Learning and doing `scalac` transformations the easy way: via unparsing

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

December 16th, 2010

Abstract

The Scala compiler supports a plugin architecture that allows third-parties to perform custom program transformations, much like the compiler itself transforms ASTs. In particular, it's possible to have a plugin whose output are `.scala` source files (for example, that's what the `jdk2ikvm` converter does). These plugins rely on an *unparsing* component, to obtain a more detailed yet easily navigable view of the structure of ASTs. More detailed because some code expansions are made explicit (expansions that help explain performance, e.g., invocations of extra methods of structural types). And easily navigable because the resulting sources can be explored with a Scala IDE (and compiled again, if so wished, although there's no difference as compared to compiling the original sources: we're always talking about so called *roundtripping* unparsing).

These notes describe the `scala.tools.unparse` component, giving examples of usage scenarios for this versatile tool, including *AST-aware pre-processors* for Scala (for example, API migration tools, style checking tools, etc.) Compared to compiler plugins that directly deliver ASTs to the next compilation phase, a pre-processor has both pros and cons: compilation rounds are slower (because input programs are typed twice: before and after pre-processing). On the plus side, the ASTs prepared by a pre-processor:

- need not contain type symbols, which will be added by the compiler on its second run; and
- are not constrained to the Scala subset that subsequent phases understand (e.g., ASTs delivered to phases after `explicitouter` should not contain `match` expressions).

The above suggests that pre-processors can serve as proofs-of-concept, later evolving to compiler plugins in case demand justifies their development.

The `unparse` sources can be found at `http://lampsvn.epfl.ch/trac/scala/browser/scala-experimental/trunk/jdk2ikvm/src/scala/tools/unparse/`

Contents

1 Getting Started

1.1 How to build

Same instructions as for building `jdk2ikvm`:

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` as follows

   ```
   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

In order to unparse a bunch of Scala source files, run `scalac` with the `jdk2ikvm` compiler plugin with command-line options:

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

Summing up, to the usual options for `jdk2ikvm`, just add `-P:jdk2ikvm:just-unparse`

1.3 Testing unparsing by roundtripping

By running with the '`-P:jdk2ikvm:just-unparse`' option, the resulting tool can pretty-print arbitrary Scala sources. What is that good for? This setting is perfect to demonstrate that `scala.tools.unparse` is behavior-preserving:

1. unparse all files found under `compiler` and `library` in scala trunk

2. compile the resulting pretty-printed files with `build.xml` from scala trunk

3. *all the tests in* `build.xml` can be used to confirm that pretty-printing was behavior-preserving.

1.4 Undoing synthetic artefacts added to ASTs till `typer`

These transformations comprise:

- those in `Parsers.scala`, including:
  - desugaring of `for` comprehensions into `filter`, `map`, and `flatMap`.
  - desugaring of `for` loops into `foreach`.
  - recovering class definition syntax, Sec. 4.1
- those in `SyntheticMethods.scala`

Consider this: if those transformations were not reversible, i.e., if the pre-image for a transformed tree couldn't be recovered, then two or more programs with different behavior would map to the same transformed program. That's not the case, ergo we can recover the structure of the input sources (not its layout nor its before-desugaring representation, but what matters for compiling again: a program with the same structure and behavior as that given by the source file).

2 When unparsed output looks different from your input ... it's on purpose

2.1 Inlined XML

For example, the following:

```
  val b = <book>Zen and the art of yo-yo</book>
```

is unparsed into:

```
  val b : scala.xml.Elem = new scala.xml.Elem((null), ("book"), scala.xml.Null, scala.$scope,
            ({
              val $buf : scala.xml.NodeBuffer = new scala.xml.NodeBuffer()
              $buf.&+(new scala.xml.Text(("Zen and the art of yo-yo")))
              $buf
             }: scala.xml.Node))
```

TODO: Upon trying to compile the above, we get an error (missing implicit conversion?)

```
type mismatch;
 found   : scala.xml.NodeBuffer
 required: scala.xml.Node
                         $buf
                         ^
```

2.2 Long form for existential types

The unparser prints:

```
Class[_$1] forSome { type _$1 }
```

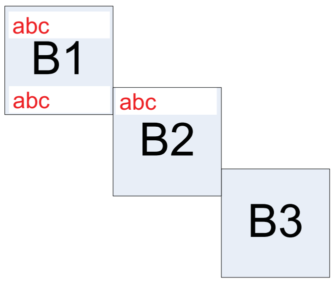
rather than the semantically equivalent `Class[_]`. This is on purpose, so as to allow greping for existentials. The chosen output syntax does however elide type bounds when they are default:

