jdk2ikvm and next steps for Scala.NET (bonus: a preview of scala.tools.unparse)

Miguel Garcia
http://lamp.epfl.ch/~magarcia/ScalaNET/

LAMP, EPFL

2011-01-18

Outline

Last time we reviewed some goals for Scala.NET:

- interoperate with assemblies emitted by other compilers
- deal with CLR specifics (e.g., unsigned integrals, `structs` and `address-of`, overflow-checking arithmetic)
- support compiler plugins, LINQ, play nice with .NET tooling, IDEs.

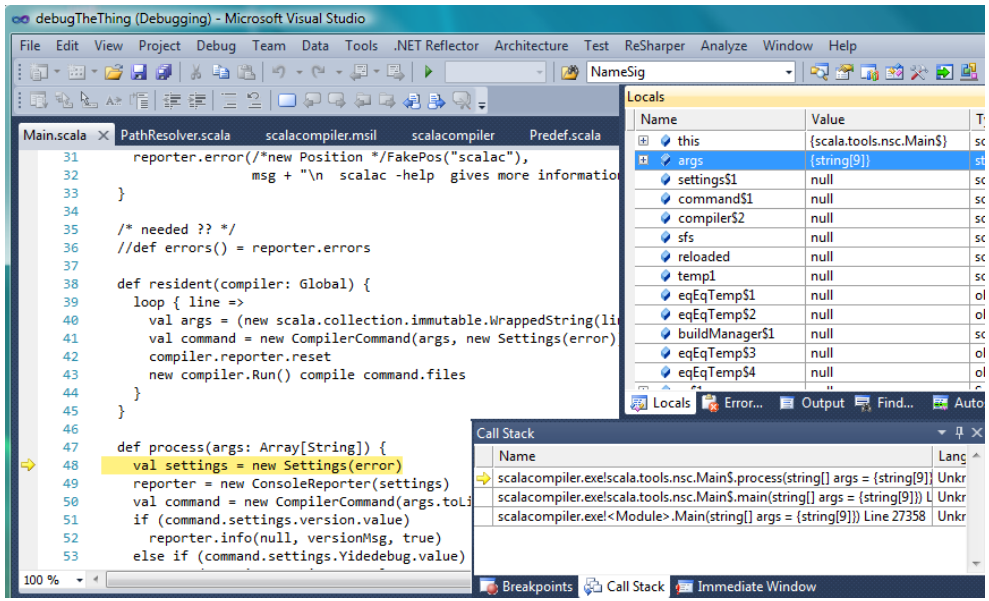
We also looked at IKVM and the way it automates platform migration:

- interplay of the *IKVM library* (`.dll` with JDK-like API); and the `ikvmc` compiler, which performs a fair amount of rewriting on the way from `jar` to `exe`,
- rewritings that we dub *the JDK to IKVM conversion recipe*[1]

[1] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/jdk2ikvmPartA.pdf`

There's nothing wrong with the screenshot below ("*patched compiler*").

- After all, it does not show the architectural drift that had accrued with respect to `forJVM` mode.
- We started shoehorning the JDK-to-IKVM conversion into the compiler well before having a clear picture about its full extent (hint: the pseudocode summary of the conversion takes 8 pages).
- Please note: It's easy to be wise after the fact.



JDK-to-IKVM not only *can* be formulated at the level of Scala sources: doing so adds value beyond "just" avoiding arhitectural drift.

The way jdk2ikvm does it (sample conversion: instance-method receiver turned into first arg of class-static invocation)

```
1 object HelloWorld {
2   val x = (new String(Array('h','e','l','l','o'))
3             indexOf 2)
4 }
5
6
7
```

```
1 object HelloWorld {
2   val x = (java.lang.String.instancehelper_indexOf(
3               java.lang.String.newhelper(
4                 Array('h','e','l','l','o')) , 2)
5             )
6 }
7
```

Range positions (-Yrangepos) can nest, so must patches[2]

```
          x5 = "abc" substring (0   ,    3)
app             |-------------------------|[341:367]
fun             |---------------|           [341:356]
quali           |----|                      [341:346]
arg0                          ||            [358:359]
arg1                                ||      [365:366]
```

[2] http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q1/ValidatePositions.pdf

Another conversion step: adding co-overrides for `String` and `Object`.

