More about the internal workings of `GenMSIL`

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

August 4th, 2010

Abstract

Notes for future reference about existing functionality in Scala.NET.

Contents

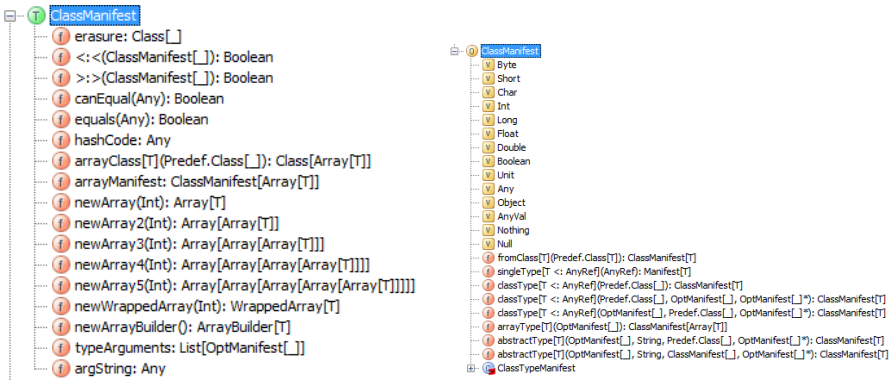
1 Intro

These notes focus on aspects not covered in previous write-ups: *Notes on GenMSIL*[1], *Notes on GenICode*[2], and *Exception handling: from ICode to CIL*[3].

1.1 Generic array creation

```
def mkArray[T](elems: Seq[T])(implicit cm: ClassManifest[T]): Array[T] = {
 val result = new Array[T](elems.length)
 var i = 0
 for (elem <- elems) {
  result(i) = elem
  i += 1
 }
 result
}
```

[1] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q2/GenMSIL.pdf`
[2] `http://www.sts.tu-harburg.de/~mi.garcia/ScalaCompilerCorner/GenICode01.pdf`
[3] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q2/ExceptionHandling.pdf`



(a) (b)

Figure 1: `ClassManifest`

The resulting code has to handle all array creations, including those where the element type is a valuetype.

One of the `ClassManifest.newArray` overloads is invoked (Figure 1):

```
def mkArray[T >: Nothing <: Any](elems: Seq[T], cm: scala.reflect.ClassManifest[T]): Array[T] = {
  val result: Array[T] = cm.newArray(elems.length());
  var i: Int = 0;
  elems.foreach[Unit]({
    final <synthetic> class $anonfun extends scala.runtime.AbstractFunction1[T,Unit] {
      def this(): anonymous class $anonfun = {
        $anonfun.super.this();
        ()
      };
      final def apply(elem: T): Unit = {
        result.update(i, elem);
        i = i.+(1)
      }
    };
    (new anonymous class $anonfun(): (T) => Unit)
  });
  result
};
```

For comparison, one of the possible ways on .NET is:

```
.method public hidebysig instance !!T[] dynarrcreation<T>() cil managed
{
  // Code size      15 (0xf)
  .maxstack 1
  .locals init ()
  IL_0000: nop
  IL_0001: ldc.i4.s  10
  IL_0003: newarr    !!T
  IL_000e: ret
} // end of method Program::dynarrcreation
```

1.2 `flatten` and `liftcode` are JVM phases

In retrospect, it only makes sense: there's no need in CLR to make all classes top-level (`flatten`). On JVM, `liftcode` runs after `refchecks` and before `uncurry`, while `flatten` runs after `constructors` and before `mixin` and `cleanup`.

```
trait JavaPlatform extends Platform[AbstractFile] {
  . . .
  def platformPhases = List(
    flatten,   // get rid of inner classes
    liftcode,  // generate reified trees
    genJVM     // generate .class files
  ) ::: depAnalysisPhase
  . . .
```

2 Bits and pieces of `GenMSIL`

What we're talking about:
`http://lampsvn.epfl.ch/trac/scala/browser/scala/trunk/src/compiler/scala/tools/nsc/backend/msil/GenMSIL.scala`

`initAssembly()` serves the purpose of setting `outDir` and `srcPath`, give a name to the assembly (`assemName`), and creating the `massembly` to own the `mmodule` which in turn will own almost all type builders about to be emitted (optionally owning one global method also, the entry point). `mmodule` does not directly own the `TypeBuilder`s for nested classes:

```
  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)
  }
```

The above snippet comes from `createTypeBuilder(IClass)`. It takes 72 LOC but it basically adds pairs to `CLRTypes.types` (from `IClass.symbol` to `TypeBuilder`). BTW, it adds no pairs for the opposite direction (`CLRTypes.sym2type`) because, when emitting CIL, lookups only go from symbols to stuff in `ch.epfl.lamp.compiler.msil`. There are no reverse maps for fields, constructors, or methods.

