

Where implicits come from, and Constant Folding Redux

© Miguel Garcia, STS, Hamburg University of Technology
<http://www.sts.tu-harburg.de/people/mi.garcia>

December 26, 2009

Contents

1	End-to-end example	1
2	Where implicits come from	6
3	Spotting implicit invocations at the AST level	7

Abstract

These notes are part of a series discussing the operation of the Scala compiler. Taken together, they help to ease the learning curve when exploring `scalac`, although each installment describes, in a piecemeal fashion, only some aspects of `scalac`'s operation. This document continues the discussion from *Untangling Scala ASTs*, covering (1) the program transformation by which implicit invocations are inserted, as well as (2) a late simplification of ASTs (after the `cleanup` phase). This simplification applies (once more) constant folding and `if-then-else` reduction, to trees that were generated after their initial application. The simplification is more instructive than useful, as a later phase (`icode`) would have performed them anyway, albeit at a lower level of abstraction.

1 End-to-end example

The program of interest is shown in Listing 1

The single line: `@BeanInfo case class Address(street: String, city: String)` is expanded internally into the class definition sketched in Figure 1. Running the compiler with `-Xprint:cleanup` allows seeing that two methods in the expansion, `equals` and `productElement`, contain code like:

```
override def productElement(x$1: Int): java.lang.Object = {  
  var temp5: Int = x$1;  
  (temp5: Int) match {  
    case 0 => {  
      if (true) { body%0(){ Address.this.street() } }  
      else throw new MatchError(scala.Int.box(temp5).toString())  
    }  
  }  
  ...  
}
```

Listing 1: A program that will be expanded by the compiler with invocations to implicits and boilerplate methods for case classes

```

package reflectionTest
import scala.reflect.BeanInfo
@BeanInfo case class Address(street: String, city: String)
object Test {
  def main(args: Array[java.lang.String]): Unit = {
    val a = Address("street", "city")
    import java.beans._
    val pds = Introspector.getBeanInfo(classOf[Address]).getPropertyDescriptors
    println(pds.map(_.getReadMethod).mkString(" ", "))
  }
}

```

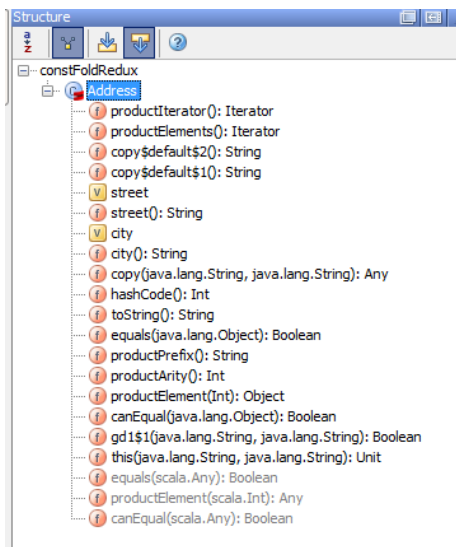


Figure 1: Case class after expansion

We'll make that code look nicer! By applying the “recipe” for transformation phases (previously covered in the *Cleanup* and *Specialize Reports*).

All we need is a compiler plugin :-). The more or less standard plumbing is shown in Listing 2. The transformation proper appears in Listing 3. As can be seen, `CFReduxTransformer` handles the following cases:

```

case x @ If(_, _, _) => transformIf(x)
case Apply(Select(Literal(x1), op1), List(Literal(y1))) => transformOp(tree)
case Select(Literal(x2), op2) => transformOp(tree)
case Block(stats, expr) if ((expr != EmptyTree) && stats.isEmpty) =>
  { someReductionApplied = true; expr }
case _ => super.transform(tree)

```

Constant folding proper is done in `transformOp(tree)`, by invoking functionality from `ConstantFolder.scala` (as done during type-checking). `transformIf(x)` was also found by cherry-picking (it is originally applied during `refchecks`). The last conversion replaces a block with its single statement.