Below, in `jdk2ikvm`:

```
private def coOverrides(tree: Tree) {
  tree match {
    case dd: DefDef if (dd.symbol.isMethod) && (dd.symbol.isSourceMethod) && (!dd.symbol.isSynthetic)
       &&    (dd.symbol.overriddenSymbol(ObjectClass) != NoSymbol)
       &&    (dd.symbol != Object_clone)
       &&    (dd.symbol != Array_clone)
    =>
      dd.symbol.name match {
        case nme.hashCode_ =>
          spliceAfter(dd, "override def GetHashCode() = { this.hashCode() }")
        case nme.toString_ =>
          spliceAfter(dd, "override def ToString() = { this.toString() }"   )
        case nme.equals_   =>
          spliceAfter(dd, "override def Equals(that: Object) = { this.equals(that) }"     )
        case nme.finalize_ =>
          spliceAfter(dd, "override def Finalize() = { this.Finalize() }"   )
        case _ =>
          warning(dd.pos, "j.l.Object override not co-overridden with .NET counterpart:\n"+asString(dd))
      }
    case _ => ()
  }
}
```

Compare with the thrill (yes, right) of adding trees and symbols in the *patched compiler*.

```
359  def addMissingObjectContract(msym: Symbol) { /* FOR BOOTSTRAP ONLY */
360    if (clazz.isAbstractClass) return
361    val implReady = ts exists { case DefDef(_, ddname, _, ddvparamss, _, _)
362                                   => (ddname == msym.name) && (msym.paramss.head.length == ddvpa
363                                 case _ => false }
364    if (implReady) return
365    if (!inheritsAbstrDecl(msym)) return
366    if (definesOrInheritsImplOtherThanAnyRef(msym)) return
367    val t : Tree = msym.name match {
368      case nme.hashCode_ => msilObjectMethod("hashCode", "GetHashCode")
369      case nme.toString_ => msilObjectMethod("toString", "ToString")
370      case nme.equals_   => msilObjectMethod("equals"  , "Equals")
371    }
372    ts += t
373  }
374
375  def msilOverrideMethod(name: Name, targetmsym: Symbol, rhs: Tree): Tree = {
376    val paramtypes  = targetmsym.tpe.paramTypes
377    val method      = syntheticMethod(
378      name, 0, makeTypeConstructor(paramtypes, targetmsym.tpe.resultType)
379    )
380    typer typed { DEF(method) === rhs }
381  }
382
383  def addMissingIEnumerableContract() { /* FOR BOOTSTRAP ONLY */
384    if (clazz.isAbstractClass) return
385    val msym = getMember(IKVMSCIEnumerable, "GetEnumerator")
386    if (!inheritsAbstrDecl(msym)) return
387    if (definesOrInheritsImplOtherThanAnyRef(msym)) return
388    val jlIterable = definitions.getClass("java.lang.Iterable")
389    val argss = List(List(This(clazz)))
390    val rhs : Tree = if (clazz.info.baseClasses contains jlIterable) {
391      val ikvmIterableEnumerator = definitions.getClass("ikvm.lang.IterableEnumerator")
392      New(TypeTree(ikvmIterableEnumerator.tpe), argss)
393    } else {
394      val ikvmMapEnumerator = definitions.getClass("ikvm.lang.MapEnumerator")
395      New(TypeTree(ikvmMapEnumerator.tpe), argss)
396    }
397    ts += msilOverrideMethod("GetEnumerator", msym, rhs)
398  }
399
400  // similar to msilObjectMethod, expect that here we invoke (1) on this a (2) jdk method
401  def msilDetourMethod(classWithTarget: Symbol, jdkName: Name, dotnetName: Name): Tree = {
402    val target      = getMember(classWithTarget, jdkName)
403    val paramtypes  = target.tpe.paramTypes
404    val method      = syntheticMethod(
405      dotnetName, 0, makeTypeConstructor(paramtypes, target.tpe.resultType)
406    )
407    val toAdd = typer typed {
408      DEF(method) === {
409        val thisRef: Tree = Select(This(clazz), target)
410        Apply( thisRef, method ARGNAMES )
411      }
412    }
413    toAdd
414  }
415
416  def addMissingIDisposableContract() { /* FOR BOOTSTRAP ONLY */
417    if (clazz.isAbstractClass) return
418    val msym = getMember(IKVMSCIDisposable, "Dispose")
419    if (!inheritsAbstrDecl(msym)) return
420    if (definesOrInheritsImplOtherThanAnyRef(msym)) return
421    val jioCloseable = definitions.getClass("java.io.Closeable")
422    if (clazz.info.baseClasses contains jioCloseable) {
423      val rhs : Tree = msilDetourMethod(jioCloseable, "close", "Dispose")
424      ts += rhs
425    } else {
426      scala.Console.println("could not addMissingIDisposableContract for " + clazz) // error
427    }
428  }
429
430  addMissingObjectContract(Object_equals)
```

