`cleanup` of references to structural members

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

August 30th, 2011

Abstract

`cleanup` is the last phase that transforms `Tree` nodes, before `GenICode` takes over and produces ICode, a stack-based IR.

```
      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

1 Motivation **3**
- 1.1 How `ApplyDynamic` nodes come to being in `erasure` 3

2 Type rewriting **4**
- 2.1 Adding private members 4
- 2.2 Which `Typer` to use 4

3 Two schools of thought on dynamic invocation **5**
- 3.1 Platform-specific mechanism for dynamic invocation 6

4 Term rewriting for `ApplyDynamic` **6**
- 4.1 Possibly an array 7
- 4.2 Possibly a primitive 8

5 Default case **8**
- 5.1 No caching 10
- 5.2 Single-slot most-recently-used caching 10
- 5.3 Caching for all classes seen for receiver 11
- 5.4 `Trees` for caches 11

6 Other rewritings **12**
- 6.1 VM-reflected classes for Scala class literals of value classes, including for the `Unit` class 12
- 6.2 Emptying the stack at the end of a try-block 13
- 6.3 Caching of interned symbol literals 13
- 6.4 Wrapped arrays turned into normal arrays 15

7 Type parameters and Structural types **16**

8 Scala.NET **17**
- 8.1 Non-fixed types of formals 17
  - 8.1.1 Translation `forJVM` 18
- 8.2 Static caches 18
- 8.3 Class literals of value classes 19
- 8.4 Arrays 19

1 Motivation

Quoting from the SLS [2, §3.2.7]:

> *A compound type* T_1 `with` *...*`with` T_n { `R` } *represents objects with members as given in the component types* T_1, ..., T_n *and the refinement* { `R` }. *A refinement* { `R` } *contains declarations and type definitions. If a declaration or definition overrides a declaration or definition in one of the component types* T_1, ..., T_n, *the usual rules for overriding (§5.1.4) apply; otherwise the declaration or definition is said to be "structural". ... A reference to a structurally defined member (method call or access to a value or variable) may generate binary code that is significantly slower than an equivalent code to a non-structural member.*

Before arriving at this phase, `ApplyDynamic` nodes were last touched by `erasure` (Sec. 1.1). Now `cleanup` has to figure out a way to efficiently implement the `ApplyDynamic` semantics, which are:

> *runtime method lookup + reflective invocation, where method lookup is based on (a) the dynamic type of the receiver and (b) static types of the formals at the callsite.*

Runtime caches [1] are the best solution in almost all cases. We review how code is emitted for that in Sec. 4. The remaining term rewritings in `cleanup` don't have anything to do with `ApplyDynamic` nor with each other (Sec. 6).

1.1 How `ApplyDynamic` nodes come to being in `erasure`

`Eraser` is a custom typer that receives "pre-erased" trees, i.e. trees that in general have null `tpe` and have been possible transformed by `PreTransformer`. One such pre-transform substitutes `ApplyDynamic` nodes for `Apply` nodes denoting a structural reference (call or access) as follows:

```
/*- pre-era-apply (6) */
  /**
   * 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.
   *
   */
  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 Type rewriting

2.1 Adding private members

Because `cleanup` adds private class members only, it gets by without being an `InfoTransform`. However, it does mutate `Type`s in symbols' `info`s (adding to `info.decls`, but never removing or altering an entry). For example:

```
/*- during addStaticVariableToClass(), as part of creating the method cache. */
currentClass.info.decls enter varSym

/*- during addStaticMethodToClass(), as part of creating the method cache. */
currentClass.info.decls enter methSym

/*- during getSymbolStaticField(),
    caching an interned symbol for lock-free access after the first access. */
currentClass.info.decls enter stfieldSym
```

2.2 Which `Typer` to use

The only transformer in this phase (`CleanUpTransformer`) doesn't subclass `TypingTransformer`, keeping track instead of a transformer-local `Typer` instance (called `localTyper`), obtained from `global.analyzer.typer` as shown below:

```
private var localTyper: analyzer.Typer = null
. . .
override def transform(tree: Tree): Tree = tree match {
    . . .
    case Template(parents, self, body) =>
        localTyper = typer.atOwner(tree, currentClass)
```

The thus managed transformer-local `Typer` is the right one most of the time, however in one occassion (shown below) the right one is `typer.atOwner(tree, currentClass)`:

```
/* Returns the symbol and the tree for the symbol field interning a reference to a symbol 'synmname'.
 * If it doesn't exist, i.e. the symbol is encountered the first time,
 * it creates a new static field definition and initialization and returns it.
 */
private def getSymbolStaticField(pos: Position, symname: String, rhs: Tree, tree: Tree): Symbol =
  symbolsStoredAsStatic.getOrElseUpdate(symname, {
    val theTyper = typer.atOwner(tree, currentClass) /*- now this Typer is needed, not localTyper. */
    . . .
```

Instead of managing by itself the typer to use (as `CleanUpTransformer` does) a transformer extending the abstract class `TypingTransformer` gets that for free (the typer is updated upon visiting a `Template` or a `PackageDef`), due to the interplay of the `transform` override and the `atOwner` method overloads in `TypingTransformer`:

