How the **flatten** phase works

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

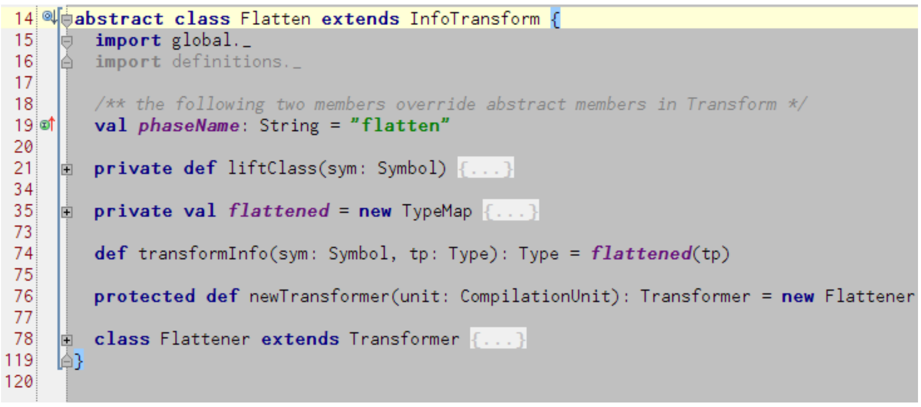
July 4th, 2011

Abstract

flatten runs in forJVM mode because Java bytecode doesn't accomodate nested types (unlike Microsoft's CLR, with the caveat that a nested class in CLR is not an inner class with access to a unique outer instance, instead CLR class nesting serves namespace structuring and grants access privileges only). Given the focus of this phase, its implementation allows seeing first hand "type re-attribution" (in contrast to other phases where most of the code deals with "tree rewriting").

```
      phase name id  description
      ---------- --  -----------
          parser  1  parse source into ASTs, perform simple desugaring
           namer  2  resolve names, attach symbols to named trees
  packageobjects  3  load package objects
           typer  4  the meat and potatoes: type the trees
  superaccessors  5  add super accessors in traits and nested classes
         pickler  6  serialize symbol tables
       refchecks  7  reference/override checking, translate nested objects
        liftcode  8  reify trees
         uncurry  9  uncurry, translate function values to anonymous classes
       tailcalls 10  replace tail calls by jumps
      specialize 11  @specialized-driven class and method specialization
   explicitouter 12  this refs to outer pointers, translate patterns
         erasure 13  erase types, add interfaces for traits
        lazyvals 14  allocate bitmaps, translate lazy vals into lazified defs
      lambdalift 15  move nested functions to top level
    constructors 16  move field definitions into constructors
/*------------------------------------------------------------------------*/
         flatten 17  eliminate inner classes
/*------------------------------------------------------------------------*/
           mixin 18  mixin composition
         cleanup 19  platform-specific cleanups, generate reflective calls
           icode 20  generate portable intermediate code
         inliner 21  optimization: do inlining
        closelim 22  optimization: eliminate uncalled closures
             dce 23  optimization: eliminate dead code
             jvm 24  generate JVM bytecode
        terminal 25  The last phase in the compiler chain
```

Contents



```
abstract class Flatten extends InfoTransform {
  import global._
  import definitions._

  /** the following two members override abstract members in Transform */
  val phaseName: String = "flatten"

  private def liftClass(sym: Symbol) {...}

  private val flattened = new TypeMap {...}

  def transformInfo(sym: Symbol, tp: Type): Type = flattened(tp)

  protected def newTransformer(unit: CompilationUnit): Transformer = new Flattener

  class Flattener extends Transformer {...}
}
```

Figure 1: Sec. 1

1 On tree descent: collecting keys

The structure of this transform is depicted in Figure 1.

We start with the transform's transformer, `Flattener` (the difference between a `Transform` and an `InfoTransform` was covered in the `lazyvals` write-up).

Moving from root to leaves, a map is populated to hold keys for packages that can potentially hold classes to unnest (there may be no classes to unnest).

```
override def transform(tree: Tree): Tree = {
  tree match {
    case PackageDef(_, _) =>
      liftedDefs(tree.symbol.moduleClass) = new ListBuffer
    case Template(_, _, _) if tree.symbol.owner.hasPackageFlag =>
      liftedDefs(tree.symbol.owner) = new ListBuffer
    case _ =>
  }
  postTransform(super.transform(tree))
}
```

Although it's unnesting what this phase is about, the map is called `liftedDefs`, just like the map in `lambdalift` that serves a completely different purpose.

```
private val liftedDefs = new mutable.HashMap[Symbol, ListBuffer[Tree]]
```

2 On the way up

2.1 Eliding, Collecting trees to relocate

Before returning a `Tree` on the way back to the root, it's `postTransform`'s chance to make a difference:

- any `ClassDef` with `isNestedClass` symbol is sent into oblivion (i.e., an `EmptyTree` takes its place but the original tree will stay for a while in the `liftedDefs` map, under the key of its future home). This pattern also matches a nested "`object P`" as described in Sec. 2.1.1.

