Lowering inner classes to nested classes

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

August 30th, 2011

Abstract

The `explicitouter` phase lowers inner classes into nested classes (rephrasing references to outer instances in the process). Not only can the following phases forget about inner classes, they also need not worry about the distinction nested vs. top-level (for an exception see `lambdalift`). Nested classes, at least in `forJVM` mode, also go away: `flatten` goes after classes whose symbol `isNestedClass`.

BTW, inner classes can refer (sneakily if you will) to type params or member types of an enclosing class. This doesn't matter in `forJVM` mode because `erasure` will wallpaper over all that, but I'm mentioning this for the attentive Scala.NET reader.

Additionally, `explicitouter` sets `matchTranslation` in motion to emit code for pattern matching. That's outside the scope of these notes.

```
     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

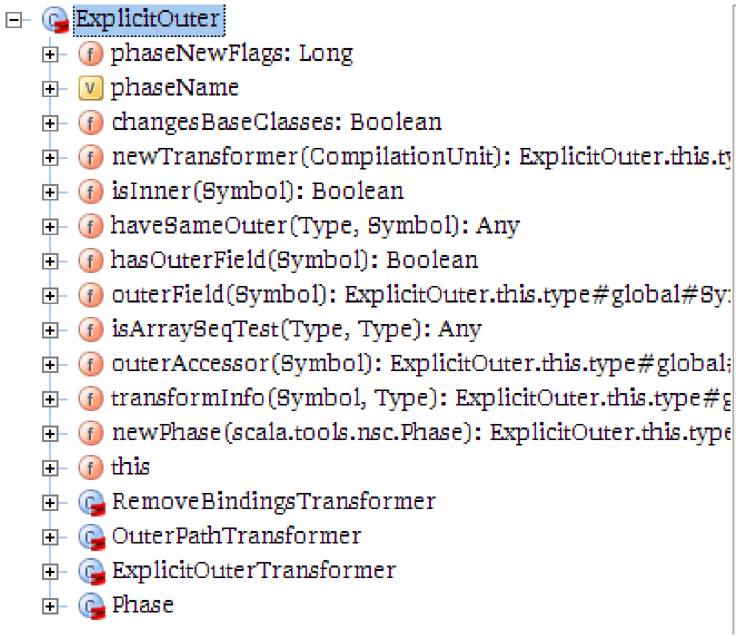


Figure 1: `ExplicitOuter`, Sec. 1

1 Intro

As with any `InfoTransform` there are both term rewriting (performed in an `Transformer.transform()` override) and type rewriting (in the form of symbol `info` rewriting, realized by an `InfoTransform.transformInfo()` override).

In the case of `ExplicitOuter`, term rewriting relies on updated types (the snippet below belongs to `ExplicitOuterTransformer`, Figure 1):

```
/** The transformation method for whole compilation units */
override def transformUnit(unit: CompilationUnit) {
  atPhase(phase.next)(super.transformUnit(unit))
}
```

2 Terminology

- **outer param**: the constructor param receiving the outer instance, Sec. 2.1.
- **outer field**: a protected field to make the outer param available after constructors have run, Sec. 2.2.
- **outer accessor**: a getter used internally to obtain the outer instance. For traits, it's abstract. In a non-trait class, its body picks the right outer field (inherited or not). Details in Sec. 2.3.

2.1 Outer param

A piece of information that comes handy during term rewriting is the current constructor's param for the outer instance (`outerParam` for short). As shown below, it's defined inside instance constructors of inner classes except in any class local to those constructors.

