Haskell and Scala

comparison

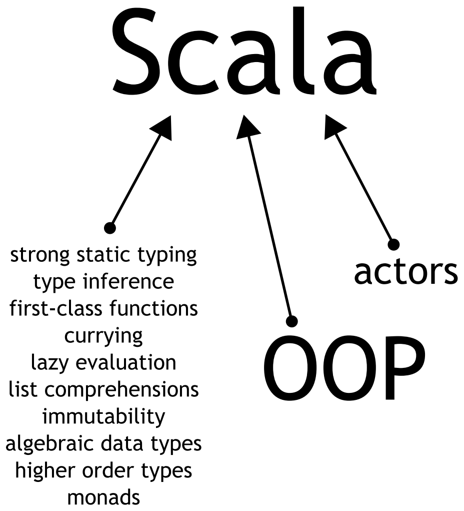
Adam Szlachta

March 20, 2013

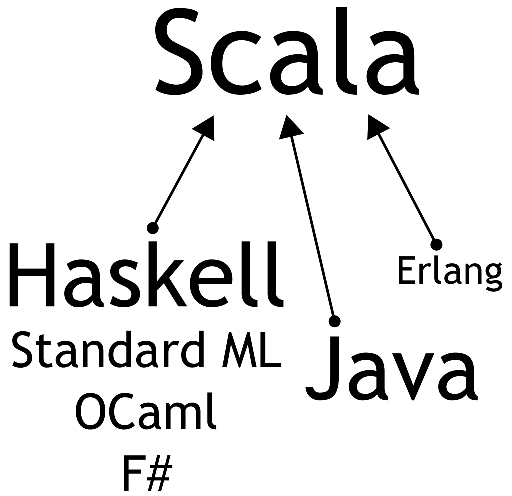
ver. 1.0

adam.szlachta@gmail.com

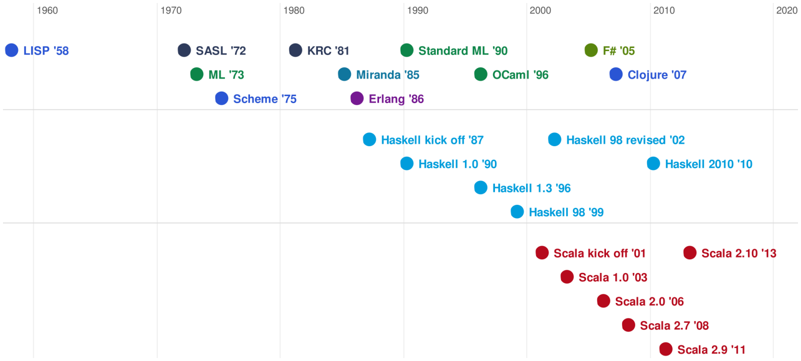
Features incorporated in Scala



Languages which influenced Scala



Functional programming languages history



Functional programming

- avoiding side effects
- avoiding state (mutable data)
- referential transparency and lazy evaluation
- first-class functions
- based on theories
  - λ-calculus (α-conversion, β-reduction, η-conversion)
  - category theory

Hello, World!

```haskell
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

Haskell

```scala
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, World!")
  }
}
```

Scala

Referential transparency

From Wikipedia

Referential transparency is a property whereby an expression can be replaced by its value without affecting the program.

Example:

```
text = reverse "redrum"
```

can be replaced with:

```
text = "murder"
```

Functon definition

```haskell
triangleArea :: Double -> Double -> Double -> Double
triangleArea a b c =
    let s = (a + b + c) / 2 in
    sqrt (s * (s - a) * (s - b) * (s - c))
```

Haskell

```scala
def triangleArea(a: Double, b: Double, c: Double): Double = {
  val s = (a + b + c) / 2
  return Math.sqrt (s * (s - a) * (s - b) * (s - c))
}
```

Scala

Currying

```
add x y = x + y

add5 = add 5

print $ add5 10
```

Haskell

```
def add(x: Int)(y: Int) = x + y

def add5 = add(5)_

println (add5(10))
```

Scala

Map, fold and lambda expressions

```haskell
add1 :: [Int] -> [Int]
add1 xs = map (\x -> x + 1) xs

sum :: [Int] -> Int
sum xs = foldr (\x y -> x + y) 0 xs


add1 :: [Int] -> [Int]
add1 xs = map (+ 1) xs

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

Haskell

```scala
def add1(xs: List[Int]): List[Int] = xs.map(x => x + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)((x, y) => x + y)


def add1(xs: List[Int]): List[Int] = xs.map(_ + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)(_ + _)
```

Scala

Point-free notation

Standard notation:

```
double x = 2*x

sum xs = foldr (+) 0 xs
```

Point-free notation:

```
double = (2*)

sum = foldr (+) 0
```

Syntax

	Haskell	Scala	Python	Java
semicollons	optional	optional	optional	obligatory
curly brackets	optional	yes***	no	yes
significant indentation	yes	no	yes	no
type inference	yes	yes	dynamic	no
functions definitions	whitespace	()*	()	()
functions call	whitespace	()**	()	()
point-free notation	yes	no	no	no

* optional for arity-0
** optional for arity-0 and arity-1
*** optional for purely functional bodies (but without val definitions)

Strict and non-strict semantics

Meaning

Lazy evaluation means evaluating expression only when it is needed.

Meaning

Non-strictness means that the evaluation of expressions proceed from the outside (e.g. from '$+$' in $(a + (b * c))$). Usually identified with lazy evaluation.

Note

Useless for not purely functional computations!