- references to a nested object are replaced with another one "after relocation" (that's what the "`atPhase(phase.next)`" is for) because at that point the navigation path to reach the object will be different (with fewer `Selects`), and because `mkAttributedRef` bases the tree it builds on `sym.owner.thisType`. Example in Sec. 2.1.2.

```
private def postTransform(tree: Tree): Tree = {
  val sym = tree.symbol
  val tree1 = tree match {
    case ClassDef(_, _, _, _) if sym.isNestedClass =>
      liftedDefs(sym.toplevelClass.owner) += tree
      EmptyTree
    case Select(qual, name) if (sym.isStaticModule && !sym.owner.isPackageClass) =>
      atPhase(phase.next) {
        atPos(tree.pos) {
          gen.mkAttributedRef(sym)
        }
      }
    case _ =>
      tree
  }
  tree1 setType flattened(tree1.tpe)
}
```

2.1.1 It reads "`object P`" but it's a `ClassDef`

One more detail about `postTransform`. The pattern

```
case ClassDef(_, _, _, _) if sym.isNestedClass =>
```

also matches "`object P`" in the program below. Why? Because although `parser` delivers a `ModuleDef` for it, `refchecks` replaces it with a `ClassDef`.

```
class C {
  class D
  object P
}
```

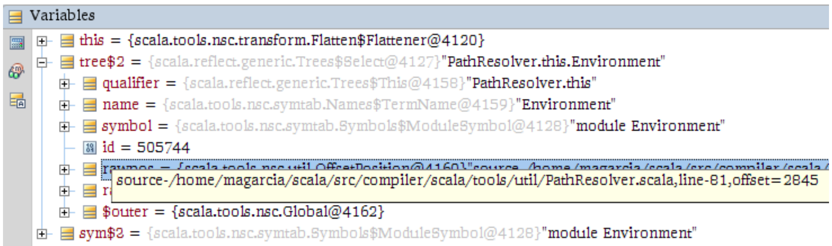
Quoting from `refchecks`:

```
/** Eliminate ModuleDefs.
 *   - A top level object is replaced with their module class.
 *   - An inner object is transformed into a module var, created on first access.
 *
 *  In both cases, this transformation returns the list of replacement trees:
 *   - Top level: the module class accessor definition
 *   - Inner: a class definition, declaration of module var, and module var accessor
 */
```

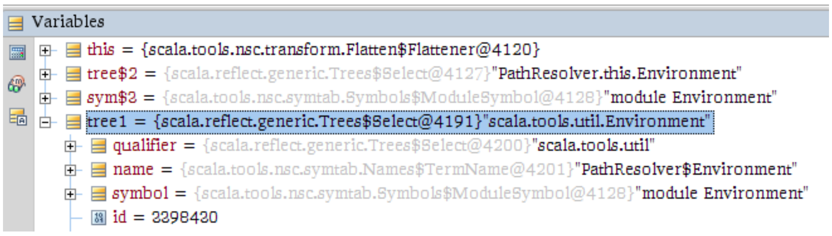
(Frequently, `scalac` phases can't be understood in isolation).

2.1.2 Rewriting

The `case Select` in Sec. 2.1 "rewrites" a tree node. For example, originally:



After rewriting:



2.2 Pasting

Given that `postTransform` collected on tree descent classes for unnesting, they will be ready in `liftedDefs` by the time the following runs, and thus are "pasted" ("relocated" sounds better?) in their new home (a package):

```
override def transformStats(stats: List[Tree], exprOwner: Symbol): List[Tree] = {
  val stats1 = super.transformStats(stats, exprOwner)
  if (currentOwner.isPackageClass) stats1 ::: liftedDefs(currentOwner).toList
  else stats1
}
```

3 Sometime later: re-typing trees

Remember the very last line of `Flattener.postTransform()`? It maps the type of every tree (unnested or not) with a `TypeMap` (i.e., sthg that extends `Function1[Type, Type]`):

```
private def postTransform(tree: Tree): Tree = {
  val sym = tree.symbol
  . . .
  tree1 setType flattened(tree1.tpe)
}
```

and not to forget the overload in `Flatten`:

```
def transformInfo(sym: Symbol, tp: Type): Type = flattened(tp)
```

There are four cases that `flattened` handles (`TypeRef`, `ClassInfoType`, `MethodType`, and `PolyType`) as covered in Sec. 3.2 to Sec. 3.4.

