Type erasure in Scala for JVM

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

August 26th, 2011

Abstract

The `erasure` phase consists of several moving parts. We review how:

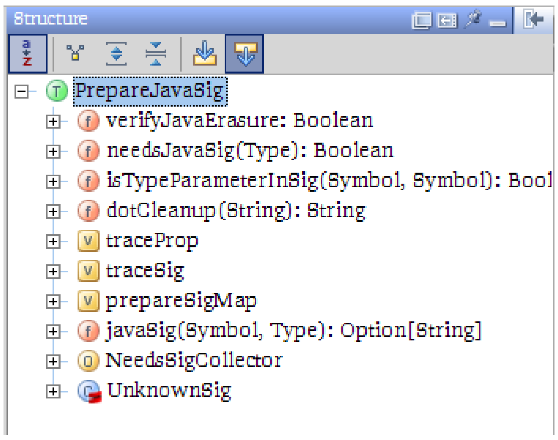
- the custom re-typer (`Eraser`) and
- a `TypingTransformer` (`PreTransformer`)

put trees in good shape so that they can emerge well-typed from the `erasure` phase. In addition to `Eraser` and "`atPhase(phase.next)`", re-typing involves "`transformInfo`". Also related, another write-up covers `AddInterfaces`[1].

```
           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
```

[1] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q3/AddInterfaces.pdf`

Contents



```
Structure
PrepareJavaSig
    verifyJavaErasure: Boolean
    needsJavaSig(Type): Boolean
    isTypeParameterInSig(Symbol, Symbol): Bool
    dotCleanup(String): String
    traceProp
    traceSig
    prepareSigMap
    javaSig(Symbol, Type): Option[String]
    NeedsSigCollector
    UnknownSig
```

Figure 1: Sec. 1

1 Eraser, the modifier typer that retypes and adapts trees on the fly

`Eraser` is a custom typer that receives "pre-erased" trees (Sec. 2) and does two things with them:

1. inserts casting, boxing, and unboxing operations.
2. types them, via
    - functionality inherited from `Typer`, and
    - custom `setType` on tree nodes.

In order to simplify the presentation, some anonymous classes have been given names, and methods moved from their original enclosing class into the nearest enclosing class where they are used (`PreTransformer` or `Eraser`). With that `Erase` and `ErasureTransformer` become slimmer. Also for readability, the `erasure` phase can be uncluttered by creating a trait `PrepareJavaSig` with the contents depicted in Figure 1.

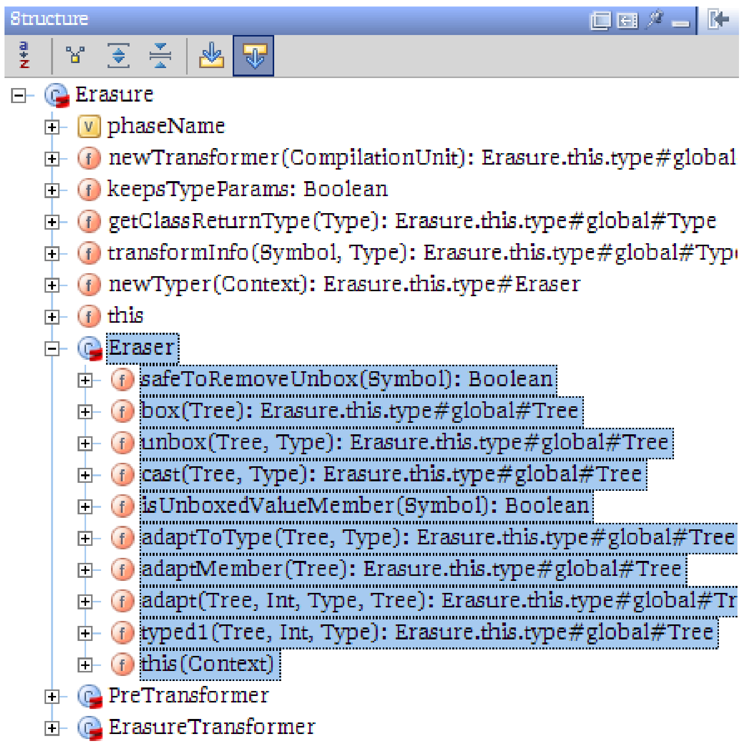
The resulting structure of `Eraser` is depicted in Figure 2.

The two main tools at `Eraser`'s disposal (`adaptToType()` and `adaptMember()`) are discussed in Sec. 1.1 and Sec. 1.2 resp.

1.1 `adaptToType()`

1. inserts boxing, unboxing, and cast ops to make the incoming `tree.tpe` conform to the expected type,
2. wraps in a zero-args `Apply` the invocation of a nullary method (a situation detected via "`tree.tpe.isInstanceOf[MethodType] && tree.tpe.params.isEmpty`")

Because `adaptToType` expects its first argument to be typed, recursive invocations with a different (usually, boxed or casted) tree have to fulfill that



```
Structure
Erasure
  phaseName
  newTransformer(CompilationUnit): Erasure.this.type#global
  keepsTypeParams: Boolean
  getClassReturnType(Type): Erasure.this.type#global#Type
  transformInfo(Symbol, Type): Erasure.this.type#global#Typ
  newTyper(Context): Erasure.this.type#Eraser
  this
  Eraser
    safeToRemoveUnbox(Symbol): Boolean
    box(Tree): Erasure.this.type#global#Tree
    unbox(Tree, Type): Erasure.this.type#global#Tree
    cast(Tree, Type): Erasure.this.type#global#Tree
    isUnboxedValueMember(Symbol): Boolean
    adaptToType(Tree, Type): Erasure.this.type#global#Tree
    adaptMember(Tree): Erasure.this.type#global#Tree
    adapt(Tree, Int, Type, Tree): Erasure.this.type#global#Tr
    typed1(Tree, Int, Type): Erasure.this.type#global#Tree
    this(Context)
  PreTransformer
  ErasureTransformer
```

Figure 2: Sec. 1

precondition, i.e. the utility methods `box`, `unbox`, and `cast` should type trees before returning them.

