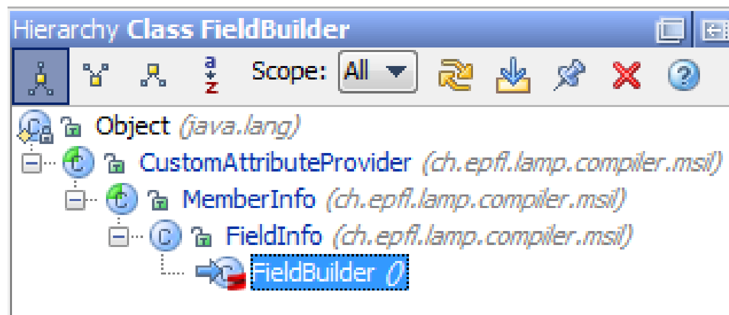

Figure 3: `FieldBuilder` hierarchy

`addModuleInstanceField` retrieves a `TypeBuilder` for the `iclass.symbol` not by using `msilTypeFromSym` (as we've seen so far) but with the help of `getType`. With the `TypeBuilder` thus obtained, a `FieldBuilder` for `MODULE$` is created. Also in `GenMSIL.BytecodeGenerator` there's `mapType(sym: Symbol, mType: MsilType)` which simply adds a pair to the `clrTypes.types` map.

Comparing the similarly named `addModuleInstanceField` in `GenMSIL` and `GenJVM` reveals them to do the same, with the addition that in MSIL the `clrTypes.fields` map (from `Symbol` to `FieldInfo`) is used to track the just created `FieldBuilder` (Figure 3 shows `FieldBuilders` to be `FieldInfos`) using the `iclass.symbol` as key (therefore, `clrTypes.fields` tracks only `MODULE$` fields).

4.2 . . . and now the GenJVM story

The static module field is added just after creating the current `JClass`. The relevant code is shown in Figure 7 on p. 15 (we'll revisit a lot that code fragment), where it reads:

```
if (isStaticModule(c.symbol) || serialVUID != None || clasz.bootstrapClass.isDefined) {
  if (isStaticModule(c.symbol))
      addModuleInstanceField; /*- <-- here */
  addStaticInit(jclass, c.lookupStaticCtor)
  . . .
```

4.3 Static initializers

As a reminder, Figures 4 and 5 quote from the JVM spec about initializers.

Reading a few more lines in Figure 7 shows that a *static initializer* is added whenever the IClass `isStaticModule`, has a serialization version ID, or participates in `invokedynamic` (Sec. 9.1), or fulfills none of these conditions but `containsStaticCtor`. That means, a static module always has a static constructor.

Of both similary named `addStaticInit`, the `GenMSIL` version sports longer comments, so we start with it (reproduced below).

Unlike its `GenJVM` counterpart, static initializers are added here by `GenMSIL` only for IClasses that are static modules. Such static initializer contains three IL instructions: (1) `newobj` invoking the primary constructor of the (static module)

3.9 Specially Named Initialization Methods

At the level of the Java virtual machine, every constructor (JLS3 §8.8) appears as an *instance initialization method* that has the special name <init>. This name is supplied by a compiler. Because the name <init> is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Instance initialization methods may be invoked only within the Java virtual machine by the *invokespecial* instruction, and they may be invoked only on uninitialized class instances. An instance initialization method takes on the access permissions (JLS3 §6.6) of the constructor from which it was derived.

A class or interface has at most one *class or interface initialization method* and is initialized (§5.5) by invoking that method. The initialization method of a class or interface is static, takes no arguments, and has the special name <clinit>.[5] This name is supplied by a compiler. Because the name <clinit> is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Class and interface initialization methods are invoked implicitly by the Java virtual machine; they are never invoked directly from any Java virtual machine instruction, but are invoked only indirectly as part of the class initialization process.

Figure 4: From the JVM spec (1 of 2)

5.5 Initialization

Initialization of a class or interface consists of executing its class or interface initialization method (§3.9). A class or interface may be initialized only as a result of:

- The execution of any one of the Java virtual machine instructions *new*, *getstatic*, *putstatic*, or *invokestatic* that references the class or interface. All of these instructions reference a class directly or indirectly through either a field reference or a method reference. Upon execution of a *new* instruction, the referenced class or interface is initialized if it has not been initialized already. Upon execution of a *getstatic*, *putstatic*, or *invokestatic* instruction, the class or interface that declared the resolved field or method is initialized if it has not been initialized already.