```
/**
 *   [ ">:" TypeTree ]  [ "<:" TypeTree ]
 */
override def TypeBoundsTree(tree: TypeBoundsTree, lo: Tile, hi: Tile): Tile = {
  def skipDefault(t: Tree, symForDefault: Symbol, longForm: Tile) = {
    if (t.symbol eq symForDefault) EmptyTile
    else longForm
  }
  val loPart = skipDefault(tree.lo, definitions.NothingClass, ">: " ~ lo ~ Blank)
  val hiPart = skipDefault(tree.hi, definitions.AnyClass, "<: " ~ hi ~ Blank)
  loPart ~ hiPart
}
```

3 Under the hood

3.1 Tiling of Text Blocks

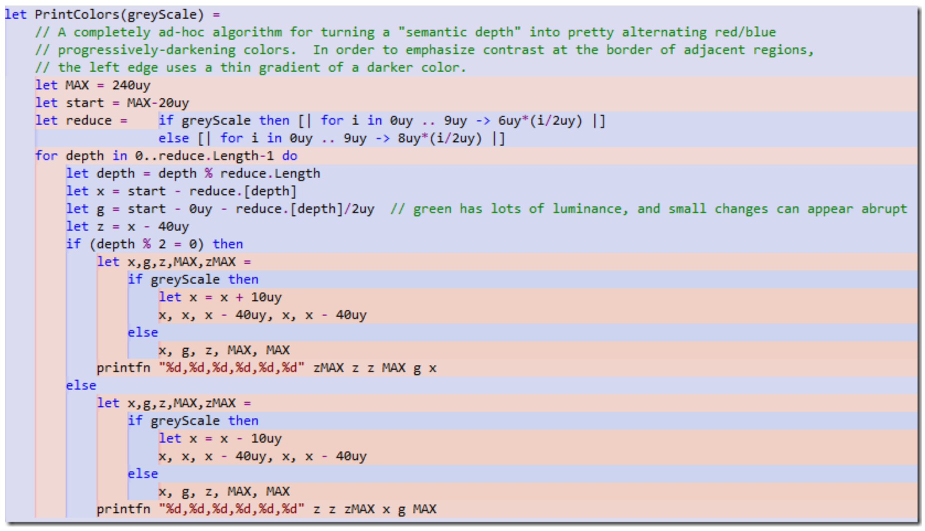
Rather than writing a full-fledged pretty-printer (go to `scala-refactoring` for that) our unparser adopts a text blocks metaphor (ie., rectangular text regions that can be juxtaposed). Allowing for arbitrary layout of text blocks would result in non-parseable output, a situation we avoid when serializing rows of text blocks (where blocks are potentially rectangular, i.e., with elements with height > 1). In the following example, blocks B1 to B3 are serialized as depicted in the diagram:



The utilities in `TextTiling.scala` allow obtaining the output show below for the following snippet:

```
  TilePrinter.newConsoleTilePrinter print tileF

  def vTunnel(c: Int) = {
    val str = c.toString + c.toString + c.toString
    Flow(Vert, Array.fill(3)(StrTile(str)).toList)
  }
  def tileF = {
    val t23 = row(vTunnel(2), vTunnel(3))
    val t234 = column(t23, vTunnel(4))
    row(vTunnel(1), t234)
  }
```



```
let PrintColors(greyScale) =
    // A completely ad-hoc algorithm for turning a "semantic depth" into pretty alternating red/blue
    // progressively-darkening colors.  In order to emphasize contrast at the border of adjacent regions,
    // the left edge uses a thin gradient of a darker color.
    let MAX = 240uy
    let start = MAX-20uy
    let reduce =    if greyScale then [| for i in 0uy .. 9uy -> 6uy*(i/2uy) |]
                    else [| for i in 0uy .. 9uy -> 8uy*(i/2uy) |]
    for depth in 0..reduce.Length-1 do
        let depth = depth % reduce.Length
        let x = start - reduce.[depth]
        let g = start - 0uy - reduce.[depth]/2uy  // green has lots of luminance, and small changes can appear abrupt
        let z = x - 40uy
        if (depth % 2 = 0) then
            let x,g,z,MAX,zMAX =
                if greyScale then
                    let x = x + 10uy
                    x, x, x - 40uy, x, x - 40uy
                else
                    x, g, z, MAX, MAX
            printfn "%d,%d,%d,%d,%d,%d" zMAX z z MAX g x
        else
            let x,g,z,MAX,zMAX =
                if greyScale then
                    let x = x - 10uy
                    x, x, x - 40uy, x, x - 40uy
                else
                    x, g, z, MAX, MAX
            printfn "%d,%d,%d,%d,%d,%d" z z zMAX x g MAX
```

Figure 1: FSharp lexical nesting colorizer, `http://lorgonblog.wordpress.com/2010/11/18/f-source-code-structural-colorizer-available/`