In more detail, "based on the incoming `tree.tpe` and the expected type" means that the those `Type` instances are compared using:

- `isValueClass(sym: Symbol)`
- `def <:<(that: Type): Boolean`

```
/** Adapt `tree` to expected type `pt`.
 *
 *  @param tree the given tree
 *  @param pt   the expected type
 *  @return     the adapted tree
 */
private def adaptToType(tree: Tree, pt: Type): Tree = {
  if (settings.debug.value && pt != WildcardType)
    log("adapting " + tree + ":" + tree.tpe + " : " + tree.tpe.parents + " to " + pt)//debug

  if (tree.tpe <:< pt) /*- already well-typed, do nothing. */
    tree
  else if (isValueClass(tree.tpe.typeSymbol) && !isValueClass(pt.typeSymbol)) {
    /*- box before type-adapting */
    adaptToType(box(tree), pt)
  } else if (tree.tpe.isInstanceOf[MethodType] && tree.tpe.params.isEmpty) {
    /*- wraps in a zero-args Apply the invocation of a nullary method */
    assert(tree.symbol.isStable, "adapt "+tree+":"+tree.tpe+" to "+pt)
    adaptToType(Apply(tree, List()) setPos tree.pos setType tree.tpe.resultType, pt)
  } else if (pt <:< tree.tpe) /*- downcast to expected type */
    cast(tree, pt)
  else if (isValueClass(pt.typeSymbol) && !isValueClass(tree.tpe.typeSymbol)) {
    /*- unbox before type-adapting */
    adaptToType(unbox(tree, pt), pt)
  } else /*- downcast to expected type */
    cast(tree, pt)
}
```

1.2 adaptMember()

`Eraser.adaptMember()` performs three kinds of rewritings:

1. A cast of `Any` into a Scala value class is equivalent to unboxing of an adapted qualifier, Sec. 1.2.1.
2. After erasure, ASTs will be devoid of references to `Any` or `AnyVal` (the "phantom types"). This requires adapting `Select`s: due to `Any` member, due to boxing mismatch, and in a few other cases too. Details in Sec. 1.2.2.
3. recover "non-erased" type to desugar `SelectFromArray` trees, Sec. 1.2.3.

```
TODO Confirm:
``After erasure, ASTs will be devoid of references to Any or AnyVal (the "phantom types")''.
What about Any_equals?
```

1.2.1 `Any.asInstanceOf[AnyVal]` same-thing-as unbox adapted receiver

The trees for type tests and casts (unlike all other type applications) were left in place by `PreTransformer.preErase()` (Sec. 2.1.1). It's true that their `tpe` was set to null right after that invocation in `PreTransformer.transform()`, but that's another story.

Regarding the type argument, back in `PreTransformer.preErase()` class literals were rewritten into their erasure, and by now its `tpe` is not null.

As the snippet below shows, whenever the type-arg is a Scala value class a rewriting is performed *that elides the type cast yet preserves semantics.* Otherwise the original `asInstanceOf` tree is left as-is.

- We're in the case *<some-Any-value>*`.asInstaceOf[`*<value-class-V>*`]`:
  1. The receiver of the type cast is re-typed with an expected type of `ObjectClass.tpe`, which results in *a new receiver* ("boxing of the original receiver").
  2. An unbox invocation *into the value class denoted by the type-arg* is emitted taking the new receiver as input.
  3. Summing up, the emitted "`unboxToV(box(receiver))`" allows preserving semantics upon eliding the type-cast:
     - the unbox-box composition will succeed iff the original receiver has a runtime value of the given *<value-class-V>*.
     - in particular, when the original receiver evaluates to `null`, a non-null `AnyRef`, or some non-`V` valueclass, the `unboxToV` will fail.

```
case Apply(TypeApply(sel @ Select(qual, name), List(targ)), List())
if tree.symbol == Any_asInstanceOf =>
  val targClass = targ.tpe.typeSymbol

  if (isValueClass(targClass)) {
      val qual1 = typedQualifier(qual, NOmode, ObjectClass.tpe) // need to have an expected type, see #3037
      unbox(qual1, targ.tpe)
  } else tree
```

```
TODO The snippet above was cleaned up a bit from the version in trunk.
```

1.2.2 Adapt `Select`s: due to `Any` member, due to boxing mismatch, and in a few other cases too

In order to get rid of references to members of `Any` or `AnyVal` (the "phantom types") it's necessary to replace a `Select` (or adapt its receiver) in the following situations:

```
  (1) Rewrite Select of an Any member (except constructor) to reference its Object counterpart:
      (a) x.asInstanceOf[T] on Any becomes x.$asInstanceOf[T] with $asInstanceOf in class Object.
      (b) x.isInstanceOf[T] on Any becomes x.$isInstanceOf[T] with $isInstanceOf in class Object.
      (c) x.m where m is some other member of Any becomes x.m where m is a member of class Object.
          This includes in particular:
            (c.I) ‘x == y‘ for == on Any becomes ‘x equals y‘ with equals  in class Object.
            (c.II) ‘x != y‘ for != on Any becomes ‘!(x equals y)‘ with equals in class Object.
```

```
(2) Rewrite remaining Selects in two steps:

   Firstly, adapt (in some cases) the receiver (aka "qualifier") based on its erased tpe:
   (1st-a) "boxed expected"
          x.m where  x has unboxed value type T, and
                     m is not a directly translated member of T
             becomes T.box(x).m
   (1st-b) "unboxed expected"
          x.m where  x is a reference type, and
                     m is a directly translated member of value type T
             becomes x.TValue().m

   Secondly, "adapt the selector" if needed (below, `y` is the result of adapting `x` as above):
   (2nd-a) y.m where y is a boxed type, and
                     m is a member of an unboxed class
             becomes y.n where n is the corresponding member of the boxed class.
   (2nd-b) y.m where m is nullary method
             becomes y.m()
   (2nd-c) y.m where /*- TODO */
             becomes /*- TODO */
```

1.2.3 Remove SelectFromArray

The first part of the "`SelectFromArray` duet" was performed in one of the pre-erase `Apply` transforms (Sec. 2.2.2) and the second part is performed here.

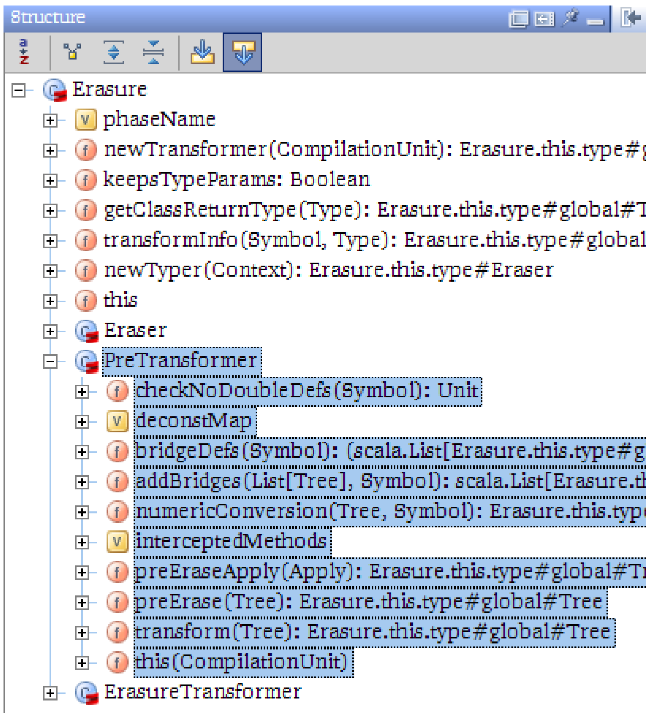
```
/**
 *  (adapt-member 3) SelectFromArray is used only during erasure, it's time to remove it.
 *
 *  The comment in @see PreTransformer.preEraseApply (where SelectFromArray trees are introduced) reads:
 *    "store exact array erasure in map to be retrieved later when we might need to do the cast in adaptMember"
 *
 */
case SelectFromArray(qual, name, erasure) =>
  var qual1 = typedQualifier(qual)
  if (!(qual1.tpe <:< erasure)) qual1 = cast(qual1, erasure)
  Select(qual1, name) copyAttrs tree
```

