

# Code walkthrough of the UnCurry phase (Scala 2.8)

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

November 24, 2009

## Contents

<b>1</b>	<b>What's in and what's out</b>	<b>1</b>
<b>2</b>	<b>Eliminating anonymous functions by applying Closure Conversion</b>	<b>2</b>
<b>3</b>	<b>Eta Expansion, or how to handle partial function applications</b>	<b>3</b>
3.1	Syntax: perspective from the programmer's point of view . . . .	4
3.2	Perspective from the compiler point of view . . . . .	4
<b>4</b>	<b>An example where first the transformation for partial function app is performed, then that for closure conversion</b>	<b>5</b>

### Abstract

Previous walkthroughs have explored some phases of the Scala compiler (CleanUp, SpecializeTypes). Continuing in their tradition, the present notes sketch some aspects of another phase, UnCurry. And the usual caveat applies: these notes are biased to newcomers. Accomplished Scala compiler hackers might instead consider contributing a phase description of their own, or extending this one, thanks!

## 1 What's in and what's out

The UnCurry phase performs several transformations, two of which are summarized in these notes (closure conversion and eta-expansion).

The other transformations are listed below:

```
/** - uncurry all symbol and tree types (@see UnCurryPhase)
 * - for every curried parameter list: (ps_1) ... (ps_n) ==> (ps_1, ..., ps_n)
 * - for every curried application: f(args_1)...(args_n) ==> f(args_1, ..., args_n)
 * - for every type application: f[Ts] ==> f[Ts]() unless followed by parameters
 * - for every use of a parameterless function: f ==> f() and q.f ==> q.f()
 * - for every def-parameter: x: => T ==> x: () => T
 * - for every use of a def-parameter: x ==> x.apply()
 * - for every argument to a def parameter 'x: => T':
```

```

*       if argument is not a reference to a def parameter:
*         convert argument 'e' to (expansion of) '() => e'
* - for every repeated Scala parameter 'x: T*' --> x: Seq[T].
* - for every repeated Java parameter 'x: T...' --> x: Array[T], except:
*   if T is an unbounded abstract type, replace --> x: Array[Object]
* - for every argument list that corresponds to a repeated Scala parameter
*   (a_1, ..., a_n) => (Seq(a_1, ..., a_n))
* - for every argument list that corresponds to a repeated Java parameter
*   (a_1, ..., a_n) => (Array(a_1, ..., a_n))
* - for every argument list that is an escaped sequence
*   (a_1:_) => (a_1) (possibly converted to sequence or array, as needed)
* - convert implicit method types to method types
* - convert non-trivial catches in try statements to matches
* - convert non-local returns to throws with enclosing try statements.
*/

```

Several transformations prepare the ground for later ones, for example:

```

// a local variable that is mutable and free somewhere later should be lifted
// as lambda lifting (coming later) will wrap 'rhs' in an Ref object.

```

Lambda lift and explicit outer are explained in <http://lamp.epfl.ch/~paltherr/phd/altherr-phd.pdf>. Explicit outer is performed before lambda lift.

## 2 Eliminating anonymous functions by applying Closure Conversion

In a nutshell, closure conversion rewrites an anonymous function into the instantiation of a function trait. What do function traits look like? For example, the trait `Function5` is defined as:

---

```

trait Function5[-T1, -T2, -T3, -T4, -T5, +R] extends AnyRef { self =>
  def apply(v1:T1,v2:T2,v3:T3,v4:T4,v5:T5): R
  override def toString() = "<function5>"

  /** f(x1,x2,x3,x4,x5) == (f.curry)(x1)(x2)(x3)(x4)(x5)
   */
  def curry: T1 => T2 => T3 => T4 => T5 => R = {
    (x1: T1) => ((x2: T2, x3: T3, x4: T4, x5: T5) => self.apply(x1,x2,x3,x4,x5)).curry
  }
}

```

---

In more detail, closure conversion takes as input a program containing functions with free variables, and returns an equivalent program containing only closed functions.

The application of closure conversion to a particular anonymous function (a.k.a. a function object, or a lambda expression) is realized by the Scala compiler in `UnCurry.scala` (stretching from line 319 to 394)

---

```

def transformFunction(fun: Function): Tree

```

---

This implementation of closure conversion handles two different cases:

First, transform a function object of the form  $(x_1, \dots, x_n) \Rightarrow \text{body}$  of type `FunctionN[T1, .., TN, R]` to

---

```
class $anon() extends Object() with FunctionN[T1, .., TN, R] with ScalaObject {
  def apply(x1: T1, ..., xN: TN): R = body
}
new $anon()
```

---

Second, transform a function object of the form  $(x \Rightarrow \text{body})$  of type `PartialFunction[T, R]` where ‘body’ is of the form  $x \text{ match } \{ \text{case } P_i \text{ if } G_i \Rightarrow E_i \}_{i=1..n}$  to

---

```
class $anon() extends Object() with PartialFunction[T, R] with ScalaObject {
  def apply(x: T): R = (x: @unchecked) match {
    { case Pi if Gi => Ei }i=1..n
  }
  def isDefinedAt(x: T): boolean = (x: @unchecked) match {
    case P1 if G1 => true
    ...
    case Pn if Gn => true
    case _ => false
  }
}
new $anon()
```

---

However, if one of the patterns  $P_i \text{ if } G_i$  is a default pattern, `isDefinedAt` is generated as follows

---

```
def isDefinedAt(x: T): boolean = true
```

---

As can be seen from the two cases above, the newly introduced class definition appears at the point where the original function literal appeared, i.e. these classes are not yet hoisted into top-level classes. Because of this, ‘body’ itself need not be rewritten: the lexical scope is the same for the old and new expressions (function literal and its closure conversion, resp.)