Listing 2: After cleanup, the redux of constant folding and if-then-else reduction
(1 of 2)

```
package constFoldRedux

import scala.tools.nsc.{plugins, Global, Phase}
import plugins.PluginComponent
import scala.tools.nsc.{ast, symtab, util, Global}
import plugins.Plugin

object CFReduxPlugin
{
    val PluginName = "CFRedux"
}
class CFReduxPlugin(val global: Global) extends Plugin
{
    import global._
    import definitions.UnitClass

    val name = CFReduxPlugin.PluginName
    val description = "A plugin to apply constant folding once again" +
        " (after typing) this time after cleanup."
    val components = List[PluginComponent](Component)

    /* For source compatibility between 2.7.x and 2.8.x */
    private object runsBefore { def :: (s: String) = s }
    private abstract class CompatiblePluginComponent(afterPhase: String) extends PluginComponent
    {
        val runsAfter = afterPhase :: runsBefore
    }
    private object Component extends CompatiblePluginComponent("cleanup")
    {
        val global = CFReduxPlugin.this.global
        val phaseName = CFReduxPlugin.this.name
        def newPhase(prev: Phase) = new CFReduxPhase(prev)
    }

    private class CFReduxPhase(prev: Phase) extends Phase(prev)
    {
        def name = CFReduxPlugin.this.name
    }
    var someReductionApplied = false

    override def run {
        for (unit <- currentRun.units; if !unit.isJava) {
            val tBefore = unit.body
            var tAfter = tBefore
            do {
                someReductionApplied = false
                tAfter = CFReduxTransformer.transform(tAfter)
            } while (someReductionApplied)
            unit.body = tAfter
        }
    }
}
```

Listing 3: After cleanup, the redux of constant folding and if-then-else reduction
(2 of 2)

```
object CFReduxTransformer extends Transformer {
  override def transform(tree: Tree) = tree match {
    case x @ If(_, _, _) =>
      transformIf(x)
    case Apply(Select(Literal(x1), op1), List(Literal(y1))) =>
      transformOp(tree)
    case Select(Literal(x2), op2) =>
      transformOp(tree)
    case Block(stats, expr) if ((expr != EmptyTree) && stats.isEmpty) =>
      { someReductionApplied = true; expr }
    case _ => super.transform(tree)
  }

  private def transformOp(tree: Tree): Tree = {
    someReductionApplied = true
    constfold(tree)
  }

  /* cut&pasted from RefChecks.transformIf */
  private def transformIf(tree: If): Tree = {
    val If(cond, thenpart, elsepart) = tree
    def unitIfEmpty(t: Tree): Tree =
      if (t == EmptyTree) Literal(()).setPos(tree.pos).setType(UnitClass.tpe)
      else t

    cond.tpe match {
      case ConstantType(value) =>
        val res = if (value.booleanValue) thenpart else elsepart
        someReductionApplied = true;
        unitIfEmpty(res)
      case _ => super.transform(tree)
    }
  }
}
}
```

12	<caseaccessor> <paramaccessor> private[this] val city: java.lang.String = _;	12
13	<stable> <caseaccessor> <paramaccessor> def city(): java.lang.String = _;	13
14	<synthetic> def copy(street: java.lang.String = street, city: java.lang.String = city):	14
15	Address: Address = ScalaRuntime.this.hashCode(Address.this);	15
16	override def toString(): java.lang.String = ScalaRuntime.this.toString(Address.this);	16
17	private[this] def hashCode(): Int = ScalaRuntime.this.hashCode(Address.this);	17
18	var temp1: java.lang.Object = xs1;	18
19	if (temp1.\$asInstanceOf?[ReflectionTest.Address]())	19
20	{	20
21	var temp2: ReflectionTest.Address = temp1.\$asInstanceOf?[ReflectionTest.Address]	21
22	{	22
23	var temp3: java.lang.String = temp2.street();	23
24	var temp4: java.lang.String = temp2.city();	24
25	var temp5: java.lang.String = temp2.city();	25
26	{	26
27	val city\$1: java.lang.String = temp4;	27
28	{	28
29	val streets\$1: java.lang.String = temp3;	29
30	if (Address.this.gdIs(streets\$1, city\$1))	30
31	{	31
32	val streets\$1: java.lang.String = temp3;	32
33	{	33
34	val city\$1: java.lang.String = temp4;	34
35	body%0(streets\$1, city\$1){	35
36	xs1.\$asInstanceOf?[ReflectionTest.Address]().canEqual(Address.thi	36
37)};	37
38	}	38
39	}	39
40	}	40
41	else	41
42	{	42
43	false	43
44	}	44
45	}	45
46	}	46
47	else	47
48	{	48
49	throw new MatchError(temp1.toString())	49
50	}	50
51	};	51
52	};	52
53	};	53
54	};	54
55	};	55
56	};	56
57	};	57
58	};	58
59	};	59
60	};	60
61	};	61
62	};	62
63	};	63
64	};	64
65	};	65
66	};	66
67	};	67
68	};	68
69	};	69
70	};	70
71	};	71
72	};	72
73	};	73

Figure 2: After and Before Constant Folding Redux (Code with green background was optimized away)