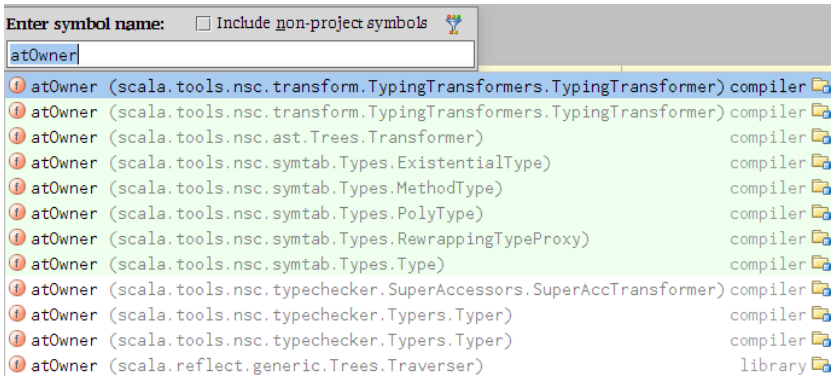
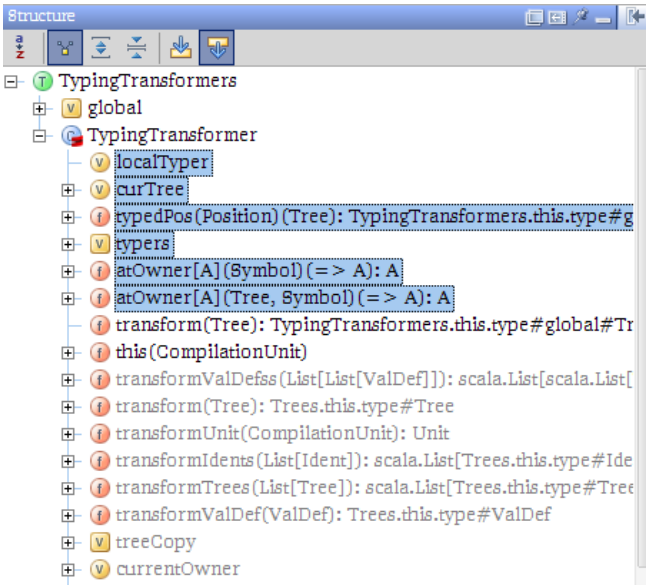


Figure 1: Sec. 2



If this is the first time you hear about `atOwner`, perhaps you should also know that there are many of them, some returning a `Typer` (e.g. the overloads in `Typers#Typer`), others `Unit` (e.g. in `Traverser`), and yet others whatever its by-name param evaluates to (e.g. in `Transformer`). Please fest on the details (Figure 1).

3 Two schools of thought on dynamic invocation

Most of the brainpower of `cleanup` goes to lowering references to structural members (this section and Sec. 4), but the `transform()` override in `CleanUpTransformer` also performs other rewritings (Sec. 6).

When lowering `ApplyDynamic` nodes, there's a main choice between:

1. VM-specific target code, Sec. 3.1

2. a more portable mechanism, which uses the appropriate reflection API (JDK or .NET), Sec. 4.

That "main choice" is made within the "`case ad@ApplyDynamic(qual0, params)`" clause of `CleanUpTransformer`'s `transform()` method (Listing 3).

The first mechanism above is picked via "`invoke-dynamic`". All other options below pick the second mechanism:

```
val refinementMethodDispatch =
     ChoiceSetting ("-Ystruct-dispatch", "policy", "structural method dispatch policy",
                    List("no-cache", "mono-cache", "poly-cache", "invoke-dynamic"),
                    "poly-cache")
```

3.1 Platform-specific mechanism for dynamic invocation

It's usually hard to resist the "solution looking for problem" bug. Initially, `cleanup` performed its magic using only the reflection API of the target platform, yet preserving semantics across platforms. We call this magic the "platform-independent mechanism" (Sec. 4).

However, once `InvokeDynamic` on the JVM (and the *Dynamic Language Runtime* on the CLR) appeared on scene, motivation was there for `cleanup` to leave `ApplyDynamic` as-is (Listing 1), letting `GenICode` translate it into a VM-specific version of "dynamic invocation".

In the case of JVM:

```
  case ApplyDynamic(qual, args) =>
    assert(!forMSIL)
    ctx.clazz.bootstrapClass = Some("scala.runtime.DynamicDispatch")
    val ctx1 = genLoad(qual, ctx, ObjectReference)
    genLoadArguments(args, tree.symbol.info.paramTypes, ctx1)
    ctx1.bb.emit(CALL_METHOD(tree.symbol, InvokeDynamic), tree.pos)
    ctx1
```

```
TODO How does the above fare when:
     - the receiver is an array (array-op case, as well as maybe-array case),
     - the receiver is a primitive value (primitive-op case, as well as maybe-primtive case)
```

4 Term rewriting for `ApplyDynamic`

Inside the `else` branch in Listing 1 can be found the entry point to all-things platform-independent rewriting of structural calls: `callAsReflective(List[Type], Type)`. Falling under this category:

1. the receiver is possibly an array (runtime info needed), Sec. 4.1.
2. the receiver is possibly a primitive value (runtime info needed), Sec. 4.2.
3. default case, Sec. 5. For this case (only), and depending on the caching strategy, the emitted code is one of:
   (a) No caching, Sec. 5.1.

