Finding the source code fragment that corresponds to a given AST-level type-reference

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

January 22th, 2011

Abstract

The `jdk2ikvm` transformation requires, among others:

- replacing some type-ref occurrences for others (e.g., `extends CharSequence` should be replaced with `extends java.lang.CharSequence.__Interface`).
- erasing type args in references to types that in JDK take type params.

Automatic API migration tools (such as `jdk2ikvm`) are regarded more useful if they preserve the layout of the original source code. For the case at hand, doing so involves tracing type references in ASTs back to their parse tree counterparts (thanks to range positions), as summarized in these notes. Although most examples are framed in the context of `jdk2ikvm`, the approach discussed here is generally applicable when refactoring, migrating, or pre-processing Scala source code.

Contents

1 Background

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

- *Learning and doing scalac transformations the easy way: via unparsing*[1]

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*[2]
- *A source-level, automatic API migration that preserves layout (a story of range positions)*[3]
- *Unparsing types the Scaladoc way*[4]

2 Motivation: "Upcastings" needed in `jdk2ikvm`

The following six concrete value types are to be mapped as shown:

```
val upcastings = Map(ThrowableClass -> "System.Exception",
                   jlExceptionClass -> "System.Exception",
                   jlErrorClass    -> "System.Exception",
                   ComparableClass -> "System.IComparable",
                   ObjectClass     -> "Object",
                   StringClass     -> "String")
```

In general, visit subnodes of the node kinds covered in Sec. 3.1, to get hold of their range positions. For example:

```
class B extends java.util.List[java.lang.String]

// should be translated to:

class B extends java.util.List[String]
```

This translation requires visiting sub-nodes of an `AppliedTypeTree` (Sec. 3.2).

2.1 Compiler-supported type maps (we want range positions instead)

The term "type mapping" is used in the context of the compiler as discussed for completeness in this subsection. It's not the technique we'll employ, because range positions are not copied over from parse trees to `tpe`'s.

An description of `TypeMap` can be found in §4.2 of `http://www.scala-lang.org/sid/5`.

A `TypeMap` has a `TypeMapTransformer` (Figure 1), which can be applied with `TypeMap.mapOver`:

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

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

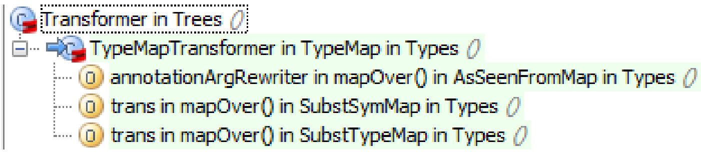
[3] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q1/ValidatePositions.pdf`

[4] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q1/TypesScaladocWay.pdf`

```
/** Map a tree that is part of an annotation argument.
 * If the tree cannot be mapped, then invoke giveup().
 * The default is to transform the tree with
 * TypeMapTransformer.
 */
def mapOver(tree: Tree, giveup: ()=>Nothing): Tree =
  (new TypeMapTransformer).transform(tree)

/** This transformer leaves the tree alone except to remap
 * its types. */
class TypeMapTransformer extends Transformer {
  override def transform(tree: Tree) = {
    val tree1 = super.transform(tree)
    val tpe1 = TypeMap.this(tree1.tpe)
    if ((tree eq tree1) && (tree.tpe eq tpe1))
      tree
    else
      tree1.shallowDuplicate.setType(tpe1)
  }
}
```

Also check,

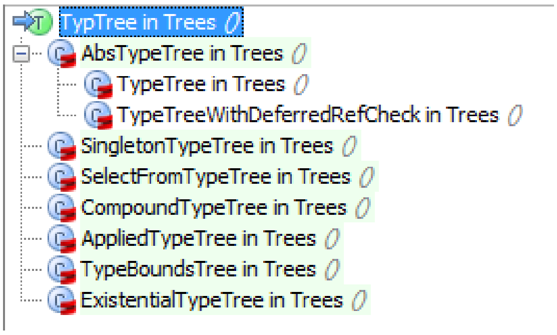


```
Transformer in Trees
  TypeMapTransformer in TypeMap in Types
    annotationArgRewriter in mapOver() in AsSeenFromMap in Types
    trans in mapOver() in SubstSymMap in Types
    trans in mapOver() in SubstTypeMap in Types
```

