The `jdk2ikvm` source-to-source converter

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

December 14th, 2010

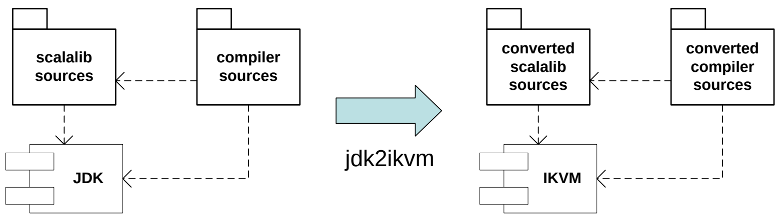
Abstract

Our previous prototype (in the form of a patched compiler) applied the *JDK to IKVM* conversion *on the fly*, emitting .NET assemblies as output. That division of labor turned out to be inflexible (Sec. 8.2).

Thus we factor out the JDK-to-IKVM functionality into a compiler plugin (`jdk2ikvm`) that outputs Scala.NET source files, given JDK-based counterparts as input. The Scala.NET compiler compiles them as usual (IKVM-dependencies are now explicit in source code, and the IKVM library can be linked as any other). After removing the special-casing for IKVM, the original architecture of `scalac` is gained back, sharing again most of the codebase between the JVM and .NET backends. As before, `scalac` can also run as a cross-compiler.

We field-test `scalac` and `jdk2ikvm` for bootstrapping, which comprises:

- output Scala.NET sources from unmodified JDK-based trunk sources
- let the cross-compiler produce `scalacompiler.exe` from them
- from then on, use `scalacompiler.exe` (not the cross-compiler) to compile the output of `jdk2ikvm`



Nota bene: we refer to `jdk2ikvm` as a source-to-source converter to emphasize its input-output behavior, however it does not limit itself to surface syntax, operating instead on typed ASTs that are later pretty-printed.

Nota bene 2: In more detail, given ASTs typed in `forJVM` mode, the plugin trades some subtrees for *untyped parse trees*. Once pretty-printed, they are compiled (and thus typecheckd) in `forMSIL` mode. `jdk2ikvm` does not re-type after transform (it can't retype its own output: the IKVM library is a `.dll`, not a `.jar`).

Sec. 1.3 covers possibilities opened up by `jdk2ikvm` as blueprint for other language-aware pre-processors for Scala, and in connection with `Scalify`, a prototype *Java* $\rightarrow$ *Scala* converter.

An accompanying document, *Learning and doing `scalac` transformations the easy way: via unparsing*[1], describes unparsing in more detail.

[1] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/Unparsing.pdf`

Contents

1 Intro

We reuse file utilities adapted from Mark Harra's Scala X-Ray, `http://www.scala-lang.org/node/1509`, to interact with the compiler and to output a tree of source files whose structure mirrors that of the input.

And of course we build upon `scalac`'s extensible support for tree traversing, transformation, and building.

At a high level, the `jdk2ikvm` transformation is quite concise:

```
def generateOutput()
{
       for(unit <- currentRun.units)
       {

         BlockFlattener.flattenBlocks(unit)

         val noOfChanges = if(justUnparse) 0
                           else rewriteTrees(unit)

         val sourceFile = unit.source.file.file
         val relativeSourcePath = getRelativeSourcePath(sourceFile)
         val outputFile = new File(outputDirectory.get, relativeSourcePath)
         outputFile.getParentFile.mkdirs()

         if ( justUnparse || (noOfChanges > 0) ) {
           val refactoredTile = UnparseTreeFolder.xform(unit.body)
           unparse.TilePrinter.writeTo(refactoredTile, outputFile)
         } else {
       val f: java.io.File = unit.source.file.file
       FileUtil.write(new java.io.FileInputStream(f), outputFile)
     }
       }
}
```

As can be seen, all transformations are applied locally to subtrees within the current compilation unit, yet require in general knowledge about types in all compilation units processed in the compiler run (thus, we run after `refchecks`). No re-typing is performed after transformation, just pretty-printing.