```
override def transform(tree: Tree): Tree = {
  val savedOuterParam = outerParam
  try {
    tree match {
      case Template(_, _, _) =>
        outerParam = NoSymbol
      case DefDef(_, _, _, vparamss, _, _) =>
        if (tree.symbol.isClassConstructor && isInner(tree.symbol.owner)) {
          outerParam = vparamss.head.head.symbol
          assert(outerParam.name startsWith nme.OUTER, outerParam.name)
        }
      case _ =>
    }
    super.transform(tree)
  }
  finally outerParam = savedOuterParam
}
```

How come the `ValDef` for an `outerParam` is always there by the time this override runs? As will be seen in Sec. 3.2, there's an override of this override that gets to run first, and in so doing adds the `ValDef` in question.

2.2 Outer field

Each non-trait inner class gets a protected field added for the outer instance, unless the field's contents would match one already declared in a superclass. Two utility methods, `hasOuterField(clazz: Symbol)` and `haveSameOuter()` (discussed below) find out whether an outer field should be added or not. The name of an outer field is fixed to:

```
val OUTER_LOCAL: NameType       = "$outer " // note the space
```

not to be confused with the suffix for outer accessors (Sec. 2.3):

```
val OUTER: NameType             = "$outer"
```

It's easy to create input that makes the helper function `haveSameOuter(parent: Type, clazz: Symbol)` indicate *another* outer field should be added (they are distinguishable by their owner, having identical names). For example, for `N` below, adding an outer field in addition to the outer field owned by `I1` (because `I1` and `N` have different owners):

```
class O {
  class I1
  class I2 {
    class N extends I1
  }
}
```

Listing 1: Sec. 2.2

```
class I2 extends java.lang.Object with ScalaObject {

  class N extends O#I1 with ScalaObject {

    /*- not shown: another $outer field (whose type is O) that is inherited from I1. */
    protected <synthetic> <paramaccessor> val $outer: O#I2 = _;

    /*- the source class for this outer accessor is N,
        therefore its return type is N's owner. */
    <synthetic> <stable> def O$I2$N$$$outer(): O#I2 = N.this.$outer;

    def this($outer: O#I2): O#I2#N = {
      if ($outer.eq(null))
        throw new java.lang.NullPointerException()
      else
        N.this.$outer = $outer;
      N.super.this($outer.O$I2$$$outer());
      /*- above, an outer accessor is invoked
       (returns O, the type of the owner of the source class for the invoked accessor).*/
      ()
    }
  };

  protected <synthetic> <paramaccessor> val $outer: O = _;
  <synthetic> <stable> def O$I2$$$outer(): O = I2.this.$outer;
  def this($outer: O): O#I2 = {
    if ($outer.eq(null))
      throw new java.lang.NullPointerException()
    else
      I2.this.$outer = $outer;
    I2.super.this();
    ()
  }
};
```

After `constructors` has run (i.e., once code has been emitted to assign constructor params to fields) the situation for `I2` and `N` is depicted in Listing 1.

2.3 Outer accessor

Continuing with the same example, `N` has two outer accessors:

```
// inherited from I1, with source class I1:
<synthetic> <stable> def O$I1$$$outer() : O = I1.this.$outer;

// with source class N
<synthetic> <stable> def O$I2$N$$$outer(): O#I2 = N.this.$outer;
```

Good to know about outer accessors:

1. there's one for each direct outer class (each base class a.k.a "`sourceClass`" could in principle have its own direct outer class, that's why the name of an outer accessor is prefixed with the fully qualified name of the "`sourceClass`").

2. Outer accessors are stable.

```
TODO Does stable imply final?
```

2.4 Looking up outer fields and accessors

Provided `hasOuterField(clazz)`, a `clazz` is all that's needed to find its outer field:

```
private def outerField(clazz: Symbol): Symbol = {
  val result = clazz.info.member(nme.OUTER_LOCAL)
  assert(result != NoSymbol, "no outer field in "+clazz+" at "+phase)

  result
}
```

The lookup story is only slightly longer for outer accessors. The name of an outer accessor includes `nme.OUTER` as suffix (not to be confused with `nme.OUTER_LOCAL` which names every outer field). However, no dedicated map is needed to keep the correspondence *class symbol* → *outer accessor symbol*. Instead, lookup is based on name, which can be univocally determined given the 'base' `clazz`:

```
/** The expanded name of `name` relative to this class `base` with given `separator` */
def expandedName(name: TermName, base: Symbol, separator: String = EXPAND_SEPARATOR_STRING): TermName =
  newTermName(base.fullName('$') + separator + name)
```

Therefore, the 'base' `clazz` suffices as lookup key (well, almost always, sometimes resorting to `Symbol.outerSource`):