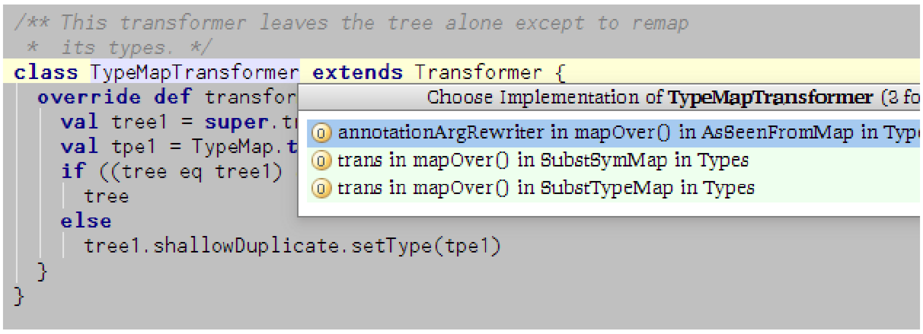


Figure 2: Sec. 3.1

3.1 Idioms around `TypeMap` and `TypeTraverser`

Basically, there are `TypeMaps` (with an inner `TypeMapTransformer`, which "leaves the tree alone except to remap its types") and `TypeTraverser`. Subtypes of `TypeMapTransformer` are used in:

1. the abstract `TypeMap` itself:

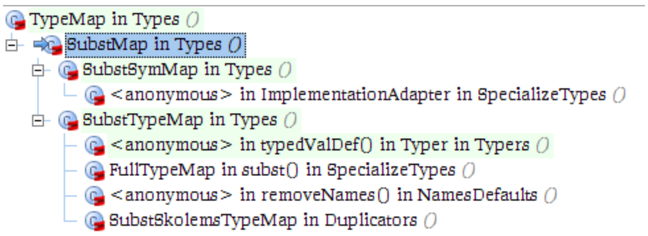
```
/** Map a tree that is part of an annotation argument.
 *  If the tree cannot be mapped, then invoke giveup().
 *  The default is to transform the tree with
 *  TypeMapTransformer.
 */
def mapOver(tree: Tree, giveup: ()=>Nothing): Tree =
  (new TypeMapTransformer).transform(tree)
```

2. in `annotationArgRewriter` as part of `AsSeenFromMap`

3. in `SubstSymMap`

4. in `SubstTypeMap`

There are too many `TypeMap` subclasses so we just show:



There are fewer `TypeTraverser` subclasses, Figure 3 depicts them all.

3.2 `TypeRef`

```
case TypeRef(pre, sym, args) if (pre.typeSymbol.isClass && !pre.typeSymbol.isPackageClass) =>
  assert(args.isEmpty) /*- Scala.NET alarm: not necessarily the case
    (although `flatten` isn't forMSIL, it's still useful to track its dependencies on `erasure`) */
```

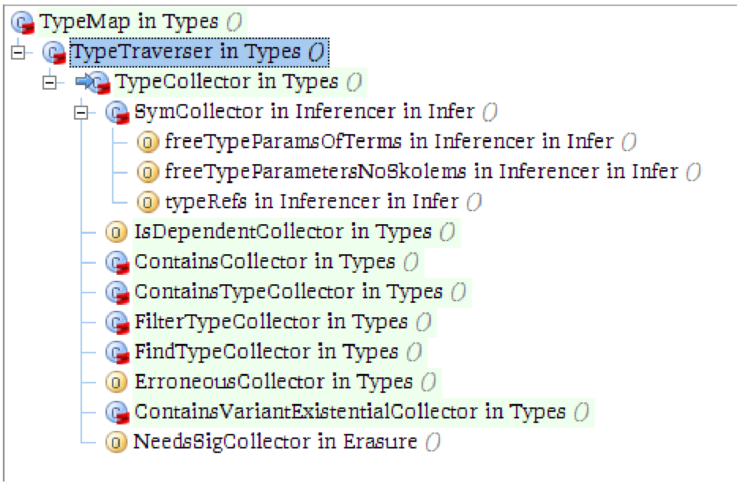


Figure 3: Sec. 3.1

```
  assert(sym.toplevelClass != NoSymbol, sym.ownerChain)
  typeRef(sym.toplevelClass.owner.thisType, sym, args)
```