1.1 How to build `jdk2ikvm`

To build `jdk2ikvm` from sources:

1. compile all Scala source files from `http://lampsvn.epfl.ch/trac/scala/browser/scala-experimental/trunk/jdk2ikvm`
2. say the resulting classfiles are found at `myplugins\jdk2ikvm\classes`
3. prepare the `jdk2ikvm.jar`

   ```
   del jdk2ikvm.jar
   jar -cf jdk2ikvm.jar -C myplugins\jdk2ikvm\classes scala -C myplugins\jdk2ikvm\resources\ .
   ```

4. where `myplugins\jdk2ikvm\resources` contains the plugin manifest `scalac-plugin.xml`

   ```
   <plugin>
          <name>jdk2ikvm</name>
   ```

```
        <classname>scala.tools.jdk2ikvm.JDK2IKVMPlugin</classname>
    </plugin>
```

5. that's it.

1.2 How to run

Regarding command-line options, the following are needed:

```
-Ystop:superaccessors /*- given that the plugin runs right after typer */
-sourcepath bla\bla\src
-P:jdk2ikvm:output-directory:bla\bla\out
-Xplugin where\to\find\jdk2ikvm.jar
```

1.3 Possibilities opened up by `jdk2ikvm`

`jdk2ikvm` opens up intriguing possibilities in connection with `Scalify`, a tool to automatically translate from Java to Scala:

- `http://wiki.jvmlangsummit.com/Scalify`
- `http://video.google.com/videoplay?docid=-3493190786110154189#`

Additionally, `jdk2ikvm` can also serve as blueprint for other *language-aware pre-processors* (as opposed to typing-oblivious pre-processors limited to macro expansions and the like). As a more encompassing example, the mythical *Scala.NET* → *C#* converter also fits in this category: `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q2/threeaddress.pdf`

2 Under the hood

At its core, `jdk2ikvm` applies a pipeline of AST transformers just like the `foldLeft` example below shows for a few transformers (`strObjTransformer`, `addMissingObjectContract` etc.):

```
def rewriteTrees(unit: CompilationUnit) = {
  if (!unit.isJava) {
    val pipeline = List( strObjTransformer, addMissingObjectContract,
                         magicIfaceTransformer, exceptionsTransformer, veryLastTransformer )
    ChangedTreePipeliner.clear(pipeline)
    val op = (t: Tree, f: Transformer) => f.transform(t)
    unit.body = pipeline.foldLeft(unit.body)(op)
    ChangedTreePipeliner.noOfChanged(pipeline)
  } else 0
}
```

where a `Transformer.transform` does *not* perform an in-place destructive update but returns instead tree only whose changed nodes are new. An excursion about this appears in Sec. 2.1.

Before diving into IKVM-specific transforms, you might want to take a look at a more gentle introduction to AST rewriting:

StringAndObject.scala

```scala
package scala.tools
package jdk2ikvm
package transforms

import scala.tools.nsc.{ast, plugins, symtab, util, Global}
import plugins.Plugin
import java.io._
import nsc.util.{Position, SourceFile}


/**
 *  @author Miguel Garcia
 */
trait StringAndObject extends Plugin
        with refactoring.transformation.TreeFactory
        with refactoring.common.PimpedTrees
        with JDK2IKVMUtil
     /* with nsc.ast.TreeDSL */ {

  import global._
  import definitions._
  // import CODE._

  private object StrObjCallsites extends Transformer {...}

  private object StrObjCoOverrides extends Transformer {...}

  object strObjTransformer extends Transformer {
    override def transform(tree: Tree) = {
      val t1 = StrObjCallsites.transform(tree)
      val t2 = StrObjCoOverrides.transform(t1)
      t2
    }
  }

}
```

Figure 1: One way to compose transformers

- "*Constant Folding Redux*", `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/ReduxReport.pdf`

Coming back to `jdk2ikvm`, each element in the pipeline is realized by its own trait to simplify development, although they all need be applied if the output sources are to abide by IKVM usage rules.

The `foldLeft` above showed one way to apply transformers in sequence, while Figure 1 shows another way (a single transformer may compose others). The snippet collapses the source code of a transformer that rewrites some callsites (`StrObjCallsites`) and another that adds co-overrides. When `rewriteTrees` invokes `strObjTransformer.transform(Tree)`, both of them are invoked.

