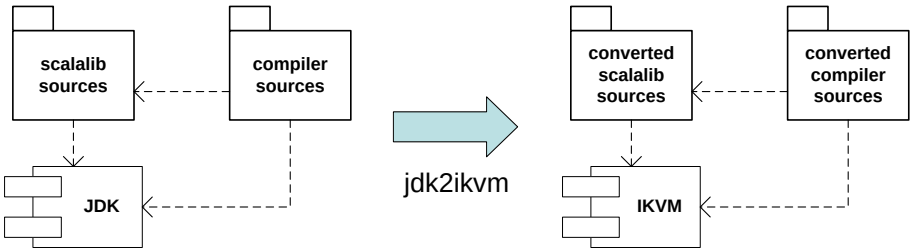
Bootstrapping Scala.NET via `jdk2ikvm`

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

April 24nd, 2011

Abstract

Bootstrapping the compiler on .NET requires for `scalacompiler.exe` to compile its own sources. For that, `scalalib.dll` should also be available. Both assemblies above are obtained from sources which result from an automated API migration, performed by `jdk2ikvm`. This tool takes as input a tree of Scala source files with JDK depedencies, and emits a mirror of that source tree, preserving the original layout except for those places where a JDK dependency has been replaced with its IKVM counterpart. Bootstrapping has been automated, and these notes describe how that process works under the hood. If you want to use Scala.NET then none of these details are necessary: simply downloading the (already bootstrapped) distribution.




```
import System.Collections.Generic.IList

object CountAll {
  def doCount[T](sample: T, is: IList[T]): Int = {
    val enu = is.GetEnumerator
    var count = 0
    while(enu.MoveNext) {
      if(enu.C == sample) count += 1;
    }
    count
  }
}
```

```
Current
Choosing item with Tab will overwrite the rest of identifier after caret
```

Contents

1 Before delving into details: Summary of the whole process

Once `scalacompiler.exe` and `scalalib.dll` are available, using them to compile their own sources is also called "bootstrapping". But that's not the "bootstrapping" described in these notes. The "bootstrapping" described here starts with the compiler sources *for its JVM version*, obtaining as intermediate result the first `scalacompiler.exe` (which is then used to compile its own sources). Obtaining `scalacompiler.exe` "for the first time" involves using the cross-compiler.

Scala.NET bootstrapping is easier than it seems. Specially after breaking it up into four steps:

1. **API migration**. We want to boostrap the compiler, thus we need to have Scala.NET sources to compile. For that, we need to:
   (a) build and run `jdk2ikvm` (Sec. 2). This discussion will also be useful in case you want to port your own projects to Scala.NET using `jdkikvm`.
   (b) apply manually those few transformations that either can't be automated (e.g., adding .NET-specific implicits to `Predef`) as well as those that aren't yet supported by `jdk2ikvm` (and for which there's a straightforward workaround). All this in Sec. 3.
2. **Preparing pre-requisite libraries**. This is a cumbersome step, as shown by the section length. We use the following tools: `ikvmc`, `ilasm`, and `csc`. The good news is that Sec. 4 attempts to convey all the relevant details on using these tools for the task at hand.
3. **Cross-compiling**. With the previous items in place, we get into one-click territory. In this step the compiler is run on JVM (cross-compilation mode) to produce .NET assemblies, as discussed in Sec. 5.
4. **Bootstrapping proper**. This step takes as input the same source files as above, except that the .NET version of the compiler and library are run (i.e., no more cross-compilation). A behavioral-conformance test is also performed. Details in Sec. 6.

Steps 1 and 2 are independent of each other. The rest is sequential: step 3 depends on both 1 and 2; while steps 3 and 4 must be carried out in that order.

- Secs. 3 to 6 describe bootstrapping in detail with Windows commands.
- Sec. 7 sums up the process for Mono, giving command-prompt instructions (i.e., telling the how's but not the why's already included above)
- Sec. 8 discusses ideas for future work.

A note on the patches in Step 1.(b) above. They are unavoidable, even if `jdk2ikvm` handled the few corner cases it overlooks now. Why they are unavoidable can be seen when compiling `scalalib.dll`. For that to work, the library sources must include references to IKVM types. For the cross-compilation step to work (Step 3), you'd better run on JVM with a version of the Scala library that omits any IKVM types (because you're running on JVM ...). See? The patches are what keeps both the cross-compiler and the "native compiler" happy.

2 Getting the source-to-source converter ready

2.1 How to build `jdk2ikvm`

Starting from scratch as we are, we have to download and build `jdk2ikvm`:

1. compile all Scala source files from:

```
svn co
  http://lampsvn.epfl.ch/svn-repos/scala/scala-experimental/trunk/jdk2ikvm/src/scala/tools/jdk2ikvm
  jdk2ikvm
```

2. say the resulting classfiles are found in folder `myplugins\jdk2ikvm\classes`

3. prepare `jdk2ikvm.jar`

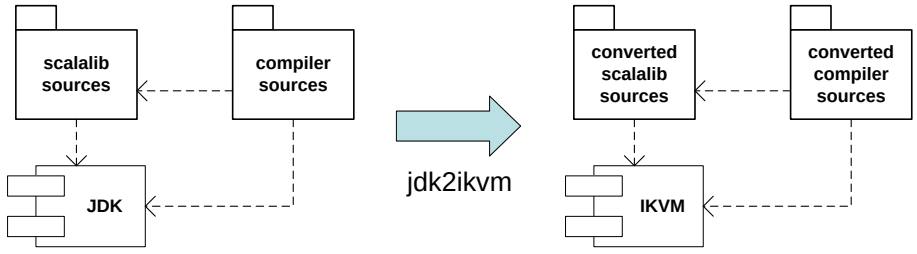
```
del jdk2ikvm.jar
jar -cf jdk2ikvm.jar -C myplugins\jdk2ikvm\classes scala -C myplugins\jdk2ikvm\resources\ .
```

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

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

2.2 Running `jdk2ikvm` on Scala `trunk`

Let's see if a picture is really worth a thousand words:



It all starts with the sources from trunk, where "`jscala`" below is used as a mnemonic for "JDK-based Scala sources"

```
svn co http://lampsvn.epfl.ch/svn-repos/scala/scala/trunk@24806 jscala
```

> All right, here's where you'll miss the separate branch for Scala.NET. We'll manually apply a few changes to overcome `jdk2ikvm` limitations that that branch has already merged. But if you're reading this, you're into compiler internals, right? More details in Sec. 3.2.

