What about an explicit formulation of module initialization as ICode-level desugarings

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

May 4th, 2010

Contents

1 Desugaring of module initialization 2
1.1 static-module iff module-instance 2
1.2 Instructions to read `MODULE$` 2
1.3 Instructions to initialize `MODULE$` 3
1.4 The `readResolve` idiom (for serialization) 3

2 Interplay between static-constructor and static class initializer 4
2.1 What `clinit` looks like, given a static-constructor 4
2.2 And what it looks like in the opposite case 5

3 Mirror class for a top-level module without companion class 6
3.1 Purpose 6
3.2 Code generation mechanics 6

4 Adding static-forwarders to existing companion class 7

A Module initialization terminology in detail 7

Abstract

In a previous write-up the possibility was mentioned about desugaring ICode in a platform-independent manner before emitting bytecode. Examples included adding forwarders, static initializers, and in general several ICode-level transformations that make object initialization explicit. The motivation for this was an envisaged simplification of the .NET backend when ICode is received in its simplest form (the same applies when consuming ICode for other purposes, for example program analysis). In these notes we describe transformations that could eventually constitute `i2i`, a new compiler phase running immediately after `GenICode` to encapsulate ICode-to-ICode desugarings and thus avoid their separate maintenance for each backend. Finally, documenting these transformations helps with the maintenance of GenMSIL.

1 Desugaring of module initialization

We'll catalog first what `GenJVM` does about module initialization (whether as ICode transformation or bytecode generation) in order to determine which of these code generation tasks are amenable to an ICode-only formulation.

The terminology of module initialization (details in Appendix A):

```
/*
 * class Foo <
 *  ^  ^ (2)  \
 *  |  | |     \
 *  | (5) |    (3)
 *  |  | |       \
 * (1) v v        \
 * object Foo (4)-> > class Foo$
 *
 * (1) companionClass
 * (2) companionModule
 * (3) linkedClassOfClass
 * (4) moduleClass
 * (5) companionSymbol
 */
```

TODO. A question for `GenJava` fans: Secs. 1.1 to 1.3 describe a bytecode-level mechanism to implement the policy *each module should be initialized with its arg-less constructor before being accessed.* Are there other mechanisms (say, Java-level) to enforce the same policy?

1.1 static-module iff module-instance

The applicabiliy condition of this transformation is platform-independent:

```
if (isStaticModule(c.symbol))
   addModuleInstanceField;
```

but its realization as of now JVM-specific:

```
def addModuleInstanceField {
  import JAccessFlags._
  jclass.addNewField(ACC_PUBLIC | ACC_FINAL | ACC_STATIC,
                     nme.MODULE_INSTANCE_FIELD.toString,
                     jclass.getType())
}
```

where `MODULE_INSTANCE_FIELD` has the value `newTermName("MODULE$")`.

1.2 Instructions to read `MODULE$`

Given that the field just added was not in ICode, no ICode instructions refer to it. Instead, translations will be done on the fly (on a per-platform-basis) as for example when emitting code for a `LOAD_MODULE`:

```
case LOAD_MODULE(module) =>
  // assert(module.isModule, "Expected module: " + module)
  if (clasz.symbol == module.moduleClass && jmethod.getName() != nme.readResolve.toString)
    jcode.emitALOAD_0()
```

```
else
  jcode.emitGETSTATIC(javaName(module) /* + "$" */ ,      // class
                      nme.MODULE_INSTANCE_FIELD.toString, // field
                      javaType(module));                  // type
```

Emitting `aload 0` as in the `then`-branch above is an optimization for accesses to `module` from methods declared in the IClass that `isStaticModule` itself. Emitting instead the `getstatic` as in the `else`-branch would be functionally equivalent. For the `case` clause above, both `jclass` (in the snippet of Sec. 1.1) and `clasz` (in the snippet above) refer to the same class.