2.1 Transforming trees without destructive updates

Although not framed in the context of `scalac` ASTs, the following (visual!) depiction conveys the idea quite effectively (Figure 2):

> *In recent discussions on the scala-internals and scala-xml mailing lists, there were calls for a mutable, DOM-like XML model, where nodes hold references to their parent element, so that all standard*

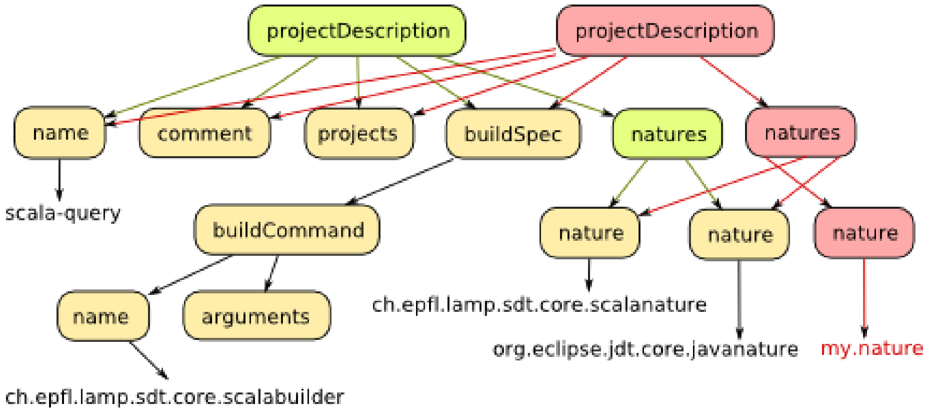


Figure 2: A Zipper for scala.xml, reproduced from `http://szeiger.de/blog/2009/12/27/a-zipper-for-scala-xml/`

> *XPath axes can be supported. In this post Id like to present a different representation which is based on the immutable and persistent scala.xml model, is completely immutable itself, yet allows navigation along all axes and "mutation". It is based on the Zipper technique which was first described by Gérard Huet*
>
> *http://szeiger.de/blog/2009/12/27/a-zipper-for-scala-xml/*

3 Big picture of the conversion recipe

A summary description of each set of transformations will be added as this document progresses, in the meantime the most complete description can be found in previous write-ups at *The Scala Compiler Corner (Reloaded)*, in particular:

- "*Mental adjustments demanded by IKVM's Object Model Mapping*" (October 2010), `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/ikvmify2.pdf`
- "*Transforming JDK-based Scala sources to use IKVM instead*" (September 2010), `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q3/ikvmcMining.pdf`

4 Transforms for the `String` and `Object` contracts

These transformations comprise:

- callsites that originally targeted instance methods are rewritten to target static helpers instead ("`instancehelpers`")
- instantiations (but not super constructor invocations) are rewritten to invoke "`newhelpers`"
- co-overrides are added for the following non-sealed methods in `j.l.Object`:

```
val t = msym.name match {
  case nme.hashCode_ => callerForSibling(dd, "GetHashCode")
  case nme.toString_ => callerForSibling(dd, "ToString" )
  case nme.equals_ => callerForSibling(dd, "Equals" )
  case nme.finalize_ => callerForSibling(dd, "Finalize" )
}
```

- type references to `j.l.{String, Object}` are rewritten to `scala.{String, Object}`.
  - in particular, in `extends` clauses, `j.l.Object` type references are rewritten to `System.Object`
  - the following occurrence of `java.lang.String` as RHS in a type alias in `Predef.scala`:

    ```
    type String      = java.lang.String
    ```

    should be rewritten as:

    ```
    type String      = System.String
    ```

TODO: `AddMissingJLObjOverrides`

TODO: remember that `super.hashCode` had to be rewritten?

5 Magic for interfaces

The transformations around interfaces are grouped into "*extra interfaces*" (Sec. 5.1), "*implied interfaces*" (Sec. 5.2), and "*upcast to extra*" (Sec. 5.3).

5.1 Extra interfaces

There's only one pair *(JDK interface, extra .NET interface)* to transform:

- `java.lang.Comparable`
- `System.IComparable`

A template that explicitly lists `j.l.Comparable` in its `extends` clause will get a `DefDef` added to its body:

```
override def CompareTo(that: Object) = this.compareTo(that)
```

and that's it (no interface needs to be added, because the extra interface is already inherited).