```
// blanks shown as dots for better appreciation
111
111
111222
...222
...222333
......333
......333
...444
...444
...444
```

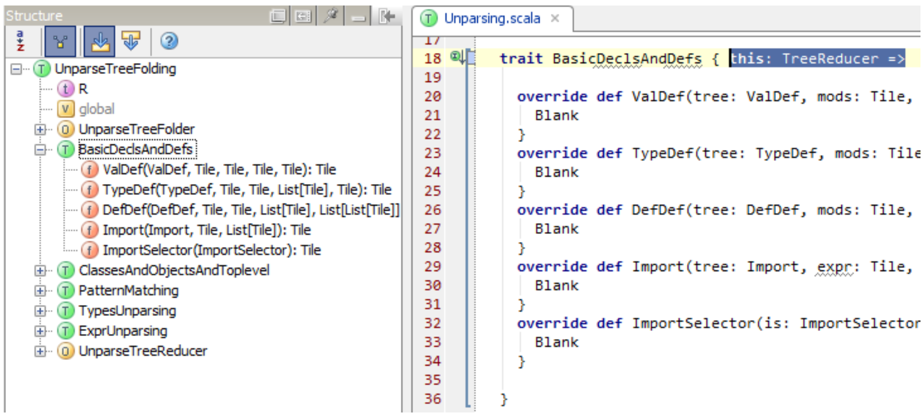
You might want to explore another variation of text tiling, this time to color-code lexical scoping (Scala IDEs like IntelliJ show that on the vertical bar, but colors as in Figure 1 provide for a more dramatic effect).

3.2 *Visiting* (yes, visiting) Scala ASTs

We want to fold a tree into a single value (a text block) and for that it's handy to have a traverser that hands over the results of folding children (post-order traversal) as realized in `TreeFolding.scala`. The post-order processing part if performed by a so called `TreeReducer`, which for a `PackageDef` is invoked as follows:

```
def PackageDef(tree: PackageDef, pid: RefTree, stats: List[Tree]): R =
  treeRedu.PackageDef(
    tree, transform(pid),
    transformTrees(stats)
  )
```

where `transform()` and `transformTrees()` are hosted in `TreeFolder`, and the signature of the reducer for `PackageDefs` is:



```
UnparseTreeFolding
    R
    global
    UnparseTreeFolder
    BasicDeclsAndDefs
        ValDef(ValDef, Tile, Tile, Tile, Tile): Tile
        TypeDef(TypeDef, Tile, Tile, List[Tile], Tile): Tile
        DefDef(DefDef, Tile, Tile, List[Tile], List[List[Tile]]
        Import(Import, Tile, List[Tile]): Tile
        ImportSelector(ImportSelector): Tile
    ClassesAndObjectsAndToplevel
    PatternMatching
    TypesUnparsing
    ExprUnparsing
    UnparseTreeReducer
```

```
trait BasicDeclsAndDefs { this: TreeReducer =>

  override def ValDef(tree: ValDef, mods: Tile,
    Blank
  }
  override def TypeDef(tree: TypeDef, mods: Tile
    Blank
  }
  override def DefDef(tree: DefDef, mods: Tile,
    Blank
  }
  override def Import(tree: Import, expr: Tile,
    Blank
  }
  override def ImportSelector(is: ImportSelector
    Blank
  }