Listing 1: Two mechanisms to lower structural calls: platform-dependent (Sec. 3.1) and portable (Sec. 4)

```
if (settings.refinementMethodDispatch.value == "invoke-dynamic") {
  localTyper.typed(treeCopy.ApplyDynamic(ad, transform(qual), transformTrees(params)))
}
else {

  /* ### BODY OF THE TRANSFORMATION -> remember we're in case ad@ApplyDynamic(qual, params) ### */

  val t: Tree = ad.symbol.tpe match {
    case MethodType(mparams, resType) =>
      assert(params.length == mparams.length)
      typedPos {
        val sym = currentOwner.newValue(ad.pos, mkTerm("qual")) setInfo qual0.tpe
        qual = safeREF(sym)

        BLOCK(
          VAL(sym) === qual0,
          callAsReflective(mparams map (_.tpe), resType)
        )
      }
  }

  /* We return the dynamic call tree, after making sure no other
   * clean-up transformation are to be applied on it. */
  transform(t)
}
```

(b) Single-slot most-recently-used caching, Sec. 5.2.

(c) Caching for all actual types seen for receiver, Sec. 5.3.

Summing up, `callAsReflective()` makes a choice as shown below, because different code has to be emitted to operate on a primitive vs. an array vs. an object:

```
localTyper typed (
  if (isMaybeBoxed && isJavaValueMethod) genValueCallWithTest
  else if (isArrayMethodSignature && isDefinitelyArray) genArrayCall
  else if (isArrayMethodSignature && isMaybeArray) genArrayCallWithTest
  else genDefaultCall
)
```

4.1 Possibly an array

We arrived here taking one of the branches in Sec. 4.

The entry points for this case are `genArrayCallWithTest()` and `genArrayCall()`:

```
/** A conditional Array call, when we can't determine statically if the argument is
 *  an Array, but the structural type method signature is consistent with an Array method
 *  so we have to generate both kinds of code.
 */
def genArrayCallWithTest =
  IF ((qual GETCLASS()) DOT nme.isArray) THEN genArrayCall ELSE genDefaultCall
```

```
/** A native Array call. */
def genArrayCall = fixResult(
  methSym.name match {
    /*- let's keep in mind that 'args' was defined as 'qual :: params' */

    case nme.length =>
      /*- scala.Int.box( scala.runtime.ScalaRunTime.array_length(<qual>) ) */
      REF(boxMethod(IntClass)) APPLY (REF(arrayLengthMethod) APPLY args)

    case nme.update =>
      /*- scala.runtime.ScalaRunTime.array_update(
            <qual>,
            scala.Int.unbox(<params(0)>),
            <params(1)>
          ) */
      REF(arrayUpdateMethod) APPLY List(args(0), (REF(unboxMethod(IntClass)) APPLY args(1)), args(2))

    case nme.apply =>
      /*- scala.runtime.ScalaRunTime.array_apply(
            <qual>,
            scala.Int.unbox(<params(0)>)
          ) */
      REF(arrayApplyMethod) APPLY List(args(0), (REF(unboxMethod(IntClass)) APPLY args(1)))

    case nme.clone_ =>
      /*- scala.runtime.ScalaRunTime.array_clone( <qual> ) */
      REF(arrayCloneMethod) APPLY List(args(0))
  },
  mustBeUnit = methSym.name == nme.update
)
```

4.2 Possibly a primitive

We arrived here taking one of the branches in Sec. 4.

The entry point for this case is `genValueCallWithTest()` and `genValueCall()`:

```
def genValueCallWithTest = {
  val (operator, test) = getPrimitiveReplacementForStructuralCall(methSym.name)
  IF (test) THEN genValueCall(operator) ELSE genDefaultCall
}

/** A possible primitive method call, represented by methods in BoxesRunTime. */
def genValueCall(operator: Symbol) = fixResult(REF(operator) APPLY args)
```

5 Default case

We arrived here taking one of the branches in Sec. 4.

The entry point for this case is `genDefaultCall()` (Listing 2).

Forgetting about caching for a moment, the tree emitted for the "default case" is of the form "*`getClass()` on the receiver, then `getMethod()` on that class, then `invoke()` on that method*", but in fact that's the "default" case.

There's a minor variation in the emitted code when the method to invoke returns a primitive value (because erasure consistently adapts `ApplyDynamic` nodes by adding boxing around them). Quoting from source:

Listing 2: Sec. 5

```
def genDefaultCall = {

  /*- symbol for
        invoke() in j.l.reflect.Method
      or
        Invoke() in System.Reflection.MethodInfo
   */
  val invokeName = MethodClass.tpe member nme.invoke_

  /*- tree denoting whatever cache is supposed to be used at runtime (including none). */
  def cache      = safeREF(reflectiveMethodCache(ad.symbol.name.toString, paramTypes))

  /*- tree denoting a Method object, retrieved from cache
      using as key the actual class of the receiver of the ApplyDynamic. */
  def lookup     = Apply(cache, List(qual GETCLASS))

  /*- tree denoting an object array,
      whose elems are the actual args of the ApplyDynamic. */
  def invokeArgs = ArrayValue(TypeTree(ObjectClass.tpe), params)

  /*- tree denoting reflective invocation. */
  def invocation = (lookup DOT invokeName)(qual, invokeArgs)

  /*- trees to piece together another tree,
        try { method.invoke } catch { case e: InvocationTargetExceptionClass => throw e.getCause() }
      which constitute the return value for genDefaultCall()
   */
  val invokeExc  = currentOwner.newValue(ad.pos, mkTerm("")) setInfo InvocationTargetExceptionClass.tpe
  def catchVar   = Bind(invokeExc, Typed(Ident(nme.WILDCARD), TypeTree(InvocationTargetExceptionClass.tpe)))
  def catchBody  = Throw(Apply(Select(Ident(invokeExc), nme.getCause), Nil))

  /*- wrapping (or not) the try expr via fixResult(). */
  fixResult(TRY (invocation) CATCH { CASE (catchVar) ==> catchBody } ENDTRY)
}
```