2 PreTransformer, or the art of preparing trees for Eraser

Although `PreTransformer` (Figure 3) runs before `Eraser`, discussing them in the opposite order (as done in this write-up) allows highlighting those places in `Eraser` where some pre-processing (on the part of `PreTransformer`) is expected before "it's too late". We've seen the following examples, amongst others:

- `SelectFromArray`

```
/**
 *  (adapt-member 3) SelectFromArray is used only during erasure, it's time to remove it.
 *
 *  The comment in @see PreTransformer.preEraseApply (where SelectFromArray trees are introduced) reads:
 *    "store exact array erasure in map to be retrieved later when we might need to do the cast in adaptMem
 *
```



```
Erasure
  phaseName
  newTransformer(CompilationUnit): Erasure.this.type#
  keepsTypeParams: Boolean
  getClassReturnType(Type): Erasure.this.type#global#T
  transformInfo(Symbol, Type): Erasure.this.type#global
  newTyper(Context): Erasure.this.type#Eraser
  this
  Eraser
  PreTransformer
    checkNoDoubleDefs(Symbol): Unit
    deconstMap
    bridgeDefs(Symbol): (scala.List[Erasure.this.type#g
    addBridges(List[Tree], Symbol): scala.List[Erasure.t
    numericConversion(Tree, Symbol): Erasure.this.typ
    interceptedMethods
    preEraseApply(Apply): Erasure.this.type#global#T
    preErase(Tree): Erasure.this.type#global#Tree
    transform(Tree): Erasure.this.type#global#Tree
    this(CompilationUnit)
  ErasureTransformer
```

Figure 3: Sec. 2

```
  */
```

- handling of invocations to `##` and `getClass` on `Any` and `Object` in `preEraseApply`

```
// Methods on Any/Object which we rewrite here while we still know
// what is a primitive and what arrived boxed.
```

2.1 All pre-erase transforms except those on `Apply` nodes

Not that the following tells anything new over previous write-ups, but the layout is more readable now. Please convince yourself:

`PreTransformer.preErase()` does most of the work that `PreTransformer` is credited for. What else does `PreTransformer` do? Just some `tpe`-nullifying in anticipation of the retyping by `Eraser`. To recap:

```
override def transform(tree: Tree): Tree = {
  if (tree.symbol == ArrayClass && !tree.isType)
    tree
  else {
    val tree1 = preErase(tree)
    tree1 match {
      case EmptyTree | TypeTree() =>
        tree1 setType erasure(NoSymbol, tree1.tpe)
      case DefDef(_, _, _, _, tpt, _) =>
        val result = super.transform(tree1) setType null
        tpt.tpe = erasure(tree1.symbol, tree1.symbol.tpe).resultType
        result
      case _ =>
        super.transform(tree1) setType null
    }
  }
}
```

Back to `preErase()`. It transforms a single node as described below, without recursing into children (yes, that also holds for those `TypeApply` transforms where the receiver is pre-erased, because in that case that child node is the single node being transformed, its containing node is discarded).

1. Erasure of type params

   - Remove all type parameters in class and method definitions.
   - Remove all abstract and alias type definitions.

   Note: `EmptyTree`s are emitted for `TypeDef` but removed in `Transformer.transformStats()` (which is invoked for the stmts in `PackageDef`, `Template`, and `Block`).

2. Remove all type applications other than type tests and casts. Details in Sec. 2.1.1. This *has to do* with casting of arrays, as discussed in Sec. 2.1.2.

3. Pre-erase `Apply` applications. This is delegated to `preEraseApply()`, Sec. 2.2

4. Corner case: Given a selection `q.s`, where the owner of `s` is not accessible but the type symbol of `q`'s type `qT` is accessible, insert a cast "`q.asInstanceOf[qT].s`". This prevents illegal access errors (see `#4283`).

5. Template-level transforms:

   - Check that there are no double definitions in a template.
   - Add bridge definitions to a template.

6. Remember the non-erased `tpe` of a match selector, rewrite class literals into their erasure.

2.1.1 Type applications

As a warm-up, let's dissect the realization of "remove all type applications other than type tests and casts":

```
/* pre-era (2): remove all type applications other than type tests and casts */

case TypeApply(fun, args) if (fun.symbol.owner != AnyClass &&
                              fun.symbol != Object_asInstanceOf &&
                              fun.symbol != Object_isInstanceOf) =>
  preErase(fun)
```

The condition above on `fun.symbol`, given the pattern `TypeApply(fun, args)`, prevent all type tests and casts from being erased. In detail, "`fun.symbol.owner != AnyClass`" is shorthand for the more explicit

```
fun.symbol != Any_isInstanceOf &&
fun.symbol != Any_asInstanceOf
```

Because `Any_getClass` takes no type-params it won't appear inside a `TypeApply`.

Note: Some type applications are removed before reaching here. For example, `preEraseApply()` rewrites into `ApplyDynamic`, including those invocations giving explicit type arguments, and in so doing removes the type application (Sec. 2.2.6).

2.1.2 Sidenote: *<array>*`.asInstanceOf[`*<BlaBla>*`]`

Briefly:

- Casting arrays requires no rewriting (this subsection). Sec. 2.2.4 discusses the few cases where `asInstanceOf` rewritings are performed.
- Type-testing of array values may require rewriting (Sec. 2.2.1).

At this point I couldn't resist cut & pasting form the spec:

> *First, unlike arrays in Java or C#, arrays in Scala are not co-variant. That is,* `S <: T` *does not imply* `Array[S] <: Array[T]` *in Scala. However, it is possible to cast an array of* `S` *to an array of* `T` *if such a cast is permitted in the host environment. For instance* `Array[String]` *does not conform to* `Array[Object]`*, even though* `String` *conforms to* `Object`*. However, it is possible to cast an expression of type* `Array[String]` *to* `Array[Object]`*, and this cast will succeed without raising a* `ClassCastException`*. Example:*