```
def outerAccessor(clazz: Symbol): Symbol = {

  val firstTry = clazz.info.decl(nme.expandedName(nme.OUTER, clazz))

  if (firstTry != NoSymbol && firstTry.outerSource == clazz)
    firstTry
  else
    clazz.info.decls find (_.outerSource == clazz) getOrElse NoSymbol
}
```

3 Term rewriting

Before delving into `ExplicitOuterTransformer` proper (Sec. 3.2 ff.) we look at some of the utility methods it relies on (Sec. 3.1).

Figure 2 on p. 7 gives an overview of the classes involved in term rewriting.

3.1 Utility methods

`OuterPathTransformer`, the superclass of `ExplicitOuterTransformer`, factors out utility methods also needed by `LambdaLifter`:

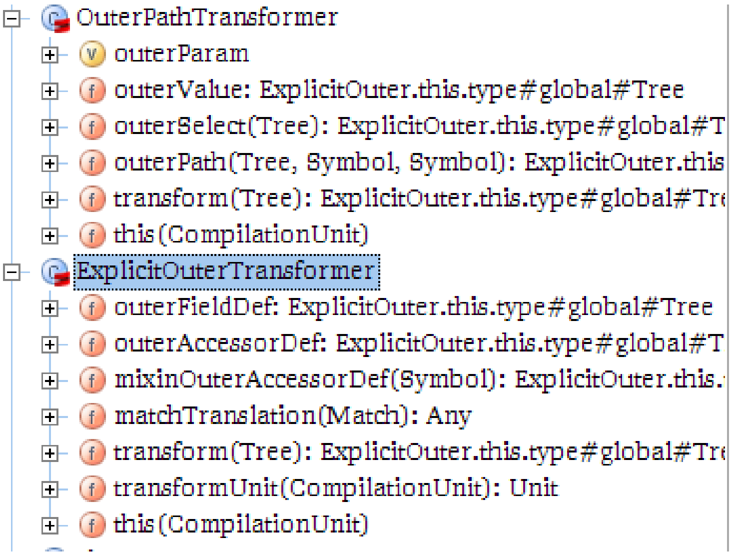
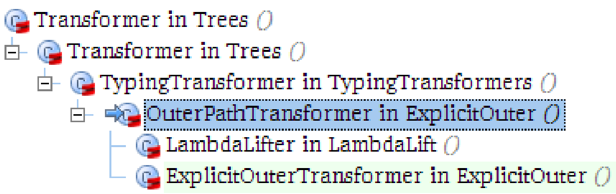


Figure 2: `ExplicitOuterTransformer`, Sec. 3



Besides keeping the `OuterPathTransformer`'s `outerParam` up-to-date, three utility tree builders are provided, summarized below:

```
/** The first outer selection from currently transformed tree.
 *  The result is typed but not positioned.
 */
protected def outerValue: Tree =
  if (outerParam != NoSymbol) ID(outerParam)
  else outerSelect(THIS(currentClass))

/** Select and apply outer accessor from 'base'
 *  The result is typed but not positioned.
 *  If the outer access is from current class and current class is final
 *  take outer field instead of accessor
 */
private def outerSelect(base: Tree): Tree = {
  . . .
}
/** The path
 *  <blockquote><pre>'base'.$outer$$C1 ... .$outer$$Cn</pre></blockquote>
 *  which refers to the outer instance of class to of
 *  value base. The result is typed but not positioned.
 */
protected def outerPath(base: Tree, from: Symbol, to: Symbol): Tree = {
  . . .
}
```

```
/** The definition tree of the outer accessor of current class
 */
def outerAccessorDef: Tree = {
  val outerAcc = outerAccessor(currentClass)
  var rhs: Tree =
    if (outerAcc.isDeferred) EmptyTree
    else This(currentClass) DOT outerField(currentClass)