> *`invoke()` needs an array of `AnyRefs` that are the method's arguments. The erasure phase guarantees that any parameter passed to a dynamic apply is compatible (through boxing). Boxed `Ints` et al. is what `invoke()` expects when the applied method expects `Ints`, hence no change needed there.*
>
> *In the end, the result of `invoke()` must be fixed, again to deal with arrays. This is provided by `fixResult()`. `fixResult()` will cast the invocation's result to the method's return type, which is generally ok, except when this type is a value type (`Int` et al.) in which case it must [be boxed] because ... erasure made sure the result is expected to be an `AnyRef`. [in contrast, indexing an array of primitives leaves a primitive on the operand stack]*

Details:

```
def fixResult(tree: Tree, mustBeUnit: Boolean = false) =
  if (mustBeUnit || resultSym == UnitClass) BLOCK(tree, REF(BoxedUnit_UNIT)) // boxed unit
  else if (resultSym == ObjectClass) tree                                    // no cast necessary
  else tree AS_ATTR boxedResType                                             // cast to expected type
```

Listing 3: Main transform method, Sec. 3 and Sec. 5

```
override def transform(tree: Tree): Tree = tree match {
  . . .
  case ad@ApplyDynamic(qual0, params) =>
  . . .
  case Template(parents, self, body) =>
  . . .
```

Bringing back caching into the picture: the lowering for the "default case" comprises actually the two case handlers shown in Listing 3:

1. The first `case` clause expands the `ApplyDynamic` node proper. It's here choices are made:
   (a) VM-dependent or not, Sec. 3
   (b) for the latter between primitive, array, or object receiver, Sec. 4
   (c) and finally for the last one between caching strategy, Sec. 5.
2. The second `case` clause adds `Tree`s for the caches to support method lookup. These `Tree`s (if any) were built by the previous rewriting Sec. 5.4.

5.1 No caching

There's no cache (in the sense of a cache indexed by the actual type of the receiver) as in `mono-cache` or `poly-cache`. Instead, a helper method is emitted, which in turn accesses a pre-computed array (containing types of formals) and the static field holding this array is called (somewhat confusingly) "`reflParams$Cache`".

```
  case NO_CACHE =>

      /* Implementation of the cache is as follows for method "def xyz(a: A, b: B)":

        var reflParams$Cache: Array[Class[_]] = Array[JClass](classOf[A], classOf[B])

        def reflMethod$Method(forReceiver: JClass[_]): JMethod =
          forReceiver.getMethod("xyz", reflParams$Cache)

      */
```

5.2 Single-slot most-recently-used caching

```
case MONO_CACHE =>

  /* Implementation of the cache is as follows for method "def xyz(a: A, b: B)"
     (but with a SoftReference wrapping reflClass$Cache, similarly in the poly Cache) :

    var reflParams$Cache: Array[Class[_]] = Array[JClass](classOf[A], classOf[B])

    var reflMethod$Cache: JMethod = null
```

```
  var reflClass$Cache: JClass[_] = null

  def reflMethod$Method(forReceiver: JClass[_]): JMethod = {
    if (reflClass$Cache != forReceiver) {
      reflMethod$Cache = forReceiver.getMethod("xyz", reflParams$Cache)
      reflClass$Cache = forReceiver
    }
    reflMethod$Cache
  }

*/
```

5.3 Caching for all classes seen for receiver

```
case POLY_CACHE =>

  /* Implementation of the cache is as follows for method "def xyz(a: A, b: B)"
     (SoftReference so that it does not interfere with classloader garbage collection, see ticket
     #2365 for details):

    var reflParams$Cache: Array[Class[_]] = Array[JClass](classOf[A], classOf[B])

    var reflPoly$Cache: SoftReference[scala.runtime.MethodCache] = new SoftReference(new EmptyMethodCache())

    def reflMethod$Method(forReceiver: JClass[_]): JMethod = {
      var method: JMethod = reflPoly$Cache.find(forReceiver)
      if (method != null)
        return method
      else {
        method = ScalaRunTime.ensureAccessible(forReceiver.getMethod("xyz", reflParams$Cache))
        reflPoly$Cache = new SoftReference(reflPoly$Cache.get.add(forReceiver, method))
        return method
      }
    }

  */
```