1.3 Instructions to initialize `MODULE$`

Another instruction expanded on the fly to "do more" is a `CALL_METHOD` with `SuperCall(_)` invocation style:

```
case call @ CALL_METHOD(method, style) =>
  ...
  style match {
    ...
    case SuperCall(_) =>
        jcode.emitINVOKESPECIAL(owner, jname, jtype)
        // we initialize the MODULE$ field immediately after the super ctor
    if (isStaticModule(clasz.symbol) && !isModuleInitialized &&
        jmethod.getName() == JMethod.INSTANCE_CONSTRUCTOR_NAME &&
        jname == JMethod.INSTANCE_CONSTRUCTOR_NAME)
    {
          isModuleInitialized = true;
          jcode.emitALOAD_0();
          jcode.emitPUTSTATIC(jclass.getName(),
                              nme.MODULE_INSTANCE_FIELD.toString,
                              jclass.getType());
     }
```

The snippet above explains the assignment `isModuleInitialized = false` near the beginning of `genCode(m: IMethod)`.

The applicability condition of the above transformation is:

> *An invocation `invc` to the super constructor (where `invc` occurs in a module constructor) is followed by instructions to assign `this` to the module-instance.*

TODO: Argue about `putstatic` being emitted (at least / at most) once, thus achieving exactly-once initialization for an accessed module.

1.4 The `readResolve` idiom (for serialization)

The compiler mentions `nme.readResolve` at the locations depicted in Figure 1.

```
if (clazz.isModuleClass && hasSerializableAnnotation(clazz)) {
  // If you serialize a singleton and then deserialize it twice,
  // you will have two instances of your singleton, unless you implement
  // the readResolve() method (see http://www.javaworld.com/javaworld/
  // jw-04-2003/jw-0425-designpatterns_p.html)
  // question: should we do this for all serializable singletons, or (as currently done)
  // only for those that carry a @serializable annotation?
  if (!hasImplementation(nme.readResolve)) ts += readResolveMethod
```

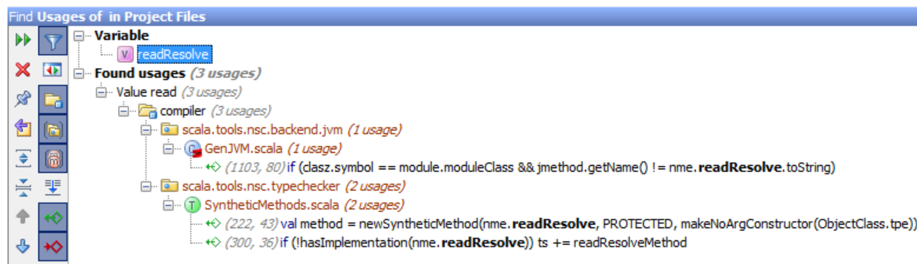

Figure 1: Places in the compiler where `nme.readResolve` is mentioned

```
}
```

TODO: come up with a platform-independent formulation of this idiom, or flag it as requiring separate maintenance on each platform.

2 Interplay between static-constructor and static class initializer

An IClass, whether static-module or not, may contain a static constructor, which serves as basis for the single `clinit` in the resulting `.class` (that's the "operational semantics" part in the title of this write-up). The opcodes making up that `clinit` are added by `addStaticInit(JClass, Option[IMethod])`, which is invoked only when one of the following holds:

- `if (isStaticModule(c.symbol) || serialVUID != None || clasz.bootstrapClass.isDefined)`, or
- none of the above but `if (c.containsStaticCtor) addStaticInit(jclass, c.lookupStaticCtor)`

TODO Q: Does a static-module always have a static-constructor defined? If not, `addStaticInit` may be invoked (under the first predicate, for some compiler run) with an `Option[IMethod]` of `None`. Apparently, the answer is "not always", otherwise there would be no `case None => legacyStaticInitializer(cls, clinit)` in `addStaticInit`.

2.1 What `clinit` looks like, given a static-constructor

In the snippet below, we elide two things:

- the case where `bootstrapClass.isDefined` (to install a handler for JDK 7's `invokedynamic` as part of `clinit`'s side-effects).
- the injection of opcodes to materialize the serialization idiom. Because it's platform-specific we don't discuss it (other than by noticing that this emission happens in the `case Some(m)` branch of the `mopt match` in `addStaticInit` (and only there), therefore the "serialization idiom" will be emitted only for classes with an already existing static constructor).