LINE 359

- Target audience for jdk2ikvm: Developers with a JDK-based Scala codebase who want to migrate to .NET
  - either as a one-time migration
    Please note: impossible with the *patched-compiler* approach
    (i.e., only the Scala.NET codebase is maintained afterwards); or
  - supporting both platforms in parallel.

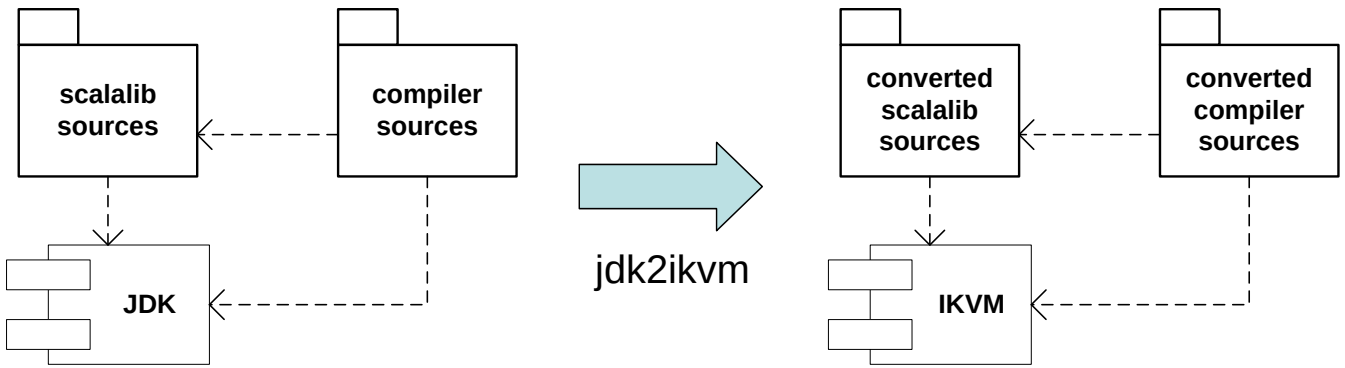
- Ideas for the future:
  - the migration path
    (Java on JDK) $\rightarrow$ *(Scala on IKVM + (.NET or Mono))*
    now requires ("only") a more complete Java-to-Scala translator
    (existing prototypes: scalify[3], jatran[4], java2scala[5])
  - "same-platform" API migration tools
    - from java.io to revamped scala.io
    - from Java to Scala Collections, etc.

  so as to progressively break ties, moving towards a Scala platform

[3] http://github.com/paulp-etc/scalify
[4] http://code.google.com/p/jatran/
[5] http://java2scala.svn.sourceforge.net/

Bootstrapping, using `jdk2ikvm`:

1. output Scala.NET sources from unmodified JDK-based trunk
2. cross-compile them to obtain `scalacompiler.exe`
3. use `scalacompiler.exe` (not the cross-compiler) to compile the output of `jdk2ikvm`



The fine print: how it's going.