  /** If we don't re-type the tree, we see self-type related crashes like #266.
   */
  localTyper typed {
    (DEF(outerAcc) withPos currentClass.pos withType null) === rhs
  }
}
```

Figure 3: `outerAccessorDef`, Sec. 3.3

3.2 Adding outer params

```
case DefDef(mods, name, tparams, vparamss, tpt, rhs) =>
  if (sym.isClassConstructor) {
    val clazz = sym.owner

    val vparamss1 =
      if (isInner(clazz)) { /*- a constructor of non-trait inner class */

        /*- symbol infos appear rewritten by now, that's why the setInfo works */
        val outerParam =
          sym.newValueParameter(sym.pos, nme.OUTER) setInfo outerField(clazz).info

        /*- updated first value-args list followed by remaining value-args lists */
        ((ValDef(outerParam) setType NoType) :: vparamss.head) :: vparamss.tail
      } else vparamss

    super.transform(treeCopy.DefDef(tree, mods, name, tparams, vparamss1, tpt, rhs))
  } else
    super.transform(tree)
```

3.3 Adding outer fields and outer accessors

Distilled from Listing 2 on p. 9:

- A non-trait inner class gets an outer field (see `outerFieldDef`).
- An inner class which is not an interface and is not static gets an outer accessor (see `outerAccessorDef`, Figure 3 on p. 8). Please notice it returns an abstract method for traits.
- A non-trait inner class can get for each class it mixes-in a mixin-outer-accessor (see `mixinOuterAccessorDef`). Details in Sec. 3.4.

3.4 Details on outer accessors for mixins

Say we have a non-interface inner trait, e.g. `T1` below:

Listing 2: Sec. 3.3

```
case Template(parents, self, decls) =>
  val newDefs = new ListBuffer[Tree]
  atOwner(tree, currentOwner) {
    if (!currentClass.isInterface || (currentClass hasFlag lateINTERFACE)) {
      if (isInner(currentClass)) {
        if (hasOuterField(currentClass))
          newDefs += outerFieldDef // (1a)
        newDefs += outerAccessorDef // (1)
      }
      if (!currentClass.isTrait)
        for (mc <- currentClass.mixinClasses)
          if (outerAccessor(mc) != NoSymbol)
            newDefs += mixinOuterAccessorDef(mc)
    }
  }
  super.transform(
    treeCopy.Template(tree, parents, self,
                  if (newDefs.isEmpty) decls else decls ::: newDefs.toList)
  )
```

```
class O {

  trait T1 { def doSthg(x: Any) = x.toString }

  class I2 {
    class N extends T1
  }

}
```

Because `AddInterfaces` hasn't run yet, there's no distinction `implClass-lateINTERFACE`. An abstract outer accessor gets added to the mixin by `outerAccessorDef` (Sec. 3.3):

```
abstract trait T1 extends java.lang.Object with ScalaObject {
  def /*T1*/$init$(): Unit = {
    ()
  };
  def doSthg(x: Any): java.lang.String = x.toString();
  <synthetic> <stable> def O$T1$$$outer(): O /*- abstract outer accessor */
};
```

And classes mixing-in that mixin get an implementation for it, added by `mixinOuterAccessorDef` (which is depicted in Figure 4 on p. 10):

```
  class N extends java.lang.Object with O.this.T1 with ScalaObject {
    def this($outer: I2.this.type): I2.this.N = {
      N.super.this();
      ()
    };
    protected <synthetic> <paramaccessor> val $outer: I2.this.type = _;
    <synthetic> <stable> def O$I2$N$$$outer(): O.this.I2 = N.this.$outer;

    /*- implementation for T1's (a trait) outer accessor. */
    <synthetic> <stable> def O$T1$$$outer(): O =
      N.this.O$I2$N$$$outer().O$I2$$$outer().asInstanceOf[O]()
```

