

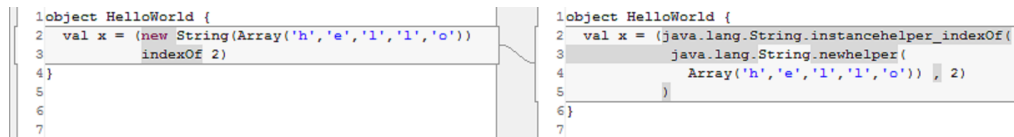
# A source-level, automatic API migration that preserves layout (a story of range positions)

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

January 10<sup>th</sup>, 2011

## Abstract

Whether you're interested in unparsing, pretty-printing, or refactoring of Scala ASTs, chances are high you'll have to do with *tree positions*, in particular of the range variety as results from `-Yrangepos`. This write-up summarizes some findings about them, gained while developing an API migration tool (`jdk2ikvm`).



```
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
```

## Contents

<b>1</b>	<b>Background</b>	<b>2</b>
1.1	How to build and run . . . . .	2
1.2	Motivation . . . . .	2
1.3	Implementing the “String instance helpers” transformation . . .	3
<b>2</b>	<b>Invariants for tree positions</b>	<b>4</b>
<b>3</b>	<b>validatePositions</b>	<b>5</b>
3.1	Enclosing-enclosed tree pairs . . . . .	6
3.2	Overlap tests for children of ranged trees . . . . .	6
3.3	Tests for solid descendants . . . . .	6
<b>4</b>	<b>Debug session</b>	<b>7</b>
4.1	Accessing a package object . . . . .	7
4.1.1	Solution . . . . .	8
4.2	Accessing <code>this</code> . . . . .	9
4.2.1	After <code>typer</code> . . . . .	9
4.2.2	After <code>parser</code> . . . . .	9
4.2.3	Where it gets transformed . . . . .	9

# 1 Background

## 1.1 How to build and run

Build and run instructions for `jdk2ikvm` can be found in Sec. 1 of:

- *Learning and doing scalac transformations the easy way: via unparsing*<sup>1</sup>

Other related write-ups:

- *Bits and pieces of information about the parser, namer, and typer phases that turn out to be necessary just to be able to unparse Scala ASTs*<sup>2</sup>
- *Unparsing types the Scaladoc way*<sup>3</sup>

## 1.2 Motivation

Say we want to perform the following transformation (as required by the *JDK to IKVM conversion recipe*):

```
/* Example:
 *      "abc".length
 * -->
 *      java.lang.String.instancehelper_length("abc")
 */
```

In order to preserve layout, for different cases of `Apply` nodes (with and without args, parens around args or not, using dot notation or not), we have to deal with examples as the following:

(Please notice that invocations of JDK methods do not make use of named arguments, thus we need not consider them).

```
      x1 = "abc".substring(0, AboutPositions.this.padding)
app      |-----| [106:153]
fun      |-----| [106:121]
quali    |----| [106:111]
arg0     || [122:123]
arg1     |-----| [125:152]
```

```
      x2 = "def".length
app      |-----| [185:197]
fun      |-----| [185:197]
quali    |----| [185:190]
```

```
      x3 = "xyz".isEmpty()
app      |-----| [236:251]
fun      |-----| [236:249]
quali    |----| [236:241]
```

<sup>1</sup><http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/Unparsing.pdf>

<sup>2</sup><http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/unparsing.pdf>

<sup>3</sup><http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q1/TypesScaladocWay.pdf>

```

                x4 = "xyz" indexOf 'n'
app             |-----| [290:307]
fun             |-----| [290:303]
quali          |----| [290:295]
arg0            |--| [304:307]

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

```

### 1.3 Implementing the “String instance helpers” transformation

We have to gather first the ranges for those sub-expressions that might be rewritten by other transformations. Afterwards, place them in the order they should appear in the output, with any additional (fixed) text as `Insert` patch commands.

It remains to explain how a “patch tree” organizes the “patch commands” it receives (by nesting patch commands for sub-expressions within that for its containing expression, where “nesting” follows the interval inclusion relationship, where “interval” is a closed non-empty integer interval, where the “integer” denotes an offset in the `Array[Char]` given by `SourceFile.content`).

