Volatile fields, Sub-line step debugging, and a few TODOs (plugins, properties)

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

April 5th, 2011

Abstract

Notes about implementation aspects of Scala.NET. Unless you're hacking the compiler these notes should be of no consequence to you :-)

Contents

1 Handling `volatile` fields

Those notes document how volatile fields are handled by `GenMSIL` and during metadata-parsing. Once `GenMSIL` is replaced to emit binary assemblies this implementation will have to be revisited. And documenting is useful anyway.

1.1 How it's done `forJVM`

- nothing special done in `ClassfileParser.parseField()` for volatile fields.
- when emitting Java bytecode, it's enough to mark as such the definition of a volatile field. In contrast, MSIL also requires to prefix with `volatile.` each read/write to the field.

```
// from GenJVM:
def genField(f: IField) {
  if (settings.debug.value)
    log("Adding field: " + f.symbol.fullName)

  val attributes = f.symbol.annotations.map(_.atp.typeSymbol).foldLeft(0) {
    case (res, TransientAttr) => res | ACC_TRANSIENT
    case (res, VolatileAttr) => res | ACC_VOLATILE
    case (res, _)          => res
  }

  . . .
```

1.2 Background

In terms of ILAsm syntax, a field is marked as volatile as follows:

```
.field private int32 modreq( [mscorlib]'System.Runtime.CompilerServices.IsVolatile' ) 'f'
```

Volatile fields are a special case of `modreq`, a required custom modifier. Quoting from the CIL spec, Partition II, §7.1.1:

> *Custom modifiers, defined using `modreq` (required modifier) and `modopt` (optional modifier), are similar to custom attributes (§21) except that modifiers are part of a signature rather than being attached to a declaration. Each modifer associates a type reference with an item in the signature.*

Practicalities: quoting from a discussion[1] on the difference between "type equivalence" and "signature matching" in CLR:

> *It means, that `typeof(string)` is the same as `typeof(string modopt(NonNullType))` at runtime (except signature matching).*
>
> ***Furthermore Reflection was designed not as managed Meta-Data API, but rather as runtime type information.*** *Therefore Reflection takes loaded types as parameters and that leads to the results you can see.*

[1] `http://connect.microsoft.com/VisualStudio/feedback/details/282406/modopts-not-supported-by-generics-in-clr`

More background:

- http://weblog.ikvm.net/PermaLink.aspx?guid=82
- http://jasper-22.blogspot.com/2010/11/subterranean-il-custom-modifiers.html
- Ch. 8 in the *Expert IL* book [1].

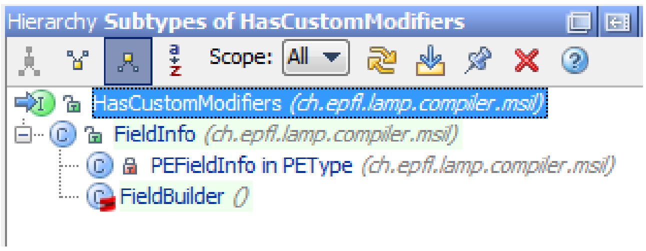
1.3 Keeping track of custom mods: `PECustomMod` helped by `CustomModifier`

As the ILAsm syntax suggests, one type is "*marked*" with one or more "*custom mods*", where each "*custom mod*" in turn comprises a "*marker type reference*" and the indication whether the marker is required or optional. We keep track of all this in `PECustomMod`:

```
/**
 * A PECustomMod holds the info parsed from metadata per the CustomMod production in Sec. 23.2.7, Partition II.
 * */
public final class PECustomMod {

    public final Type marked;
    public final CustomModifier[] cmods;
```

Those locations that can marked with custom modifiers (fow now in `FieldInfo`, should also be added to `ParameterInfo`, and `PropertyInfo`) implement a tag interface `HasCustomModifiers`:



1.4 GenMSIL

Emitting a field access:

```
def loadFieldOrAddress(field: Symbol, isStatic: Boo
  if (settings.debug.value)
    log(msg + " with owner: " + field.owner +
        " flags: " + Flags.flagsToString(field.owne
  var fieldInfo = fields.get(field) match {
    case Some(fInfo) => fInfo
    case None =>
      val fInfo = getType(field.owner).GetField(msi
      fields(field) = fInfo
      fInfo
  }
  if (fieldInfo.IsVolatile) {
    mcode.Emit(OpCodes.Volatile)
  }
  if (!fieldInfo.IsLiteral) {
```

Before that, the field was created. Please notice that some `sym.annotations` result in CLR attributes, while others in CLR custom modifiers:

```
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 fieldTypeWithCustomMods =
      new PECustomMod(msilType(sym.tpe), customModifiers(sym.annotations))
    val fBuilder = mtype.DefineField(msilName(sym),
                                     fieldTypeWithCustomMods,
                                     attributes)
    fields(sym) = fBuilder
    addAttributes(fBuilder, sym.annotations)
  } // all iclass.fields iterated over
```

1.5 Comparison with `System.Reflection.Emit`

In `System.Reflection.Emit`, a few factory methods take custom modifiers as input. For example:

```
public FieldBuilder DefineField(
        string fieldName,
        Type type,
        Type[] requiredCustomModifiers,
        Type[] optionalCustomModifiers,
        FieldAttributes attributes
)
```

A disadvantage of separately defining (and later retrieving) optional and required cmods is that the occurrence order is lost. We avoid that by having a single array hold all custom modifers (fow now in `FieldInfo`, should also be added to `ParameterInfo`, and `PropertyInfo`):

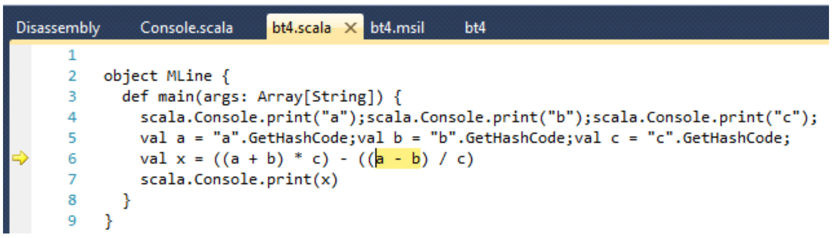
```
// once they are added,
// they are added all at once
// and never modified
public final CustomModifier[] cmods = null;
```

Also in `System.Reflection.Emit`, three classes allow `GetOptionalCustomModifiers` and `GetRequiredCustomModifiers`, and we add them to our API too.

```
TODO add HasCustomModifiers support to
  ParameterInfo and
  PropertyInfo.
```

2 Sub-line step debugging (without `-Yrangepos`)

Here's a screen capture conveying how it works:



2.1 Background

On CLR, debuggers can highlight a *text range* with each debug step, thus giving better feedback when debugging closures, for example. ILAsm has syntax for this [1, p. 403]:

> *The* `.line <start_line>[,<end_line>][:<start_col> [,<end_col>]] [<file_name>]` *directive identifies the line and column in the original source file that are responsible for the IL code that follows the .line directive.*

Quoting from "Compiling in Debug Mode" [1, Ch. 19]:

- *If your compiler generates ILAsm source code, it must insert* `.language` *and* `.line` *directives at the appropriate points.*
- *If you are round-tripping a module compiled from a high-level language, use the disassembler option* `/LINENUM` *(or* `/LIN`*).*
- *In any case, don't forget to use one of the PDB-generating options of the ILAsm compiler:* `/DEB`*,* `/DEB=OPT`*,* `/DEB=IMP`*, or* `/PDB` *(the last option generates the PDB file but doesn't emit the* `DebuggableAttribute`*).*

Sidenotes:

- Related forum: "Building Development and Diagnostic Tools for .Net"[2].
- Using `System.Reflection` to emit sub-line range information[3]:

2.2 Implementation

In `GenMSIL`, we now have:

```
for (instr <- block) {
  try {
    val currentLineNr = instr.pos.line
    val skip = if(instr.pos.isRange) instr.pos.sameRange(lastPos) else (currentLineNr == lastLineNr);
    if(!skip) {
      val fileName = if(dbFilenameSeen) "" else {dbFilenameSeen = true; ilasmFileName(clasz)};
```

[2] `http://social.msdn.microsoft.com/Forums/en/netfxtoolsdev/threads` forum

[3] `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/ScalaNetBackend.pdf`

```
      if(instr.pos.isRange) {
        val startLine = instr.pos.focusStart.line
        val endLine  = instr.pos.focusEnd.line
        val startCol = instr.pos.focusStart.column
        val endCol   = instr.pos.focusEnd.column
        mcode.setPosition(startLine, endLine, startCol, endCol, fileName)
      } else {
        mcode.setPosition(instr.pos.line, fileName)
      }
      lastLineNr = currentLineNr
      lastPos = instr.pos
    }
  } catch { case _: UnsupportedOperationException => () }
```

When emitting `.line`, it's enough to include the full filename just once per method, thus reducing filesize. That's what `dbFilenameSeen` is for.

In `ILPrinterVisitor`, source locations for instructions are printed as-is (they are strings by that time), as shown next:

```
val label = itL.next
val oOpt = code.lineNums.get(label)
if (oOpt.isDefined) {
  println(".line     " + oOpt.get)
}
```

because `lineNums` is

```
val lineNums = scala.collection.mutable.Map.empty[Label, String]
```

The ready-made string for the source location is provided by calling a `setPosition` overload in `ILGenerator`.

For all of the above to work in the `GenMSIL` backend, the following is needed during parsing (in `SourceFileParser`):

```
def r2p(start: Int, mid: Int, end: Int): Position =
  if(forMSIL) new util.RangePosition(source, start, mid, end)
  else rangePos(source, start, mid, end)
```

2.3 FYI: Why we want to do without `-Yrangepos`

```
TODO Current support is fine for small programs,
    but the compiler crashes with -Yrangepos when compiling, say, the library.
```

Thus the following won't do (in `nsc.Main`):

```
val compiler =
  if (settings.Yrangepos.value && (settings.target.value != "msil"))
    new interactive.Global(settings, reporter)
  else new Global(settings, reporter)
```

During parsing, the following overrides determine whether offset or range positions are created:

- What is overridden, `nsc.symtab.Positions`:

```
package scala.tools.nsc
package symtab

import scala.tools.nsc.util.{ SourceFile, Position, OffsetPosition, NoPo

trait Positions {
self: scala.tools.nsc.symtab.SymbolTable =>

  def rangePos(source: SourceFile, start: Int, point: Int, end: Int) =
    new OffsetPosition(source, point)
```

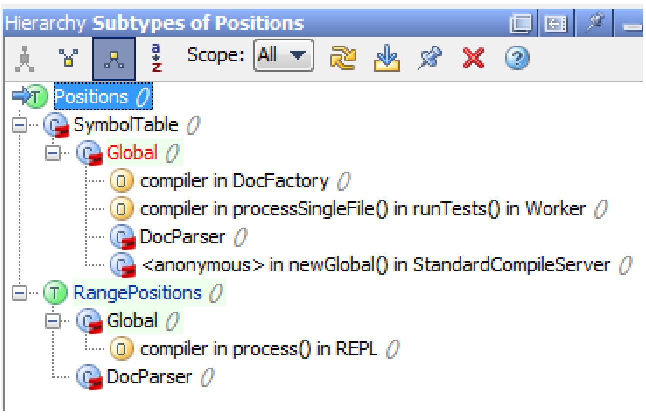
- as follows in trait `RangePositions`

```
trait RangePositions extends Trees with Positions {
self: scala.tools.nsc.Global =>

  case class Range(pos: Position, tree: Tree) {
    def isFree = tree == EmptyTree
  }

  override def rangePos(source: SourceFile, start: Int, point: Int, end: Int) =
    new RangePosition(source, start, point, end)
```