Please download the patches from `http://lamp.epfl.ch/~magarcia/ScalaNET/2011Q2/patches1.zip` and apply them with:

```
patch -p0 -i ForAssemblyVisibility.patch

patch -p0 -i ForCompileByEXE.patch

patch -p0 -i TricksThatJ2KCannot.patch

ant build
```

As with most compiler plugins, `jdk2ikvm` requires a few command-line options when run:

```
-Ystop:superaccessors /*- given that the plugin runs right after typer */
-sourcepath bla\bla\src
-P:jdk2ikvm:output-directory:bla\bla\out
-d bla\bla\out /*- yes, again the same output folder as given above */
-Xplugin where\to\find\jdk2ikvm.jar
-Yrangepos
```

The option `-Yrangepos` has an interesting effect on the Abstract Syntax Trees (ASTs) that the compiler builds: nodes record the text interval (i.e., start-end positions in the input files) for program elements. This information is used by other tools to implement refactorings, and in general allows implementing source-level transformations, as exemplified by `jdk2ikvm`.

```
    scala.tools.nsc.interactive.RangePositions performs in validatePositions() a number of checks,
    some of which fail but whose failure is nonetheless not important for jdk2ikvm.

    One of my patches to RangePositions results in errors being more readable.
    Whether those improved error messages are shown or not,
    the important thing is that no ValidateException is thrown.
    The following lines should therefore be commented out, as shown below:

/* if(validationErrors > 0)
    throw new ValidateException("There were "+validationErrors+" tree-positions validation errors.") */
```

For the purposes of bootstrapping, the JDK-to-IKVM conversion has to be applied to three folders in the `src` folder of the Scala trunk: `msil`, `library`, and `compiler`.

3 Getting sources ready for the cross-compiler

After `jdk2ikvm` has run on `jscala` (and before feeding those sources to the cross-compiler) a few patches are needed. These patches fall in two categories:

- additions mandated by .NET. These are obviously *nowhere* to be found in the JDK-based sources. Sec. 3.1.
- patches to circumvent `jdk2ikvm` shortcomings. All right, a few corner cases are not (yet) automated by `jdk2ikvm`. In the meantime, Sec. 3.2 describes what to do.

It all boils down to having 50 files manually changed before they reach the cross-compiler, out of 1065 files automatically converted by `jdk2ikvm`. The following zip archive contains the patched sources:
`http://lamp.epfl.ch/~magarcia/ScalaNET/2011Q2/patches2.zip`

3.1 Changes and additions mandated by .NET and by IKVM

The ".NET-mandated" changes are only natural. For example, in `Predef` the RHS for the `String` type alias is now `System.String`. As another example, `ScalaObject` now extends `System.Object`.

Other changes are not so much .NET-specific but rather due to our start situation (JDK-based Scala sources). For example, the following implicits are added in `Predef` to account for the preferred reflection abstractions of JVM and .NET:

```
implicit def class2type(c : java.lang.Class) : System.Type = ikvm.runtime.Util.getInstanceTypeFromClass(c)

implicit def type2class(t : System.Type) : java.lang.Class = ikvm.runtime.Util.getFriendlyClassFromType(t)
```

The mapping between Java and .NET exception classes also requires runtime object wrapping:

```
  implicit def systemExceptionToJLException(cause: System.Exception) = new java.lang.Exception(cause)

  implicit def systemExceptionToJLError(cause: System.Exception) = new java.lang.Error(cause)
```

Arrays in .NET are instances of `System.Array`:

```
  implicit def arrayToSystemArray(xs: Array[_]): System.Array = {
    xs.asInstanceOf[System.Array]
  }
```

Let's not forget some additions to `ScalaRunTime`:

```
  def _ToString(x: Product) = _toString(x)

  def _GetHashCode(x: Product) = _hashCode(x)

  def _Equals(x: Product, y: Any) = _equals(x, y)
```

The remaining additions to `Predef` are motivated in implementation choices of IKVM. Most notably, representing `java.lang.CharSequence` not as an interface but as a valuetype, as shown in Listing 1 on p. 23.

All this is mentioned because, being now in `scalalib.dll`, the above definitions could also be in scope when writing your own Scala.NET programs. And yes, given that `scalalib.dll` depends on the IKVM library, your Scala.NET programs will also depend on it. It's all a matter of transitivity, you see.

3.2 Tricks that `jdk2ikvm` can't (so far)

`jdk2ikvm` is only 2KLOC heavy, and thus there are a few (special) cases where it leaves the input as-is, and manual rewriting is needed afterwards (as signalled by Scala.NET when attempting to compile those files).

In some occassions, the consequences of IKVM's erased types exceed what `jdk2ikvm` automates (red background means manually modified):

```
48  <!    if (cache containsKey key) cache.get(key).asInstanceOf[List[Fileish]] else Nil
[48] !>   if (cache containsKey key) cache.get(key) else Nil
49
50      private def add(key: String, value: Fileish) = {
51  <!    if (cache containsKey key) cache.replace(key, value :: cache.get(key).asInstanceOf[List[Fileish]])
[51] !>   if (cache containsKey key) cache.replace(key, value :: cache.get(key))
52        else cache.put(key, List(value))
53      }
54      override def toString = "Sources(%d dirs, %d jars, %d sources)".format(
55  <!    dirs.size, jars.size, cache.asScala.asInstanceOf[Map[String, List[Fileish]]].values map (_.length) sum
[55] !>   dirs.size, jars.size, cache.asScala.values map (_.length) sum
```

(The above shows that Linux has `meld`[1], and Windows `windiff`.)

Othertimes, it's IKVM's decision to give an array type to what a JDK signature shows as a repeated param (most of the time, the required rewriting is performed, but not in the case shown below):

```
if (completion ne NoCompletion) {
  val argCompletor: ArgumentCompleter =
    new ArgumentCompleter(new JLineDelimiter, scala.Array ( scalaToJline(completion.completer() ) ))
    new ArgumentCompleter(new JLineDelimiter, scalaToJline(completion.completer()))
  argCompletor setStrict false

  this addCompleter argCompletor
  this setAutoprintThreshold 400 // max completion candidates without warning
}
```

4 Getting non-Scala libraries ready

As will be seen in Sec. 5, a number of libraries are required to carry out the next step in bootstrapping (cross-compiling). How to build those libraries is the topic of this section.

4.1 Non-Scala sources and libraries in the distribution

4.1.1 Java-only libraries that can be taken as-is

For the Java sources described below, their `.jar` files are converted to `.dll`s using `ikvmc`, the bytecode-level compiler of IKVM.