5.4 Trees for caches

```
TODO

  /* Some cleanup transformations add members to templates (classes, traits, etc).
   * When inside a template (i.e. the body of one of its members), two maps
   * (newStaticMembers and newStaticInits) are available in the tree transformer. Any mapping from
   * a symbol to a MemberDef (DefDef, ValDef, etc.) that is in newStaticMembers once the
   * transformation of the template is finished will be added as a member to the
   * template. Any mapping from a symbol to a tree that is in newStaticInits, will be added
   * as a statement of the form "symbol = tree" to the beginning of the default
   * constructor. */
  case Template(parents, self, body) =>
```

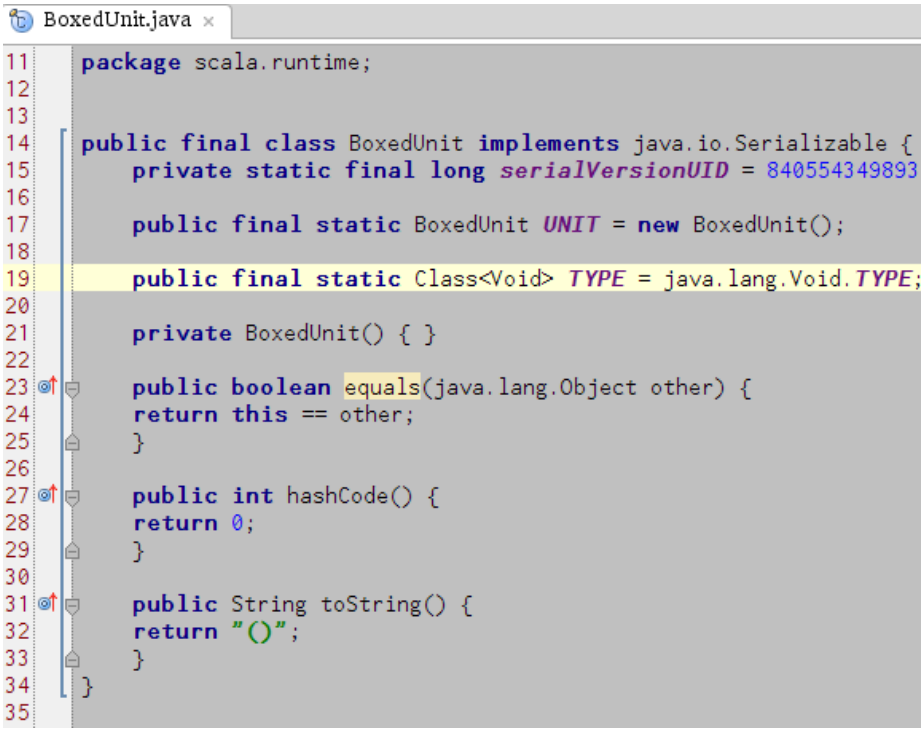


Figure 2: Sec. 6.1

6 Other rewritings

6.1 VM-reflected classes for Scala class literals of value classes, including for the `Unit` class

And the code emitted for the following class literals is ...

1. Check the following REPL session:

   ```
   scala> ().getClass
   res1: java.lang.Class[Unit] = void
   ```

   To recap, `BoxedUnitModule` is (the static part of) `scala.runtime.BoxedUnit`. Details in Figure 2.

2. For all other Scala values classes, and on the JVM, the snippet below emits things like "`java.lang.Boolean.TYPE`". REPL-wise:

   ```
   scala> java.lang.Boolean.TYPE
   res2: java.lang.Class[java.lang.Boolean] = boolean
   ```

```
case Literal(c) if (c.tag == ClassTag) && !forMSIL=>
  val tpe = c.typeValue
  typedWithPos(tree.pos) {
    if (isValueClass(tpe.typeSymbol)) {
      if (tpe.typeSymbol == UnitClass)
```

```
case theTry @ Try(block, catches, finalizer)
  if theTry.tpe.typeSymbol != definitions.UnitClass &&
     theTry.tpe.typeSymbol != definitions.NothingClass =>
  val tpe = theTry.tpe.widen
  val tempVar = currentOwner.newVariable(theTry.pos, mkTerm(nme.EXCEPTION_RESULT_PREFIX)).setInfo(tpe)
  def assignBlock(rhs: Tree) = super.transform(BLOCK(Ident(tempVar) === transform(rhs)))

  val newBlock    = assignBlock(block)
  val newCatches  = for (CaseDef(pattern, guard, body) <- catches) yield
    (CASE(super.transform(pattern)) IF (super.transform(guard))) ==> assignBlock(body)
  val newTry      = Try(newBlock, newCatches, super.transform(finalizer))

  typedWithPos(theTry.pos)(BLOCK(VAL(tempVar) === EmptyTree, newTry, Ident(tempVar)))
```

Figure 3: Sec. 6.2

```
        REF(BoxedUnit_TYPE)
      else
        Select(REF(boxedModule(tpe.typeSymbol)), nme.TYPE_)
    }

    else tree
  }
```

6.2 Emptying the stack at the end of a try-block

Quoting a source comment. Snippet in Figure 3:

> *Rewrite all try blocks with a result* `!=` {`Unit`, `All`} *such that they store their result in a local variable. The catch blocks are adjusted as well. The try tree is subsituted by a block whose result expression is that variable.*

Talking about emptying the stack, there's a compiler plugin ("`imp`"[1]) to lower ASTs into a stackless, three-address-like IR, similar in spirit to the Jimple IR in McGill's Soot framework.