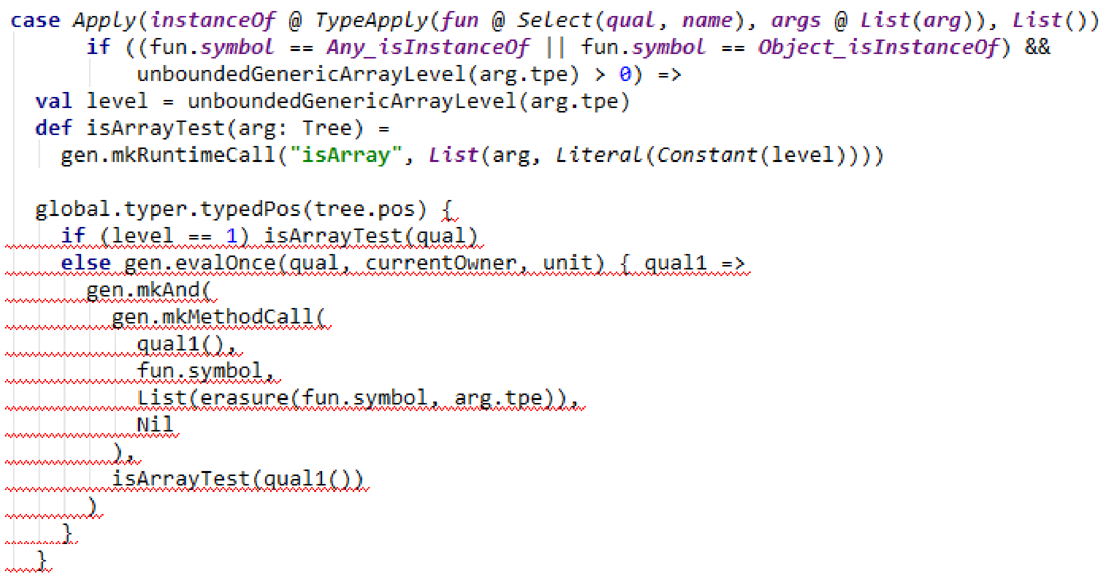
```
val xs = new Array[String](2)
// val ys: Array[Object] = xs // **** error: incompatible types

val ss = new Array[String](2)

def f[T](xs: Array[T]): Array[String] =
  if (xs.isInstanceOf[Array[String]]) xs.asInstanceOf[Array[String])
  else throw new Error("not an instance")

f(ss) // returns ss
```

2.2 Pre-erasing `Apply` nodes

To recap, we extracted from `preErase()` (Sec. 2.1) all those rewritings of `Apply` nodes, putting them together in `preEraseApply()` which is described in this section.

These rewritings comprise six subcases, each of them deserving its own sub-sub-section (Sec. 2.2.1 to Sec. 2.2.6).

1. Rewrite type-tests on generic arrays.
2. Rewrite member selections (including constructor) on arrays:
   (a) Rewrite calls to `apply`/`update`/`length`/`clone` on generic arrays
   (b) Do the pre-erase part of the `SelectFromArray` duet, the 2nd part done by `adaptMember()`
3. Two methods on `Any` and `Object` (`##` and `getClass`) are rewritten while we still know what is a primitive and what arrived boxed.



```
case Apply(instanceOf @ TypeApply(fun @ Select(qual, name), args @ List(arg)), List())
      if ((fun.symbol == Any_isInstanceOf || fun.symbol == Object_isInstanceOf) &&
          unboundedGenericArrayLevel(arg.tpe) > 0) =>
  val level = unboundedGenericArrayLevel(arg.tpe)
  def isArrayTest(arg: Tree) =
    gen.mkRuntimeCall("isArray", List(arg, Literal(Constant(level))))

  global.typer.typedPos(tree.pos) {
    if (level == 1) isArrayTest(qual)
    else gen.evalOnce(qual, currentOwner, unit) { qual1 =>
      gen.mkAnd(
        gen.mkMethodCall(
          qual1(),
          fun.symbol,
          List(erasure(fun.symbol, arg.tpe)),
          Nil
        ),
        isArrayTest(qual1())
      )
    }
  }
```

Figure 4: Sec. 2.2.1

4. (a) Elide a type cast statically known to succeed, or
   (b) Enforce numeric-conversion semantics.

5. (a) Rewrite type tests against singleton types, or
   (b) Optimize type tests against `RefinedType`.

6. Make dynamic applications explicit, removing any type application in the process.

Background: Type tests and casts were exempted when removing type applications (`PreTransformer.preErase()`, Sec. 2.1.1).

- Type tests are rewritten as discussed in Sec. 2.2.1 and Sec. 2.2.5

- Type casts are rewritten as discussed in Sec. 2.2.4

2.2.1 Rewrite `isInstanceOf[`*generic-array-type*`]`

Informally, a "*generic-array-type*" is an array type with a (nested) parameterized component type. The rewriting shown in Figure 4 acts on trees of the form `isInstanceOf[Array[`X`]]` where

1. X is a type param (in general, an abstract type), or

2. X is of the form `Array[`X`]`, and so on recursively.

Compare before and after `erasure`:

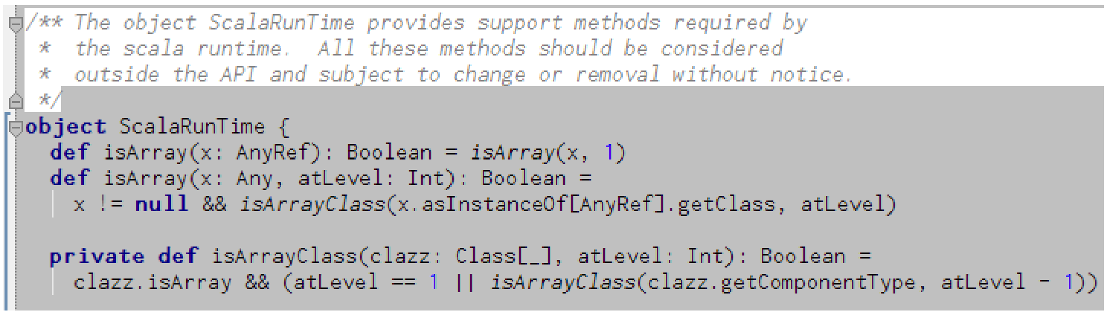
1. Before

```
def iaOfChar(arg: Any): Boolean = arg.isInstanceOf[Array[Char]]();

def iaOfT[T](arg: Any): Boolean = arg.isInstanceOf[Array[T]]();

def iaOfaOfT[T](arg: Any): Boolean = arg.isInstanceOf[Array[Array[T]]]();
```

2. After

```
def iaOfChar(arg: java.lang.Object): Boolean =
  arg.$isInstanceOf[Array[Char]]();

def iaOfT(arg: java.lang.Object): Boolean =
  runtime.this.ScalaRunTime.isArray(arg, 1);

def iaOfaOfT(arg: java.lang.Object): Boolean =
  arg.$isInstanceOf[Array[java.lang.Object]]().&&(runtime.this.ScalaRunTime.isArray(arg, 2));
```

The target of that desugaring lives in `scala.runtime.ScalaRunTime`:



```
/** The object ScalaRunTime provides support methods required by
 *  the scala runtime.  All these methods should be considered
 *  outside the API and subject to change or removal without notice.
 */
object ScalaRunTime {
  def isArray(x: AnyRef): Boolean = isArray(x, 1)
  def isArray(x: Any, atLevel: Int): Boolean =
    x != null && isArrayClass(x.asInstanceOf[AnyRef].getClass, atLevel)

  private def isArrayClass(clazz: Class[_], atLevel: Int): Boolean =
    clazz.isArray && (atLevel == 1 || isArrayClass(clazz.getComponentType, atLevel - 1))
```

2.2.2 Member selections (incl. constructor) on arrays (generic or not)

1. Rewrite calls to `apply`/`update`/`length`/`clone` on generic arrays.
   Example: `genarr(i) = iter.next()`
   becomes: `scala.runtime.ScalaRunTime.array_update(genarr, i, iter.next())`

2. Temporarily wrap the receiver (thus preserving its non-erased tpe) of a `Select` on non-generic array (ie. of an `<init>`/`apply`/`update`/`length`/`clone` on the qualifier) because only at `adaptMember`-time (Sec. 1.2.3) can be known whether a down-cast from *the original* `tpe` to the erased type is needed.