*Armed with the intuition above*, I hope the code below makes some sense (we have to start somewhere):

```

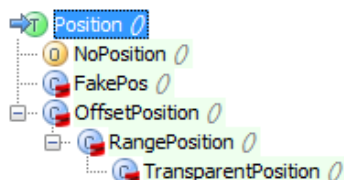
val buf = mutable.ListBuffer.empty[PatchCmd]
// new (static) receiver
buf += new Insert("java.lang.String.instancehelper_" + originalName + "(")
// new first argument
buf += new Patcheable(qualiPos.start, qualiPos.end - 1)
// existing arguments if any
if(!args.isEmpty) {
  buf += new Insert(", ")
  val headPos = args.head.pos.asInstanceOf[RangePosition]
  val lastPos = args.last.pos.asInstanceOf[RangePosition]
  buf += new Patcheable(headPos.start, lastPos.end - 1)
}
buf += new Insert(")")
patchtree.tryPatch(appPos.start, appPos.end - 1, buf.toList, dropGapsOK = true)

```

After exploring the code for a “patch tree” you might wonder why a custom data structure was used rather than a self-balancing tree of closed intervals. In order to use a `TreeSet`, a total order is necessary, while interval-inclusion determines a poset. Although an artificial `Ordering` could still have been defined by adding, say, “arrival order” to the less-or-equal criteria, in fact we want to leverage a feature of the rewritings that results from the pre-order traversal of ASTs we follow: enclosing expressions are rewritten first, i.e. rewritings with the longest ranges send first “patch commands” to the patch tree. Once the nodes for them have been added, rewritings for sub-expressions just result in more

leaves added. The resulting custom data structure is fast, both for additions and during traversal for serializing into the output file.

## 2 Invariants for tree positions



Quoting from `RangePositions.scala`:

### *Handling range positions*

*atPos*, the main method in this trait, will add positions to a tree, and will ensure the following properties:

- *INV-A*: All nodes between the root of the tree and nodes that already have positions will be assigned positions.
- *INV-B*: No node which already has a position will be assigned a different range; however a `RangePosition` might become a `TransparentPosition`.
- *INV-C*: The position of each assigned node includes the positions of each of its children.
- *INV-D*: The positions of all solid descendants of children of an assigned node are mutually non-overlapping.

Here, the solid descendant of a node are:

- If the node has a `TransparentPosition`, the solid descendants of all its children
- Otherwise, the singleton consisting of the node itself.

Quoting from `scala.tools.nsc.util.Position`:

The `Position` class and its subclasses represent positions of ASTs and symbols. Except for `NoPosition` and `FakePos`, every position refers to a `SourceFile` and to an offset in the sourcefile (its 'point'). For batch compilation, that's all. For interactive IDE's there are also `RangePositions` and `TransparentPositions`. A `RangePosition` indicates a start and an end in addition to its point. `TransparentPositions` are a subclass of `RangePositions`. Range positions that are not transparent are called opaque. Trees with `RangePositions` need to satisfy the following invariants.

- *INV1*: A tree with an offset position never contains a child with a range position

```

override def validatePositions(tree: Tree) {

  def desc(t: Tree, prefix : String = "") = {...}

  def wfBroken(rule: String, tree: Tree, encltree: Tree = EmptyTree) {...}

  /** range in range only, coverage by enclosing, and non-overlap of ranged-over) */
  def rangedTreeTests(tree: Tree, encltree: Tree) {...}

  def validate(tree: Tree, encltree: Tree): Unit = {
    if (tree.isEmpty) return
    // ranged or not, tests about (enclosing) definedness
    if (!tree.pos.isDefined) wfBroken("non-empty implies defined position", tree)
    if (!encltree.pos.isDefined && tree.pos.isDefined) wfBroken("undefined can't enclose defined")
    // tests for range trees only
    rangedTreeTests(tree, encltree)
    // skip transparent descendants, recurse to solid descendants only
    for (ct <- tree.children flatMap solidDescendants) validate(ct, tree)
  }

  validate(tree, tree)
}

```

Figure 1: validatePositions, Sec. 2

- *INV2: If the child of a tree with a range position also has a range position, then the child's range is contained in the parent's range.*
- *INV3: Opaque range positions of children of the same node are non-overlapping (this means their overlap is at most a single point).*

...

### 3 validatePositions

At its core, validation (Figure 1) looks at non-empty trees and its *solid descendants*. Two kinds of tests are performed: (a) those for enclosing-enclosed tree pairs, and (b) overlap tests for children; as reviewed in the next two subsections. Tests for solid descendants do not involve actually any more tests, the just recursively perform (a) and (b) as shown in Sec. 3.3.