What is left is already formulated as a platform-independent ICode transformation (where `m` is the static-constructor `IMethod`):

```
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)))
}

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

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

Summing up, before emitting (with `genCode(m)`) bytecodes for the static-constructor, a `BasicBlock` is added to the `IMethod.code.blocks`:

1. the till-then last block is made to jump to the new last block,
2. in case this is the static-constructor of a static-module, two ICode instructions are appended (to initialize the *enclosing object*):
   - `NEW(REFERENCE(m.symbol.enclClass))`
   - `CALL_METHOD(m.symbol.enclClass.primaryConstructor, Static(true))`

The above depends on a primary constructor being always available in the `enclClass` of the static-constructor of a static-module. In general, a `primaryConstructor` is available in every `IClass` (or so I conclude from the out-commented `assert (c != NoSymbol)` below):

```
/** The primary constructor of a class */
def primaryConstructor: Symbol = {
  var c = info.decl(
    if (isTrait || isImplClass) nme.MIXIN_CONSTRUCTOR
    else nme.CONSTRUCTOR)
  c = if (c hasFlag OVERLOADED) c.alternatives.head else c
  //assert(c != NoSymbol)
  c
}
```

2.2 And what it looks like in the opposite case

```
private def legacyStaticInitializer(cls: JClass, clinit: JExtendedCode) {
  if (isStaticModule(clasz.symbol)) {
    clinit.emitNEW(cls.getName())
    clinit.emitINVOKESPECIAL(cls.getName(),
                             JMethod.INSTANCE_CONSTRUCTOR_NAME,
                             JMethodType.ARGLESS_VOID_FUNCTION)
  }

  serialVUID match { /*- serialization idiom OMITTED */ }
```

```
  if (clasz.bootstrapClass.isDefined)
    { legacyEmitBootstrapMethodInstall(clinit) /*- JDK 7 invokeDynamic */ }

  clinit.emitRETURN()
}
```

3 Mirror class for a top-level module without companion class

3.1 Purpose

Mirror classes are emitted to allow accessing object members (from Java or C#) in a concise manner. For example, given:

```
object Foo { def bar = 1 }
```

without a mirror class one would have to write `int x = Foo$.MODULE$.bar();` With the help of a static forwarder class that cuts down to: `int x = Foo.bar();`
No code emitted by the compiler does ever invoke those forwarders.

3.2 Code generation mechanics

A static-module can be top-level, i.e.

```
  def isTopLevelModule(sym: Symbol): Boolean =
    atPhase (currentRun.picklerPhase.next) {
      sym.isModuleClass && !sym.isImplClass &&
      !sym.isNestedClass /*- !sym.isNestedClass is important to being top-level :-) */
    }
```

Moreover, one such top-level static-module may lack a companion class. What's special about top-level static-modules lacking a companion class? Why, nothing, except that a mirror-class is emitted for them. The mirror-class is not shown when `-Xprint:icode -Xprint-icode` is used (Listing 1) because mirror classes (and the forwarders they contain) are emitted in a platform-dependent manner.

```
/** Dump a mirror class for a top-level module. A mirror class is a class containing
 *  only static methods that forward to the corresponding method on the MODULE instance
 *  of the given Scala object.
 */
def dumpMirrorClass(clasz: Symbol, sourceFile: String) {
  import JAccessFlags._
  val moduleName = javaName(clasz) // + "$"
  val mirrorName = moduleName.substring(0, moduleName.length() - 1)
  val mirrorClass = fjbgContext.JClass(ACC_SUPER | ACC_PUBLIC | ACC_FINAL,
                                   mirrorName,
                                   "java.lang.Object",
                                   JClass.NO_INTERFACES,
                                   sourceFile)
  addForwarders(mirrorClass, clasz)
  val ssa = scalaSignatureAddingMarker(mirrorClass, clasz.companionSymbol)
  addAnnotations(mirrorClass, clasz.annotations ++ ssa)
  emitClass(mirrorClass, clasz)
}
```

4 Adding static-forwarders to existing companion class

Similar to the forwarders added to a mirror class (Sec. 3), the forwarders added here also simplify interoperability from Java or C#. But remember: no code emitted by the compiler ever invokes those forwarders :-)

The applicability condition in this case is tested on a non-interface IClass (i.e. the following does not hold for it: `isStaticModule(c.symbol) || serialVUID != None || clasz.bootstrapClass.is`
The applicability condition comprises both tests below:

- the IClass in question is non-interface IClass with a companion module
- the IClass is top-level

If these conditions are met, static forwarders are added (to this companion class) for those object members whose names does not conflict with existing members (in this companion class).