}
```

Figure 2: Code organization of the unparser, Sec. 3.1

```
def PackageDef(tree: PackageDef, pid: R, stats: List[R]): R
```

where `R` is an abstract type of your choice.

In case the default visit order established by `TreeFolder` is not deemed appropriate in some situation, the methods for the tree nodes in question can be overridden (such as method `PackageDef` in `TreeFolder`).

In terms of the `Transformer` idiom of `scala.tools.nsc.ast.Trees`, we would override instead the `transform()` method (which switches based on the node's shape) and call the super version for other cases than the ones of interest. For our purposes, `Transformer` is too restrictive (`transform` returns a `Tree`, we want to return a text block).

But visitors are a matter of taste! (I'll keep using mine). Besides, the resulting code organization allows for easy IDE navigation (Figure 2).

Other perspectives on visiting trees:

- The Visitor Pattern as a Reusable, Generic, Type-Safe Component, `http://ropas.snu.ac.kr/~bruno/papers/VisitorComponent.pdf`
- Strategic Programming, `http://homepages.cwi.nl/~ralf/eosp/`

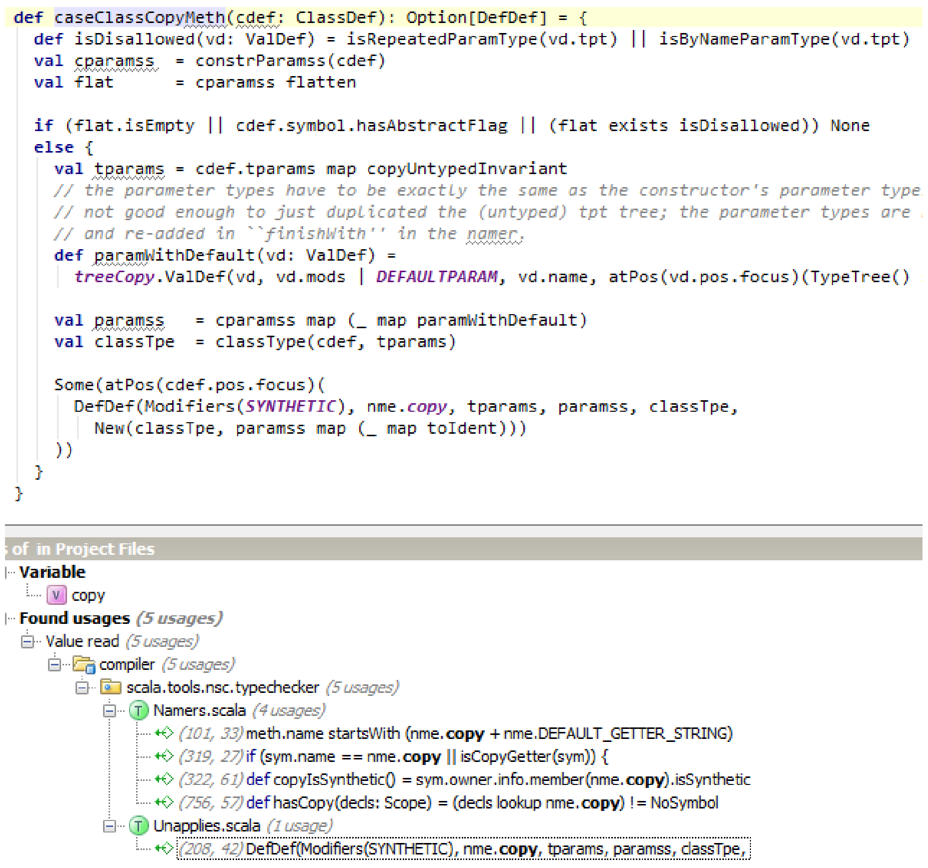
3.3 Unparsing the `copy` methods of case classes

The code in charge of emitting those methods (`caseClassCopyMeth(cdef: ClassDef)`) lives in `Unapplies.scala` as shown in Figure 3.

4 Unparsing recipe

The syntax to emit for an AST node may depend on its parent nodes, as follows:

- Some AST shapes are reused for different purposes (eg, a `ValDef` can stand for a template member, block local, or method value param). In these cases, their serialization usually depends on the context where their occurrences occur (in the `ValDef` example, the unparsing for both template member and block local are the same, unlike for method value param).



```
def caseClassCopyMeth(cdef: ClassDef): Option[DefDef] = {
  def isDisallowed(vd: ValDef) = isRepeatedParamType(vd.tpt) || isByNameParamType(vd.tpt)
  val cparamss  = constrParamss(cdef)
  val flat      = cparamss flatten

  if (flat.isEmpty || cdef.symbol.hasAbstractFlag || (flat exists isDisallowed)) None
  else {
    val tparams = cdef.tparams map copyUntypedInvariant
    // the parameter types have to be exactly the same as the constructor's parameter type
    // not good enough to just duplicated the (untyped) tpt tree; the parameter types are
    // and re-added in ``finishWith'' in the namer.
    def paramWithDefault(vd: ValDef) =
      treeCopy.ValDef(vd, vd.mods | DEFAULTPARAM, vd.name, atPos(vd.pos.focus)(TypeTree()

    val paramss   = cparamss map (_ map paramWithDefault)
    val classTpe  = classType(cdef, tparams)

    Some(atPos(cdef.pos.focus)(
      DefDef(Modifiers(SYNTHETIC), nme.copy, tparams, paramss, classTpe,
        New(classTpe, paramss map (_ map toIdent)))
    ))
  }
}
```