The big picture:

```

def validate(tree: Tree, encltree: Tree): Unit = {
  if (tree.isEmpty) return
  val brokenRules = ListBuffer.empty[String]
  // ranged or not, tests about (enclosing) definedness
  if (!tree.pos.isDefined) brokenRules += "Non-empty implies defined position"
  if (!encltree.pos.isDefined && tree.pos.isDefined) brokenRules += "Undefined can't enclose defined"
  // tests for range trees only
  rangedTreeTests(tree, encltree, brokenRules)
  reportBroken(tree, encltree, brokenRules)
  // skip transparent descendants, recurse to solid descendants only
  for (ct <- tree.children flatMap solidDescendants) validate(ct, tree)
}

```

### 3.1 Enclosing-enclosed tree pairs

Most of the enclosing-enclosed tests apply to a ranged enclosed tree, but the following is expected of all trees:

```
// ranged or not, tests about (enclosing) definedness
if (!tree.pos.isDefined)
  brokenRules += "Non-empty implies defined position"

if (!encltree.pos.isDefined && tree.pos.isDefined)
  brokenRules += "Undefined can't enclose defined"
```

Three enclosing-related tests are for ranged trees only:

```
/** range in range only, coverage by enclosing, and non-overlap of ranged-over) */
def rangedTreeTests(tree: Tree, encltree: Tree, brokenRules: ListBuffer[String]) {
  if (!tree.pos.isRange) return
  if (!encltree.pos.isRange) brokenRules += "A range must have a range parent"
  if (!(encltree.pos.includes tree.pos)) brokenRules += "Enclosing must cover enclosed"
  // The positions of all solid descendants of children of an assigned node are mutually non-overlapping.
  . . . findOverlapping, see below
}
```

### 3.2 Overlap tests for children of ranged trees

```
// The positions of all solid descendants of children of an assigned node are mutually non-overlapping.
findOverlapping(tree.children flatMap solidDescendants) match {
  case List() => ;
  case xs => {
    reportBroken("Overlapping trees "+xs.map { case (x, y) => (x.id, y.id) }.mkString("", ", ", ""))
    for((x, y) <- xs) {
      describe(x, "First overlapping ")
      describe(y, "Second overlapping ")
    }
  }
}
```

### 3.3 Tests for solid descendants

What is tested for solid descendants has been reviewed above (Sec. 3.1 and Sec. 3.2), invoked recursively:

```
def validate(tree: Tree, encltree: Tree): Unit = {
  if (tree.isEmpty) return
  val brokenRules = ListBuffer.empty[String]
  // ranged or not, tests about (enclosing) definedness
  if (!tree.pos.isDefined) brokenRules += "Non-empty implies defined position"
  if (!encltree.pos.isDefined && tree.pos.isDefined) brokenRules += "Undefined can't enclose defined"
  // tests for range trees only
  rangedTreeTests(tree, encltree, brokenRules)
  reportBroken(tree, encltree, brokenRules)
  /*- skip transparent descendants, recurse to solid descendants only */
  for (ct <- tree.children flatMap solidDescendants) validate(ct, tree) /*- <----- HERE */
}
```

## 4 Debug session

### 4.1 Accessing a package object

- The AST for the expression `math.max(stacksize, that.getStacksize())` has a `Select` of the form `scala.math.package` which is not ranged, while its enclosed `scala.math` is ranged. Thus we get “Range in range only broken”.
- And given that the enclosing node has an offset position, the enclosed range (which is non-zero length) isn’t covered, thus earning us another error message.

After parser, a `math.max(1, 2)` expressions looks as follows:

```
Apply( // sym=<none>, tpe=null
  Select( // sym=<none>, sym.tpe=<notype>, tpe=null
    Ident("math"), // sym=<none>, sym.tpe=<notype>, tpe=null,
    "max"),
  List( // 2 arguments(s)
    Literal(Constant(1)),
    Literal(Constant(2))
  )
)
```

i.e. there’s as of yet no `Select` with `nme.PACKAGEkw`. That’s added in `Typers.scala`, by `makeAccessible`:

```
/** Make symbol accessible. This means:
 * If symbol refers to package object, insert ‘.package’ as second to last selector.
 * (exception for some symbols in scala package which are dealiased immediately)
 * Call checkAccessible, which sets tree’s attributes.
 * Also note that checkAccessible looks up sym on pre without checking that pre is well-formed
 * (illegal type applications in pre will be skipped -- that’s why typedSelect wraps the resulting tree in a Tree)
 * @return modified tree and new prefix type
 */
private def makeAccessible(tree: Tree, sym: Symbol, pre: Type, site: Tree): (Tree, Type) =
```

Figure 2 shows the after-transform result, with the offset position dominating a range position, where `atPos` was given a `tree.pos.focusStart` as the following excerpt shows:

```
val qual = typedQualifier { atPos(tree.pos.focusStart) {
  tree match {
    case Ident(_) => Ident(nme.PACKAGEkw)
    case Select(qual, _) => Select(qual, nme.PACKAGEkw)
    case SelectFromTypeTree(qual, _) => Select(qual, nme.PACKAGEkw)
  }
}}
```