```
/** The definition tree of the outer accessor for class mixinClass.
 *
 *  @param mixinClass The mixin class which defines the abstract outer
 *                    accessor which is implemented by the generated one.
 *  @pre mixinClass is an inner class
 */
def mixinOuterAccessorDef(mixinClass: Symbol): Tree = {
  val outerAcc = outerAccessor(mixinClass) overridingSymbol currentClass
  assert(outerAcc != NoSymbol)
  val path =
    if (mixinClass.owner.isTerm) THIS(mixinClass.owner.enclClass)
    else gen.mkAttributedQualifier(currentClass.thisType baseType mixinClass prefix)

  localTyper typed {
    (DEF(outerAcc) withPos currentClass.pos) === {
      // Need to cast for nested outer refs in presence of self-types. See ticket #3274.
      transformer.transform(path) AS_ANY outerAcc.info.resultType
    }
  }
}
```

Figure 4: `mixinOuterAccessorDef`, Sec. 3.4

```
  };
```

3.5 Path to outer instance

Quoting from the SLS:

> *The expression `C.this` is legal in the statement part of an enclosing class or object definition with simple name `C`. It stands for the object being defined by the innermost such definition. If the expression's expected type is a stable type, or `C.this` occurs as the prefix of a selection, its type is `C.this.type`, otherwise it is the self type of class `C`.*

A reference `C.this` where `C` refers to an outer class is replaced by a selection `this.$outer$$C1 ... .$outer$$Cn`

```
case This(qual) =>
  if (sym == currentClass || sym.hasModuleFlag && sym.isStatic) tree
  else atPos(tree.pos)(outerPath(outerValue, currentClass.outerClass, sym)) // (5)
```

One thing to note is the array of "is-static-something" inspectors on `Symbol` (Listing 3). Another thing to note is how to compose a path to the target outer instance, given its class and the `currentClass`. That means making sense of:

```
outerPath(outerValue, currentClass.outerClass, sym)
```

```
TODO Explain why the symbols for path components are there by the time the path is built.
```

3.6 Call to constructor of an inner class

Two cases may arise:

Listing 3: Sec. 4.2

```
/** Is this symbol a module variable?
 *  This used to have to test for MUTABLE to distinguish the overloaded
 *  MODULEVAR/SYNTHETICMETH flag, but now SYNTHETICMETH is gone.
 */
final def isModuleVar = hasFlag(MODULEVAR)

/** Is this symbol static (i.e. with no outer instance)? */
final def isStatic: Boolean = hasFlag(STATIC) || isRoot || owner.isStaticOwner

/** Is this symbol a static constructor? */
final def isStaticConstructor: Boolean = isStaticMember && isClassConstructor

/** Is this symbol a static member of its class? (i.e. needs to be implemented as a Java static?) */
final def isStaticMember: Boolean = hasFlag(STATIC) || owner.isImplClass

/** Does this symbol denote a class that defines static symbols? */
final def isStaticOwner: Boolean = isPackageClass || isModuleClass && isStatic
```

1. an inner-class constructor (either primary or auxiliary) invokes an auxiliary constructor

2. otherwise (in this case, the "receiver" of the invocation denotes the outer instance)

In both cases, the list of actual arguments is extended by prefixing an expression denoting the outer instance. In the first case, such outer instance is given by the outer param of the invoker. Otherwise it can be obtained from the type (necessarily a singleton type) of the qualifier `qual` in the invocation "`Apply(sel @ Select(qual, nme.CONSTRUCTOR), args)`".

Well, the above conveys the idea minus a few details for the second case. To close that loophole, its source comment is reproduced next:

> *A call to a constructor* `Q.<init>(args)` *or* `Q.$init$(args)` *where* `Q != this` *and the constructor belongs to a non-static class is augmented by an outer argument. E.g.* `Q.<init>(OUTER, args)` *where* `OUTER` *is the qualifier corresponding to the singleton type* `Q`

The source code of the rewriting (covering both cases) appears in Listing 4 on p. 12.

4 Type rewritings done during term rewriting

In a way, these rewritings are about types (and can only keep the program well-typed), but are performed during term rewriting (i.e., in `ExplicitOuterTransformer.transform()`):

1. Marking type symbols as `public`, Sec. 4.1.

2. Marking members accessed from inner classes as `public`, Sec. 4.2.

Listing 4: Sec. 3.6