Lazy values

```
lazyArgument = g (f x)

lazyArgument = g $ f x

strictArgument = g $! f x
```

Haskell

```
lazy val lazyValue = g(f(x))

val strictValue = g(f(x))
```

Scala

Infinite streams

Haskell

```
take 10 [1..]

[1,2,3,4,5,6,7,8,9,10]
```

Scala

```
Stream.from(1).take(10).toList

List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Algebraic data types

Haskell

```haskell
data Boolean = True | False

data List a = Nil | Cons a (List a)

data Tree a = Empty
            | Leaf a
            | Node (Tree a) (Tree a)
```

Scala

```scala
trait Boolean
case class True extends Boolean
case class False extends Boolean

trait List[A]
case class Nil[A]() extends List[A]
case class Cons[A](v: A, l: List[A]) extends List[A]

trait Tree[A]
case class Empty[A]() extends Tree[A]
case class Leaf[A](v: A) extends Tree[A]
case class Branch[A](l: Tree[A], r: Tree[A]) extends Tree[A]
```

Pattern matching

```haskell
data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)

treeToString :: Show a => Tree a -> String
treeToString t = case t of
    Empty -> "empty"
    Leaf a -> "leaf " ++ show a
    Branch l r -> "branch[" ++ treeToString l ++
                        " " ++ treeToString r ++ "]"

print $ treeToString $ Branch (Branch (Leaf 2) (Leaf 3)) (Leaf 4)
```

Haskell

```scala
trait Tree[A]
case class Empty[A]() extends Tree[A]
case class Leaf[A](v: A) extends Tree[A]
case class Branch[A](l: Tree[A], r: Tree[A]) extends Tree[A]

def treeToString[A](t: Tree[A]): String = t match {
  case Empty() => "empty"
  case Leaf(a) => "leaf " + a
  case Branch(l, r) => "branch[" + treeToString(l) +
                             " " + treeToString(r) + "]"
}

println(treeToString(Branch(Branch(Leaf(2), Leaf(3)), Leaf(4))))
```

Scala

Default implementations

```
class Equal a where
    (===), (/==) :: a -> a -> Bool
    x /== y = not $ x === y
```

Haskell

```
trait Equal[_] {
  def ===(x: Equal[_]): Boolean
  def /==(x: Equal[_]): Boolean = !(this === x)
}
```

Scala

Default implementations

Haskell

```
instance Eq a => Equal (Tree a) where
    Empty === Empty = True
    Leaf x === Leaf y = x == y
    Branch l1 r1 === Branch l2 r2 = l1 === l2 && r1 === r2
    _ === _ = False
```

Scala

```
trait Tree[A] extends Equal[A]
case class Empty[A]() extends Tree[A] {
  def ===(x: Equal[_]): Boolean = x match {
    case Empty() => true
    case _ => false
  }
}
case class Leaf[A](v: A) extends Tree[A] {
  def ===(x: Equal[_]): Boolean = x match {
    case Leaf(v1) => v == v1
    case _ => false
  }
}
case class Branch[A](l: Tree[A], r: Tree[A]) extends Tree[A] {
  def ===(x: Equal[_]): Boolean = x match {
    case Branch(l1, r1) => l === l1 && r === r1
    case _ => false
  }
}
```

List comprehensions

```
[x | i <- [0..10], let x = i*i, x > 20]

genSquares :: [Int]
genSquares = do
    i <- [0..10]
    let x = i*i
    guard (x > 20)
    return x
```

Haskell

Works in any monadic context.

```
for { i <- List.range(0, 11); x = i*i; if x > 20 } yield x

def genSquares(): List[Int] = for {
  i <- List.range(0, 11)
  x = i*i
  if x > 20
} yield x
```

Scala

Works for any type implementing map/flatMap/filter.

Monadic notation

Haskell

```
do
    x <- Just 8                     Just 8 >>= \x ->
    y <- fun1 x                     fun1 x >>= \y ->
    z <- fun2 y                     fun2 y >>= return
    return z

do
    x <- Just 8                     Just 8 >>= \x ->
    y <- fun1 x                     fun1 x >>= \y ->
    fun2 y                          fun2 y
```

Scala

```
for {
  x <- Some(8)                  Some(8).flatMap (x =>
  y <- fun1(x)                  fun1(x).flatMap (y =>
  z <- fun2(y)                  fun2(y).map     (z =>
} yield z                           z)))
```

I/O isolation

```haskell
getLine :: IO String
getLine = ...
putStr :: String -> IO ()
putStr = ...

getLineWithPrompt :: String -> IO String
getLineWithPrompt prompt = do
    putStr prompt
    getLine

line :: IO String
line = getLineWithPrompt "> "
```

Haskell

```scala
object Console {
  def readLine(): String = { ... }
  def print(obj: Any) { ... }
}

def getLineWithPrompt(prompt: String): String = {
  Console.print(prompt)
  Console.readLine()
}

val line: String = getLineWithPrompt("> ")
```

Scala

Features comparison

	Haskell	Scala	Java
strong static typing	yes	yes	yes
type inference	yes	yes	no
higher order types	yes	yes	yes**
algebraic data types	yes	yes (verbose)	no
infinite streams	yes	yes	no*
strict semantics	optional	default	yes
lazy evaluation	default	optional	no
currying	default	optional	no
lambda expressions	yes	yes	no*
immutability	enforced	not enforced	not enforced
side effects isolation	yes	no	no
default implementations	yes	yes	no*

* will be in Java 8
** not as good as in Haskell/Scala