To confirm, after `typer` our tree now looks like:

```
Apply( // sym=method max, tpe=Int, tpe.sym=class Int, tpe.sym.owner=package scala
  Select( // sym=method max, sym.owner=class MathCommon, sym.tpe=(x: Int,y: Int)Int, tpe=(x: Int,y: Int)Int, tpe.sym=class Int, tpe.sym.owner=package scala
    Select( // sym=package object math, sym.owner=package math, sym.tpe=object scala.math.package, tpe=math.package
      Select( // sym=package math, sym.owner=package scala, sym.tpe=package math, tpe=math.type, tpe.sym=package math
        Ident("scala"), // sym=package scala, sym.owner=package <root>, sym.tpe=package scala, tpe=type, tpe.sym=package scala
        "math"),
      "math"),
    "max"),
  List( // 2 arguments(s)
    Literal(Constant(1)),
    Literal(Constant(2))
  )
)
```

```

qual = {scala.reflect.generic.Trees$Select@2338} "scala.math.package"
├─ qualifier = {scala.reflect.generic.Trees$Select@1843} "scala.math"
│  └─ qualifier = {scala.reflect.generic.Trees$Ident@2304} "scala"
│     └─ name = {scala.tools.nsc.symtab.Names$TermName@2305} "math"
│        └─ symbol = {scala.tools.nsc.symtab.Symbols$ModuleSymbol@2306} "package math"
│           └─ id = 119
│              └─ rawpos = {scala.tools.nsc.util.RangePosition@2307} "RangePosition(bt4.scala, 37, 37, 41)"
│                 └─ rawtpe = {scala.tools.nsc.symtab.Types$Nanon$3@1842} "math.type"
│                    └─ $outer = {scala.tools.nsc.interactive.Global@2308}
├─ name = {scala.tools.nsc.symtab.Names$TermName@2344} "package"
├─ symbol = {scala.tools.nsc.symtab.Symbols$ModuleSymbol@2345} "package object math"
└─ id = 121
   └─ rawpos = {scala.tools.nsc.util.OffsetPosition@2346} "source: z:\scalaproj\simple\backend\src\bt4.scala,line-3,offset=57"
      └─ rawtpe = {scala.tools.nsc.symtab.Types$Nanon$3@2347} "math.package.type"
         └─ $outer = {scala.tools.nsc.interactive.Global@2308}

```

Figure 2: Sec. 4.1

```

    "package"),
    "max"),
    List( // 2 arguments(s)
      Literal(Constant(1)),
      Literal(Constant(2))
    )
  )
)

```

#### 4.1.1 Solution

The atPos invoked above is this one:

```

def atPos[T <: Tree](pos: Position)(tree: T): T = {
  posAssigner.pos = pos
  posAssigner.traverse(tree)
  tree
}

```

not the one in RangePositions.scala:

```

/** Position a tree.
 * This means: Set position of a node and position all its unpositioned children.
 */
override def atPos[T <: Tree](pos: Position)(tree: T): T =
  if (pos.isOpaqueRange) {
    if (!tree.isEmpty && tree.pos == NoPosition) {
      tree.setPos(pos)
      val children = tree.children
      if (children.nonEmpty) {
        if (children.tail.isEmpty) atPos(pos)(children.head)
        else setChildrenPos(pos, children)
      }
    }
    tree
  } else {
    super.atPos(pos)(tree)
  }

```



## 4.2 Accessing this

Another popular error is:

```
[treepos] Rule "Non-empty implies defined position" broken by:
non-synthetic tree [119], of type This with <nopos> position, located at [NoPosition]<nopos>
  immutable.this
In context:
non-synthetic tree [120], of type Select with opaque-range position, located at [66:70]bt4.scala
  immutable.this.List
```

### 4.2.1 After typer

```
Apply( // sym=method apply, tpe=List[Int], tpe.sym=class List, tpe.sym.owner=package immutable
  TypeApply( // sym=method apply, tpe=(xs: Int*)List[Int], tpe.sym=<none>
    Select( // sym=method apply, sym.owner=object List, sym.tpe=[A](xs: A*)List[A], tpe=[A](xs: A*)List[A], tpe
      Select( // sym=object List, sym.owner=package immutable, sym.tpe=object List, tpe=scala.collection.immuta
        This("immutable"), // sym=package immutable, sym.owner=package collection, sym.tpe=package scala.collec
          "List"),
        "apply"),
      List(
        TypeTree() // sym=class Int, tpe=Int, tpe.sym=class Int, tpe.sym.owner=package scala
      )
    ),
    List( // 2 arguments(s)
      Literal(Constant(3)),
      Literal(Constant(4))
    )
  )
)
```

### 4.2.2 After parser

```
Apply( // sym=<none>, tpe=null
  Ident("List"), // sym=<none>, sym.tpe=<notype>, tpe=null,
  List( // 2 arguments(s)
    Literal(Constant(3)),
    Literal(Constant(4))
  )
)
```

### 4.2.3 Where it gets transformed

Has to do with TypeApply in Typers.scala, details coming soon :-)