2 Where implicits come from

For a number of reasons, the `main()` method from Listing 1 (p. 2) gets expanded to:

```
def main(args: Array[java.lang.String]): Unit = {  
  
  // original : val a = Address("street", "city")  
  val a: reflectionTest.Address = new reflectionTest.Address("street", "city");  
  
  // original :  
  // import java.beans._  
  // val pds = Introspector.getBeanInfo(classOf[Address]).getPropertyDescriptors  
  val pds: Array[java.beans.PropertyDescriptor] =  
    java.beans.Introspector.getBeanInfo(classOf[reflectionTest.Address])  
    .getPropertyDescriptors();  
  
  // original :  
  // println(pds.map(_.getReadMethod).mkString(", "))  
  scala.this.Predef.println(  
    scala.this.Predef.refArrayOps(  
      scala.this.Predef.refArrayOps(  
        pds.$asInstanceOf[Array[java.lang.Object]]()  
      )  
    ).map(  
      (new anonymous class $anonfun$main$1(): Function1)  
      ,  
      scala.this.Array.canBuildFrom(  
        reflect.this.ClassManifest.classType(  
          classOf[java.lang.reflect.Method])  
        )  
      )  
    ).$asInstanceOf[Array[java.lang.Object]]()  
    )  
    .mkString(", ")  
  ); // yes, I know parens aren't balanced but don't ask why  
  ()  
};
```

In retrospect, I remember reading about automatic conversions like the one from `pds` (an `Array[java.beans.PropertyDescriptor]`) into something that supports `map` (i.e., the conversion performed by `Predef.refArrayOps`). However I did not remember (from Scala collections) that the following was going to be added as 2nd argument to `map`:

```
scala.this.Array.canBuildFrom(  
  reflect.this.ClassManifest.classType(  
    classOf[java.lang.reflect.Method]  
  )  
)  
)
```

What follows are the “bits and pieces of knowledge” I’ve gathered so far on how invocations for implicits are added by the compiler. Needless to say, the coverage in this section is not exhaustive.

First, two entry points to find implicits are shown in Listing 4 and Listing 5.

Listing 4: One of the entry points to find implicits

```
// from Implicits.scala

/** Search for an implicit value. See the comment on 'result' at the end of class 'ImplicitSearch'
 * for more info how the search is conducted.
 * @param tree      The tree for which the implicit needs to be inserted.
 *                  (the inference might instantiate some of the undetermined
 *                  type parameters of that tree.
 * @param pt        The expected type of the implicit.
 * @param reportAmbiguous Should ambiguous implicit errors be reported?
 *                  False iff we search for a view to find out
 *                  whether one type is coercible to another.
 * @param isView    We are looking for a view
 * @param context   The current context
 * @return          A search result
 */
def inferImplicit(
  tree: Tree,
  pt: Type,
  reportAmbiguous: Boolean,
  isView: Boolean,
  context: Context): SearchResult = {
  if (traceImplicits && !tree.isEmpty && !context.undetparams.isEmpty)
    println("typing implicit with undetermined type params: "+context.undetparams+"\n"+tree)
  val result = new ImplicitSearch(tree, pt, isView, context.makeImplicit(reportAmbiguous)).bestImplicit
  context.undetparams = context.undetparams filterNot (result.subst.from contains _)
  result
}
```

Calls to method `inferImplicit()` can be found in both `Implicits.scala` and `Typers.scala`, an example of the latter is:

```
def findManifest(tp: Type, full: Boolean) = atPhase(currentRun.typerPhase) {
  inferImplicit(
    EmptyTree,
    appliedType((if (full) FullManifestClass else PartialManifestClass).typeConstructor, List(tp)),
    true, false, context)
}
```

But then `findManifest()` above is also called from `UnCurry.scala`. Not sure how late during compilation implicits might still be resolved.

With a breakpoint on `inferImplicit()`, an implicit with type

```
(Array[java.beans.PropertyDescriptor]) => ?{val map: ?}
```

is searched for the tree `Ident(pds)` as shown in Figure 3.

This debug session is starting to get interesting, and therefore it's a pity that I have to leave it at that. But you know, feel free to explain yourself the rest of this story at *The Scala Compiler Corner*. See you there!

3 Spotting implicit invocations at the AST level

“The attentive reader” will spot a common pattern (shown in Listing 4) to slip-in a custom iterator. Tool support wouldn't hurt in spotting automatic usages of implicits.

Listing 5: Another of the entry points to find implicits (from `Typers.scala`)

```

/** Infer an implicit conversion ('view') between two types.
 * @param tree      The tree which needs to be converted.
 * @param from      The source type of the conversion
 * @param to        The target type of the conversion
 * @param reportAmbiguous Should ambiguous implicit errors be reported?
 *                  False iff we search for a view to find out
 *                  whether one type is coercible to another.
 */
def inferView(tree: Tree, from: Type, to: Type, reportAmbiguous: Boolean): Tree = {
  if (settings.debug.value) log("infer view from "+from+" to "+to)//debug
  if (phase.id > currentRun typerPhase.id) EmptyTree
  else from match {
    case MethodType(_, _) => EmptyTree
    case OverloadedType(_, _) => EmptyTree
    case PolyType(_, _) => EmptyTree
    case _ =>
      def wrapImplicit(from: Type): Tree = {
        val result = inferImplicit(tree, functionType(List(from), to), reportAmbiguous, true, context)
        if (result.subst != EmptyTreeTypeSubstituter) result.subst traverse tree
        result.tree
      }
      val result = wrapImplicit(from)
      if (result != EmptyTree) result
      else wrapImplicit(appliedType(ByNameParamClass.typeConstructor, List(from)))
  }
}

```

In terms of ASTs, Scala X-Ray shows one way to detect whether a method invocation (to an implicit) was inserted by the compiler (Listing 6).