3.3 ClassInfoType

Upon entry to

```
def transformInfo(sym: Symbol, tp: Type): Type
```

only some combinations of symbol and its type are possible:

- a `ModuleClassSymbol` (e.g., for a package) has a type given by a `ClassInfoType`
- a `ClassSymbol` may show up with a `ClassInfoType` or a `PolyType`
- a `MethodSymbol` has always a `MethodType` to represent its type.
- a `TermSymbol` goes together with a (`Unique`)`TypeRef`.

It's useful to keep in mind that both (a) what gets unnested; and (b) the new home of sthg that got unnested; well both those things, have a `ClassInfoType`.

One more pre-requisite to understand what follows:

```
/** A class representing a class info
 */
case class ClassInfoType(
  override val parents: List[Type],
  override val decls: Scope,
  override val typeSymbol: Symbol) extends CompoundType
{
  . . .
}
```

Regarding the re-mapping for `ClassInfoType`, there are two sub-cases:

```
private def liftClass(sym: Symbol) {
  if (!(sym hasFlag LIFTED)) {
    sym setFlag LIFTED
    atPhase(phase.next) {
      log("re-enter " + sym + " in " + sym.owner)
      assert(sym.owner.isPackageClass, sym) //debug
      val scope = sym.owner.info.decls
      val old = scope lookup sym.name
      if (old != NoSymbol) scope unlink old
      scope enter sym
    }
  }
}
```

Figure 4: Sec. 3.3

1. a package may have become home to `ClassDefs` which were unnested, Therefore the package's `info`'s `Scope` needs to include entries for those `ClassDefs`.

2. An unnested `ClassDef` N was unnested from some `ClassDef` C. Therefore N should not appear `atPhase(phase.next)` among the entries in the scope of the `info` of C.

Both situations are handled in a single case handler:

```
case ClassInfoType(parents, decls, clazz) =>
```

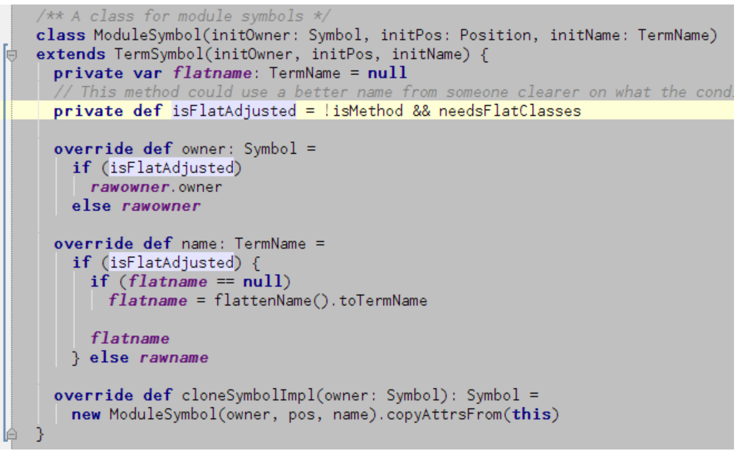
Both of the above are achieved by iterating over the `decls` of N,

- Case 2.1: `if(sym.isTerm && !sym.isStaticModule)` keep it (i.e., add it to the scope being populated).

- Case 2.2: `if(sym.isClass)` don't enter it in the scope being populated. Instead `liftClass(sym)`, to enter it in its now home. That's possible because of the way `ClassSymbol` ***overrides*** `owner` (and, there's more to it! Sec. 3.3.1):

```
override def owner: Symbol =
  if (needsFlatClasses) rawowner.owner
  else rawowner
```

```
case ClassInfoType(parents, decls, clazz) =>
  var parents1 = parents
  val decls1 = new Scope
  if (clazz.isPackageClass) { /*- item (1) above */
    atPhase(phase.next)(decls.toList foreach (sym => decls1 enter sym))
  } else {   /*- item (2) above */
    val oldowner = clazz.owner
    atPhase(phase.next)(oldowner.info)
    parents1 = parents mapConserve (this)
    for (sym <- decls.toList) {
      if (sym.isTerm && !sym.isStaticModule) { /*- item (2.1) above */
        decls1 enter sym
        if (sym.isModule) sym.moduleClass setFlag LIFTED // Only top modules
```



```scala
/** A class for module symbols */
class ModuleSymbol(initOwner: Symbol, initPos: Position, initName: TermName)
extends TermSymbol(initOwner, initPos, initName) {
  private var flatname: TermName = null
  // This method could use a better name from someone clearer on what the cond
  private def isFlatAdjusted = !isMethod && needsFlatClasses

  override def owner: Symbol =
    if (isFlatAdjusted)
      rawowner.owner
    else rawowner

  override def name: TermName =
    if (isFlatAdjusted) {
      if (flatname == null)
        flatname = flattenName().toTermName

      flatname
    } else rawname

  override def cloneSymbolImpl(owner: Symbol): Symbol =
    new ModuleSymbol(owner, pos, name).copyAttrsFrom(this)
}
```