6.3 Caching of interned symbol literals

Scala symbol literals (§1.3.7 in SLS) are realized via a library class, `scala.Symbol`, and a global `UniquenessCache[String, scala.Symbol]` (Figure 4). Just to clarify, `scala.Symbol` (the library class) has nothing to do with `scala.reflect.internal.Symbol` (the compiler class).

Detailed source comment! Enjoy:

> *For instance, say we have a Scala class:*
>
> ```
> class Cls {
>   // ...
>   def someSymbol = 'symbolic
>   // ...
> }
> ```

[1] Moving Scala ASTs one step closer to C, `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q2/Moving3A.pdf`

```scala
package scala

/** This class provides a simple way to get unique objects for equal strings.
 *  Since symbols are interned, they can be compared using reference equality.
 *  Instances of `Symbol` can be created easily with Scala's built-in quote
 *  mechanism.
 *
 *  For instance, the [[http://scala-lang.org/#_top Scala]] term `'mysym` will
 *  invoke the constructor of the `Symbol` class in the following way:
 *  `Symbol("mysym")`.
 *
 *  @author  Martin Odersky, Iulian Dragos
 *  @version 1.8
 */
final class Symbol private (val name: String) extends Serializable {
  /** Converts this symbol to a string.
   */
  override def toString(): String = "'" + name

  @throws(classOf[java.io.ObjectStreamException])
  private def readResolve(): Any = Symbol.apply(name)
}

object Symbol extends UniquenessCache[String, Symbol]
{
  protected def valueFromKey(name: String): Symbol = new Symbol(name)
  protected def keyFromValue(sym: Symbol): Option[String] = Some(sym.name)
}
```

Figure 4: Sec. 6.3

After transformation, this class looks like this:

```
class Cls {
  private "static" val <some_name>$symbolic = Symbol("symbolic")
  // ...
  def someSymbol = <some_name>$symbolic
  // ...
}
```

The reasoning behind this transformation is the following. Symbols get interned – they are stored in a global map which is protected with a lock. The reason for this is making equality checks quicker. But calling `Symbol.apply`*, although it does return a unique symbol, accesses a locked object, making symbol access slow. To solve this, the unique symbol from the global symbol map in* `Symbol` *is accessed only once during class loading, and after that, the unique symbol is in the static member. Hence, it is cheap to both reach the unique symbol and do equality checks on it.*

To give an impression of what `scala.UniquenessCache` does, it relies on:

```
import java.lang.ref.WeakReference
import java.util.WeakHashMap
import java.util.concurrent.locks.ReentrantReadWriteLock
```

Figure 5: Sec. 6.4

6.4 Wrapped arrays turned into normal arrays

```
// This transform replaces Array( Predef.wrapArray(Array(...)), <manifest>)
// with just Array(...)
case Apply(appMeth, List(Apply(wrapRefArrayMeth, List(array)), _))
if (wrapRefArrayMeth.symbol == Predef_wrapRefArray &&
    appMeth.symbol == ArrayModule_overloadedApply.suchThat {
      _.tpe.resultType.dealias.typeSymbol == ObjectClass
    }) =>
  super.transform(array)
```

The `ArrayModule_overloadedApply` overload of interest in object `scala.Array` is shown below (other overloads in Figure 5):

```
/** Creates an array with given elements.
 *
 *  @param xs the elements to put in the array
 *  @return an array containing all elements from xs.
 */
def apply[T: ClassManifest](xs: T*): Array[T] = {
  val array = new Array[T](xs.length)
  var i = 0
  for (x <- xs.iterator) { array(i) = x; i += 1 }
  array
}
```

```
TODO Example where the above AST is emitted.
```

7 Type parameters and Structural types

[1, p. 4] Scala disallows some interactions between abstract types and structural types, and the Scala.Net compiler will follow suite. We look at the reasons for this restriction after looking at examples.

In a nutshell:

> *Within a method declaration in a structural refinement, the type of any value parameter may only refer to type parameters of the method itself. This restriction does not apply to the function's result type.*

Examples of rejected programs:

- "*error: Parameter type in structural refinement may not refer to an abstract type defined outside that refinement*"

```
def g2[T] (x : { def f(t : T): Boolean },
```

- "*error: Parameter type in structural refinement may not refer to a type member of that refinement*"

```
def g3   (x : { type T; def t: T; def f(a: T): Boolean } ) = x.f(x.t)
```

In contrast, the following are OK:

```
def g1 (x : { def f[T](a: T): Int }) = x.f[Int](4)

def g4[T] (x : { def f(a: Int): T }) = x.f(4)
```

To see why the restriction makes sense, consider what you would do if you were the compiler :-) At the end of the day, given the static types at a "structural callsite" the compiler has to emit code for the reflection API to return a method reification. However (playing the devil's advocate) it would be possible to obtain at runtime the actual types of actual arguments. Yes, and in that case a method lookup based on those would amount to dynamic dispatch on arguments, unlike what the SLS specifies.