And now for something different:

Have you heard about "unparsing"?

Unparsing is like pretty-printing, except that

- as-seen-from type information is made explicit in the output (inspired by Scaladoc),
- desugarings introduced by `parser`, `namer`, and `typer` are also made explicit.

Additionally,

- unparsed code compiles and behaves the same as the program it was obtained from.

However, it's not required for the unparsed program:

- to be binary compatible with code compiled against the original program, nor
- to resemble the original layout.

Example:

```
/*- original */
def listItemsToHtml(items: Seq[Block]) =
  items.foldLeft(xml.NodeSeq.Empty){ (xmlList, item) =>
    item match {
      case OrderedList(_, _) | UnorderedList(_) =>  // html requires sub ULs to be put into
        xmlList.init ++ <li>{ xmlList.last.child ++ blockToHtml(item) }</li>
    }
}
```

```
/*- unparsed */
def listItemsToHtml(items : scala.collection.Seq[scala.tools.nsc.doc.model.comment.Block]) =
  items.foldLeft[scala.xml.NodeSeq](scala.xml.NodeSeq.Empty)(
    ((xmlList : scala.xml.NodeSeq,
      item : scala.tools.nsc.doc.model.comment.Block) => (item match {
        case (scala.tools.nsc.doc.model.comment.OrderedList(_, _) | scala.tools.nsc.doc.model.comment.UnorderedList(_)) =>
          xmlList.init.++[scala.xml.Node, scala.xml.NodeSeq](new scala.xml.Elem((null), ("li"), scala.xml.Null, scala.$scope,
            ({ $buf = new scala.xml.NodeBuffer()
              buf.&+(xmlList.last.child.++[scala.xml.Node, Any](M.blockToHtml(item))(collection.Seq.canBuildFrom[scala.xml.Node
              $buf }: _*)))(xml.NodeSeq.canBuildFrom) ))))
```

Why would someone want to read unparsed code?

- for one, to visualize what a given phase does (I've always wanted to know what `specialize` does to my program :-)
- Admittedly, benefit inverse with expertise. Put more bluntly,

  The fact that unparsing is not useful for experts
  does not mean
  it's not useful for many other developers.

- yes, forward jumps have to be defunctionalized using an explicit state machine[6].

However, the main benefit of unparsing may come from another angle: improving the economics[7] of compiler-plugin development.

---

[6] http://www.scala-lang.org/node/7423

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

An "*unparsing AST-aware pre-processor*"[8] is a compiler plugin with a `Transformer` that trades some subtrees for *non-typed parse trees*.

Compared to "traditional" compiler plugins:

- Cons: longer wall-clock time (two compiler runs).
- Pros:
  - take a break from the thrill of adding term and type symbols; and
  - not constrained to the Scala subset that later phases understand (e.g., ASTs after `explicitouter` should do without `Match`es).

Claim: the above amounts to an orders-of-magnitude speedup for first-time compiler-plugin developers.

Target niche: pre-processors as proofs-of-concept. In case demand justifies development, evolution path exists to full-fledged plugins (with expected code reuse of over 50%).

[8] For an example see `http://lampsvn.epfl.ch/trac/scala/browser/scala-experimental/trunk/jdk2ikvm/src/scala/tools/jdk2ikvm/UnparsingJDK2IKVM.scala`

Warning: entering brainstorm zone . . .