3 Type refs and the parser

3.1 AST nodes of interest

Looking at Parsers.scala:

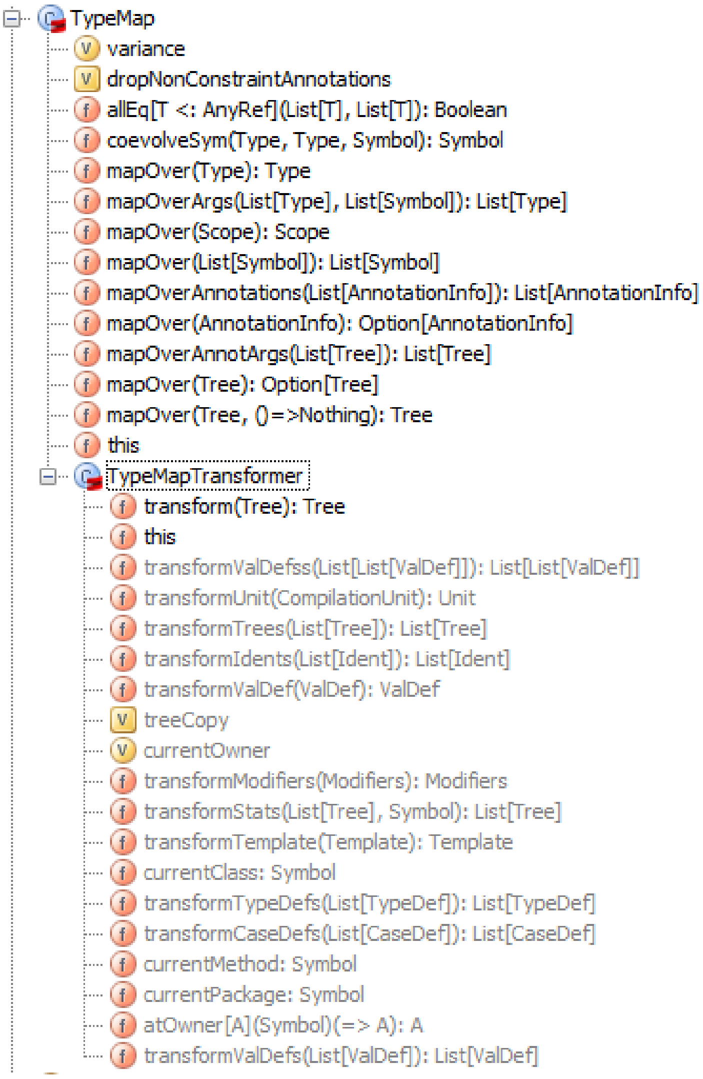


```
TypTree in Trees
  AbsTypeTree in Trees
    TypeTree in Trees
    TypeTreeWithDeferredRefCheck in Trees
  SingletonTypeTree in Trees
  SelectFromTypeTree in Trees
  CompoundTypeTree in Trees
  AppliedTypeTree in Trees
  TypeBoundsTree in Trees
  ExistentialTypeTree in Trees
```

```
  /** Singleton type, eliminated by RefCheck */
  case class SingletonTypeTree(ref: Tree) extends TypTree

  /** Type selection <qualifier> # <name>, eliminated by RefCheck */
  case class SelectFromTypeTree(qualifier: Tree, name: TypeName) extends TypTree with RefTree

  /** Intersection type <parent1> with ... with <parentN> { <decls> }, eliminated by RefCheck */
  case class CompoundTypeTree(templ: Template) extends TypTree
```



```
TypeMap
    variance
    dropNonConstraintAnnotations
    allEq[T <: AnyRef](List[T], List[T]): Boolean
    coevolveSym(Type, Type, Symbol): Symbol
    mapOver(Type): Type
    mapOverArgs(List[Type], List[Symbol]): List[Type]
    mapOver(Scope): Scope
    mapOver(List[Symbol]): List[Symbol]
    mapOverAnnotations(List[AnnotationInfo]): List[AnnotationInfo]
    mapOver(AnnotationInfo): Option[AnnotationInfo]
    mapOverAnnotArgs(List[Tree]): List[Tree]
    mapOver(Tree): Option[Tree]
    mapOver(Tree, ()=>Nothing): Tree
    this
    TypeMapTransformer
        transform(Tree): Tree
        this
        transformValDefss(List[List[ValDef]]): List[List[ValDef]]
        transformUnit(CompilationUnit): Unit
        transformTrees(List[Tree]): List[Tree]
        transformIdents(List[Ident]): List[Ident]
        transformValDef(ValDef): ValDef
        treeCopy
        currentOwner
        transformModifiers(Modifiers): Modifiers
        transformStats(List[Tree], Symbol): List[Tree]
        transformTemplate(Template): Template
        currentClass: Symbol
        transformTypeDefs(List[TypeDef]): List[TypeDef]
        transformCaseDefs(List[CaseDef]): List[CaseDef]
        currentMethod: Symbol
        currentPackage: Symbol
        atOwner[A](Symbol)(=> A): A
        transformValDefs(List[ValDef]): List[ValDef]
```