Figure 5: From the JVM spec (2 of 2)

IClass, (2) getting rid of the reference thus created on top of the stack (we're interested only on side-effects), and (3) returning.

```
/** Adds a static initializer which creates an instance of the module class
 *  (calls the primary constructor).
 *
 *  A special primary constructor will be generated (notInitializedModules)
 *  which stores the new instance in the MODULE$ field right after the super call.
 */
private def addStaticInit(sym: Symbol) {
  val tBuilder = getType(sym).asInstanceOf[TypeBuilder]

  val staticInit = tBuilder.DefineConstructor(
    (MethodAttributes.Static | /*- <-- MethodAttributes.Static, therefore parameterless */
     MethodAttributes.Public).toShort,
    CallingConventions.Standard,
    MsilType.EmptyTypes)

  val sicode = staticInit.GetILGenerator()

  val instanceConstructor = constructors(sym.primaryConstructor)

  // there are no constructor parameters. Assuming the constructor takes no parameter
  // is fine: we call (in the static constructor) the constructor of the module class,
  // which takes no arguments - an object definition cannot take constructor arguments.

  sicode.Emit(OpCodes.Newobj, instanceConstructor)

  // the stsfld is done in the instance constructor, just after the super call.

  sicode.Emit(OpCodes.Pop)

  sicode.Emit(OpCodes.Ret)
}
```

Coming back to Figure 7 (`GenJVM`), the emitted static inits don't look so simple. As already noticed those initializers are added to any IClass that `containsStaticCtor` (an `IMethod`) and it is that `IMethod` that gives the body of the static initializer.

TODO BTW, also in `GenJVM`, `legacyStaticInitializer(cls, clinit)` emits instructions pretty much similar to `GenMSIL`'s `addStaticInit` (with the addition of setting the static field `serialVersionUID`). I guess the JVM version evolved and now the MSIL version has some catch up to do.

In a nutshell, a static initializer is emitted by `GenJVM` as follows (this is an example of an ICode-to-ICode transformation, save for the final `genCode(m)`, you see what I'm saying, platform-specific issues are intermingled with platform-independent ones):

```
  val oldLastBlock = m.code.blocks.last
  val lastBlock = m.code.newBlock
  oldLastBlock.replaceInstruction(oldLastBlock.length - 1, JUMP(lastBlock))

  if (isStaticModule(clasz.symbol)) {
    // call object's private ctor from static ctor
    lastBlock.emit(NEW(REFERENCE(m.symbol.enclClass)))
    lastBlock.emit(CALL_METHOD(m.symbol.enclClass.primaryConstructor, Static(true)))
    }
```

```
  . . . add serialVUID
  . . . bootstrapClass stuff

  lastBlock.emit(RETURN(UNIT))
  lastBlock.close

  method = m
  jmethod = clinitMethod
  genCode(m)
```

4.4 dumpMirrorClass

We're not yet done with Figure 7.

The equivalent `GenMSIL` functionality can be found in `genClass`:

```
if (isTopLevelModule(sym)) {
  if (sym.companionClass == NoSymbol)
    dumpMirrorClass(sym)
  else
    log("No mirror class for module with linked class: " +
        sym.fullName)
}
```

TODO

5 Adding constructors

After the long, long, bunch of activities that Figure 7 sparks, comes the following snippet in `GenJVM`'s `genClass`:

```
if (clasz.bootstrapClass.isDefined) jclass.setBootstrapClass(clasz.bootstrapClass.get)
clasz.fields foreach genField
clasz.methods foreach genMethod /*- <-- constructors are added here */
```

`GenJVM`'s `genMethod` returns without doing anything for an `IMethod` that `isStaticCtor` (we saw in Sec. 4.3 how those methods are handled). For an argument `m` such that `m.symbol.isClassConstructor`, the return type is set to `JType.VOID`. Other than that it's handled like any other method. BTW, the following shows that `Symbol.isConstructor` encompasses the mutually exclusive `isClassConstructor` and `isMixinConstructor`.

```
final def isClassConstructor = isTerm && (name == nme.CONSTRUCTOR)
final def isMixinConstructor = isTerm && (name == nme.MIXIN_CONSTRUCTOR)
final def isConstructor = isTerm && (name == nme.CONSTRUCTOR) || (name == nme.MIXIN_CONSTRUCTOR)
```

TODO Most of the work being done in `addRemoteException` appears in fact platform-independent (adding a Scala-level annotation that later triggers adding a platform-specific exception to the method exception list). If so, it could be moved to an ICode-to-ICode transformation, before classfile serialization takes over

Like `GenJVM`, `GenMSIL` also adds constructors while iterating over methods (in `createClassMembers0`). In essence, as follows:

```
for (m: IMethod <- iclass.methods) {

  val methodSym = m.symbol
  val ownerType = mtype // should be == to getType(sym.enclClass).asInstanceOf[TypeBuilder]
  var paramTypes = msilParamTypes(methodSym)
  val attr = msilMethodFlags(methodSym) // i.e. methodSym.tpe.paramTypes.map(msilType).toArray

  if (m.symbol.isClassConstructor) {
    val constr =
      ownerType.DefineConstructor(attr, CallingConventions.Standard, paramTypes)
    for (i <- 0.until(paramTypes.length)) {
      constr.DefineParameter(i, ParameterAttributes.None, msilName(m.params(i).sym))
    }
    mapConstructor(sym, constr)
    addAttributes(constr, sym.annotations) // not implemented yet, look in GenJVM for inspiration
  } else {

. . .
```