- `forkjoin`: Contains `scala.concurrent.forkjoin`. These Java classes[2] are required by `scala.collection.parallel` and `scala.actors`.
- `msil`: The Java files under `ch.epfl.lamp.compiler.msil`[3]. Required by `GenMSIL` and `TypeParser`.

Conversion is done simply:

```
ikvmc -target:library msil.jar     -out:msil.dll
ikvmc -target:library forkjoin.jar -out:forkjoin.dll
```

[1] `http://meld.sourceforge.net/`

[2] `http://lampsvn.epfl.ch/trac/scala/browser/scala/trunk/src/forkjoin`

[3] `http://lampsvn.epfl.ch/trac/scala/browser/scala/trunk/src/msil/ch/epfl/lamp/compiler/msil`

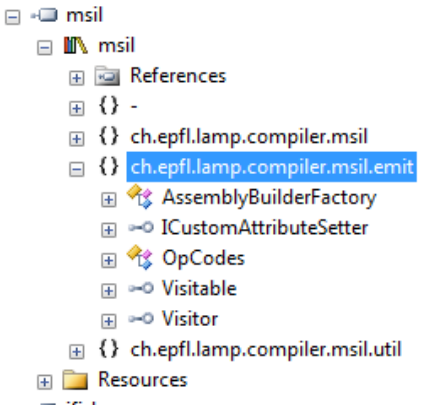


Figure 1: Contents of `ch.epfl.lamp.compiler.msil.emit` in `msil.dll`, Sec. 4.1.1

FYI: In the resulting `msil.dll`, package `ch.epfl.lamp.compiler.msil.emit` (shown in Figure 1) contains "less" classes than its counterpart in `msil.jar` (shown in Figure 2). That's OK for our purposes. The "missing" classes will be found in `scalacompiler.exe`: given that they are `.scala` classes, we'll compile them together with the other `.scala` classes that make up the compiler.

4.1.2 `jfisl` and assembly roundtripping

As in the previous sub-section, we use `ikvmc` to obtain this `.dll`. The only difference is that some methods marked `protected` in their Java version have to be made public in their assembly version. The contents of this `.dll` are shown in Figure 3 on p. 10.

In the resulting `.dll` two files are found that have no Java counterpart: `scala.reflect.ScalaSignatureAttribute` and `scala.reflect.ScalaLongSignatureAttribute`. They are added by `ikvmc`, and we leave them there.

- Regarding `scala.math.ScalaNumber`, unlike its Java counterpart, its methods should be marked public. If only the classfile is available, after compiling with `ikvmc` those methods can still be made public, via "*assembly roundtripping*" (details below). First disassemble with `ildasm`, patch the resulting `.msil` file, and re-assemble.

- `scala.runtime.ArrayRuntime`: Make both the class and all its methods public, so that they are visible from other assemblies (as for example from `scalalib.dll`).

Assembly roundtripping is described in:

- Ch. 19 of the *Expert .NET 2.0 IL Assembler* book [1], in Secs. "Principles of Round-Tripping" and "Creative Round-Tripping"

- `http://blogs.msdn.com/b/jmstall/archive/2006/01/13/debug-roundtripping.aspx`

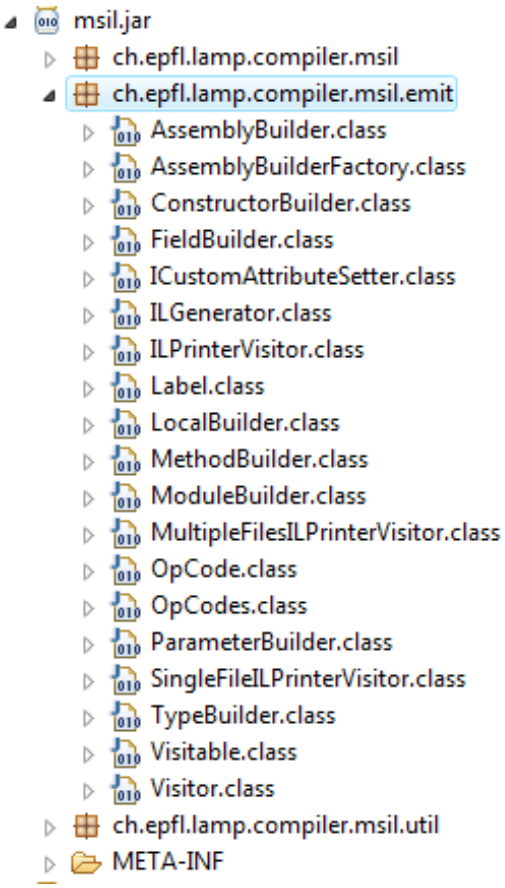


Figure 2: Contents of `ch.epfl.lamp.compiler.msil.emit` in `msil.jar`, Sec. 4.1.1

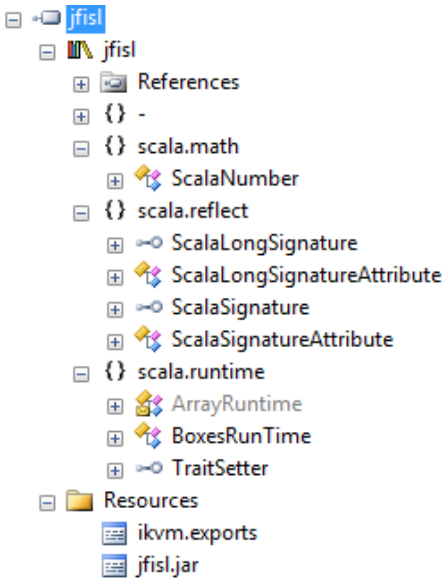


Figure 3: Sec. 4.1.2

4.1.3 `scalaruntime.dll`

The contents of this `.dll` are shown in Figure 4 on p. 11. The `.dll` is obtained using `ikvmc` as in Sec. 4.1.1. We discuss them separately because there's no ready-made `.jar` as in that section. Instead, on JVM, these files are distributed in the `.jar` for the Scala library.