```
s of  in Project Files
Variable
  copy
Found usages (5 usages)
  Value read (5 usages)
    compiler (5 usages)
      scala.tools.nsc.typechecker (5 usages)
        Namers.scala (4 usages)
          (101, 33) meth.name startsWith (nme.copy + nme.DEFAULT_GETTER_STRING)
          (319, 27) if (sym.name == nme.copy || isCopyGetter(sym)) {
          (322, 61) def copyIsSynthetic() = sym.owner.info.member(nme.copy).isSynthetic
          (756, 57) def hasCopy(decls: Scope) = (decls lookup nme.copy) != NoSymbol
        Unapplies.scala (1 usage)
          (208, 42) DefDef(Modifiers(SYNTHETIC), nme.copy, tparams, paramss, classTpe,
```

Figure 3: Unparsing the copy methods of case classes (Sec. ??)

- Otherwise reversing the grammar production is context-independent.

To handle context-dependency the pattern discussed in Sec. 3.2 was adopted. The rest of this section highlights some "syntax recovery recipes", usually by describing first the corresponding "syntax desugaring recipe" previously applied by the compiler.

4.1 Recovering class definition syntax

The parser performs the following desugaring (by invoking method `Template` in `Trees.scala`)

```
/** Generates a template with constructor corresponding to
 *
 *  constrmods (vparams1_) ... (vparams_n) preSuper { presupers }
 *  extends superclass(args_1) ... (args_n) with mixins { self => body }
 *
 *  This gets translated to
 *
 *  extends superclass with mixins { self =>
 *    presupers' // presupers without rhs
 *    vparamss  // abstract fields corresponding to value parameters
 *    def <init>(vparamss) {
 *      presupers
 *      super.<init>(args)
 *    }
 *    body
 *  }
 */
```

4.1.1 Params to the class' main constructor and arguments to the superclass' constructor (*supercall arguments*)

Of the arguments to

```
def Template(parents: List[Tree], self: ValDef, constrMods: Modifiers,
             vparamss: List[List[ValDef]],
             argss: List[List[Tree]],
             body: List[Tree], superPos: Position)
```

the following need to be recovered during unparsing:

```
vparamss: List[List[ValDef]] // param lists for the main constructor
argss: List[List[Tree]]    // arg  lists for the first parent
```

The relevant portions of the desugaring are:

1. a `DefDef` representing the main constructor is always added to the body of a non-trait class, with name `nme.CONSTRUCTOR`. This constructor comes before any other because of the way the `Template` instance is built (it's the only element in the `constrs` list below):

   ```
   Template(parents, self, gvdefs ::: vparamss2.flatten ::: constrs ::: etdefs ::: rest)
   ```

   This `DefDef`'s value params mirror the `vparamss` of the concrete syntax, but:

(a) each param's original modifiers have been ANDed as shown below, and

```
var vparamss1 =
  vparamss map
  (vps => vps.map { vd =>
    atPos(vd.pos.focus) {
      ValDef(
        Modifiers(vd.mods.flags &
                    (IMPLICIT | DEFAULTPARAM | BYNAMEPARAM) | PARAM | PARAMACCESSOR)
           withAnnotations vd.mods.annotations,
        vd.name,
        vd.tpt.duplicate,
        vd.rhs.duplicate
     ) // closes ValDef
    } // closes atPos
   } // closes vd =>
  ) // closes vparamss map
```

(b) an empty list may have been pre-pended to list of params to recover. Please notice that the resulting `DefDef` is the only place from which the original args for the first parent can be recovered (`argss` below):

```
// convert (implicit ... ) to ()(implicit ... ) if its the only parameter section
if (vparamss1.isEmpty || !vparamss1.head.isEmpty && vparamss1.head.head.mods.isImplicit)
  vparamss1 = List() :: vparamss1;
val superRef: Tree = atPos(superPos) {
  Select(Super(tpnme.EMPTY, tpnme.EMPTY), nme.CONSTRUCTOR)
}
val superCall = (superRef /: argss) (Apply)
List(
  atPos(wrappingPos(superPos, lvdefs ::: argss.flatten)) (
    DefDef(constrMods,
            nme.CONSTRUCTOR,
            List(),
            vparamss1,
            TypeTree(),
            Block( lvdefs ::: List(superCall),
                    Literal( () )
                 )
           )
   )
)
```

2. traits don't have constructor params, so there's nothing to recover in this case. However, similarly to the case above, a `DefDef` may be added for a trait class, but only if some member `!treeInfo.isInterfaceMember`. In this case, the `DefDef`'s name is `nme.MIXIN_CONSTRUCTOR`, its type params list is empty, and its value params list is `List(List())`

There's no shortcut to the above. One might be tempted to recover the main constructor's params by checking a `ValDef`'s `mods.isCaseAccessor || mods.isParamAccessor` after noticing:

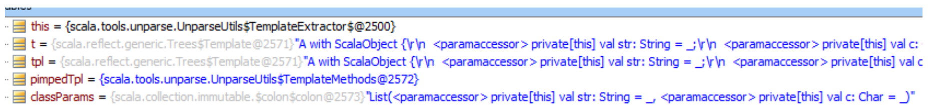
```
// vparamss2 are used as field definitions for the class. remove defaults
val vparamss2 = vparamss map (vps => vps map { vd =>
  treeCopy.ValDef(vd, vd.mods &~ DEFAULTPARAM, vd.name, vd.tpt, EmptyTree)
})
```