```
case Apply(sel @ Select(qual, nme.CONSTRUCTOR), args) if isInner(sel.symbol.owner) =>
  val outerVal = atPos(tree.pos)(qual match {
    // it's a call between constructors of same class
    case _: This =>
      assert(outerParam != NoSymbol)
      ID(outerParam)
    case _       =>
      gen.mkAttributedQualifier(qual.tpe.prefix match {
        case NoPrefix => sym.owner.outerClass.thisType
        case x       => x
      })
  })
  super.transform(treeCopy.Apply(tree, sel, outerVal :: args))
```

```
TODO What does the following do? (in the catch-all handler of the term transform)

  val x = super.transform(tree)
  if (x.tpe eq null) x
  else x setType transformInfo(currentOwner, x.tpe)
```

4.1 Marking type symbols as `public`

```
TODO Looks like this rewriting is unrelated to inner classes. Could it be performed in another phase?.
```

```
override def transform(tree: Tree): Tree = {
  val sym = tree.symbol
  if (sym != null && sym.isType) { //(9)
    sym setNotFlag PRIVATE
    sym setNotFlag PROTECTED
  }
```

```
TODO Perhaps this guarantees those type symbols can be used as type arguments anywhere,
     so that they always comply with "Bounds on type params cannot be less visible than the type param's owner i

     Really? Is that the reason?
```

4.2 Marking members accessed from inner classes as `public`

These rewritings are a concession to `flatten`, or better said to the JVM. In contrast, on the CLI, nested classes enjoy privileged access to their nesting classes and their members.

- Remove private modifier from class members `M` that are accessed from an inner class.
- Remove protected modifier from class members `M` that are accessed (without a super qualifier) from an inner class or trait.

```
/** Remove private modifier from symbol `sym`s definition. If `sym` is a
 *  term symbol rename it by expanding its name to avoid name clashes
 */
final def makeNotPrivate(base: Symbol) {
  if (this.isPrivate) {
    setFlag(notPRIVATE)
    // Marking these methods final causes problems for proxies which use subclassing. If people
    // write their code with no usage of final, we probably shouldn't introduce it ourselves
    // unless we know it is safe. ... Unfortunately if they aren't marked final the inliner
    // thinks it can't inline them. So once again marking lateFINAL, and in genjvm we no longer
    // generate ACC_FINAL on "final" methods which are actually lateFINAL.
    if (isMethod && !isDeferred)
      setFlag(lateFINAL)
    if (!isStaticModule && !isClassConstructor) {
      expandName(base)
      if (isModule) moduleClass.makeNotPrivate(base)
    }
  }
}
```

Figure 5: `TypeWrapper` hierarchy

- Remove private modifiers from members of traits. A caveat: unlike the other items, this one is also necessary in Scala.NET: trait members may result in `implClass` members, which are going to be accessed from other classes.

  `TODO What about protected members in traits, then? (TODO ticket examples?).`

```
case Select(qual, name) =>
  if (currentClass != sym.owner) // (3)
    sym.makeNotPrivate(sym.owner)
  val qsym = qual.tpe.widen.typeSymbol
  if (sym.isProtected && //(4)
      (qsym.isTrait || !(qual.isInstanceOf[Super] || (qsym isSubClass currentClass))))
    sym setFlag notPROTECTED
  super.transform(tree)
```

`makeNotPrivate` does more than `setFlag(notPRIVATE)` on `sym`. And why does `sym.owner` appear as argument anyway? Figure 4.2 on p. 12.

5 Type rewriting

1. During `MethodType` rewriting (Listing 5 on p. 14):
   (a) Make all super accessors and modules in traits non-private, mangling their names.
   (b) Add an outer parameter to the formal parameters of a constructor in a non-trait inner class;
   (c) Remove protected flag from all members of traits.
2. During `ClassInfoType` rewriting (Listing 6 on p. 15):
   (a) Add an outer accessor to every inner class that is not an interface. The added outer accessor is abstract for traits, concrete otherwise (term rewriting will add the method body).

Listing 5: Sec. 5