4.1.4 A touch of C#: `CSharpFilesForBootstrap.dll`

Two C# files are needed for bootstrapping (afterwards, Scala versions replace them). They can be compiled with:

```
csc /target:library /out:CSharpFilesForBootstrap.dll Comparator.cs SymtabAttribute.cs
```

One minor caveat, though. Before compiling, please rename the only method in `Comparator` to `Equals`:

```
namespace scala.runtime {

  using System;

  public class Comparator {
    public static bool Equals(object a, object b) { /*- i.e. uppercase first letter. */
      . . .
```

Their SVN location, and an indication why the compiler needs them (i.e, where they are loaded into the symbol table):

1. `scala.runtime.Comparator`[4]. Needed In trait `MSILPlatform`:

[4] `http://lampsvn.epfl.ch/trac/scala/browser/scala-msil/trunk/src/dotnet-library/scala/runtime/Comparator.cs`

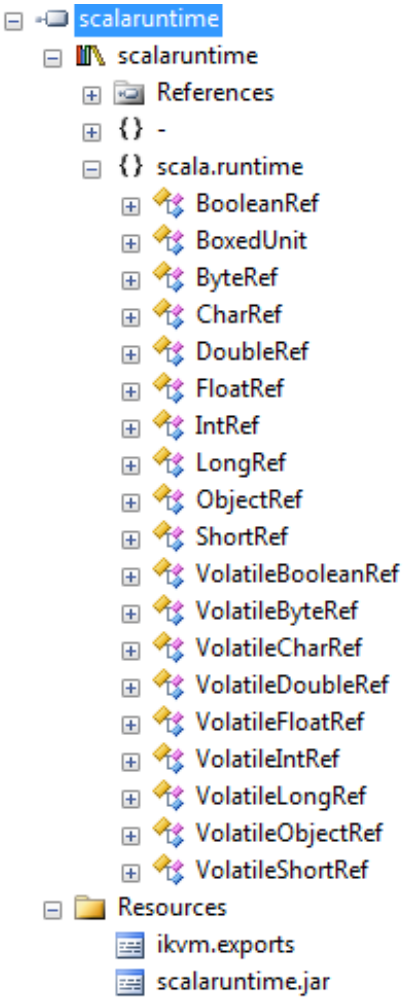


Figure 4: Sec. 4.1.3

```
lazy val externalEquals = getMember(ComparatorClass.companionModule, nme.equals_)
```

where `ComparatorClass` is defined in `Definitions.scala`:

```
lazy val ComparatorClass = getClass("scala.runtime.Comparator")
```

under section ".NET backend" of that file.

2. `scala.runtime.SymtabAttribute`[5] Needed in `CLRTypes.scala`:

```
SCALA_SYMTAB_ATTR = getTypeSafe("scala.runtime.SymtabAttribute")
val bytearray: Array[Type] = Array(Type.GetType("System.Byte[]"))
SYMTAB_CONSTR = SCALA_SYMTAB_ATTR.GetConstructor(bytearray)
SYMTAB_DEFAULT_CONSTR = SCALA_SYMTAB_ATTR.GetConstructor(Type.EmptyTypes)
```

4.2 Other libraries in the distribution

In order to bootstrap Scala on JVM, a few extra `.jar`s from other open-source projects are needed. For bootstrapping on CLR, those libraries are converted to `.dll`s using `ikvmc`. How will the resulting `.dll`s be used (or not) by Scala.NET at runtime? Let's see:

- `ant`: Required by the Ant tasks defined in `scala.tools.ant`
- `fjbg`: Required by `GenJVM`. It's unlikely that `scalacompiler.exe` will be used in `forJVM` mode, but given that we are compiling all `trunk/src/compiler` sources we also need `fjbg.dll` (at least, at compile time).
- `jline`: Required by the REPL, `scala.tools.nsc.interpreter`. The sources of `jline` are hosted in Scala trunk[6].

4.3 IKVM and .NET pre-requisites

The sources emitted by `jdk2ikvm` contain in general dependencies on IKVM and .NET `.dll`s. For the Scala library and compiler, those dependencies are as follows:

1. The folder shown in Figure 5 contains those IKVM `.dll`s that are referred from the sources output by `jdk2ikvm`. Feel free to add more IKVM `.dll`s. Not surprisingly, those files are named starting with '`IKVM`'.
2. As to .NET, the dependencies are (I'm using `ver 2:0:0:0`):
   - `mscorlib.dll`.
   - `System.dll` and `System.configuration.dll`.
   - `System.Xml.dll`. Needed after converting the `scala.xml` package.

[5] `http://lampsvn.epfl.ch/trac/scala/browser/scala-msil/trunk/src/dotnet-library/scala/runtime/SymtabAttribute.cs`

[6] `http://lampsvn.epfl.ch/trac/scala/browser/scala/trunk/src/jline`

5 Cross-compiling

We're half-way through the bootstrapping process. By the end of this step, we'll have assemblies for the Scala library and compiler. These assemblies will be used in Sec. 6 to compile their own sources.

5.1 Library

At this point we have:

- Scala sources that contain only dependencies on the .NET Framework and IKVM libraries (Sec. 3).
- `.dll`s for the Java source files in the distribution (Sec. 4.1), libraries from other open-source projects (Sec. 4.2), and some libraries from IKVM and .NET (Sec. 4.3).

With that, the cross-compiler can process all source files that constitute the Scala library.

Assuming the required libraries (Figure 5) are found in folder `DirA`, the following allows obtaining `scalalib.msil`, which can be assembled afterwards with ILAsm. We invoke `scalac` in `forJVM` mode (the default).

```
scalac
-sourcepath %OUT_TOP%\src\library /*- one of the three output folders of jdk2ikvm */
-d %MSIL_OUT% /*- folder where the textual file scalalib.msil will be written */
@%OUT_TOP%\out-src-library.txt /*- a listing of all .scala files converted by jdk2ivkm from \src\library */
-target:msil -Ystruct-dispatch:no-cache -no-specialization
-Xassem-name scalalib -Xassem-extdirs DirA
-Ydebug /*- to better appreciate compilation progress */
```

Upon invoking the ILAsm assembler, we ask also for debug symbols to be generated in `scalalib.pdb`. To use a visual debugger, the `.pdb` file is placed alongside the assembly to debug (Sec. 8.1).

```
ilasm /DLL /DEBUG scalalib.msil
```

5.2 Compiler

In addition to the libraries shown in Figure 5, the compiler sources contain references to the Scala library. That's why this time the cross-compilation run will use `-Xassem-extdirs DirB`, where "`DirB`" contains the same libraries as `DirA` plus `scalalib.dll` (the output from Sec. 5.1).