Figure 1: Mapping types, Sec. 2.1

```
/** Applied type <tpt> [ <args> ], eliminated by RefCheck */
case class AppliedTypeTree(tpt: Tree, args: List[Tree]) extends TypTree

case class TypeBoundsTree(lo: Tree, hi: Tree) extends TypTree

case class ExistentialTypeTree(tpt: Tree, whereClauses: List[Tree]) extends TypTree
```

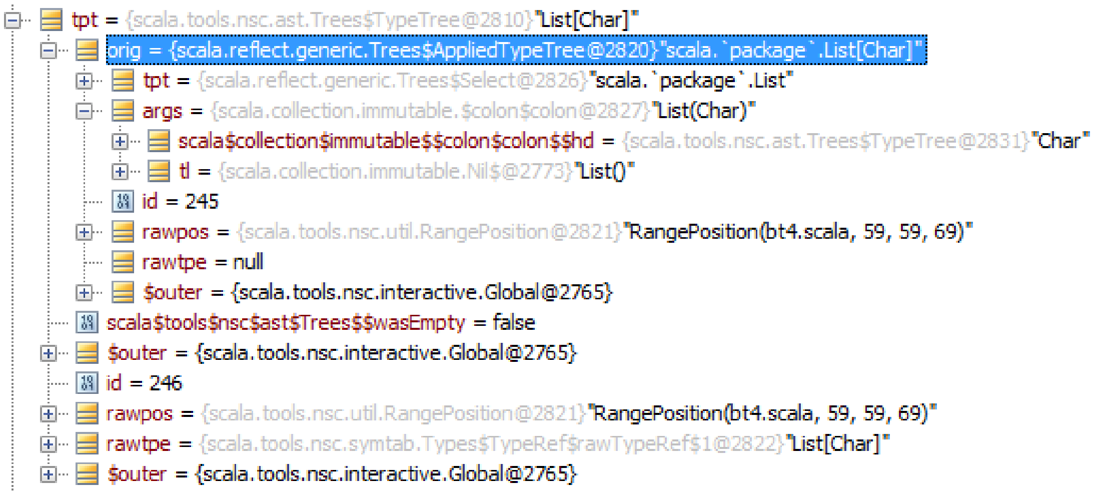
Other type refs are parsed as `Select` nodes, as the example in Sec. 3.2 shows.