```
Template(parents, self, gvdefs ::: vparamss2.flatten ::: constrs ::: etdefs ::: rest)
```

but the flattening means that the boundaries of multiple param-lists can't be recovered anymore. For example,

```
class C(str: String)(c: Char) extends A
```

results in the flat list `classParams` shown below:



```
this = {scala.tools.unparse.UnparseUtils$TemplateExtractor$@2500}
t = {scala.reflect.generic.Trees$Template@2571}"A with ScalaObject {\r\n  <paramaccessor> private[this] val str: String = _;\r\n  <paramaccessor> private[this] val c:
tpl = {scala.reflect.generic.Trees$Template@2571}"A with ScalaObject {\r\n  <paramaccessor> private[this] val str: String = _;\r\n  <paramaccessor> private[this] val c
pimpedTpl = {scala.tools.unparse.UnparseUtils$TemplateMethods@2572}
classParams = {scala.collection.immutable.$colon$colon@2573}"List(<paramaccessor> private[this] val str: String = _, <paramaccessor> private[this] val c: Char = _)"
```

4.1.2 Shortcut to fabricate `supercallArgs`

An AST-aware pre-processor (Sec. 5) need not follow all steps of preparing an AST *as the compiler does* for super-call arguments. The following Concrete Syntax Tree node (borrowed from the Scala Refactoring[1] library) can be used instead.

```
  /**
   * The call to the super constructor in a class:
   * class A(i: Int) extends B(i)
   *                         ^^^^
   */
  case class SuperConstructorCall(classId: Ident, args: List[global.Tree]) extends Tree
```

There's no similar fake node for "main constructor parameters", because piecing them together AST-wise is not as cumbersome (relatively speaking).

4.1.3 Early Defs

We want to recover `lvdefs` and `etdefs` below:

```
// method Template in Trees.scala returns

Template(parents, self, gvdefs ::: vparamss2.flatten ::: constrs ::: etdefs ::: rest)
```

which together make up the "early defs". `lvdefs` and `etdefs` result from:

```
val (edefs, rest) = body span treeInfo.isEarlyDef /*- isEarlyDef includes both ValDef's and TypeDef's */
val (evdefs, etdefs /*- <-- */ ) = edefs partition treeInfo.isEarlyValDef /*- isEarlyValDef just ValDef's */
val (lvdefs, gvdefs) = evdefs map {
  case vdef @ ValDef(mods, name, tpt, rhs) =>

    val fld = treeCopy.ValDef( /*- each fld will become a ValDef in gvdefs */
      vdef.duplicate, mods, name,
      atPos(vdef.pos.focus) { TypeTree() setOriginal tpt setPos tpt.pos.focus }, // atPos in case
      EmptyTree)

    val local = treeCopy.ValDef(vdef, Modifiers(PRESUPER), name, tpt, rhs) /*- but initializers (rhs)
                                                                             are to be found here. */
    (local, fld)
} unzip
```

[1]`http://scala-refactoring.org/`

Listing 1: Recovering loops (Sec. 4.2)

```
trait JumpsInterceptors { this: TreeFolder =>

  /**
   * (1) "while" "(" cond ")" "{" stats "}"
   *
   * (2) "do" "{" stats "}" "while" "(" cond ")"
   */
  override def LabelDef(tree: LabelDef, name: Name, params: List[Ident], rhs: Tree): Tile = {
   tree match {

     // while (cond) body ==> LabelDef($L, List(), if (cond) { body; L$() } else ())
     case laDef @ LabelDef(name, params, If(cond, thenp @ Block(stats, laCall), _)) =>
       // FYI laCall @ Apply(laCallFun : Ident, Nil)
       // For compiler trees, assert(laDef.name eq laCallFun.name) holds.
       // But we don't check that because we also want to unparse trees
       // "with just enough concrete-syntax structure in them"
       row("while ", parens(xform(cond)), blockify(transformTrees(stats)))

     // do body while (cond) ==> LabelDef($L, List(), body; if (cond) L$() else ())
     case label @ LabelDef(_, _, Block(stats, If(cond, _, _))) =>
       row("do ", blockify(transformTrees(stats)), " while ", parens(xform(cond)))