The command to obtain `scalacompiler.msil` is very similar to that in the previous section:

```
scalac
-d %MSIL_OUT%
@%OUT_TOP%\out-src-compiler.txt /*- a listing of all .scala files converted by jdk2ivkm from \src\compiler */
@%OUT_TOP%\out-src-msil.txt /*- a listing of all .scala files converted by jdk2ivkm from \src\msil */
-target:msil -Ystruct-dispatch:no-cache -no-specialization
-Xassem-name scalacompiler
-Xassem-extdirs DirB
-Ydebug
```

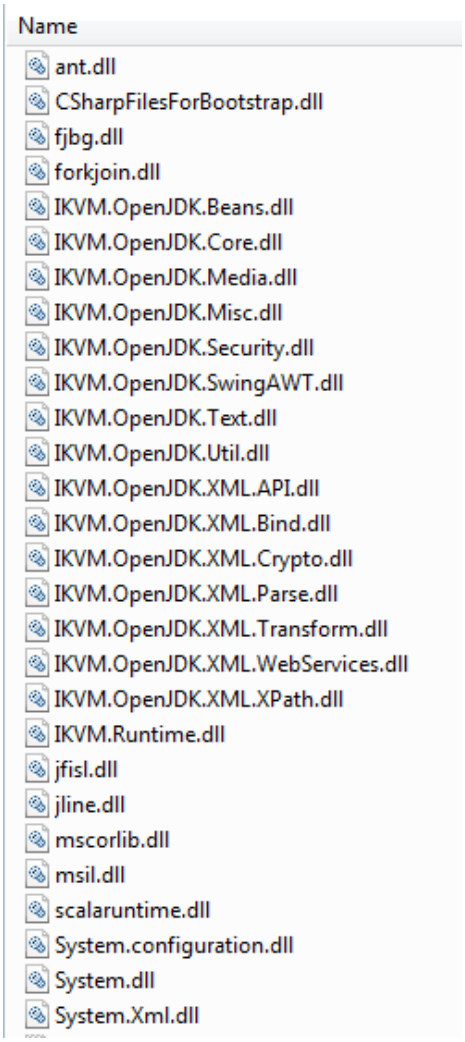


Figure 5: Non-Scala libraries ("`DirA`"), Sec. 5.1

```
-Xshow-class scala.tools.nsc.Main /*- for now, there's no dedicated option to indicate the 'entry class'
                                    (i.e., the class with the entry point: def main(args: Array[String]) )
                                    so we reuse for this purpose one of the many options that scalac offers. */
```

The ILAsm invocation to obtain the executable and its debug symbols:

```
ilasm /EXE /DEBUG scalacompiler.msil
```

```
TODO

scala.tools.nsc.ast.TreeBrowsers relies on Swing and AWT,
which IKVM supports (partially) via IKVM.OpenJDK.SwingAWT.dll.

From time to time, TreeBrowsers.scala is extended with more bells and whistles,
which may outstrip the Swing support of the IKVM version in use.
In this case, as a workaround, make TreeBrowser do nothing:
  - comment out the lines "import java.awt. . . . "
  - delete the method bodies of all browse methods in ThreeBrowsers#SwingBrowser
  - delete class ThreeBrowsers#BrowserFrame
  - delete class ThreeBrowsers#TextInfoPanel
```

6 Compiling with `scalacompiler.exe`

The last step of bootstrapping involves using `scalacompiler.exe` (the output of Sec. 5.2) to compile its own sources. In order to run it, we place in yet another folder ("`DirC`") all the libraries from `DirB` and add `scalacompiler.exe`.

1. The same command-line arguments used in Sec. 5.1 to have the cross-compiler produce `scalalib.msil` are given to `scalacompiler.exe`, to emit the same output as before (exactly the same arguments, forgetting to specify `-target:msil` will cause `scalacompiler.exe` to go out looking for `.jar` files and such).

2. Similarly, in order to have `scalacompiler.exe` emit `scalacompiler.msil`, we use the command-line arguments given in Sec. 5.2.

With that, Scala.NET bootstrapping is done.

A simple way to check whether `scalacompiler.exe` behaves as the cross-compiler involves having both emit a tree of `.msil` files for the same Scala.NET sources, and then compere results. When applying this idea to the compiler sources, the following commands can be used:

```
rem ========================================
rem exe (compiled by the cross-compiler) compiles itself. resulting exe will be used in next step.
rem ========================================

%DIRC_FOLDER%\scalacompiler.exe /*- comand-line options shown below for readability, should be on this line. */
    -Ydebug
    -d C:\temp\discard
    @C:\temp\out-jdk2ikvm\out-src-compiler.txt
    @C:\temp\out-jdk2ikvm\out-src-msil.txt
    -target:msil -Ystruct-dispatch:no-cache
    -Xassem-name scalacompiler
    -Xassem-extdirs %DIRB_FOLDER%
```

```
    -no-specialization
    -Xshow-class scala.tools.nsc.Main

del /q %DIRC_FOLDER%\scalacompiler.exe >nul 2>&1
del /q %DIRC_FOLDER%\scalacompiler.pdb >nul 2>&1

ilasm /QUIET /DEBUG C:\temp\discard\scalacompiler.msil /output=%DIRC_FOLDER%\scalacompiler.exe

peverify %DIRC_FOLDER%\scalacompiler.exe

rem ========================================
rem use exe (compiled by the exe that was compiled by the cross-compiler) to print multiple msil
rem ========================================

c:
cd c:\temp\out-jdk2ikvm\src
del /s /q C:\temp\ilasm\Multiple-Exe\*.msil >nul 2>&1
%DIRC_FOLDER%\scalacompiler.exe /*- comand-line options shown below for readability, should be on this line. */
    -Ygen-javap C:\temp\ilasm\Multiple-Exe
    -d C:\temp\discard
    @C:\temp\out-jdk2ikvm\out-src-compiler.txt
    @C:\temp\out-jdk2ikvm\out-src-msil.txt
    -target:msil -Ystruct-dispatch:no-cache
    -Xassem-name bootstrapped
    -Xassem-extdirs %DIRB_FOLDER%
    -no-specialization
    -sourcepath c:\temp\out-jdk2ikvm\src

rem ========================================
rem use cross-compiler to print multiple msil
rem ========================================

c:
cd c:\temp\out-jdk2ikvm\src

del /s /q C:\temp\ilasm\Multiple-CrossCompiler\*.msil >nul 2>&1

"C:\jdk\bin\java" /*- comand-line options shown below for readability, should be on this line. */
    -Dfile.encoding=UTF-8 -Xbootclasspath/a:%LIBRARIES%
    -Xms512M -Xmx1236M -Xss1M -XX:MaxPermSize=128M
    scala.tools.nsc.Main
    -Ygen-javap C:\temp\ilasm\Multiple-CrossCompiler
    -d C:\temp\discard
    @C:\temp\out-jdk2ikvm\out-src-compiler.txt
    @C:\temp\out-jdk2ikvm\out-src-msil.txt
    -target:msil -Ystruct-dispatch:no-cache
    -Xassem-name bootstrapped
    -Xassem-extdirs %DIRB_FOLDER%
    -no-specialization
    -sourcepath c:\temp\out-jdk2ikvm\src

rem http://stackoverflow.com/questions/605522/print-time-in-a-batch-file-milliseconds
```