5.1.1 Examples: anonymous class

```
val myComparable = new Comparable[String] {
  def compareTo(that: String) = 1
}
```

```
val myComparable: java.lang.Object with java.lang.Comparable[String] = {
  new $anon
  class $anon extends System.Object with java.lang.Comparable[String] {
    override def CompareTo(that: String ) = this.compareTo(that)
    def compareTo(that: scala.String ): Int = 1
  }
}
```

5.2 Implied interfaces

A template that explicitly lists in its `extends` clause one of the interfaces:

- `j.l.Iterable`
- `j.l.Closeable`

gets another interface added to that list, resp.:

- `System.Collections.IEnumerable`
- `System.IDisposable`

as well as a `DefDef` added to its body (as shown below) delegating to the developer-provided override, as shown below.

In the case of `j.l.Iterable`, that override is called not directly but from an IKVM-provided helper:

```
override def GetEnumerator = { new ikvm.lang.IterableEnumerator(this) }
```

```
override def Dispose = { this.close }
```

5.3 Upcast to extra interface

Two transformations are performed here:

5.3.1 String comparison semantics

```
     receiver.compareTo(that)
-->
     java.lang.Comparable.__Helper.compareTo(receiver, that)
```

When given a `System.IComparable` object `x`, `x` could be in particular a `System.String` posing for a `java.lang.String`, and therefore[2]:

> *Java's `String.compareTo` is specified more strictly than `Comparable.compareTo`. The interface method simply returns zero, positive, or negative integer, but the `String` version [should] return the difference between the mismatching characters (or zero, if the strings match.) To handle this correctly, when you call `Comparable.compareTo`, you're actually*

[2] `http://weblog.ikvm.net/PermaLink.aspx?guid=db9ee4f9-3b84-40a3-ac53-acb5e27ddf19`

calling a static method `Comparable.__Helper.compareTo` that first does a check to see if the object you're comparing is a `String` and, if so, it calls a static helper method that implements the specified Java comparison algorithm.

5.3.2 Rewrite standalone type refs

```
      java.lang.Comparable
-->
      System.IComparable
```

For interoperability, `System.IComparable` (i.e., the extra interface) should be favored (in explicit type references) over `java.lang.Comparable` (i.e., the JDK interface that carries the extra baggage). That way, existing .NET binaries unaware about IKVM can interoperate, ignoring the more specific interface.

5.4 Ghost interfaces

Summary of IKVM's *nested helper types*:

- IKVM types with a `__Helper` class: `java.lang.Comparable`
- IKVM types with a `__Interface` interface: `java.io.Serializable`, `java.lang.Cloneable`, `java.lang.CharSequence`

A *ghost interface* is a JDK interface for which no same-name *interface* but a *struct* exists in IKVM. The term refers to: `j.l.CharSequence`, `j.l.Cloneable`, and `java.io.Serializable`.

The transformations for `java.io.Serializable` are different from those for the other ghost interfaces, thus this section focuses only on `j.l.CharSequence` and `j.l.Cloneable`.

5.4.1 Standalone type references to `Cloneable` and `CharSequence`

In general, standalone type references to `Cloneable` and `CharSequence` remain as-is (and thus denote a struct type after translation) but in the following contexts a certain rewriting applies:

- implementing a ghost interface, rewrite as follows:
  - `extends Cloneable`
    → `extends java.lang.Cloneable.__Interface`
  - `extends CharSequence`
    → `extends java.lang.CharSequence.__Interface`
    In this latter case, remember to implement `toString()` if not already overridden. Not necessary on JDK (as it is inherited from `j.l.Object`) but after translation `System.Object` will not provide an implementation. IKVM includes `toString()` among the methods to implement in `java.lang.CharSequence.__Interface`
- instantiating an anonymous class, rewrite as follows:

    - new Cloneable { ...}
      → new java.lang.Cloneable.__Interface { ...}
    - new CharSequence { ...}
      → new java.lang.CharSequence.__Interface { ...}

- TODO Adding `toString` is done by the transforms for the `String` and `Object` contracts. Anyway, what should be done is: (a) YES, there's an override. In this case, add `ToString` that `callSibling`; (b) NO, there's no override: add just `toString()` whose body invokes the `instancehelper_toString(this)`

The transformations above are achieved by:

```
case tpl @ Template(parents, self, body) if parents.exists(p => p.symbol == jlCharSequenceClass) =>
  val newParents = transformTrees(parents)
  val newSelf = transformValDef(self)
  val newStats = transformStats(tpl.body, tpl.symbol)
  val newParents2 = newParents map {
    case p if (p.symbol == jlCharSequenceClass) => jlCharSequenceSelector
    case p => p
  }
  treeCopy.Template(tpl, newParents2, newSelf, newStats)
```

5.4.2 Static accesses

Ghost types correspond to JDK interfaces with no static fields (FYI: the IKVM version of `j.l.CharSequence`, a valuetype, does have static methods for example the `==` operator overload mentioned below).