Figure 5: Sec. 3.3

```scala
        // Nested modules (MODULE flag is reset so we access through lazy):
        if (sym.isModuleVar && sym.isLazy) sym.lazyAccessor.lazyAccessor setFlag LIFTED
      } else if (sym.isClass) { /*- item (2.2) above */
        liftClass(sym)
        if (sym.needsImplClass) liftClass(erasure.implClass(sym))
      }
    }
  }
  ClassInfoType(parents1, decls1, clazz)
```

3.3.1 Flatten-aware `ClassSymbol` and `ModuleSymbol`

We saw `needsFlatClasses` in Sec. 3.3 and its effect on `ClassSymbol.owner`. There's more to it (*Find usages* tells the whole story):

```scala
/**
 * Returns the rawInfo of the owner. If the current phase has flat classes,
 * it first applies all pending type maps to this symbol.
 *
 * assume this is the ModuleSymbol for B in the following definition:
 *   package p { class A { object B { val x = 1 } } }
 *
 * The owner after flatten is "package p" (see "def owner"). The flatten type map enters
 * symbol B in the decls of p. So to find a linked symbol ("object B" or "class B")
 * we need to apply flatten to B first. Fixes #2470.
 */
private final def flatOwnerInfo: Type = {
  if (needsFlatClasses)
    info
  owner.rawInfo
}
```

Also related to this, there's `isFlatAdjusted` in `ModuleSymbol` (Figure 5), this time affecting `ModuleSymbol`'s `def owner` and `def name` behavior.

Listing 1: Sec. 3.4

```
/** A type function or the type of a polymorphic value (and thus of kind *).
 *
 * Before the introduction of NullaryMethodType, a polymorphic nullary method
 * (e.g, def isInstanceOf[T]: Boolean)
 * used to be typed as PolyType(tps, restpe),
 * and a monomorphic one as PolyType(Nil, restpe)
 *
 * This is now: PolyType(tps, NullaryMethodType(restpe)) and NullaryMethodType(restpe)
 * by symmetry to MethodTypes: PolyType(tps, MethodType(params, restpe)) and MethodType(params, restpe)
 *
 * Thus, a PolyType(tps, TypeRef(...)) unambiguously indicates
 *   a type function (which results from eta-expanding a type constructor alias).
 * Similarly, PolyType(tps, ClassInfoType(...)) is a type constructor.
 *
 * A polytype is of kind * iff its resultType is a (nullary) method type.
 */
case class PolyType(override val typeParams: List[Symbol], override val resultType: Type)
    extends Type {

  //assert(!(typeParams contains NoSymbol), this)

  // used to be a marker for nullary method type, illegal now (see @NullaryMethodType)
  assert(typeParams nonEmpty, this)

  . . .
```

3.4 The rest: `MethodType`, `PolyType`, etc.

Background on `PolyType` in Listing 1.

```
case MethodType(params, restp) =>
  val restp1 = apply(restp)
  if (restp1 eq restp) tp else copyMethodType(tp, params, restp1)
case PolyType(tparams, restp) =>
  val restp1 = apply(restp);
  if (restp1 eq restp) tp else PolyType(tparams, restp1)
case _ =>
  mapOver(tp)
```

```
/*- Scala.NET alarm. For example, */
package p
  class C[X] {
    class D[Y]
  }

   /*- We said "both situations are handled with a single case handler" */
   case ClassInfoType(parents, decls, clazz) =>

   /*- but without erasure the info of a ClassDef.symbol can be
       a type constructor that should be handled via */
   case PolyType(typeParams, resultType) =>

/*- (although ‘flatten‘ isn’t forMSIL,
    it’s still useful to track its dependencies on ‘erasure‘) */
```