7 And now with Mono

7.1 Environ variables

```
export JAVA_OPTS="-Xms1536M -Xmx1536M -Xss1M -XX:MaxPermSize=192M -XX:+UseParallelGC"
export ANT_OPTS="-Xms1536M -Xmx1536M -Xss1M -XX:MaxPermSize=192M -XX:+UseParallelGC"
```

7.2 Getting sources ready for jdk2ikvm

The .patch files below can be downloaded from
http://lamp.epfl.ch/~magarcia/ScalaNET/2011Q2/patches1.zip

```
svn co http://lampsvn.epfl.ch/svn-repos/scala/scala/trunk@24806 scala

patch -p0 -i ForAssemblyVisibility.patch

patch -p0 -i ForCompileByEXE.patch

patch -p0 -i TricksThatJ2KCannot.patch

ant all.clean build

cp scala/build/pack/lib/scala-library.jar scala/lib/
cp scala/build/pack/lib/scala-compiler.jar scala/lib/
```

7.3 Downloading and building jdk2ikvm

```
svn co http://lampsvn.epfl.ch/svn-repos/scala/scala-experimental/trunk/jdk2ikvm jdk2ikvm

./scala/bin/scalac -d classes `find jdk2ikvm/src/scala/tools/jdk2ikvm -name *.scala`

jar -cf jdk2ikvm.jar -C ./classes/ scala -C ./jdk2ikvm/resources/ .
```

7.4 Getting sources ready for the cross-compiler

7.4.1 Prepare folder to hold files converted by jdk2ikvm

```
rm -rf output
mkdir output
mkdir output/msil
mkdir output/library
mkdir output/compiler
```

7.4.2 Converting with jdk2ikvm the msil, library, and compiler folders in trunk/src

1. ```
   ./scala/bin/scalac /*- on the same line */
       -Ystop:superaccessors
       -sourcepath ./scala/src/msil
       -P:jdk2ikvm:output-directory:$HOME/output/msil
       -d $HOME/output/msil
       -Xplugin jdk2ikvm.jar
       -Yrangepos `find scala/src/msil -name *.scala`
   ```

2. similary for `library` and `compiler` (adapt correspondingly `-sourcepath`, `-P:jdk2ikvm:output-directory:`, `-d` , and `find`)

3. In order to compile the compiler sources, a few libraries are needed:

   ```
   -classpath ./scala/lib/fjbg.jar:./scala/lib/jline.jar:./scala/lib/ant/ant.jar
   ```
   ```

7.4.3 Manually applying patches that `jdk2ikvm` does not automate

Copy the contents of `patches2.zip`[7] into the '`output`' folder (i.e., overwrite some files emitted by `jdk2ikvm`).

Before overwriting those files with their patched version, comparing them can be useful (say, using `meld`). Given that `meld` does not support "ignore whitespace" (in particular the end-of-line Windows vs. Unix styles) At the `patched-src` folder, run:

```
fromdos `find . -name *.scala`
```

7.5 Getting dlls ready for the cross-compiler

1. Use `ikvmc` on existing jars: `msil, forkjoin, fjbg, jline, ant`.

   ```
   ikvmc -target:library -out:$HOME/DirA/msil.dll $HOME/scala/build/libs/msil.jar
   ikvmc -target:library -out:$HOME/DirA/fjbg.dll $HOME/scala/lib/fjbg.jar
   ikvmc -target:library -out:$HOME/DirA/jline.dll $HOME/scala/lib/jline.jar
   ikvmc -target:library -out:$HOME/DirA/ant.dll $HOME/scala/lib/ant/ant.jar
   ikvmc -target:library -out:$HOME/DirA/forkjoin.dll $HOME/scala/lib/forkjoin.jar
   ```

2. Obtaining `scalaruntime.dll`:

   ```
   javac -d classes `find $HOME/scala/src/library/ -name *.java`
   jar -cf scalaruntime.jar -C ./classes .
   ikvmc scalaruntime.jar
   ```

3. Obtaining `CSharpFilesForBootstrap.dll` patch Comparator.cs: rename method 'equals' to 'Equals'

   ```
   mcs Comparator.cs SymtabAttribute.cs -target:library
   ```

7.6 Cross-compiling

If multiple `.msil` files are wanted, add `-Ygen-javap <output_folder>` to the compilation line below, while also keeping the `-d` option. The setting `-Xshow-class` is reused by `GenMSIL` to pick the class with the entrypoint (i.e., the class with the main method)

1. from `/usr/lib/mono/2.0` copy `mscorlib.dll` to `DirA` . Similarly for `System.dll` and `System.Xml.dll`. And also `System.Drawing.dll` , needed for the compiler's `TreeBrowser` (but see Note in Sec. 5.2)

2. bring into `DirA` the `.dll`s of the ikvm version in use, `http://www.ikvm.net/download.html`

```
./scala/bin/scalac /*- on the same line */
   -sourcepath $HOME/output/library `find $HOME/output/library -name *.scala`
   -target:msil -Xassem-name scalalib -Ystruct-dispatch:no-cache -no-specialization
   -Xassem-extdirs $HOME/dirA
   -Ydebug
```

[7] `http://lamp.epfl.ch/~magarcia/ScalaNET/2011Q2/patches1.zip`

```
    -d $HOME/output/library

./scala/bin/scalac /*- on the same line */
    `find $HOME/output/compiler -name *.scala` `find $HOME/output/msil -name *.scala`
    -target:msil
    -Xassem-name scalacompiler
    -Ystruct-dispatch:no-cache -no-specialization
    -Xassem-extdirs $HOME/dirB -Ydebug
    -Xshow-class scala.tools.nsc.Main
    -d $HOME/output/compiler
```

7.7 Assembling

1. instead of .NET's `peverify` use `pedump --verify error <assembly>`
2. instead of `.pdb` files, `.mdb` files are used (`pdb2mdb` can be used to obtain the Mono version from its .NET counterpart, or compile with `/debug` using `ilasm2`).