5.4.3 Instance method invocations

The JDK versions of `Serializable` and `Cloneable` define no methods of their own, so the issue of how to map calls to them does not arise. On the other hand, `CharSequence` has instance methods of its own, but textual occurrences of invocations can remain as-is, due to the views below (with them, before-translation invocations on `CharSequence` instance methods will find at runtime a conformant receiver).

```
// Scala.NET code to be compiled against IKVM

implicit def refToStructCharSequence
(i: java.lang.CharSequence.__Interface): java.lang.CharSequence = {
    val c : java.lang.CharSequence = new java.lang.CharSequence() // default init
    c.__<ref> = i
    c
  }

implicit def sstringToStructCharSequence
(s: System.String): java.lang.CharSequence = {
    val c : java.lang.CharSequence = new java.lang.CharSequence() // default init
    c.__<ref> = s
    c
  }
```

```
// Scala.NET code to be compiled against IKVM
```

```
implicit def refToStructCloneable
(i: java.lang.Cloneable.__Interface): java.lang.Cloneable = {
   val c : java.lang.Cloneable = new java.lang.Cloneable() // default init
   c.__<ref> = i
   c
 }
```

5.4.4 `==` and `!=`

Please notice that `==` and `!=` between two `CharSequence`s should bind to the following methods. This is taken care of by `TypeParser`.

```
// C# code showing part of IKVM's java.lang.CharSequence API

public static bool operator ==(CharSequence sequence1, CharSequence sequence2)
{ return (sequence1.__<ref> == sequence2.__<ref>); }

public static bool operator !=(CharSequence sequence1, CharSequence sequence2)
{ return (sequence1.__<ref> != sequence2.__<ref>); }
```

5.4.5 Type casts and checks

We're talking about: `isInstanceOf[C]`, `asInstanceOf[C]`, `classOf[C]`

The cases to consider when translating "`arg.isInstanceOf[C]`" and "`arg.asInstanceOf[C]`" for ghosts are:

1. `C` is ghost:
   `arg.isInstanceOf[java.lang.CharSequence]`
   → `java.lang.CharSequence.IsInstanceOf(arg)`

   `arg.isInstanceOf[java.lang.Cloneable]`
   → `java.lang.Cloneable.IsInstanceOf(arg)`

   similarly for `asInstanceOf`.

2. `C` is array of ghost. Call the static `IsInstanceArray(arg, rank)` on the ghost's type.

   TODO: what about `asInstanceOf[`*`array of ghost`*`]`

TODO In contrast, `classOf[]` should be left as-is (??)

The transformations above are achieved by:

```
// rewrites isInstanceOf[CharSequence] and isInstanceOf[Cloneable]
case app @ TypeApply(fun, args) if (
   {
     val guard0 = (fun.symbol.overriddenSymbol(AnyClass) == Any_isInstanceOf)
     if (guard0 && (args.head.tpe != null)) {
       val typeArg = args.head.tpe.typeSymbol
       ghostClasses contains typeArg
     } else false
   }
 ) =>
 val typeArg = args.head.tpe.typeSymbol
```

```
if (typeArg == jlCharSequenceClass)
  instanceOfGhost(jlCharSequenceSelector, app)
else
  instanceOfGhost(jlCloneableSelector, app)
```

6 Exceptions

Before looking at the rewriting rules, let's review the context where those rules apply:

- All of `j.l.Throwable`, `j.l.Exception`, and `j.l.Error` "get mapped to" `System.Exception` (because there's a correspondence between `j.l.Throwable` and `S.Exception` methods, and because neither `j.l.Exception` nor `j.l.Error` add methods of their own).
- IKVM's `j.l.Throwable` is derived from `System.Exception` and thus does not conform to `j.l.Object`.