```
/** Add forwarders for all methods defined in 'module' that don't conflict with
 *  methods in the companion class of 'module'. A conflict arises when a method
 *  with the same name is defined both in a class and its companion object (method
 *  signature is not taken into account). If 3rd argument cond is supplied, only
 *  symbols for which cond(sym) is true are given forwarders.
 */
def addForwarders(jclass: JClass, module: Symbol) { addForwarders(jclass, module, _ => true) }
```

The code listing for `addForwarders` (with applicability conditions for each static-forwarder) appears in Listing 2 and that for adding a particular forwarder in Listing 3. In both cases, code is emitted for JVM only.

TODO What the ICode-level formulation of the above is.

A Module initialization terminology in detail

```
/** The class with the same name in the same package as this module or case class factory. */
final def companionClass: Symbol = {
  if (this != NoSymbol)
    flatOwnerInfo.decl(name.toTypeName).suchThat(_ isCoDefinedWith this)
  else NoSymbol
}
```

```
/** The module or case class factory with the same name in the same package as this class. */
final def companionModule: Symbol =
  if (this.isClass && !this.isAnonymousClass && !this.isRefinementClass)
    companionModule0
  else NoSymbol
```

```
/**  For a module class: its linked class
 *   For a plain class: the module class of its linked module.
 *
 *  For instance
 *    object Foo
 *    class Foo
 *
```

```
 *  Then object Foo has a ‘moduleClass’ (invisible to the user, the backend calls it Foo$
 *  linkedClassOfClass goes from class Foo$ to class Foo, and back.
 */
final def linkedClassOfClass: Symbol =
  if (isModuleClass) companionClass else companionModule.moduleClass
```

```
/** If symbol is an object defition, it’s implied associated class, otherwise NoSymbol */
override def moduleClass: Symbol =
  if (hasFlag(MODULE)) referenced else NoSymbol
```

```
/** For a module its linked class, for a class its linked module or case factory otherwise. */
final def companionSymbol: Symbol =
  if (isTerm) companionClass
  else if (isClass)
    companionModule0
  else NoSymbol
```

Listing 1: No mirror-class in sight so far

```
object TopLevelStaticModuleWithoutCompanionClass extends java.lang.Object, ScalaObject {

  // fields:
        var tlsmwocc: Int

  // methods
  def tlsmwocc(): Int {
        locals:
        startBlock: 1
        blocks: [1]
        1:
          19  THIS
          19  LOAD_FIELD variable tlsmwocc
          19  RETURN (INT)
        }


  def tlsmwocc_$eq(x$1: Int (INT)): Unit {
        locals: parameter of setter tlsmwocc_=
        startBlock: 1
        blocks: [1]
        1:
          19  THIS
          19  LOAD_LOCAL parameter of setter tlsmwocc_=
          19  STORE_FIELD variable tlsmwocc (dynamic)
          19  RETURN (UNIT)
        }


  def <init>(): object TopLevelStaticModuleWithoutCompanionClass {
        locals:
        startBlock: 1
        blocks: [1]
        1:
          18  THIS
          18  CALL_METHOD java.lang.Objectjava.lang.Object.<init> (super())
          19  THIS
          19  CONSTANT (Constant(-1))
          19  STORE_FIELD variable tlsmwocc (dynamic)
          19  RETURN (UNIT)
        }