In case the above isn't clear enough, here goes the source code:

```
case Apply(fn @ Select(qual, name), args) if (fn.symbol.owner == ArrayClass) =>
  if (unboundedGenericArrayLevel(qual.tpe.widen) == 1)
    // convert calls to apply/update/length/clone on generic arrays to
    // calls of ScalaRunTime.array_xxx method calls
    global.typer.typedPos(tree.pos) { gen.mkRuntimeCall("array_"+name, qual :: args) }
  else
    // store exact array erasure in map to be retrieved later when we might
    // need to do the cast in adaptMember
    treeCopy.Apply(
      tree,
      SelectFromArray(qual, name, erasure(tree.symbol, qual.tpe)).copyAttrs(fn),
      args)
```

2.2.3 Rewrite calls to `##` and `getClass` on `Any` and `Object`

Two methods on `Any` and `Object` (`##` and `getClass`) are rewritten as follows (while we still know what is a primitive and what arrived boxed):

- implement semantics of `##` for `null` and `Unit`
- rewrite `v.getClass` (where `v` is value of Scala value class) into `scala.runtime.ScalaRunTime.anyValClass(v)` whose source comment reads:

> *Return the class object representing an unboxed value type, e.g.* `classOf[Int]`*, not* `classOf[java.lang.Integer]`*. The compiler rewrites expressions like* `5.getClass` *to come here.*

```
/*- pre-era-apply (3) */
  case Apply(fn @ Select(qual, _), Nil) if interceptedMethods(fn.symbol) =>
    if (fn.symbol == Any_## || fn.symbol == Object_##) {
      // This is unattractive, but without it we crash here on ().## because after
      // erasure the ScalaRunTime.hash overload goes from Unit => Int to BoxedUnit => Int.
      // This must be because some earlier transformation is being skipped on ##, but so
      // far I don't know what. For null we now define null.## == 0.
      qual.tpe.typeSymbol match {
        case UnitClass | NullClass                 => LIT(0)
        case IntClass                              => qual
        case s @ (ShortClass | ByteClass | CharClass) => numericConversion(qual, s)
        case BooleanClass                          => If(qual, LIT(true.##), LIT(false.##))
        case _                                     =>
          global.typer.typed(gen.mkRuntimeCall(nme.hash_, List(qual)))
      }
    }
    // Rewrite 5.getClass to ScalaRunTime.anyValClass(5)
    else if (isValueClass(qual.tpe.typeSymbol))
      global.typer.typed(gen.mkRuntimeCall(nme.anyValClass, List(qual)))
    else
      tree
```

2.2.4 Rewriting (some) type casts

Quoting from the SLS, §12.1:

> *The test* `x.asInstanceOf[T]` *is treated specially if* `T` *is a numeric value type (§12.2). In this case the cast will be translated to an application of a conversion method* `x.toT` *(§12.2.1). For non-numeric values x the operation will raise a* `ClassCastException`*.*

```
/*- pre-era-apply (4) */
  case Apply(fn, args) if (fn.symbol == Any_asInstanceOf) =>
      (fn: @unchecked) match {
```

```
      case TypeApply(Select(qual, _), List(targ)) =>
        if (qual.tpe <:< targ.tpe)
          /*- eliding a type test statically known to succeed */
          atPos(tree.pos) { Typed(qual, TypeTree(targ.tpe)) }
        else if (isNumericValueClass(qual.tpe.typeSymbol) && isNumericValueClass(targ.tpe.typeSymbol))
          /*- numeric-conversion semantics */
          atPos(tree.pos)(numericConversion(qual, targ.tpe.typeSymbol))
        else
          /*- let the backend enforce semantics :-) */
          tree
    }
```

2.2.5 Reduce type-testing for singleton and for refined types

Type tests on generic arrays are matched by a preceding `case` clause (Sec. 2.2.1). All other `Any_isInstanceOf` tests are matched here, but only some are rewritten:

- type tests against singleton types are rewritten into equality tests:

```
// todo: also handle the case where the singleton type is buried in a compound
case SingleType(_, _) | ThisType(_) | SuperType(_, _) =>

  val cmpOp = if (targ.tpe <:< AnyValClass.tpe) Any_equals else Object_eq

  atPos(tree.pos) {
    Apply(Select(qual, cmpOp), List(gen.mkAttributedQualifier(targ.tpe)))
  }
```

- Intersection types of the `RefinedType` variety involve individual type-tests some of which may be statically known to succeed, those are elided, the remaining ones are rewritten from `Any_isInstanceOf` to use `Object_isInstanceOf` instead.

  – The differences between those methods can be seen in Sec. 4.2.

  – `adaptMember()` performs that rewriting a lot (Sec. 1.2.2).

```
case RefinedType(parents, decls) if (parents.length >= 2) =>
  // Optimization: don't generate isInstanceOf tests if the static type
  // conforms, because it always succeeds. (Or at least it had better.)
  // At this writing the pattern matcher generates some instance tests
  // involving intersections where at least one parent is statically known true.
  // That needs fixing, but filtering the parents here adds an additional
  // level of robustness (in addition to the short term fix.)
  val parentTests = parents filterNot (qual.tpe <:< _)

  if (parentTests.isEmpty) Literal(Constant(true))
  else gen.evalOnce(qual, currentOwner, unit) { q =>
    atPos(tree.pos) {
      parentTests map mkIsInstanceOf(q) reduceRight gen.mkAnd
    }
  }
```

- otherwise leave the type-test as-is.

2.2.6 ApplyDynamic

Make dynamic applications easier to detect by wrapping them in a dedicated node, removing any type application in the process. `cleanup` will lower them into cache-supported reflective calls.

```
/*- pre-era-apply (6) */
  case Apply(fn, args) =>
  {
        def doDynamic(fn: Tree, qual: Tree): Tree = {
          if (fn.symbol.owner.isRefinementClass && !fn.symbol.isOverridingSymbol)
            ApplyDynamic(qual, args) setSymbol fn.symbol setPos tree.pos
          else tree
        }

    fn match {
      case Select(qual, _) => doDynamic(fn, qual)
      case TypeApply(fni@Select(qual, _), _) =>
        doDynamic(fni, qual)// type parameters are irrelevant in case of dynamic call
      case _ => tree
    }
  }
```

2.3 Explore whether some Apply rewritings can be performed before erasure

TODO

It may be useful to explore whether some of the rewritings for `Apply` nodes can be performed in a phase of its own before `erasure`, i.e. in a phase whose typing doesn't rely on `Erasure.transformInfo`.

From the six rewritings listed in Sec. 2.2, three are clear candidates (listed below). Perhaps there are more.