TODO: explain the purpose of the following at the very start of `transformFunction`.

---

```
val fun1 = deEta(fun)
if (fun1 ne fun) fun1
else // ... actual closure conversion
```

---

### 3 Eta Expansion, or how to handle partial function applications

This transformation, also performed during the `UnCurry` phase, is all about those cases where: (a) underscore is used to represent a missing argument, or (b) no underscore appears, but there’s a missing argument anyway.

As in

---

```
def add(x:Int, y:Int, z:Int) = x + y + z
val addFiveInTheMiddle = add(_:Int, 5, _:Int)
addFiveInTheMiddle(3, 1) // 9
```

---

but also in syntax shorthands for single-arg methods:

---

```
def square(in: double) = in * in
def someMethod(f: Function1[Double,Double]) = f(12)
someMethod(square)
List(1, 2) map square // which really stands for List(1, 2) map (square _)
```

---

### 3.1 Syntax: perspective from the programmer's point of view

This subsection is reproduced (with slight modifications) from <http://www.codecommit.com/blog/scala/function-carrying-in-scala>

It is possible to define a method which takes a function of  $n$  parameters and converts it to a curried function of order  $n$ . In fact, this function is already defined within the `Function` singleton in the `scala` package. We can use it to curry the first version of our `add` method:

---

```
def add(x:Int, y:Int) = x + y
val addCurried = Function.curried(add _)
add(1, 2) // 3
addCurried(1)(2) // 3
```

---

The underscore within the curried method invocation is actually a Scala syntax which tells the compiler to treat `add` as a function value, rather than a method to be invoked.

`scala.Function` also provides utility methods which allow us to reverse the process. For example, if we start with our second version of `add`, we may wish to generate a version which takes all parameters inline (in conventional fashion). This can be done using the `uncurried` method:

---

```
def add(x:Int)(y:Int) = x + y
val addUncurried = Function.uncurried(add _)
add(3)(4) // 7
addUncurried(3, 4) // 7
```

---

From the object `scala.Function`:

---

```
/** Currying for functions of arity 2. This transforms a function
 * of arity 2 into a unary function returning another unary function.
 */
def curried[a1, a2, b](f: (a1, a2) => b): a1 => a2 => b = {
  x1 => x2 => f(x1, x2)
}

/** Uncurrying for functions of arity 2. This transforms a unary function
 * returning another unary function into a function of arity 2.
 */
def uncurried[a1, a2, b](f: a1 => a2 => b): (a1, a2) => b = {
  (x1, x2) => f(x1)(x2)
}
```

---

### 3.2 Perspective from the compiler point of view

The rewriting of particular partial function applications (as in the example above)

---

```
def etaExpand(unit : CompilationUnit, tree: Tree): Tree
```

---

Rather than looking at details of this transformation, a combined example is shown next.

## 4 An example where first the transformation for partial function app is performed, then that for closure conversion

On `Namers.scala` (line 604) we can find:

---

```
vparamss.map(_.map(enterValueParam))
```

---

That innocent looking ‘`enterValueParam`’ is represented at the AST level (on entrance to the uncurry phase) as an `Ident()`, which refers to a method with signature `def enterValueParam(param: ValDef): Symbol`

Therefore, that occurrence of “`enterValueParam`” actually means:

---

```
{ ((param: Namers.this.global.ValDef) => enterValueParam(param)) }
```

---

which in fact has the type that `map` expects in that position. That was eta-expansion. Admittedly, the example does not show the resulting pattern in its full generality (as presented in SLS 6.25.5) and excerpted below:

---

```
{
  private synthetic val eta$f = p.f // if p is not stable
  ...
  private synthetic val eta$e_i = e_i // if e_i is not stable
  ...
  (ps_1 => ... => ps_m =>
    eta$f([es_1])
    ...
    ([es_m])(ps_1)...(ps_m)
  )
}
```

---

Coming back to the example about `enterValueParam`. After the eta-expansion, closure conversion comes next, which reformulates the above anonymous function as the following:

---

```
{
  final <synthetic> class $anonfun
  extends java.lang.Object
  with (Namers.this.global.ValDef) => Namers.this.global.Symbol
  with ScalaObject {
    def <init>() = {
      super.<init>();
      ()
    };
    final def apply(param: Namers.this.global.ValDef): Namers.this.global.Symbol = enterValueParam(param)
  };
  (new anonymous class $anonfun(): (Namers.this.global.ValDef) => Namers.this.global.Symbol)
}
```

---