    }
  }

}
```

TODO

4.2 Recovering loops

This is relatively well known (Listing 1) but a comment also applies when unparsing other AST nodes:

> *For trees produced by the compiler,* `assert(. . .)` *always holds. But we don't check that because we also want to unparse trees "with just enough concrete-syntax structure in them", as for example those produced by API migration tools.*

4.3 Recovering patterns in `Match` expressions

This turns out to be easier than expected (Figure 4) thanks to utility methods borrowed from `Patterns.scala`.

4.4 Unparsing lambdas

Another merry case where the source comments are as good as documentation. For example,

```
List(1, 2, 3) collect { case i: Int => i }
```

is unparsed to

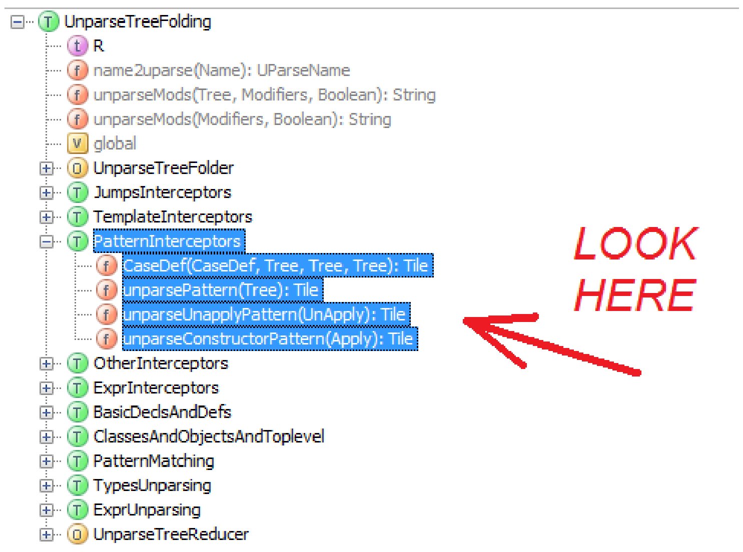

UnparseTreeFolding
R
name2uparse(Name): UParseName
unparseMods(Tree, Modifiers, Boolean): String
unparseMods(Modifiers, Boolean): String
global
UnparseTreeFolder
JumpsInterceptors
TemplateInterceptors
PatternInterceptors
CaseDef(CaseDef, Tree, Tree, Tree): Tile
unparsePattern(Tree): Tile
unparseUnapplyPattern(UnApply): Tile
unparseConstructorPattern(Apply): Tile
OtherInterceptors
ExprInterceptors
BasicDeclsAndDefs
ClassesAndObjectsAndToplevel
PatternMatching
TypesUnparsing
ExprUnparsing
UnparseTreeReducer
LOOK
HERE

Figure 4: Unparsing patterns, Sec. 4.3

```
scala.collection.immutable.List.apply[Int]((1), (2), (3)).collect[Int, Any]
( (x0$1 : Int) => x0$1 match { case i : scala.Int => i } )
(scala.collection.immutable.List.canBuildFrom[Int])
```

where

```
( (x0$1 : Int) => x0$1 match { case i : scala.Int => i } )
```

results from unparsing the `tree: Function` argument received by:

```
override def Function(tree: Function, vparams: List[ValDef], body: Tree): Tile = {
  val vparamsList = unparseFormalParams(vparams)
  forceParens(vparamsList ~ " => " ~ xform(body))
}
```

5 Beyond plain unparsing: pre-processing (resolved, typed) ASTs

Out of the box, the unparser can be used to help in understanding what phases like `specialize` do on input ASTs. With additional work, an *AST-aware preprocessor* can be built (`jdk2ikvm` serves as blueprint). There are many application areas for such processors:

- code expansion as favored by runtime-checking techniques,
  - *Temporal JML*, `http://www.eecs.ucf.edu/~fhussain/papers/temporaljmlc.pdf`
  - In general, the kind of rewriting that *Code Contracts* tools perform on bytecode, can be done by a Scala pre-processor.

`ccrewrite`, a tool for generating runtime checks from Code Contracts, `http://research.microsoft.com/en-us/projects/contracts/`

- As another example, annotations can be used to guide AST rewriting: `http://www.scala-lang.org/sid/5`
- perform partial evaluation of programs,
- re-phrase custom syntax into DSLs (of the staged or embedded varieties)
- translation of dynamically-typed languages into Scala, `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/OOScriptToScala.pdf`
- sub-expression line-debugging. Quoting from `http://article.gmane.org/gmane.comp.lang.scala.internals/4142`

  The idea is to have an option in the debugger to temporarily expand expressions written on a single line into code that expands the expression into intermediate function calls, on multiple lines. This way the debugger could step into the code and inspect intermediate function return values. Once the debugger finished the session, the code would revert to the original version.
  For example, this line

```
val r =(ls filter (_>1) sort (_<_) zipWithIndex) filter
{v=>(v._2)%2==0} map {_._1}
```

  would temporarily expand into