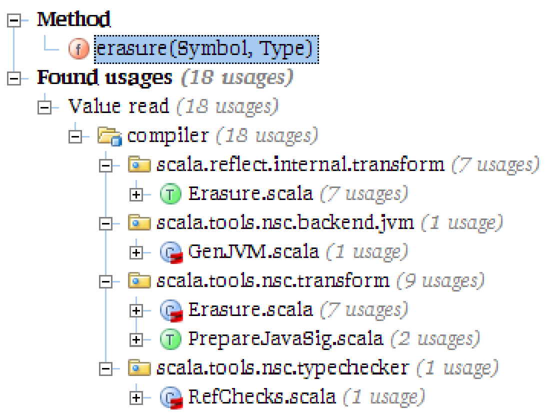
- Two methods on `Any` and `Object` (`##` and `getClass`) are rewritten while we still know what is a primitive and what arrived boxed. Sec. 2.2.3
- (a) Elide a type cast statically known to succeed, or
  (b) Enforce numeric-conversion semantics. Sec. 2.2.4
- (a) Rewrite type tests against singleton types, or
  (b) Optimize type tests against `RefinedType`. Sec. 2.2.5

Caveat: in two occassions `preEraseApply` sets erased types on trees it emits. If done before `erasure`, non-erased types should be used instead.

3 Putting it all together: type erasure for JVM

Computing a `forJVM`-erased-type is like a pipeline:

1. `def transformInfo(sym: Symbol, tp: Type): Type`
   An `InfoTransform` override in `Erasure`. It interceps a few cases (Sec. 3.1) (thus special casing them) otherwise delegates to method `erasure()`.



```
Method
  erasure(Symbol, Type)
Found usages (18 usages)
  Value read (18 usages)
    compiler (18 usages)
      scala.reflect.internal.transform (7 usages)
        Erasure.scala (7 usages)
      scala.tools.nsc.backend.jvm (1 usage)
        GenJVM.scala (1 usage)
      scala.tools.nsc.transform (9 usages)
        Erasure.scala (7 usages)
        PrepareJavaSig.scala (2 usages)
      scala.tools.nsc.typechecker (1 usage)
        RefChecks.scala (1 usage)
```

Figure 5: Sec. 3

2. `def erasure(sym: Symbol, tp: Type): Type`
   a helper method in `Erasure` that picks either a Java-aware or Scala's own way of finding an *intersection dominator*. Details in Sec. 3.2. Unlike the two helper-helpers below, this method is also used far afield from the `erasure` phase (Figure 5).

3. `javaErasure` and `scalaErasure` are two `TypeMap` objects that help the helper above (and are used by nobody else). Each of them customizes how an *intersection dominator* is computed given a list of types. The functionality they have in common (catalogued in Sec. 3.3) constitutes the bulk of "type erasure for JVM".

3.1 Stage 1: `transformInfo`

Because of the pre-erasing rewritings performed by `Erasure.PreTransformer` some symbols won't show up by the time `transformInfo()` is invoked (for example, there's nomore `Any_isInstanceOf` but only `Object_isInstanceOf`).

`transformInfo()` special-cases erasure as follows:

- type tests and casts keep their type-arg, albeit erased.
- `scala.Array` gets as `info` a new type constructor based on the previous one, but with an erased `resultType` (example shown in Sec. 4.5).
- a type var gets a `TypeBounds` `info` with `WildcardType` bounds.
- Members of `ArrayClass` have the component type erased in their `info`.

Details about the special-casing appear as inline comments:

```
def transformInfo(sym: Symbol, tp: Type): Type = {
  if (sym == Object_asInstanceOf)
    /*- For $asInstanceOf, unchanged i.e. [T]T */
    sym.info

  else if (sym == Object_isInstanceOf || sym == ArrayClass)
    /*- For $isInstanceOf    : [T]scala#Boolean */
    /*- For class Array      : [T]C where C is the erased classinfo of the Array class. */
    PolyType(sym.info.typeParams, erasure(sym, sym.info.resultType))

  else if (sym.isAbstractType)
    /*- For a type parameter : A type bounds type consisting of the erasures of its bounds. (BUT SEE BELOW) */
    TypeBounds(WildcardType, WildcardType)
    /*- TODO: However the comment for WildcardType reads:
          "An object representing an unknown type, used during type inference.
           If you see WildcardType outside of inference it is almost certainly a bug." */

  else if (sym.isTerm && sym.owner == ArrayClass) {

    if (sym.isClassConstructor) /*- For Array[T].<init> : (Int)Array[T] */
      tp match {
        case MethodType(params, TypeRef(pre, sym1, args)) =>
          MethodType(cloneSymbols(params) map (p => p.setInfo(erasure(sym, p.tpe))),
                     typeRef(erasure(sym, pre), sym1, args))
                   /*- TODO: why not erasure(pre.typeSymbol, pre) ?. */
      }
    else if (sym.name == nme.apply) /*- For Array[T].apply(Int):T : unchanged */
      tp
    else if (sym.name == nme.update)
      /*- For Array[T].update(Int, T):Unit preserve type-var in MethodType, erase the rest. */
      (tp: @unchecked) match {
        case MethodType(List(index, tvar), restpe) =>
          MethodType(List(index.cloneSymbol.setInfo(erasure(sym, index.tpe)), tvar),
                     erasedTypeRef(UnitClass))
      }
    else erasure(sym, tp) /*- For clone() and length on Array, erase */

  } else if (
    sym.owner != NoSymbol &&
    sym.owner.owner == ArrayClass &&
    sym == Array_update.paramss.head(1)) {
    /*- special case for the 2nd formal value param of Array.update: its type-var info remains non-erased.
        I.e. the type-var ‘A‘ of ArrayClass’s ‘update‘ (to recap, the info of that member is ‘(Int,A)Unit‘).
        The symbol of that formal has to be caught here since the erasure type map gets applied to every symbo
    tp
  } else {
    erasure(sym, tp)
  }
}
```

Additionally, *in my codebase* (where `AddInterfaces` is a phase of its own running right before `erasure`), there's an override in `class Erasure` (the list above refers to the overridden method in `trait Erasure`) as depicted in Figure 6.

3.2 Stage 2: `def erasure(sym: Symbol, tp: Type): Type`

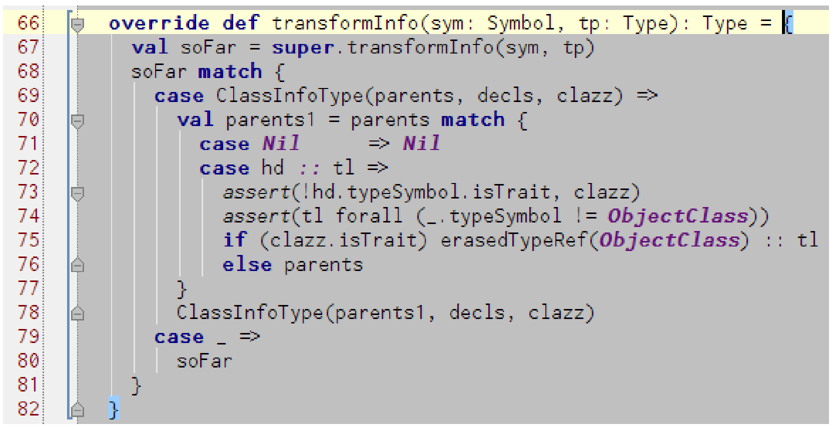
The helper method shown in Figure 7 is heavily used in `Erasure` and elsewhere (Figure 5). It picks either a Java-aware or Scala's own way of finding an *intersection dominator.*

3.3 Stage 3: the `ErasureMap` `TypeMap`

In this section, recursive invocations of "erasure" refer to what the method `erasure()` (Sec. 3.2) does, not what `transformInfo` (Sec. 3.1) does.