1. In some cases, the compiler can emit code to call that API with a list of statically known type-representatives:

   ```
   def g0(x: { def f(a: Int, b: List[Int]): Int }) = x.f(4, List(1,2,3))
   ```

2. In other cases the structural member defines a type param, but still each callsite ("`x.f[Int](4)`" in the example below) allows inferring a statically-known instantiated type:

   ```
   def g1 (x : { def f[T](a: T): Int }) = x.f[Int](4)
   ```

3. However in cases like those below, different lookups would be needed *for the same callsite* depending on type arguments provided by upstream invokers. In the example below,

```
def undecisive[T](p: { def x(t: T): Int }, t: T) = p.x(t)
```

the callsite "`p.x(t)`" should result in lookup "`getMethod("x", Array(Int))`" in one case and "`getMethod("x", Array(String))`" in another (example adapted from[2])

```
undecisive[Int](    new { def x(t: Int) = t }, 4    )   // upstream invocation 1
undecisive[String]( new { def x(t: String) = 5 }, "abc") // upstream invocation 2
```

Adapting an explanation from [1, p. 4] to the "`undecisive`" example:

> *The type variable `T` is instantiated to a concrete type every time `undecisive` is called. Therefore, the static types of `T`-typed formal params will in general be different for different calls to `x` in the expression `p.x(t)`.*

That's the correctness argument (to reiterate, the SLS specifies method dispatch based on dynamic type of the receiver and static types of the formals). But there's also an implementation-related difficulty that confirms the correctness-based design decision :-) Well, at least on the JVM:

> *On the other hand, the transformation of `ApplyDynamic` for `p.x(t)` is done only once, in the body of `undecisive`, no matter what type `T` will eventually be assigned to. The value of type variables are not available at runtime [on the JVM] so that `ApplyDynamic` cannot be compiled in a way that reconstructs the static types of the method's parameters at runtime.*

Summing up:

- *Member-local type params in structural members are allowed.*
- *Free type vars in structural members are not (except in their result type).*

8 Scala.NET

8.1 Non-fixed types of formals

The snippet below is accepted by the compiler, although it may result in structural callsites with non-fixed types for formals.

```
// accepted by the compiler
  def gy[Y] (y: Y, x : { def f[T](a: T): Int }) = x.f(y)
```

In those cases where the `T` type var is replaced by a concrete type at a callsite, we have fixed-types for formals. However, `T` can also be replaced as above by another type var (`Y` in the example). Looks like that should be rejected.

As background info, the `forJVM` translation of the structural callsite can be found in Sec. 8.1.1.

[2] `http://scala-programming-language.1934581.n4.nabble.com/scala-Structural-types-with-generic-type-question-td1992248.html`

The examples in Listing 4 show that only a subset of all feasible receivers are accepted by the compiler (e.g., `ostr`, `oint`, and `oobj` are rejected) while one receiver that is acccepted causes NPE (the `gy(null, null)` invocation). That leaves only `ogen` as an example of receiver that is both well-typed and doesn't fail at runtime.

8.1.1 Translation `forJVM`

After `erasure`:

```
def gy(y: java.lang.Object, x: java.lang.Object): Int = unbox(<apply-dynamic>(x#f, (y)))
```

and the `MethodType` in `ad.symbol.info` reads:

```
(a: java.lang.Object)Int
```

After `genDefaultCall()`, the following is emitted:

```
def gy(y: java.lang.Object, x: java.lang.Object): Int =
  scala.Int.unbox(
    {
      val qual1: java.lang.Object = x;
      {
        var exceptionResult1: java.lang.Object = _;
        try {
          exceptionResult1 =
             Test.reflMethod$Method1(qual1.getClass())
            .invoke(qual1, Array[java.lang.Object]{y})
        } catch {
          case (1 @ (_: java.lang.reflect.InvocationTargetException))
          => { exceptionResult1 = throw 1.getCause() }
        };
        exceptionResult1
      }.$asInstanceOf[java.lang.Integer]()
    }
  )
```

And the caches are:

```
final private <synthetic> <static> var reflParams$Cache1: Array[java.lang.Class] =
  Array[java.lang.Class]{classOf[java.lang.Object]};

<synthetic> <static> def reflMethod$Method1(x$1: java.lang.Class): java.lang.reflect.Method =
  x$1.getMethod("f", Test.reflParams$Cache1);
```

8.2 Static caches

Regarding Sec. 5.4: Statics are per-type-instantiation on CLR. The C# 2.0 spec worded it concisely:

> *A static variable in a generic class declaration is shared amongst all instances of the same closed constructed type, but is not shared amongst instances of different closed constructed types . . . regardless of whether the type of the static variable involves any type parameters or not.*

The CLR way: type-params owned by a class are visible in its static members.

1. If (a) a structural reference appears in a type P owning type params, and (b) the caching policy is `mono-cache` or `poly-cache`; then there will be not one (as on JVM) but in general many caches (one for each instantiation of P). A similar situation occurs when caching interned symbols (Sec. 6.3).

2. Further cache fragmentation (and thus cache misses) result on the CLR with `poly-cache` because there's a different key for each instantiated type of the receiver.