By default, `scalac` in `-target:msil` mode emits a single `.msil` file. In order to have one `.msil` file for each `.scala` file in the input, use

```
-d output/folder -Ygen-javap output/folder
```

(`scalac` in `forMSIL` mode reuses the `-Ygen-javap` setting from its `forJVM` counterpart, as it is "conceptually similar"). Say we have emitted multiple `.msil` files. They can be assembled with:

```
ilasm2 /dll /output:dirB/scalalib.dll `find $HOME/output/library -name *.msil`
```

When `/debug` is given, Mono versions 2.10.1 or earlier run into `http://bugzilla.novell.com/show_bug.cgi?id=633312`

```
ilasm2 /exe /debug /output:dirC/scalacompiler.exe /*- on the same line */
  `find $HOME/output/msil -name *.msil` `find $HOME/output/compiler -name *.msil`
```

7.8 Troubleshooting and Debugging

What if mono throws some exception when running my program? Try with:

```
mono --help-devel
mono --with-profile4=yes
```

If running an OpenSUSE guest in VirtualBox, and want to share folder with the host (say, a Windows host)[8]

```
sudo /sbin/mount.vboxsf <name_given_in_VBox_setting> /home/mg/c_temp
```

Debugging: `http://tirania.org/blog/archive/2010/Feb-20.html`

[8] `http://www.virtualbox.org/manual/ch04.html`

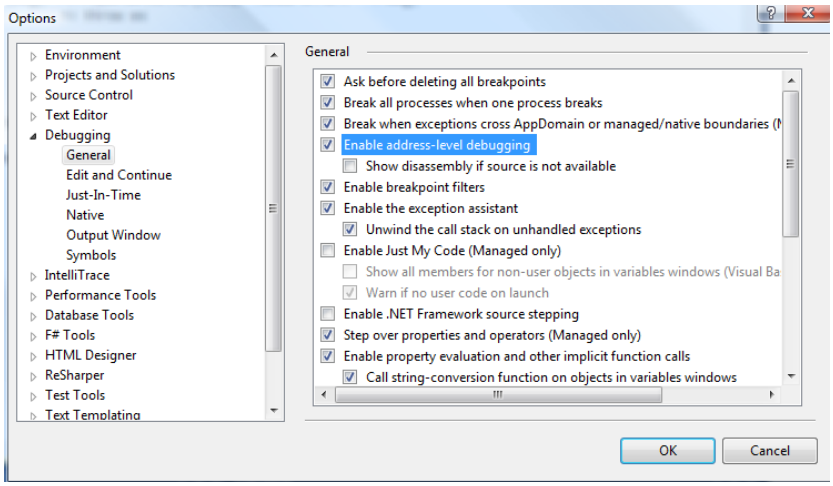


Figure 6: Sec. 8.1

8 After bootstrapping, the fun only starts

8.1 Debugging in Visual Studio

As a bonus, this section covers how to debug assemblies emitted by Scala.NET.

Another write-up[9] described how to debug either at the `.scala` or `.msil` level, and the snippets in `GenMSIL` responsible for emitting debug information. Summing up, in order to step-debug over Scala sources:

1. "*File / New Project*"
2. "*Other Project Types / Visual Studio solutions / Blank solution*"
3. right click on the new solution, "*Add existing project*" and pick `scalacompiler.exe` (on the same folder, `scalacompiler.pdb` should be found)
4. From the main menu, "*Debug / Options and Settings*", mark "*Enable address-level debugging*" as shown in Figure 6.
5. ready to go, "*Step into new instance*". Besides line breakpoints, exception breakpoints are very useful ("*Debug / Exceptions*", Figure 7).

8.2 Profiling

- CLR Inside Out Article: Profiling the .NET Garbage-Collected Heap[10]
- Quoting from *CLR Profiler for .NET Framework 4*[11]

[9] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/ikvmify4.pdf`

[10] `http://msdn.microsoft.com/en-us/magazine/ee309515.aspx`

[11] `http://www.microsoft.com/downloads/en/details.aspx?FamilyID=be2d842b-fdce-4600-8d32-a3cf74fda5e1`

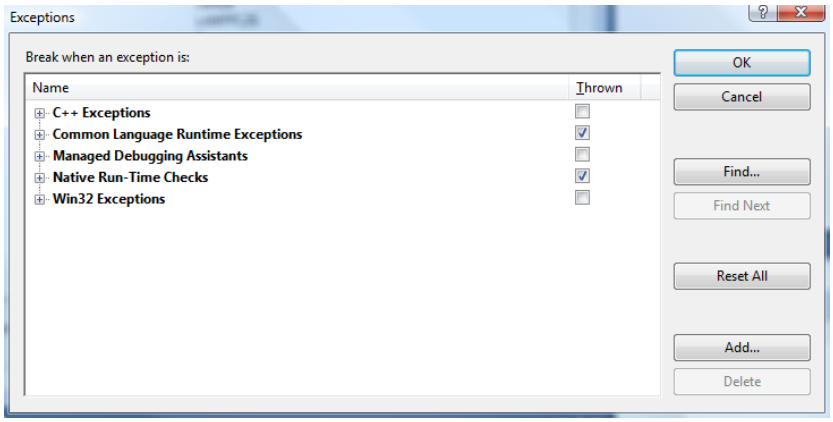


Figure 7: Sec. 8.1

> *The CLR Profiler includes a number of very useful views of the allocation profile, including a histogram of allocated types, allocation and call graphs, a time line showing GCs of various generations and the resulting state of the managed heap after those collections, and a call tree showing per-method allocations and assembly loads*

8.3 A few words about Generics: *Coming soon!*

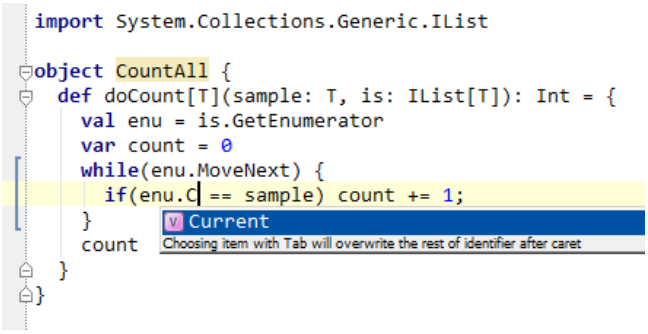
Right now, the backend of Scala.NET does not support generics. This is so because we wanted first to get bootstrapping to work (which can be done without full support for Generics: the IKVM library, just like the OpenJDK bytecode it is based on, contains "erased" types only).