- and trait `RangePositions` in turn as base class of `nsc.interactive.Global` (2nd "`Global`" below)



In detail :-) `nsc.Global` extends:

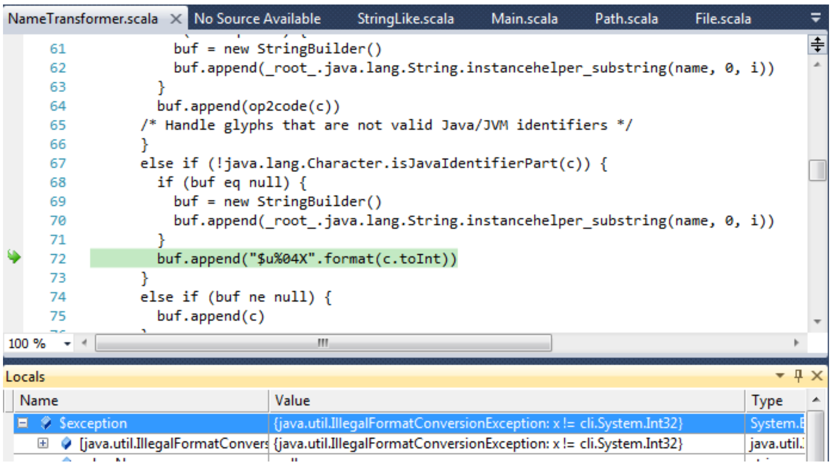
```
class Global(var settings: Settings, var reporter: Reporter) extends SymbolTable
                                                              with CompilationUnits
                                                              with Plugins
                                                              with PhaseAssembly
```

in constrast, `nsc.interactive.Global` extends:

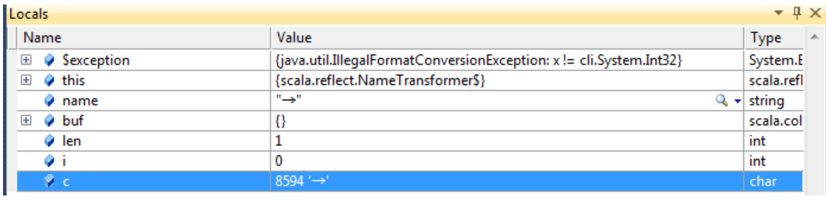
```
class Global(settings: Settings, reporter: Reporter, projectName: String = "")
  extends scala.tools.nsc.Global(settings, reporter)
     with CompilerControl
     with RangePositions
     with ContextTrees
     with RichCompilationUnits
     with Picklers
```

3 Workarounding two behavioral differences in the way we use IKVM vs. JDK

3.1 Behavioral difference 1



In detail, the above is due to `NameTransformer` receiving the Unicode "→" character:



Solution:

```
val tmp : String = {
  val h = java.lang.Integer.toHexString(c.toInt)
  "$u" + "000".take(4 - h.size) + h
}
buf.append(tmp)
```

3.2 Behavioral difference 2

In `scala.tools.nsc.io.File`:

```
  // this is a workaround for http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6503430
  // we are using a static initializer to statically initialize a java class so we don't
  // trigger java.lang.InternalErrors later when using it concurrently. We ignore all
  // the exceptions so as not to cause spurious failures when no write access is available,
  // e.g. google app engine.
  try {
    import Streamable.closing
    val tmp = JFile.createTempFile("bug6503430", null, null)
    try closing(new FileInputStream(tmp)) { in =>
      val inc = in.getChannel()
      closing(new FileOutputStream(tmp, true)) { out =>
```

```
        out.getChannel().transferFrom(inc, 0, 0)
      }
    }
    finally tmp.delete()
  }
  catch {
    case _: IllegalArgumentException | _: java.lang.IllegalStateException | _: IOException | _: java.lang.Securi
    case _ => () /*- needed because IKVM can also throw an IllegalArgument (or NumberConversion, don't remember
  }
```