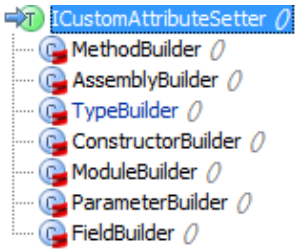
By the time `createTypeBuilder(IClass)` is invoked for a nested class that method will have been invoked for its containing class because of the way `GenICode` populates the `ICodes.classes` map (from `Symbol` to `IClass`) as shown in Listing 1. And also because `classes.values` returns values in the same order that pairs were inserted. For details, dig into:

```
  /* Override to avoid tuple allocation in foreach */
  override def values: collection.Iterable[B] = new DefaultValuesIterable {
    override def foreach[C](f: B => C) = foreachEntry(e => f(e.value))
  }
```

Listing 1: Why containing classes come first, before nested classes

```
////////////////// Code generation ///////////////////////

def gen(tree: Tree, ctx: Context): Context = tree match {
  case EmptyTree => ctx

  case PackageDef(pid, stats) =>
    gen(stats, ctx setPackage pid.name)

  case ClassDef(mods, name, _, impl) =>
    log("Generating class: " + tree.symbol.fullName)
    val outerClass = ctx.clazz
    ctx setClass (new IClass(tree.symbol) setCompilationUnit unit)
    addClassFields(ctx, tree.symbol);
    classes += (tree.symbol -> ctx.clazz) /*- containing classes visited before nested ones.
                                           Keep in mind that the flatten phase is a JVM phase. */
    unit.icode += ctx.clazz
    gen(impl, ctx)
    ctx.clazz.methods = ctx.clazz.methods.reverse // preserve textual order
    ctx.clazz.fields = ctx.clazz.fields.reverse // preserve textual order
    ctx setClass outerClass

  . . .
```

`createClassMembers(IClass)` adds one last time more pairs to `CLRTypes` maps (`fields`, `constructors`, `methods`), including attributes (TODO), and adds module-instance fields, tracking the thus created `FieldBuilders` in the `CLRTypes.fields` map using `iclass.symbol` as key.




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

Before generating CIL instructions by invoking `genMethod(IClass)` and with `CLRTypes` maps already complete, `genClass(IClass)` does the following:

- for an `IClass.symbol` annotated with `scala.cloneable`, add a `clone()` method which will invoke `MemberwiseClone` on `this` (this method is added provided no user-defined `clone()` or `Clone()` method exists)
- dump mirror class for top-level module without `companionClass` (Figure 2).
- `addSymtabAttribute(sym, tBuilder)`

Listing 2: `companionClass`

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

- `addAttributes(tBuilder, sym.annotations)`
- emit CIL for method bodies

2.1 Code blocks to CIL

`genMethod(IMethod)` starts by clearing the *local builders* map (from an `icode.Members.Local` for each value param and local variable in the method, to `LocalBuilder`), also setting the `index` of each such `Local` in the `IMethod.params` and `IMethod.locals`. This map will be populated once an `ILGenerator` is available by calling `ILGenerator.DeclareLocal(MsilType)`.

An `ILGenerator` is obtained from the method's `ConstructorBuilder` (for an `IMethod.symbol.isClassConstructor`) or `MethodBuilder` (provided that method builder is not `IsAbstract()`).

`genMethod(IMethod)` completes after invoking `genCode(IMethod)` on its argument.

The branch instructions that `genCode(IMethod)` will emit for ICode `JUMP(whereto)`, `CJUMP(success, failure, cond, kind)`, `CZJUMP(success, failure, cond, kind)`, and `RETURN(kind)` take into account whether the control transfer leaves a block covered by an exception handler or a `finally` block. For example:

```
case JUMP(whereto) =>
  val (leaveHandler, leaveFinally, lfTarget) = leavesHandler(block, whereto)
  if (leaveHandler) {
    if (leaveFinally) {
      if (lfTarget.isDefined) mcode.Emit(OpCodes.Leave, lfTarget.get)
      else mcode.Emit(OpCodes.Endfinally)
    } else
      mcode.Emit(OpCodes.Leave, labels(whereto))
  } else if (next != whereto)
    mcode.Emit(OpCodes.Br, labels(whereto))
```

2.2 `maxstack` size, `IL_hhhh`, and opcode-arg downcasting

A method's `maxstack` size is computed for us in `ILGenerator.Emit(...)` overloads, Figure 2 (those are the public ones, lowercase `emit(...)` are private).