3.2 Example: `AppliedTypeTree`

For example,

```
val charList : List[Char] = List('a','b','c')
```

results in:



```
tpt = {scala.tools.nsc.ast.Trees$TypeTree@2810}"List[Char]"
  orig = {scala.reflect.generic.Trees$AppliedTypeTree@2820}"scala.`package`.List[Char]"
    tpt = {scala.reflect.generic.Trees$Select@2826}"scala.`package`.List"
    args = {scala.collection.immutable.$colon$colon@2827}"List(Char)"
      scala$collection$immutable$$colon$colon$$hd = {scala.tools.nsc.ast.Trees$TypeTree@2831}"Char"
      tl = {scala.collection.immutable.Nil$@2773}"List()"
    id = 245
    rawpos = {scala.tools.nsc.util.RangePosition@2821}"RangePosition(bt4.scala, 59, 59, 69)"
    rawtpe = null
    $outer = {scala.tools.nsc.interactive.Global@2765}
  scala$tools$nsc$ast$Trees$$wasEmpty = false
  $outer = {scala.tools.nsc.interactive.Global@2765}
  id = 246
  rawpos = {scala.tools.nsc.util.RangePosition@2821}"RangePosition(bt4.scala, 59, 59, 69)"
  rawtpe = {scala.tools.nsc.symtab.Types$TypeRef$rawTypeRef$1@2822}"List[Char]"
  $outer = {scala.tools.nsc.interactive.Global@2765}
```

The screen capture is there just to show that the text fragment "`List`" is represented as a `Select`, while the parse node for "`Char`" now lives in its own `TypeTree` (different from that where the `AppliedTypeTree` lives).

3.3 Productions building those nodes

```
TODO: explore typ(), simpleType(), etc. in Parsers.scala
```

4 Parse tree counterpart to an AST type ref

"Standalone type references" may appear in:

1. `isInstanceOf[X]`, `asInstanceOf[X]`

2. parent in `extends` clause

3. tpt of a ValDef (local or template variable, or formal value param)
   this tpt is a TypeTree by the time we get our hands at it (i.e., after typer)

4. tpt of a DefDef

5. one of the args (i.e., a type param) in a TypeApply
   (a TypeApply *is* visited, it *is not* wrapped as orig in a TypeTree).

6. bound, be it lower or upper (context and view bounds are desugared to implicit arguments)

   - see TypeDef and TypeBounds
   - With or without parameter

7. Typed, e.g. (4 + 5): Int

8. a by-name flag can only appear in the type of a value param. Quoting from TreeInfo.scala:

   ```
   /** Is tpt a by-name parameter type? */
   def isByNameParamType(tpt: Tree) = tpt match {
     case TypeTree() =>
       definitions.isByNameParamType(tpt.tpe)
     case AppliedTypeTree(Select(_, tpnme.BYNAME_PARAM_CLASS_NAME), _) =>
       true
     case _  => false
   }
   ```

9. Only the last param in a value param list can have repeated type. Quoting from TreeInfo.scala:

   ```
   /** Is tpt of the form T* ? */
   def isRepeatedParamType(tpt: Tree) = tpt match {
     case TypeTree() =>
       definitions.isRepeatedParamType(tpt.tpe)
     case AppliedTypeTree(Select(_, tpnme.REPEATED_PARAM_CLASS_NAME), _) =>
       true
     case AppliedTypeTree(Select(_, tpnme.JAVA_REPEATED_PARAM_CLASS_NAME), _) =>
       true
     case _ => false
   }
   ```

10. import (but no rewriting in J2K operates on import clauses)

11. (a self-type falls under ValDef)

12. the ClassQualifier in

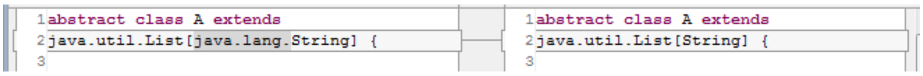
    ```
    [id '.'] 'super' [ClassQualifier] '.' id \\
    ```

13. not in a path designator (§6.4 in SLS) but in a type projection (with hash)

14. in a TypedPatten,
    be it as part of a catch clause in an enclosing Try
    or in a case clause in an enclosing Match