4 TODO: Compiler plugins

IKVM can do Java classloading on .NET, and thus it is possible to have `scala-compiler.jar` compiled by `ikvmc` into an `.exe`, and run it with `-Xplugin` to load *a compiler plugin packed (as usual) as a `.jar`*.

In this section we explore a different approach: using the bootstrapped `scalacompiler.exe` to dynamically load a compiler plugin *that was packed as `.dll`*.

First we translate the sources of a compiler plugin using `jdk2ikvm` (like, `jdk2ikvm` itself), and compile using Scala.NET into `jdk2ikvm.dll`. The following command line allows debugging its loading:

```
scalacompiler.exe
  -Ystop-after:superaccessors
  -P:jdk2ikvm:output-directory:c:\temp\discard
  -Xplugin jdk2ikvm.dll
  -sourcepath Z:\scalaproj\sn5\myplugins\jdk2ikvm\src
  -d c:\temp\discard
  @C:\temp\out-jdk2ikvm\sn5-src-jdk2ikvm.txt
  -target:msil -Ystruct-dispatch:no-cache -Xassem-name jdk2ikvm -Xassem-extdirs c:\temp\dirC -no-specialization
  -Yrangepos
```

The list of compiler plugins is built (Figure 1) but the following causes later a `ZipException`

```
/** Try to load a plugin description from the specified
 *  file, returning <code>None</code> if it does not work.
 */
private def loadDescription(jarfile: Path): Option[PluginDescription] =
  // XXX Return to this once we have some ARM support
  if (!jarfile.exists) None
  else try {
    val jar = new JarFile(jarfile.jfile)

    try {
      jar getEntry PluginXML match {
        case null  => None
        case entry =>
          val in = jar getInputStream entry
          val packXML = XML load in
          in.close()
```

```
TODO: To load plugin.xml from a .dll, use the following Assembly method

public virtual Stream GetManifestResourceStream(
       string name
)
```

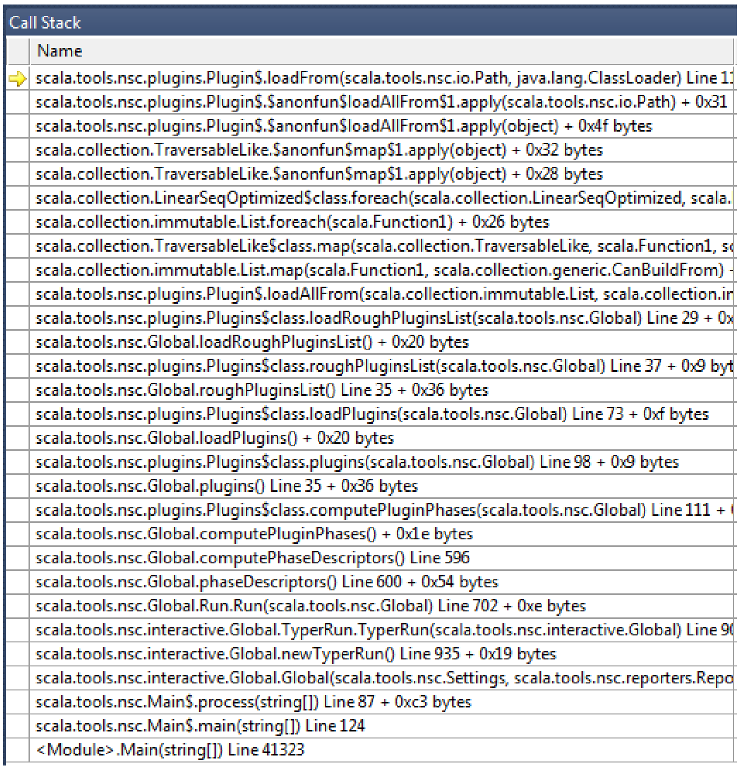


Figure 1: Sec. 4

See also:

- Microsoft .NET Framework Resource Basics,
  http://msdn.microsoft.com/en-us/library/ms950960.aspx

After that, it's time for dynamic class loading:

```
  /** Loads a plugin class from the named jar file.

   *  @return <code>None</code> if the jar file has no plugin in it or
   *                            if the plugin is badly formed.
   */
  def loadFrom(jarfile: Path, loader: java.lang.ClassLoader): Option[AnyClass] =
    loadDescription(jarfile) match {
      case None => None
      case Some(pdesc) =>
        try Some(loader loadClass pdesc.classname) catch {
          case _: java.lang.Exception =>
            println("Warning: class not found for plugin in %s (%s)".format(jarfile,
            None
      }
    }
```

5 TODO: the ILAsm `.language` directive, and language-specific *Expression Evaluators* in VS

ILAsm `.language` directive [1]:

> *The* `.language <Language_GUID>[,<Vendor_GUID>[,<Document_GUID>]]` *directive defines the source language and, optionally, the compiler vendor and the source document type. This information is used by the Visual Studio debugger, which displays source code of different languages differently.*

Example for C#:

```
.language '{3F5162F8-07C6-11D3-9053-00C04FA302A1}',
          '{994B45C4-E6E9-11D2-903F-00C04FA302A1}',
          '{5A869D0B-6611-11D3-BD2A-0000F80849BD}'
```

The *language GUID* makes VS pick an *Expression Evaluator* during debugging[4]:

> *The VS debugger selects the appropriate EE for a stack frame based on the "language" of the code at that stack frame. For your purposes, the interpreter will be a "language". A language is identified by a pair of guids: the language guid and the vendor guid.*
>
> *. . .*
>
> *When VS enters break mode and the current stack frame is in your interpreter, VS will read the language and vendor guids in your interpreter's module header, then VS will try to find an EE registered with those guids. (If anything goes wrong, it falls back to the C# EE with no warning or log of any kind.)*

```
TODO
```

6 TODO: Emitting metadata for CLR properties after collecting (getter, setter) pairs

6.1 Taking a page from `GenJVM`

CLR properties are not unlike JavaBeans getter and setter, thus we look for inspiration in `GenJVM`:

```
  var fieldList = List[String]()
  for (f <- clasz.fields if f.symbol.hasGetter;
         val g = f.symbol.getter(c.symbol);
         val s = f.symbol.setter(c.symbol);
         if g.isPublic && !(f.symbol.name startsWith "$")) // inserting $outer breaks the bean
    fieldList = javaName(f.symbol) :: javaName(g) :: (if (s != NoSymbol) javaName(s) else null) :: fieldList
```

The above is run only for an `IClass c` such that

[4] http://social.msdn.microsoft.com/Forums/en/vsx/thread/2e412c53-b24b-4506-af00-5cca6d5257a7

```
if (c.symbol hasAnnotation BeanInfoAttr)
  genBeanInfoClass(c)
```

6.2 And now in `GenMSIL`

In `createClassMembers0`, a class' fields and methods are iterated to instantiate `FieldBuilder`s and `MethodBuilder`s resp. During the iteration of methods, getter/setter correspondences can be gathered. Based on them, `PropertyBuilder`s are instantiated before `createClassMembers0` is over.

```
TODO Well-formedness of CLR properties covered in:
   - Sec 8.11.3 in Partition I
   - Sec 17 in Partition II
```

References

[1] Serge Lidin. *Expert .NET 2.0 IL Assembler.* Apress, Berkely, CA, USA, 2006.