```
val r_1 = ls.filter(_>1)
val r_2 = r_1.sort(_<_)
val r_3 = r_2.zipWithIndex
val r_4 = r_3.filter(v => (v._2) % 2 == 0)
val r = r_4.map(_._1)
```

6 TODO: Unparsing after `explicitouter` and `erasure`

The unparser has been initially designed to handle ASTs after all phases up to `explicitouter`.

- In order for the unparser to work after `erasure`, the Scala backend has to receive non-erased type arguments (that's work in progress for Scala.NET)
- In order for unparsed output to be compilable again after phase `explicitouter`, the `LabelDefs` (representing jumps) that `TransMatch` inserts have to be rephrased in Scala terms:
  - defunctionalizing forward jumps and building an explicit state machine, `http://www.scala-lang.org/node/7423`
  - Sec. 2.2 Recovering loops from LabelDef-Apply pairs, `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/ScalaMinusMinus.pdf`

Regarding the second item, some jumps can be recast right away in terms of loops:

```
// while (cond) body ==> LabelDef($L, List(), if (cond) { body; L$() } else ())
case laDef @ LabelDef(name, params, If(cond, thenp @ Block(stats, laCall), _)) =>
  // FYI laCall @ Apply(laCallFun : Ident, Nil)
  // For compiler trees, assert(laDef.name eq laCallFun.name) holds.
  // But we don't check that because we also want to unparse trees
  // "with just enough concrete-syntax structure in them"
  row("while ", parens(xform(cond)), blockify(transformTrees(stats)))

// do body while (cond) ==> LabelDef($L, List(), body; if (cond) L$() else ())
case label @ LabelDef(_, _, Block(stats, If(cond, _, _))) =>
  row("do ", blockify(transformTrees(stats)), " while ", parens(xform(cond)))
```

but in general the technique described in `http://www.scala-lang.org/node/7423` is necessary.

7 FYI: Scala subset reaching each phase

A glimpse of this is shown in Listing 2.

Listing 2: Scala subset reaching each phase, Sec. 7

```
class TreeMatchTemplate {
  // non-trees defined in Trees
  //
  // case class ImportSelector(name: Name, namePos: Int, rename: Name, renamePos: Int)
  // case class Modifiers(flags: Long, privateWithin: Name, annotations: List[Tree], positions: Map[Long, Posit
  //
  def apply(t: Tree): Unit = t match {
    // eliminated by typer
    case Annotated(annot, arg) =>
    case AssignOrNamedArg(lhs, rhs) =>
    case DocDef(comment, definition) =>
    case Import(expr, selectors) =>

    // eliminated by refchecks
    case ModuleDef(mods, name, impl) =>
    case TypeTreeWithDeferredRefCheck() =>

    // eliminated by erasure
    case TypeDef(mods, name, tparams, rhs) =>
    case Typed(expr, tpt) =>

    // eliminated by cleanup
    case ApplyDynamic(qual, args) =>

    // eliminated by explicitouter
    case Alternative(trees) =>
    case Bind(name, body) =>
    case CaseDef(pat, guard, body) =>
    case Star(elem) =>
    case UnApply(fun, args) =>

    // eliminated by lambdalift
    case Function(vparams, body) =>

    // eliminated by uncurry
    case AppliedTypeTree(tpt, args) =>
    case CompoundTypeTree(templ) =>
    case ExistentialTypeTree(tpt, whereClauses) =>
    case SelectFromTypeTree(qual, selector) =>
    case SingletonTypeTree(ref) =>
    case TypeBoundsTree(lo, hi) =>

    // survivors
    case Apply(fun, args) =>
    case ArrayValue(elemtpt, trees) =>
    case Assign(lhs, rhs) =>
    case Block(stats, expr) =>
    case ClassDef(mods, name, tparams, impl) =>
    case DefDef(mods, name, tparams, vparamss, tpt, rhs) =>
    case EmptyTree =>
    case Ident(name) =>
    case If(cond, thenp, elsep) =>
    case LabelDef(name, params, rhs) =>
    case Literal(value) =>
    case Match(selector, cases) =>
    case New(tpt) =>
    case PackageDef(pid, stats) =>
    case Return(expr) =>
    case Select(qualifier, selector) =>
    case Super(qual, mix) =>
    case Template(parents, self, body) =>
    case This(qual) =>
    case Throw(expr) =>
    case Try(block, catches, finalizer) =>
    case TypeApply(fun, args) =>
    case TypeTree() =>
    case ValDef(mods, name, tpt, rhs) =>

    // missing from the Trees comment
    case Parens(args) =>                      // only used during parsing
    case SelectFromArray(qual, name, erasure) => // only used during erasure
  }
}
```