TODO: mention IDE support to signal which implicits are in effect.

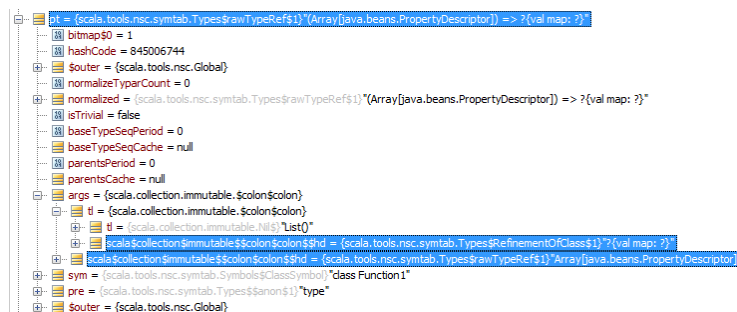


Figure 3: Hitting `inferImplicit()` first time for the sample program

Listing 6: Detecting invocations to implicits inserted by the compiler (reproduced from `Browse.scala` in Scala X-Ray)

```
// magic method #3
private def processSymbol(t: Tree, token: Token, sourceFile: File)
{
  val sym = t.symbol
  sym match
  {
    case ts: TermSymbol =>
      val sType = t match // find out the type for the term symbol
      {
        case ad: ApplyDynamic => ad.qual.tpe.memberType(ad.symbol)
        case s: Select => s.qualifier.tpe.memberType(s.symbol)
        case _ => ts.owner.thisType.memberType(ts)
      }
      if(sType != null)
      { val asString = sType match
        { // the invoked method may be implicit, flag it
          case mt: MethodType if ts.hasFlag(Flags.IMPLICIT) =>
            "implicit " + fullName(sym) + " : " + typeString(sType)
          case _ => typeString(sType)
        }
        //println("Term symbol " + sym.id + ": " + asString)
        token.tpe = TypeAttribute(asString, linkTo(sourceFile, sType.typeSymbol))
      }
    case ts: TypeSymbol =>
      val treeType = t.tpe
      val sType =
        if(treeType == NoType) ts.info
        else treeType
      //println("Type symbol " + sym.id + ": " + typeString(sType))
      if(sType != null)
        token.tpe = TypeAttribute(typeString(sType), linkTo(sourceFile, sType.typeSymbol))
    case _ => ()
  }
  if(sym != null)
  {
    if(t.isDef) token += sym.id
    else
    { linkTo(sourceFile, sym) match
      { case Some(x) => token.reference = x
        case None => token += sym.id
      }
    }
  }
}
}
```

```

62 private def scan(unit: CompilationUnit) =
63 {
64   val tokens = wrap.Wrappers.treeset[Token]
65   val scanner = new syntaxAnalyzer.UnitsScanner(unit) { override def init {} ; def parentInit = super.init }
66
67   implicit def iterator28(s: syntaxAnalyzer.UnitsScanner) =
68   {
69     class CompatIterator extends Iterator[(Int, Int, Int)]
70     {
71       def next =
72       {
73         type TD = { def offset: Int; def lastOffset: Int; def token: Int }
74         class Compat { def prev: TD = null; def next: TD = null; def offset = 0; def token = 0; def lastOffset = 0 }
75         implicit def keep27SourceCompatibility(a: AnyRef): Compat = new Compat // won't ever be called
76         val offset = s.offset
77         val token = s.token
78         s.nextToken
79         (offset, (s.lastOffset - offset) max 1, token)
80       }
81       def hasNext = s.token != Tokens.EOF
82     }
83
84     scanner.parentInit
85     new { def iterator = new CompatIterator }
86   }
87
88   /* This for loop relies on the iterator implicitly defined above (i.e. iterator28) */
89   for( (offset, length, code) <- scanner.iterator )
90   {
91     if(includeToken(code)) {
92       val text = new String(scanner.buf.slice(offset, offset + length))
93       tokens += new Token(offset, length, code, text)
94     }
95   }
96   tokens
97 }

```

Figure 4: Excerpt from Scala X-Ray, iterator28 used in scanner <- iterator