The detailed recipe appears in Secs. 6.1 to 6.3, the high-level view is:

- After translation, each catch clause declares an argument of (a subclass of) `System.Exception`. Depending on the before-translation type, the rewritten type will be:
  - `System.Exception` for `j.l.Throwable`, `j.l.Exception`, and `j.l.Error`, i.e., for Cases (1) and (2) below.
  - the same-name IKVM counterpart for all others, Case (3).
- In Case (1), `catch Throwable`, the "original catch block" may contain invocations to `Throwable` methods to be called through a `System.Exception` reference. `ikvmc` detours those invocations to `instancehelpers` in `Throwable` that check if the passed object subclasses `Throwable` and:
  - if so, (a) `callvirt` the method in question,
  - if not, calls either (b.1) the closest `System.Exception` equivalent; or (b.2) a static helper in IKVM's `java.lang.Throwable`.
- In Case (2), `catch Exception` or `catch Error`, a utility call is pre-pended to the output catch-body, to try to wrap the exception so that the wrapper conforms to the originally declared one (or rethrow it otherwise).
- In Case (3), the exception to catch is a proper subtype of `Exception` or `Error`. The rewriting is simpler because there's an IKVM counterpart for that type.

6.1 Case (1) Originally `catch Throwable`

The output catch clause looks as follows:

```
case exceptionArg : System.Exception =>
  val exception = java.lang.Throwable.__<map>(exceptionArg, true).asInstanceOf[System.Exception]
  ... original catch block ...
  /*- in this block there are before-translation invocations
        on j.l.Throwable methods that after-translation will go through
        a System.Exception receiver. */
```

6.2 Case (2) Originally `catch Exception` or `catch Error`

The output catch clause looks as follows:

```
case exceptionArg : <OriginalType> =>
  val exception = java.lang.Throwable.__<map>(exceptionArg,
                                              typeof(<ExceptionOrError>),
                                              true).asInstanceOf[<OriginalType>]
  if (exception eq null) { throw exceptionArg }
  ... original catch block ...
```

6.3 Case (3) Otherwise

"Otherwise" means of course: originally a subclass of `Exception` or `Error` is caught.

```
case exception : <OriginalType> => // Bind remains unmodified
  java.lang.Throwable.__<map>(exception, true)
  ... original catch block ...
```

7 Etc

- diff the IKVM-ready source files with the original sources.

```
mapMethod(JOBJECT, "getClass"    , jEmpty  , MIKVMJLObject , "instancehelper_getClass", mObject1)
mapMethod(JOBJECT, nme.hashCode_ , jEmpty  , MIKVMJLObject , "instancehelper_hashCode", mObject1)
mapMethod(JOBJECT, nme.equals_   , jObject1, MIKVMJLObject , "instancehelper_equals"  , mObject2)
mapMethod(JOBJECT, nme.clone_              , MOBJECT       , "MemberwiseClone"                  )
mapMethod(JOBJECT, nme.toString_ , jEmpty  , MIKVMJLObject , "instancehelper_toString", mObject1)
mapMethod(JOBJECT, nme.notify_   , jEmpty  , MMONITOR      , "Pulse"                  , mObject1)
mapMethod(JOBJECT, nme.notifyAll_, jEmpty  , MMONITOR      , "PulseAll"               , mObject1)
mapMethod(JOBJECT, nme.wait_     , jEmpty  , MIKVMJLObject , "instancehelper_wait"    , mObject1)
mapMethod(JOBJECT, nme.wait_     , jLong1  , MIKVMJLObject , "instancehelper_wait"    , Array(MOBJECT, MLONG))
// mapMethod(JOBJECT, nme.wait_     , Array(JLONG.tpe, JINT.tpe), MIKVMJLObject , "instancehelper_wait"    , Ar
mapMethod(JOBJECT, nme.finalize_           , MOBJECT       , "Finalize")
```

8 Appendix

8.1 Using another pretty-printer

Pretty-printing as implemented in `jdk2ikvm` does not keep the original layout (we use the term *unparsing* to make clear this behavior). In oder to preserve layout another pretty-printer should be integrated, as provided by Mirko Stocker's refactoring library, `http://scala-refactoring.org/`. In this case the resulting plugin will depend on `refactoring.jar`, a fact that has to be taken care of when running `scalac`: `-classpath` indicates where to look for classfiles needed by `scalac` to compile the input files, not the classfiles needed by some plugin. Solution alternatives are discussed at `http://www.scala-lang.org/node/6664`. To those alternative, I'll add yet another:

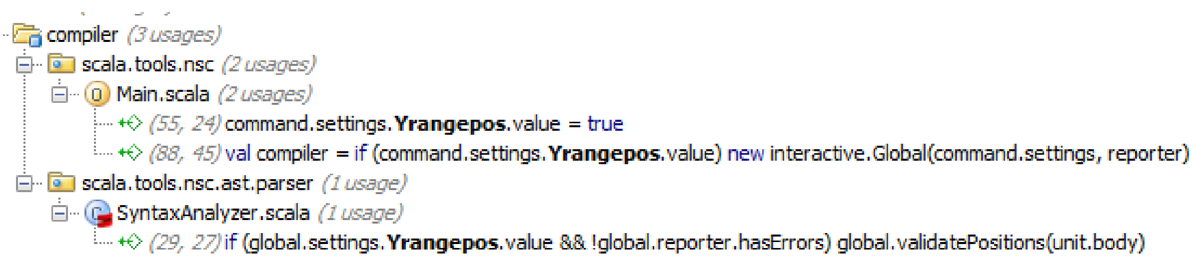
```
-Xbootclasspath/a:...;...;Z:\scalaproj\sn4\myplugins\scala-refactoring-lib.jar
```

Another straightforward way to make sure the plugin can access the refactoring classfiles is to make them part of the plugin itself.

> TODO: `-Yrangepos` also validates trees, which fails on `scala.*` definitions (as in the bootstrapping case ...). Details at `http://www.scala-lang.org/node/2755`

The command-line option `-Yrangepos` makes sure that `scalac` instantiates an `nsc.interactive.Global`:

```
val compiler = if (command.settings.Yrangepos.value) new interactive.Global(command.settings, reporter)
else new Global(command.settings, reporter)
```



The refactoring library sometimes requires the following functionality from `nsc.interactive.Global`:

```
def compilationUnitOfFile(f: AbstractFile): Option[global.CompilationUnit]
```

`scalac` uses the interactive compiler when invoked with `-Yrangepos`, otherwise an `nsc.Global` will be handed to compiler plugins such as `jdk2ikvm`. The `scala-refactoring` library supports both usage scenarios, by encapsulating in trait `CompilerAccess` the above dependency:

```
trait CompilerAccess {

  val global: tools.nsc.Global

  def compilationUnitOfFile(f: AbstractFile): Option[global.CompilationUnit]
}
```

8.2 Lessons learnt from the previous prototype

Our previous prototype (in the form of a patched compiler) applied the *JDK to IKVM* conversion *on the fly*, emitting .NET assemblies as output. That division of labor turned out to be inflexible because:

1. It forced the developer to prepare programs with JDK-like syntax when targeting .NET, moreover forcing to keep a mental model of the mappings that the patched compiler performed.

   - For example, whenever a catch block expected a `java.lang.NullPointerException`, it could happen that a `System.NullReferenceException` was thrown by external code being invoked. Question: do you know, *from looking at your source files*, if the patched compiler emitted code to also catch `NullReferenceException`?

2. As a .NET compiler, it required programs always to be linked against IKVM's library (which is not the usual case unless migrating platforms). The IKVM library in turn was necessary when implementing all of the JDK-to-IKVM conversion recipe, without which bootstrapping can't happen.

3. Running in `forMSIL` mode, it accepted programs that would have been rejected in both (a) `forJVM` mode and (b) "`forMSIL` minus patches" mode (precisely the "mode" in which .NET compilers run).

   - For example, any program that mixes API calls from JDK and .NET would be valid as per the patched compiler (which runs in `forMSIL` mode),
   - in particular programs calling the overrides added automatically to support *extra interfaces* (remember, the patched compiler implements the full JDK to IKVM conversion recipe). After offloading to `jdk2ikvm` the conversion task, the Scala.NET compiler does not add any overrides on its own, and program sources tell again all there is to know about the emitted program.

4. Intermingling the functionality of the JVM, .NET, and IKVM backends *all in one codebase* increased complexity big time.

5. Last but not least, the `jdk2ikvm` tool is useful on its own.