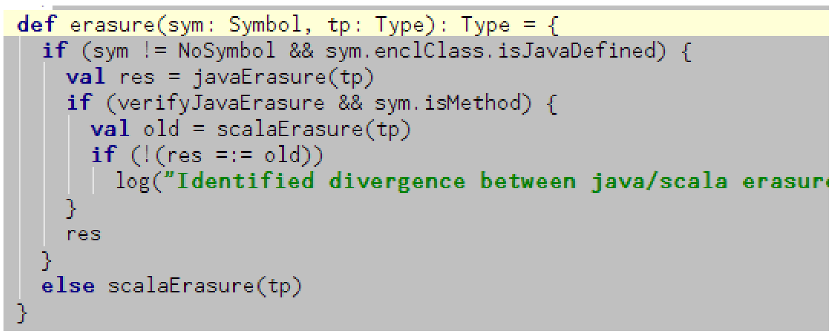
The erasure `|T|` of a type `T` is:

1. For a constant type, itself.



```
override def transformInfo(sym: Symbol, tp: Type): Type = {
  val soFar = super.transformInfo(sym, tp)
  soFar match {
    case ClassInfoType(parents, decls, clazz) =>
      val parents1 = parents match {
        case Nil      => Nil
        case hd :: tl =>
          assert(!hd.typeSymbol.isTrait, clazz)
          assert(tl forall (_.typeSymbol != ObjectClass))
          if (clazz.isTrait) erasedTypeRef(ObjectClass) :: tl
          else parents
      }
      ClassInfoType(parents1, decls, clazz)
    case _ =>
      soFar
  }
}
```

Figure 6: Sec. 3.1



```
def erasure(sym: Symbol, tp: Type): Type = {
  if (sym != NoSymbol && sym.enclClass.isJavaDefined) {
    val res = javaErasure(tp)
    if (verifyJavaErasure && sym.isMethod) {
      val old = scalaErasure(tp)
      if (!(res =:= old))
        log("Identified divergence between java/scala erasur
    }
    res
  }
  else scalaErasure(tp)
}
```

Figure 7: Sec. 3.2

2. For every other singleton type, the erasure of its supertype.

3. For array typerefs, as follows. When the typeref is to:

   (a) `scala.Array[T]` where `T` is an abstract type, `AnyRef`.

   (b) `scala.Array[Nothing]` or `scala.Array[Null]`, `Array[AnyRef]`

   (c) otherwise, `Array[|T|]`.

4. For other (non-array) typerefs, as follows. When the typeref is to:

   (a) `Any`, `AnyVal`, `scala.Singleton`, or `scala.NotNull`, its erasure is `AnyRef`.

   (b) `Unit`, its erasure is `scala.runtime.BoxedUnit`.

   (c) `P.C[Ts]` where `C` refers to a class, its erasure is `|P|.C`. (where `P` is first rebound to the class that directly defines `C`, see ticket 2585)

   (d) a non-empty type intersection (possibly with refinement)

      - in Scala, its erasure is that of the intersection dominator
      - in Java, its erasure is that of its first parent

   (e) `else apply(sym.info)` *`//alias type or abstract type`*

```
TODO looks like the above stands for:

   For a typeref C that refers to:
      - an alias type,  the erasure of the alias    of C.
      - an abstract type, the erasure of the upper bound of C.

   What about variance, ie under which circumstances does it make sense
     ‘‘the erasure of the lower bound of C’’?
```

5. For “quantified types”:

   (a) For a polymorphic type, the erasure of its result type.

   (b) For an existential type, the erasure of its result type.

6. For method types:

   (a) For a method type `(Fs)scala.Unit`, `(|Fs|)scala#Unit`
   (otherwise the result type would be rewritten as for a typeref to `Unit`).

   (b) For any other method type `(Fs)T`, `(|Fs|)|T|`.

7. For a type intersection (possibly with refinement)

   (a) Non-empty: In Scala, the erasure of the intersection dominator. In Java, the erasure of the first parent.

   (b) Empty: `java.lang.Object` (because the intersection dominator of `Nil` is `AnyRef`)

8. For an annotated type, the erasure of its underlying type (where underlying is the type without the annotation)

9. For the classinfo type of

(a) `java.lang.Object` or a Scala value class, the same type without any parents.

(b) `Array`, the same type with only `AnyRef` as parent.

```
BTW, before erasure, the info of ArrayClass is
   [T]java.lang.Object with java.io.Serializable with java.lang.Cloneable{ <...decls...> }
ie a PolyType with resultType of ClassInfoType
```

(c) any other classinfo type with parents `Ps`, the same type with parents `|Ps|`, but with duplicate references of `Object` removed.

10. for all other types, the type itself (with any sub-components erased)

3.4 Stage 4: the `Eraser` custom modifier typer

That's the topic of Sec. 1.

4 Background

4.1 The `transformInfo` story

What difference does a `transformInfo` override make? It's the Hollywood principle at its best: we never invoke it directly, but ends up being invoked under `rawinfo`. Listing 8 on p. 21 shows an anonymous class of `InfoTransformer` with a method override that invokes `transformInfo` on its outer instance (an `InfoTransform`).

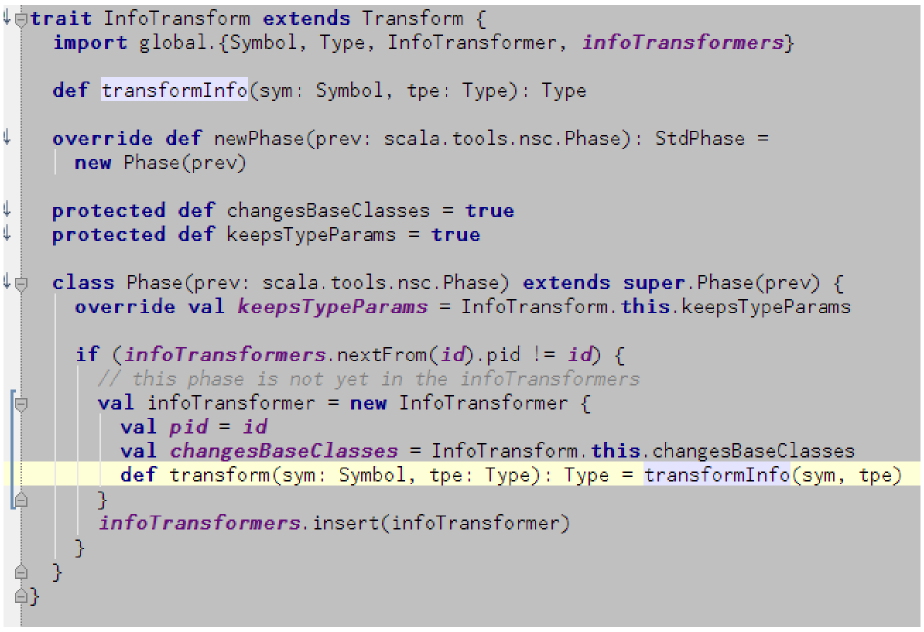
As an appetizer, here goes the comment for `InfoTransform`:

```
/**
 * An InfoTransform contains a compiler phase that transforms
 * trees and symbol infos -- making sure they stay consistent.
 * The symbol info is transformed assuming it is consistent right before this phase.
 * The info transformation is triggered by Symbol::rawInfo,
 * which caches the results in the symbol's type history.
 * This way sym.info (during an atPhase(p)) can look up what the symbol's info
 * should look like at the beginning of phase p.
 * (If the transformed info had not been stored yet,
 * rawInfo will compute the info by composing the info-transformers
 *  of the most recent phase before p, up to the transformer of the phase right before p.)
 *
 * Concretely, atPhase(p) { sym.info } yields the info *before* phase p has transformed it.
 * Imagine you're a phase and it all makes sense.
 */
```