5 Implementations

5.1 Type mappings in `jdk2ikvm`



```
abstract class A extends
java.util.List[java.lang.String] {
```

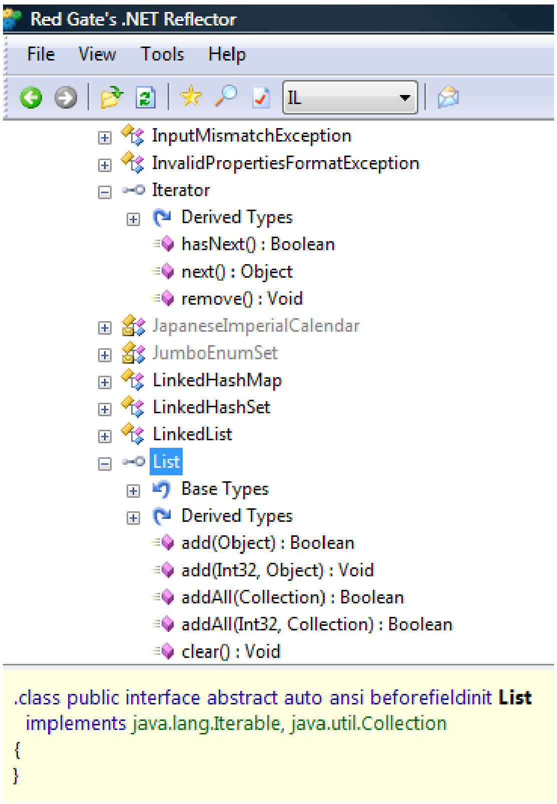
```
abstract class A extends
java.util.List[String] {
```

The big picture of `IKVMUpcaster` is shown in Figure 3. After determining whether the received node is a `TypeTree`, it goes on to visit its contents (which are skipped during normal traversals).

- A helper method (`shouldSubst(tree)`) informs whether the ranged `tree` (1) matches the substitution condition and (2) its corresponding text fragment hasn't been ruled out as an exception (e.g., occurrences of "`AnyRef`" are not rewritten to "`Object`").
- Another helper method (`trySubst`) actually performs the rewriting, if instructed to do so by `replSourceFragmentForASTType`, as discussed below.

```
/** precond: sourceFrag must have a TypeTree as dominator over the "AST node containment" hierarchy,
 *  ie. sourceFrag should *not* be visitable by a Tree traverser
 *  (which skips the 'orig' node contained in a TypeTree, and its contained nodes, and so on. */
def replSourceFragmentForASTType(sourceFrag: Tree) {
  sourceFrag match {
    case SingletonTypeTree(ref)          => replSourceFragmentForASTType(ref)
    case SelectFromTypeTree(qualifier, name) => replSourceFragmentForASTType(qualifier)
    case CompoundTypeTree(templ)         => replSourceFragmentForASTType(templ)
    case AppliedTypeTree(tpt, args) =>
      replSourceFragmentForASTType(tpt)
      for (arg <- args) { replSourceFragmentForASTType(arg) }
    case TypeBoundsTree(lo, hi) =>
      replSourceFragmentForASTType(lo)
      replSourceFragmentForASTType(hi)
    case ExistentialTypeTree(tpt, whereClauses) =>
      replSourceFragmentForASTType(tpt)
      for (wc <- whereClauses) { replSourceFragmentForASTType(wc) }
    case tt : TypeTree if (tt.original != null) =>
      replSourceFragmentForASTType(tt.original)
    case Annotated(annot, arg) =>
      // TODO children not visited on purpose although I would like to know more about them
      // warning(sourceFrag.pos, "Annotated(annot, arg) not visited in replSourceFragmentForASTType")
    case other if (shouldSubst(other)) => trySubst(other)
    case _ =>
      /* children of this node won't be visited by IKVMUpcaster.this.collectPatches,
       * because this node lives inside a TypeTree to start with.
       * We shouldn't need to visit them anyway, but the warning is there to help discover overlooked cases. */
      val substCandidates = (new CollectRangedNodes apply sourceFrag) filter (rn => shouldSubst(rn))
      if(substCandidates.nonEmpty) {
        for (rn <- substCandidates) {
          warning(rn.pos, "ranged node contained in a TypeTree not substituted by replSourceFragmentForASTType")
        }
      }
  }
}
```