6 Adding fields the `GenMSIL` way

Fields are added at the very beginning of `createClassMembers0`:

```
def createClassMembers0(iclass: IClass) {
  val mtype = getType(iclass.symbol).asInstanceOf[TypeBuilder]
  for (ifield <- iclass.fields) {
    val sym = ifield.symbol
    if (settings.debug.value)
      log("Adding field: " + sym.fullName)

    var attributes = msilFieldFlags(sym)
    val fBuilder = mtype.DefineField(msilName(sym), msilType(sym.tpe), attributes)
    fields(sym) = fBuilder
    addAttributes(fBuilder, sym.annotations)
  }
```

6.1 Adding `outerField` and a fake local for debugging purposes

As part of `genMethod` in `GenJVM`, the snippet shown in Figure 6 is run just before `genCode(m)`. I'm telling you this because in `GenMSIL` similar functionality is missing. An `IMethod.symbol` is a closure's `apply` whenever:

```
private def isClosureApply(sym: Symbol): Boolean = {
  (sym.name == nme.apply) &&
  sym.owner.hasFlag(Flags.SYNTHETIC) &&
  sym.owner.tpe.parents.exists { t =>
    val TypeRef(_, sym, _) = t;
    definitions.FunctionClass exists sym.==
  }
}
```

TODO An ICode-to-ICode transformation? Looks like.

```
if (!jmethod.isAbstract() && !method.native) {
  val jcode = jmethod.getCode().asInstanceOf[JExtendedCode]

  // add a fake local for debugging purposes
  if (emitVars && isClosureApply(method.symbol)) {
    val outerField = clasz.symbol.info.decl(nme.getterToLocal(nme.OUTER))
    if (outerField != NoSymbol) {
      log("Adding fake local to represent outer 'this' for closure " + clasz)
      val _this = new Local(
        method.symbol.newVariable(NoPosition, "this$"), toTypeKind(outerField.tpe), false)
      m.locals = m.locals ::: List(_this)
      computeLocalVarsIndex(m) // since we added a new local, we need to recompute indexes

      jcode.emitALOAD_0
      jcode.emitGETFIELD(javaName(clasz.symbol),
                         javaName(outerField),
                         javaType(outerField))
      jcode.emitSTORE(indexOf(_this), javaType(_this.kind))
    }
  }

  for (local <- m.locals if ! m.params.contains(local)) {
    if (settings.debug.value)
      log("add local var: " + local);
    jmethod.addNewLocalVariable(javaType(local.kind), javaName(local.sym))
  }

  genCode(m)
  if (emitVars)
    genLocalVariableTable(m, jcode);
}
```

Figure 6: Adding `outerField` and a fake local for debugging purposes (Sec. 6.1)

7 Adding methods in `GenMSIL`

After adding fields in `createClassMembers0`, methods are next on the line. As we saw in Sec. 5 (*Adding constructors*), both methods and constructors are added while iterating over `iclass.methods`, only that an `if-then-else` in `GenMSIL` makes a distinction that `GenJVM` skips. In essence, methods are added in `GenMSIL` as follows:

```
for (m: IMethod <- iclass.methods) {
  val methodSym = m.symbol
  val ownerType = mtype
  var paramTypes = msilParamTypes(sym)
  val attr = msilMethodFlags(sym) // i.e. sym.tpe.paramTypes.map(msilType).toArray

  if (m.symbol.isClassConstructor) {
    . . .
  } else {
    var resType = msilType(m.returnType)
    val method =
      ownerType.DefineMethod(getMethodName(methodSym), attr, resType, paramTypes)
    for (i <- 0.until(paramTypes.length)) {
      method.DefineParameter(i, ParameterAttributes.None, msilName(m.params(i).sym))
    }
    if (!methods.contains(methodSym))
      mapMethod(methodSym, method)
    addAttributes(method, methodSym.annotations) // not implemented yet, look in GenJVM for inspiration
  }
}
```

7.1 Generics, exception list, and annotations (including those for parameters)

Just before returning, the `GenJVM` version of `genMethod` deals with the following:

```
addGenericSignature(jmethod, m.symbol, clasz.symbol)
val (excs, others) = splitAnnotations(m.symbol.annotations, definitions.ThrowsClass)
addExceptionsAttribute(jmethod, excs)
addAnnotations(jmethod, others)
addParamAnnotations(jmethod, m.params.map(_.sym.annotations))
```

TODO Check how much of the above also belongs to GenMSIL but is missing there. Please notice that `genClass` in `GenMSIL` invokes `addSymtabAttribute` and `addAttributes`, the latter based on the `iclass.symbol.annotations`.