4.2 The different methods for type testing

`Object_isInstanceOf` and `Object_asInstanceOf` are synthetic methods. Under the hood:

```
lazy val Object_isInstanceOf = newPolyMethod(
     ObjectClass,
     "$isInstanceOf",
     tparam => MethodType(List(), booltype)
  ) setFlag (FINAL | SYNTHETIC)
```



```
trait InfoTransform extends Transform {
  import global.{Symbol, Type, InfoTransformer, infoTransformers}

  def transformInfo(sym: Symbol, tpe: Type): Type

  override def newPhase(prev: scala.tools.nsc.Phase): StdPhase =
    new Phase(prev)

  protected def changesBaseClasses = true
  protected def keepsTypeParams = true

  class Phase(prev: scala.tools.nsc.Phase) extends super.Phase(prev) {
    override val keepsTypeParams = InfoTransform.this.keepsTypeParams

    if (infoTransformers.nextFrom(id).pid != id) {
      // this phase is not yet in the infoTransformers
      val infoTransformer = new InfoTransformer {
        val pid = id
        val changesBaseClasses = InfoTransform.this.changesBaseClasses
        def transform(sym: Symbol, tpe: Type): Type = transformInfo(sym, tpe)
      }
      infoTransformers.insert(infoTransformer)
    }
  }
}
```

Figure 8: Sec. 4.1

```
lazy val Object_asInstanceOf = newPolyMethod(
      ObjectClass,
      "$asInstanceOf",
      tparam => MethodType(List(), tparam.typeConstructor)
  ) setFlag (FINAL | SYNTHETIC)
```

The differences with their `Any` counterparts are shown below (in all cases a type param is taken, the differences involve names and being zero-arg vs. no-arg):

```
  Any_isInstanceOf = newPolyMethod(
      AnyClass,
      nme.isInstanceOf_,
      tparam => NullaryMethodType(booltype)
  ) setFlag FINAL

  Any_asInstanceOf = newPolyMethod(
      AnyClass,
      nme.asInstanceOf_,
      tparam => NullaryMethodType(tparam.typeConstructor)
  ) setFlag FINAL
```

4.3 Type testing on member types

```
TODO Is the above accounted for in Typers, in Erasure, where?
```

Quoting from the SLS, §12.1:

The type test `x.isInstanceOf[T]` is equivalent to a typed pattern match

```
x match {
  case _: T' => true
  case _ => false
}
```

where the type `T'` is the same as `T` except if `T` is of the form `D` or `D[tps]` where `D` is a type member of some outer class `C`. In this case `T'` is `C#D` (or `C#D[tps]`, respectively) whereas T itself would expand to `C.this.D[tps]`. In other words, an `isInstanceOf` test does not check for the

4.4 What `PolyType` means

- A polymorphic nullary method
  (e.g, `Any_isInstanceOf`, whose def is akin to "`def isInstanceOf[T]: Boolean`")
  is represented as `PolyType(tps, NullaryMethodType(restpe))`
- A polymorphic (non-nullary) method (e.g, `Object_isInstanceOf`)
  is represented as `PolyType(tps, MethodType(restpe))`
- A `PolyType(tps, TypeRef(...))` indicates a type function
  (which results from eta-expanding a type constructor alias).
- `PolyType(tps, ClassInfoType(...))` is a type constructor.

Some invariants: no `NoSymbol` among `typeParams`, `typeParams` is not empty.

4.5 Examples (before erasure)

- The info of `ArrayClass` is

  ```
  [T]java.lang.Object with java.io.Serializable with java.lang.Cloneable{ <...decls...> }
  ```

  ie a `PolyType` with `resultType` of `ClassInfoType`.
- The info of `Object_asInstanceOf` is "`[T]T`" internally represented as:

  ```
  a PolyType whose resultType is a
      MethodType with
          Nil params and a
          UniqueTypeRef(NoPrefix, <type-var>, Nil) as resultType
            (where <type-var> is a TypeSymbol).
  ```

4.6 Intersection dominator

Quoting from the source code:

> *The intersection dominator (SLS 3.7) of a list of types is computed as follows.*
>
> 1. *If the list contains one or more occurrences of `scala.Array` with type parameters `El1, El2, ...` then the dominator is `scala.Array` with type parameter of `intersectionDominator(List(El1, El2, ...))`.*

2. *Otherwise, the list is reduced to a subsequence containing only types which are not subtypes of other listed types (the span.)*
3. *If the span is empty, the dominator is `Object`.*
4. *If the span contains a class `Tc` which is not a trait and which is not `Object`, the dominator is `Tc`.*
5. *Otherwise, the dominator is the first element of the span.*

4.7 Examples (after erasure)

```
def doc(tpe: Type): String = {
  val buf = new java.io.StringWriter()
  treeBrowsers.TypePrinter.toDocument(tpe).format(40, buf)
  buf.toString()
}
```

Let's record the input-output pairs to `ErasureMap.apply()`, to later display them (in the example, on entry to `constructors`). A few examples:

```
-----------------
0:      <root>#test#Test#Ti
->      <root>#test#Test#Ti
-----------------
2:      Test.this.Bar
->      <root>#test#Test#Bar
-----------------
3:      (<param> arg$outer: Test.this.type)Test.this.Foo
->      (<param> arg$outer: <root>#test#Test)<root>#test#Test#Foo
-----------------
5:      ()lang.this.Class[?0] forSome { <deferred> <existential/mixedin> type ?0 }
->      ()<root>#java#lang#Class
-----------------
6:      ()lang.this.String
->      ()<root>#java#lang#String
-----------------
7:      scala.this.Unit
->      <root>#scala#runtime#BoxedUnit
-----------------
27:     <notype>
->      <notype>
-----------------
28:     lang.this.Class[?0]
->      <root>#java#lang#Class
-----------------
31:     scala.this.Int
->      <root>#scala#Int
-----------------
```

Many examples involve moving away from path-dependent types[2].

[2] `http://weblogs.java.net/blog/cayhorstmann/archive/2011/08/05/inner-classes-scala-and-java`