TODO: rewrite inside type annotations, see http://www.scala-lang.org/sid/5

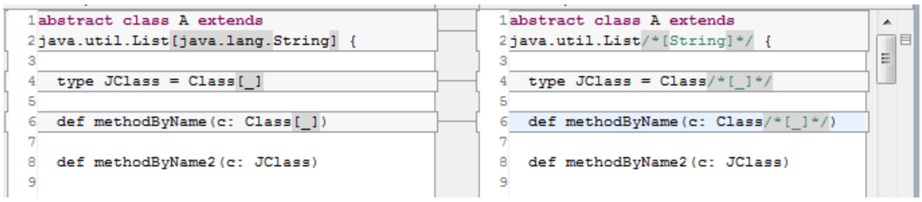


```
InputMismatchException
InvalidPropertiesFormatException
Iterator
    Derived Types
    hasNext() : Boolean
    next() : Object
    remove() : Void
JapaneseImperialCalendar
JumboEnumSet
LinkedHashMap
LinkedHashSet
LinkedList
List
    Base Types
    Derived Types
    add(Object) : Boolean
    add(Int32, Object) : Void
    addAll(Collection) : Boolean
    addAll(Int32, Collection) : Boolean
    clear() : Void
```

```
.class public interface abstract auto ansi beforefieldinit List
  implements java.lang.Iterable, java.util.Collection
{
}
```

Figure 2: Sec. 5.2

5.2 Type erasure in `jdk2ikvm`



```
abstract class A extends
java.util.List[java.lang.String] {

  type JClass = Class[_]

  def methodByName(c: Class[_])

  def methodByName2(c: JClass)

```

```
abstract class A extends
java.util.List/*[String]*/ {

  type JClass = Class/*[_]*/

  def methodByName(c: Class/*[_]*/)

  def methodByName2(c: JClass)

```

It's a fact of life that the IKVM library does away with the type params and arguments of JDK counterparts[5], as shown for example for `j.u.Iterator` and `j.u.List` in Figure 2. As usual, `ikvmc` (the IKVM compiler) takes care of wall-papering over this.

As a result, `jdk2ikvm` will similarly have to wall-paper, this time at the level of Scala source code, by deleting type args (and adding downcasts). A source file that showcases most of the required rewritings is `JavaConversions.scala`. Without those rewritings, when compiling `forMSIL` against IKVM's `.dll` we get errors like:

[5] `http://stackoverflow.com/questions/1477038/doesnt-ikvm-net-support-generics-type-parameters`

```
JavaConversions.scala:71: error: java.util.Iterator does not take type parameters
  implicit def asJavaIterator[A](i : Iterator[A]): ju.Iterator[A] = i match {
                                                  ^
```

5.3 Deconstructing types

As with the rewriting to map types, we have to traverse type constituents. However in this case a `typeSymbol` won't be enough as applicability condition for a rewriting, we also need a `tpe`. Additionally, we need a tandem decomposition of `TypTree` and `Type` ("in tandem" because we don't want to lose track of the `RangePosition` for a given `Type` value).

In all cases, the entry point is:

```
eraseTypeArgs(sourceFrag: Tree, sourceFragTpe: Type)
```

Particular cases of "in tandem deconstruction" are listed below:

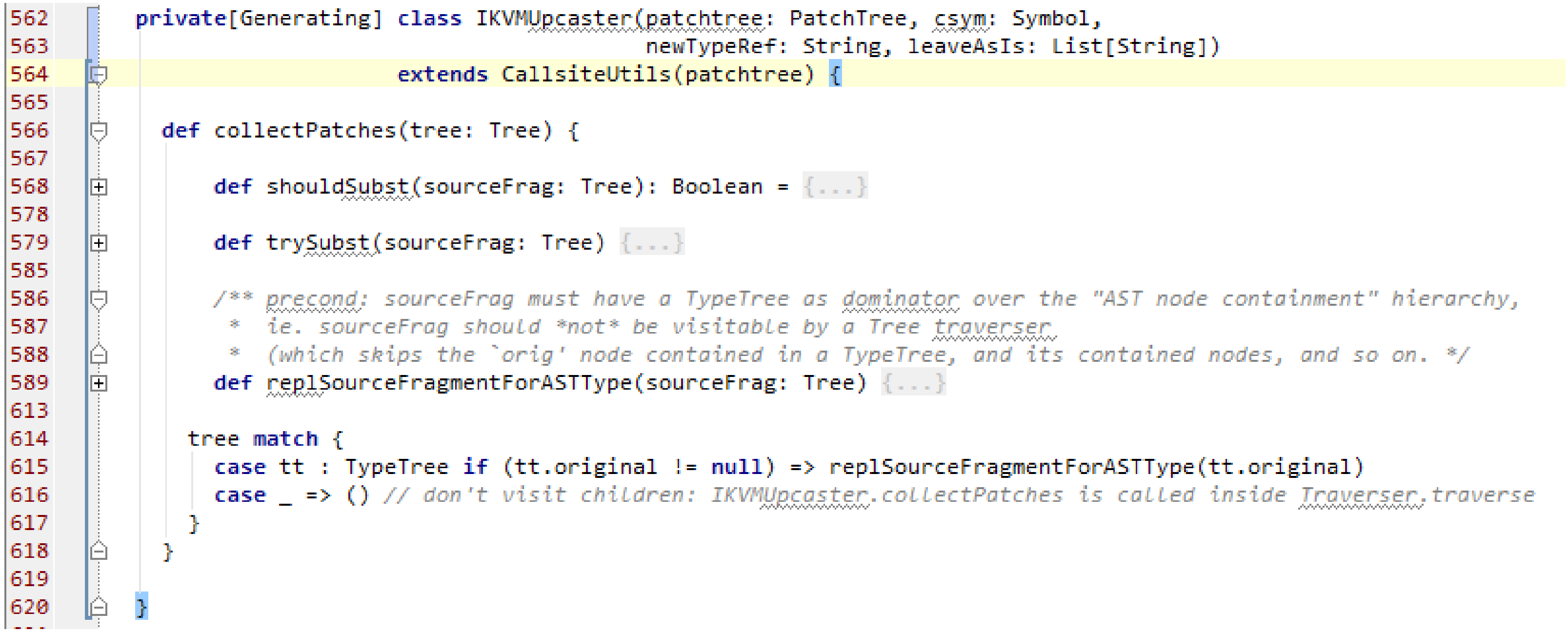
- Type application:

```
case att @ AppliedTypeTree(tpt, args) =>
  if(shouldErase(sourceFragTpe)) erase(att)
  else {
    eraseTypeArgs(tpt, sourceFragTpe.typeConstructor)
    for ( (arg, tpe) <- args zip sourceFragTpe.typeArgs) { eraseTypeArgs(arg, tpe) }
  }
```

- Type bounds:

```
case TypeBoundsTree(lo, hi) =>
  eraseTypeArgs(lo, sourceFragTpe.bounds.lo)
  eraseTypeArgs(hi, sourceFragTpe.bounds.hi)
```

- etc.



```scala
private[Generating] class IKVMUpcaster(patchtree: PatchTree, csym: Symbol,
                                       newTypeRef: String, leaveAsIs: List[String])
                    extends CallsiteUtils(patchtree) {

  def collectPatches(tree: Tree) {

    def shouldSubst(sourceFrag: Tree): Boolean = {...}

    def trySubst(sourceFrag: Tree) {...}

    /** precond: sourceFrag must have a TypeTree as dominator over the "AST node containment" hierarchy,
     *  ie. sourceFrag should *not* be visitable by a Tree traverser
     *  (which skips the `orig' node contained in a TypeTree, and its contained nodes, and so on. */
    def replSourceFragmentForASTType(sourceFrag: Tree) {...}

   tree match {
     case tt : TypeTree if (tt.original != null) => replSourceFragmentForASTType(tt.original)
     case _ => () // don't visit children: IKVMUpcaster.collectPatches is called inside Traverser.traverse
   }
  }

}
```

Figure 3: Sec. 5.1