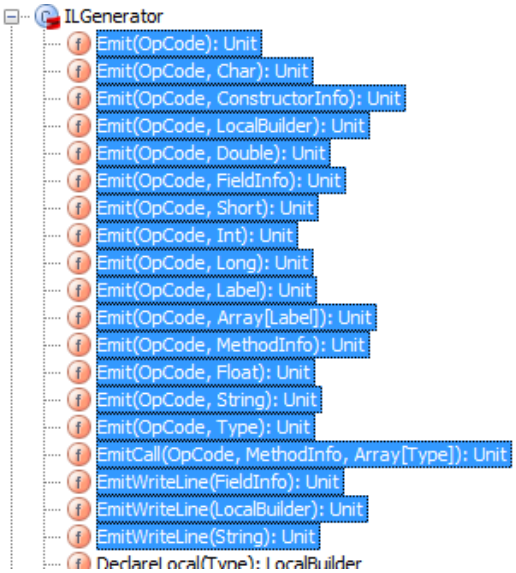


Figure 2: `ILGenerator.Emit(...)`

Whichever is invoked, all of them invoke in turn `emit(opcode: OpCode, arg: Object, overridePOPUSH: Int)`, i.e. the stack size delta (`overridePOPUSH`) is given by the invoking `ILGenerator.Emit(...)`.

What about `BasicBlocks` in diamond configuration, where each intermediate block increases the stack height by `D`, does that get added twice? No way, because the delta is added to the stack size tracked by `lastLabel`, and upon each block being generated (`genBlock` invoked) one of the first things done is `mcode.MarkLabel(labels(block))` which merges the stack height of `labels(block)` with that of `lastLabel` (all of `mcode`, `labels`, and `lastLabel` are owned by `BytecodeGenerator`).

```
// private emit with Object Argument and override POPUSH
private def emit(opcode: OpCode, arg: Object, overridePOPUSH: Int) {
  // add label, opcode and argument
  labelList.add(lastLabel)
  opcodeList.add(opcode)
  argumentList.add(arg) /*- opcode args (may be null) kept in a List[Object] */
  // compute new lastLabel (next label)
  val stackSize: Int = lastLabel.getStacksize() + overridePOPUSH
  if (stackSize < 0) {
    throw new RuntimeException("ILGenerator.emit(): Stack underflow in method: " + owner)
  }
  if (stackSize > maxstack)
    maxstack = stackSize
  var address: Int = lastLabel.getAddress() + opcode.CEE_length /*- position in method's IL stream */
  if (opcode.CEE_opcode == OpCode.CEE_SWITCH) {
    address = address + 4 * arg.asInstanceOf[Array[Label]].length
  }
  lastLabel = new Label.NormalLabel(address, stackSize)
              /*- Label.toString() has format IL_hhhh with hex digits */
  pc = pc + 1
}
```

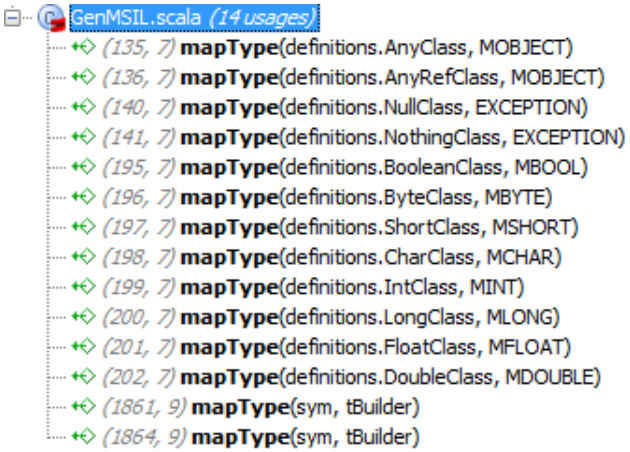
Given that opcode args are kept in a `List[Object]`, `ILPrinterVisitor.caseOpCode(OpCode)` has to downcast depending on the opcode when serializing into ILAsm. Examples:

```
def caseOpCode(opCode: OpCode) {
  . . .
  if (opCode == OpCode.Ldstr) {
    print(msilString(argument.toString()))
  } else if (opCode == OpCode.Switch) {
    // switch ( <labels> )
    print("(")
    val targets = argument.asInstanceOf[Array[Label]]
    . . .
  } else if (opCode == OpCode.Call || opCode == OpCode.Callvirt || opCode == OpCode.Jmp || opCode == OpCode.Ld
    . . .
    printSignature(argument.asInstanceOf[MethodBase])
  } else if (opCode == OpCode.Newobj) {
  . . .
```