TODO However `addAttributes` is an empty stub pending implementation.

TODO In `GenMSIL`, the auto-generation of `clone` method has to be fixed. The existing code can be found in `genClass`.

8 Translating code blocks

`Local.index` is accessed by both `GenJVM` and `GenMSIL` (and by no previous phase), where it is written in `computeLocalVarsIndex(m: IMethod)`. The `GenJVM` version of `computeLocalVarsIndex` iterates once over `m.locals`, while the `GenMSIL` version first numbers all `m.params` and then goes on to number (starting with 0) those `m.locals` not in `m.params`.

```
/** Represent local variables and parameters */
class Local(val sym: Symbol, val kind: TypeKind, val arg: Boolean) {
  var index: Int = -1
```

TODO

9 Items not covered in previous sections

9.1 `invokedynamic`

Just before emitting fields and methods, the following is invoked in `BytecodeGenerator.genClass(IClass)`:

```
if (clasz.bootstrapClass.isDefined) jclass.setBootstrapClass(clasz.bootstrapClass.get)
```

For the classfile being generated, method `setBootstrapClass` adds as classfile attribute a `JBootstrapInvokeDynamic` attribute in order to support the `invokedynamic` instruction. Two sources of information on that:

- *New JDK 7 Feature: Support for Dynamically Typed Languages in the JVM* http://java.sun.com/developer/technicalArticles/DynTypeLang/
- http://download.java.net/jdk7/docs/api/java/dyn/MethodHandle.html

9.2 Forwarders

http://gabrielsw.blogspot.com/2008/01/playing-with-scala-interoperation-with.html

http://lampsvn.epfl.ch/trac/scala/ticket/363
http://lampsvn.epfl.ch/trac/scala/ticket/1735
Try `-Xprint:icode -Xprint-icode` and then `javap -verbose` for this program:

```
class SuperTest {
  def superMethod() { }
}

class Test extends SuperTest{
  def useSuper { print(superMethod()) }
}

object Test{
  def main(args: Array[String]): Unit = {
      val t = new Test
      print(t.useSuper)
  }
}
```

9.3 GenMSIL's `createDelegateCaller(paramType, resType)`

10 Conclusions

Bringing `GenMSIL` up-to-date with respect to `GenJVM` involves effort that could be better spent developing a new phase (that I'll prototype in the next few days) to emit CLR bytecode directly. The proof of concept will initially leave out some aspects that can be added easily later (say, generation of method bodies). The assemblies thus produced can still be explored with the disassembler and with CFFExplorer. I'm interested in getting early feedback on whether IKVM allows using CCI seamlessly from the rest of the cross-compiler (previous tests show no evidence to the contrary). If the prototype works as expected, `GenMSIL` will be replaced in the short term using the techniques field-tested in the prototype.

```
224     if (isStaticModule(c.symbol) || serialVUID != None || clasz.bootstrapClass.isDefined) {
225       if (isStaticModule(c.symbol))
226           addModuleInstanceField;
227       addStaticInit(jclass, c.lookupStaticCtor)
228
229       if (isTopLevelModule(c.symbol)) {
230         if (c.symbol.companionClass == NoSymbol)
231           dumpMirrorClass(c.symbol, c.cunit.source.toString);
232         else
233           log("No mirror class for module with linked class: " +
234               c.symbol.fullName)
235       }
236     }
237     else {
238       if (c.containsStaticCtor) addStaticInit(jclass, c.lookupStaticCtor)
239
240       // it must be a top level class (name contains no $s)
241       def isCandidateForForwarders(sym: Symbol): Boolean =
242         atPhase (currentRun.picklerPhase.next) {
243           !(sym.name.toString contains '$') && (sym hasFlag Flags.MODULE) && !sym.isImplClass && !sym.isNestedClass
244         }
245
246       val lmoc = c.symbol.companionModule
247       // add static forwarders if there are no name conflicts; see bugs #363 and #1735
248       if (lmoc != NoSymbol && !c.symbol.hasFlag(Flags.INTERFACE)) {
249         if (isCandidateForForwarders(lmoc) && !settings.noForwarders.value) {
250           log("Adding forwarders to existing class '%s' found in module '%s'".format(c.symbol, lmoc))
251           addForwarders(jclass, lmoc.moduleClass)
252         }
253       }
254     }
```

Figure 7: “Code fragment” in `GenJVM.BytecodeGenerator.genClass`, rigth after creating the current `JClass`