Candidate pre-processors that come to mind:

- desugar into sentences of *a virtualized language*
- Atomicity via Source-to-Source Translation *(Hindman, Grossman)*

  http://www.eecs.berkeley.edu/~benh/atomjava.pdf

Verification-related deserves its own section:

- Temporal JML: runtime checks given temporal properties as DSL *(Hussain, Leavens)* www.eecs.ucf.edu/~fhussain/papers/temporaljmlc.pdf
- Typestates, anyone?

This is *not* to say that pre-processors are superior to libraries. See:

- Contracts for Scala (*Odersky*) dx.doi.org/10.1007/978-3-642-16612-9_5
- .NET Code Contracts research.microsoft.com/en-us/projects/contracts/

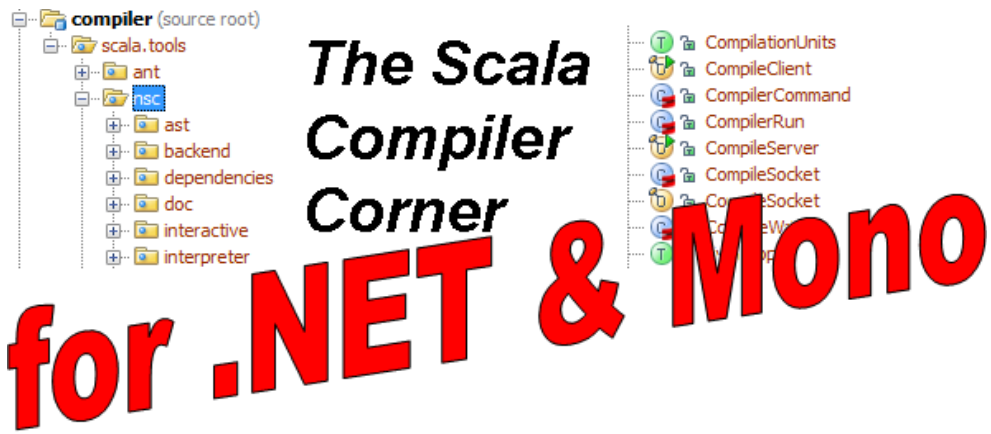
. . . leaving brainstorm zone.

Next steps

1. Hardening the compiler (stackmaps, overflow checking, unsigned integrals, “attempt to enter a try-block with non-empty stack”, etc.)
2. automated tests after running:
   trunk → `jdk2ikvm` → cross-compiler → `scalacompiler.exe`
3. “Generics in the backend”
4. emit binary assemblies as per *Common Compiler Infrastructure*[9]
5. Visual Studio Language Service

Coming soon to . . .

`http://lamp.epfl.ch/~magarcia/ScalaNET/`



[9] `http://ccimetadata.codeplex.com/`

Summary of the JDK-to-IKVM conversion (1 of 2):

1. Transforms for the `String` and `Object` contracts
   - 1.1 instance helpers, new helpers, co-overrides for non-sealed methods, add missing `j.l.Object` overrides
   - 1.2 `clone()` on arrays, `Finalize()` body-with-check.
2. Magic for interfaces
   - 2.1 Extra interfaces
   - 2.2 Implied interfaces
   - 2.3 Upcast to extra interface (string comparison semantics, rewrite standalone type refs)
3. Ghost interfaces
   - 3.1 Standalone type refs to `Cloneable` and `CharSequence`
   - 3.2 instance method invocations, == and !=
   - 3.3 Type casts and checks

Summary of the JDK-to-IKVM conversion (2 of 2):

4. Erase type arguments to all IKVM classes
5. Ignore `@throws`
6. IKVM's `Class.getMethod` and `Method.invoke` require explicit empty array for repeated param. (Similarly for other repeated params in JDK signatures)
7. Exceptions
   - 7.1 Case (1) catch Throwable
   - 7.2 Case (2) catch Exception or catch Error
   - 7.3 Case (3) otherwise