}
```

Listing 2: addForwarders

```
def addForwarders(jclass: JClass, module: Symbol, cond: (Symbol) => Boolean) {
  def conflictsIn(cls: Symbol, name: Name) =
    cls.info.members exists (_.name == name)

  /** List of parents shared by both class and module, so we don't add forwarders
   *  for methods defined there - bug #1804 */
  lazy val commonParents = {
    val cps = module.info.baseClasses
    val mps = module.companionClass.info.baseClasses
    cps.filter(mps contains)
  }
  /* The setter doesn't show up in members so we inspect the name
   * ... and clearly it helps to know how the name is encoded, see ticket #3004.
   * This logic is grossly inadequate! Name mangling needs a devotee.
   */
  def conflictsInCommonParent(name: Name) =
    commonParents exists { cp =>
      (name startsWith (cp.name + "$")) || (name containsName ("$" + cp.name + "$"))
    }

  /** Should method `m' get a forwarder in the mirror class? */
  def shouldForward(m: Symbol): Boolean =
    atPhase(currentRun.picklerPhase) (
      m.owner != definitions.ObjectClass
      && m.isMethod
      && !m.hasFlag(Flags.CASE | Flags.PRIVATE | Flags.PROTECTED | Flags.DEFERRED | Flags.SPECIALIZED)
      && !m.isConstructor
      && !m.isStaticMember
      && !(m.owner == definitions.AnyClass)
      && !module.isSubClass(module.companionClass)
      && !conflictsIn(definitions.ObjectClass, m.name)
      && !conflictsInCommonParent(m.name)
      && !conflictsIn(module.companionClass, m.name)
    )

  assert(module.isModuleClass)
  if (settings.debug.value)
    log("Dumping mirror class for object: " + module);

  for (m <- module.info.nonPrivateMembers; if shouldForward(m) ; if cond(m)) {
    log("Adding static forwarder '%s' to '%s'".format(m, module))
    addForwarder(jclass, module, m)
  }
}
```

Listing 3: addForwarder

```
/** Add a forwarder for method m */
def addForwarder(jclass: JClass, module: Symbol, m: Symbol) {
  import JAccessFlags._
  val moduleName = javaName(module) // + "$"
  val mirrorName = moduleName.substring(0, moduleName.length() - 1)
  val paramJavaTypes = m.info.paramTypes map toTypeKind
  val paramNames: Array[String] = new Array[String](paramJavaTypes.length);

  for (i <- 0 until paramJavaTypes.length)
    paramNames(i) = "x_" + i

  val mirrorMethod = jclass.addNewMethod(ACC_PUBLIC | ACC_FINAL | ACC_STATIC,
    javaName(m),
    javaType(m.info.resultType),
    javaTypes(paramJavaTypes),
    paramNames);
  val mirrorCode = mirrorMethod.getCode().asInstanceOf[JExtendedCode];
  mirrorCode.emitGETSTATIC(moduleName,
                          nme.MODULE_INSTANCE_FIELD.toString,
                          new JObjectType(moduleName));
  var i = 0
  var index = 0
  var argTypes = mirrorMethod.getArgumentTypes()
  while (i < argTypes.length) {
    mirrorCode.emitLOAD(index, argTypes(i))
    index = index + argTypes(i).getSize()
    i += 1
  }

  mirrorCode.emitINVOKEVIRTUAL(moduleName, mirrorMethod.getName(),
                              mirrorMethod.getType().asInstanceOf[JMethodType])
  mirrorCode.emitRETURN(mirrorMethod.getReturnType())

  addRemoteException(mirrorMethod, m)
  // only add generic signature if the method is concrete; bug #1745
  if (!m.hasFlag(Flags.DEFERRED))
    addGenericSignature(mirrorMethod, m, module)

  val (throws, others) = splitAnnotations(m.annotations, definitions.ThrowsClass)
  addExceptionsAttribute(mirrorMethod, throws)
  addAnnotations(mirrorMethod, others)
  addParamAnnotations(mirrorMethod, m.info.params.map(_.annotations))
}
```