8.3 Class literals of value classes

Regarding Sec. 6.1. Something like "`BoxedUnit_TYPE`" should be emitted for the class literal of `Unit`. Instead, `GenMSIL` is not aware about that special case:

```
  case CONSTANT(const) =>
    const.tag match {
      case UnitTag  => ()
      . . .
      case ClassTag =>
        mcode.Emit(OpCodes.Ldtoken, msilType(const.typeValue))
        mcode.Emit(OpCodes.Call, TYPE_FROM_HANDLE)
      . . .

And also:

    private def msilType(t: TypeKind): MsilType = (t: @unchecked) match {
      case UNIT         => MVOID
      . . .
```

8.4 Arrays

Regarding Sec. 4.1. The CLR includes two kinds of arrays:

- vectors: same as arrays on the JVM i.e., zero-based single-dimensional arrays.
- multi-dimensional arrays, where each dimension can have its own lower and upper bounds.

At runtime, arrays of both kinds are objects that conform to the abstract class `[mscorlib]System.Array`.

Quoting from the MSDN Help for the `System.Type.IsArray` property:

```
public static void Main()
{
  int [] array = {1,2,3,4};
  Type at = typeof(Array);
  Type t = array.GetType();
  Console.WriteLine("The type is {0}. Is this type an array? {1}", at, at.IsArray);
  Console.WriteLine("The type is {0}. Is this type an array? {1}", t, t.IsArray);
}
```

This code produces the following output:

```
Type is System.Array. IsArray? False
Type is System.Int32[]. IsArray? True
```

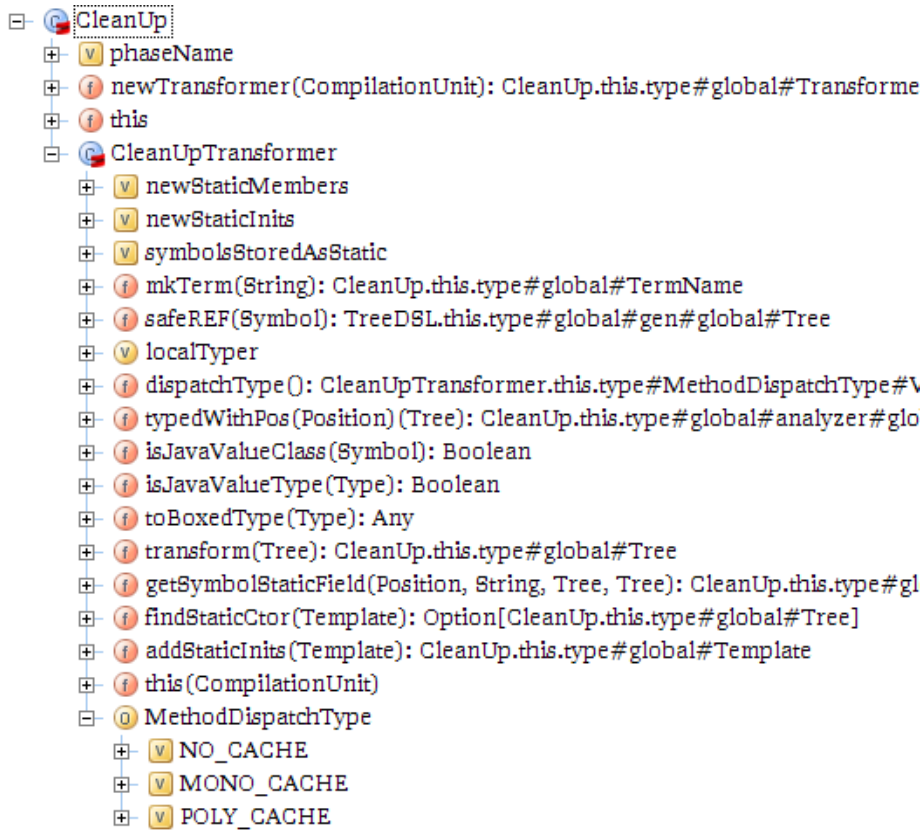


Figure 6: Sec. 1

References

[1] Gilles Dubochet and Martin Odersky. Compiling structural types on the JVM. In Ian Rogers, editor, *4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 34–41, 2009. `http://infoscience.epfl.ch/record/138931/files/2009_structural.pdf`.

[2] Martin Odersky. The Scala Language Specification. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, March 2009.

Listing 4: Sec. 8.1

```
object Test {

  def gy[Y] (y: Y, x : { def f[T](a: T): Int }) = x.f(y)

  val ostr = new { def f(a: String) = 4 }
  val oint = new { def f(a: Int) = 4 }
  val oobj = new { def f(a: Object) = 4 }
  val ogen = new { def f[T](a: T) = 4 }

  def main(args: Array[String]) {

/*-

error: type mismatch;
 found   : Test.oint.type (with underlying type java.lang.Object{def f(a: Int): Int})
 required: AnyRef{def f[T](a: T): Int}
    gy(123,  oint)
             ^


error: type mismatch;
 found   : Test.ostr.type (with underlying type java.lang.Object{def f(a: String): Int})
 required: AnyRef{def f[T](a: T): Int}
    gy("abc", ostr)
              ^


error: type mismatch;
 found   : Test.oobj.type (with underlying type java.lang.Object{def f(a: java.lang.Object): Int})
 required: AnyRef{def f[T](a: T): Int}
    gy(this, oobj)
             ^


 */

    gy(null, null) /*- accepted by compiler, results in NullPointerException at runtime. */

    gy(null, ogen) /*- runs ok. */

    gy(null, oobj.asInstanceOf[ AnyRef{ def f[T](a: T): Int } ]) /*- runs ok too. */

  }

}
```