Therefore the programs Scala.NET can as of now compile may make use of the Scala library, the IKVM library, and "pre-.NET-2.0" assemblies. Summing up, types, methods, and fields from external assemblies can be used as long as they are monomorphic.

Invoking Scala assemblies from C# (or any other .NET language) is possible, however those APIs are low-level as compared to the API seen from Scala.NET source code. Afterwards, porting any such C# code to Scala.NET will thus involve not only getting rid of "idiomatic C#" but also using Scala's libraries at the level of Scala types, instead of their lowering to CLR types. With that caveat, it's certainly possible to use for example `scalalib.dll` from C#.

8.4 Developing Scala.NET apps from your JVM IDE

No, really! For example, auto-completion and source browsing at your fingertips:



`http://lamp.epfl.ch/~magarcia/ScalaNET/2011Q2/TestDriveMD2SRC.pdf`

8.5 Acknowledgement

Big thanks to Jeroen Frijters for creating IKVM, `http://www.ikvm.net/`.

References

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

Listing 1: Sec. 3.1

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

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

implicit def nullToStructCharSequence(n: Null): java.lang.CharSequence = {
  val c : java.lang.CharSequence = new java.lang.CharSequence() // default init
  c.‘__<ref>‘ = null
  c
}

implicit def seqToCharSequence(xs: collection.IndexedSeq[Char]): java.lang.CharSequence = {
  val w = new java.lang.CharSequence.__Interface {
    def length: Int = xs.length
    def charAt(index: Int): Char = xs(index)
    def subSequence(start: Int, end: Int): java.lang.CharSequence = seqToCharSequence(xs.slice(start, end))
    override def ToString: String = xs.mkString("")
  override def toString() = { this.ToString() } /*addMissingJLObjOverrides*/
  }
  val c : java.lang.CharSequence = new java.lang.CharSequence() // default init
  c.‘__<ref>‘ = w
  c
}

implicit def arrayToCharSequence(xs: Array[Char]): java.lang.CharSequence = {
  val w = new java.lang.CharSequence.__Interface {
    def length: Int = xs.length
    def charAt(index: Int): Char = xs(index)
    def subSequence(start: Int, end: Int): java.lang.CharSequence = arrayToCharSequence(xs.slice(start, end))
    override def ToString: String = xs.mkString("")
  override def toString() = { this.ToString() } /*addMissingJLObjOverrides*/
  }
  val c : java.lang.CharSequence = new java.lang.CharSequence() // default init
  c.‘__<ref>‘ = w
  c
}
```

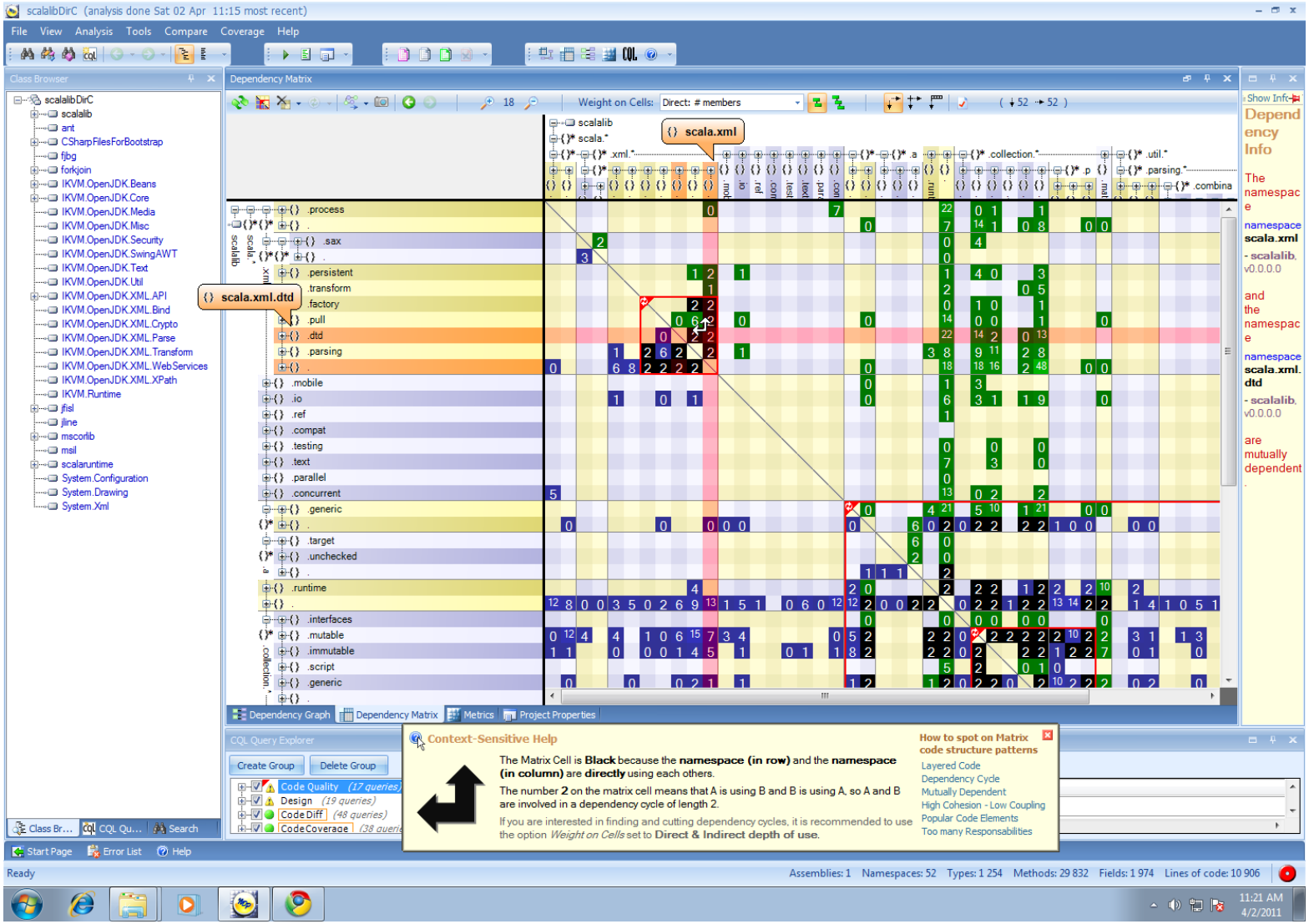


Figure 8: Dependency matrix of `scalalib.dll` shown by NDepend