```
def transformInfo(sym: Symbol, tp: Type): Type = tp match {
  case MethodType(params, restpe1) =>
    val restpe = transformInfo(sym, restpe1)
    if (sym.owner.isTrait && ((sym hasFlag (ACCESSOR | SUPERACCESSOR)) || sym.isModule)) { // 5

      /*- Make all super accessors and modules in traits non-private, mangling their names. */
      sym.makeNotPrivate(sym.owner)

    }

    /*- Remove protected flag from all members of traits.*/
    if (sym.owner.isTrait) sym setNotFlag PROTECTED // 6

    val res =
      if (sym.isClassConstructor && isInner(sym.owner)) { // 1

        /*- result 1 of 3 */
        /*- Add an outer parameter to the formal parameters of a constructor in a inner non-trait class */
        val p = sym.newValueParameter(sym.pos, "arg" + nme.OUTER)
                 .setInfo(sym.owner.outerClass.thisType)
        MethodType(p :: params, restpe)

      } else if (restpe ne restpe1)
        /*- result 2 of 3 */
        MethodType(params, restpe)
      else
        /*- result 3 of 3 */
        tp

    res
```

(b) Add a protected outer field to a non-trait inner class.

(c) Also add overriding accessor defs to every class that inherits mixin classes with outer accessor defs (unless the superclass already inherits the same mixin).

Two more type rewritings that don't fit in the above classification:

```
case PolyType(tparams, restp) =>
  val restp1 = transformInfo(sym, restp)
  if (restp eq restp1) tp else PolyType(tparams, restp1)

case _ =>
  // Local fields of traits need to be unconditionally unprivatized.
  // Reason: Those fields might need to be unprivatized if referenced by an inner class.
  // On the other hand, mixing in the trait into a separately compiled
  // class needs to have a common naming scheme, independently of whether
  // the field was accessed from an inner class or not. See #2946
  if (sym.owner.isTrait && sym.hasLocalFlag &&
          (sym.getter(sym.owner.toInterface) == NoSymbol))
    sym.makeNotPrivate(sym.owner)
  tp
```

Listing 6: Sec. 5

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

  var decls1 = decls

  if (isInner(clazz) && !clazz.isInterface) {

    /*- Add an outer accessor to every inner class that is not an interface.
     *  The added outer accessor is abstract for traits,
     *  concrete otherwise (term rewriting will add the method body).
     */

    decls1 = decls.cloneScope
    val outerAcc = clazz.newMethod(clazz.pos, nme.OUTER) // 3
    outerAcc expandName clazz

    val restpe = if (clazz.isTrait) clazz.outerClass.tpe else clazz.outerClass.thisType
    decls1 enter (clazz.newOuterAccessor(clazz.pos) setInfo MethodType(Nil, restpe))
    if (hasOuterField(clazz)) { //2

      /*- Add a protected outer field to a non-trait inner class.*/
      val access = if (clazz.isFinal) PRIVATE | LOCAL else PROTECTED
      decls1 enter (
        clazz.newValue(clazz.pos, nme.OUTER_LOCAL)
        setFlag (SYNTHETIC | PARAMACCESSOR | access)
        setInfo clazz.outerClass.thisType
      )
    }
  }

  if (!clazz.isTrait && !parents.isEmpty) {

    /*- Also add overriding accessor defs to every class that inherits
     *  mixin classes with outer accessor defs
     *  (unless the superclass already inherits the same mixin).
     */

    for (mc <- clazz.mixinClasses) {
      val mixinOuterAcc: Symbol = atPhase(phase.next)(outerAccessor(mc))
      if (mixinOuterAcc != NoSymbol) {
        if (decls1 eq decls) decls1 = decls.cloneScope
        val newAcc = mixinOuterAcc.cloneSymbol(clazz)
        newAcc resetFlag DEFERRED setInfo (clazz.thisType memberType mixinOuterAcc)
        decls1 enter newAcc
      }
    }
  }


  /*- end result */
  if (decls1 eq decls) tp else ClassInfoType(parents, decls1, clazz)
```