2.3 Type and method replacements just before CIL-emit

2.3.1 Type replacements

Not all Scala types have CLR counterparts (`scala.Any`, `scala.Null`, `scala.Nothing`). In other words, even if you didn't use `System.Object` at some place in your program, it may still show up in the emitted CIL because of:



2.3.2 Static-for-instance method replacements

`BytecodeGenerator.initMappings()` invokes `mapType` and `mapMethod` to replace whatever `MsilType` resp. `MethodInfo` appeared as value in `CLRTypes`'s maps, so that by the time `ILPrinterVisitor` is run their replacements are found in symbol → `msil` lookups.

For `MethodInfo`s, their signatures usually match but there's also:

```
/*
 * add mapping between method with name and paramTypes of clazz to
 * method with newName and newParamTypes of newClass (used for instance
```

```
 * for "wait")
 */
private def mapMethod(
  clazz:    Symbol,   name:   Name,    paramTypes:    Array[Type],
  newClass: MsilType, newName: String, newParamTypes: Array[MsilType]) {
    val methodSym = lookupMethod(clazz, name, paramTypes)
    assert(methodSym != null, "cannot find method " + name + "(" +
          paramTypes + ")" + " in class " + clazz)
    mapMethod(methodSym, newClass, newName, newParamTypes)
  }

/*
```

which is invoked for example with:

```
     mapMethod(JOBJECT, nme.notifyAll_, jEmpty,
               MMONITOR, "PulseAll",    mObject1)
```

i.e. a no-args instance method is mapped to a one-arg static method:

```
// from java.lang.Object
   public final void notifyAll()

// from System.Threading.Monitor
   public static void PulseAll(
        Object obj
   )
```

In the example above, a `MethodInfo` is found by looking in the `MsilType` given by `MMONITOR`, for a method named `"PulseAll"` with a param-list having types `List(MMOBJECT)`.

Static-for-instance method replacements result in verifiable CIL. First, they are tracked in `BytecodeGenerator.dynToStatMap`:

```
    // java instance methods that are mapped to static methods in .net
    // these will need to be called with OpCodes.Call (not Callvirt)
    val dynToStatMapped: HashSet[Symbol] = new HashSet()
```

And second, although the stack top is ok, a static method can't be invoked with `callvirt`:

```
val methodInfo: MethodInfo = getMethod(msym)
(style: @unchecked) match {
  case SuperCall(_) =>
    mcode.Emit(OpCodes.Call, methodInfo)
  case Dynamic =>
    if (dynToStatMapped(msym)) {
      mcode.Emit(OpCodes.Call, methodInfo) /*- a static method can't be invoked with callvirt */
    } else {
      mcode.Emit(OpCodes.Callvirt, methodInfo)
    }
  case Static(_) =>
    mcode.Emit(OpCodes.Call, methodInfo)
}
```

Note: the above works only for a no-arg instance method mapped to a static one-arg method.

2.3.3 Method replacements for `scala.AnyRef` and `scala.String`

All method replacements entered in `BytecodeGenerator.initMappings()` are for method symbols in `definitions.ObjectClass` or `definitions.StringClass`.

In `StandardDefinitions`, there are definitions for top types:

```
// top types
def AnyClass  : Symbol
def AnyValClass: Symbol
def AnyRefClass: Symbol
def ObjectClass: Symbol
```

and for fundamental reference classes:

```
// fundamental reference classes
def SymbolClass : Symbol
def StringClass : Symbol
def ClassClass : Symbol
```

In `Definitions.init`, methods symbols (with flag `FINAL`) are entered for `ObjectClass` and `StringClass`, in addition to whatever `TypeParser` may find in assembly metadata. Additionally, some method symbols are also entered for `scala.Any`.

```
// members of class scala.Any
   Any_==
   Any_!=
   Any_equals
   Any_hashCode
   Any_toString
   Any_##
   Any_isInstanceOf
   Any_asInstanceOf

// members of class java.lang.{ Object, String }
   Object_##
   Object_==
   Object_!=
   Object_eq =
   Object_ne =
   Object_synchronized =

// members of scala.String
   String_+
```