Haskell and Scala

comparison

Adam Szlachta

March 20, 2013

ver. 1.0
adam.szlachta@gmail.com

Introduction

- Haskell and Scala introduction
- History
- Functional programming essence
- Syntax and features comparison
- Scala libraries influenced by Haskell
- Summary

Features incorporated in Scala

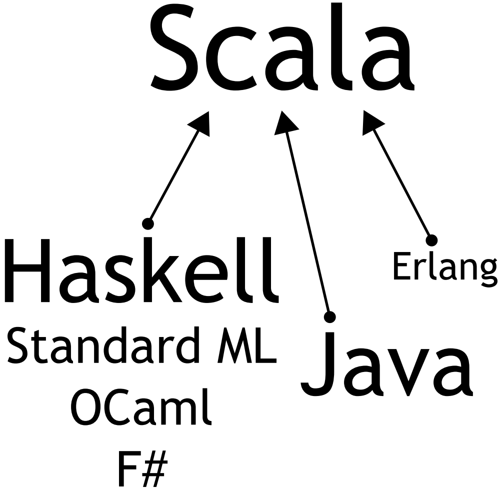
Scala

strong static typing
type inference
first-class functions
currying
lazy evaluation
list comprehensions
immutability
algebraic data types
higher order types
monads

OOP

actors

Languages which influenced Scala



Haskell logo and name

From Wikipedia

Lambda calculus (also written as λ-calculus or called "the lambda calculus") is a formal system in mathematical logic and computer science for expressing computation by way of variable binding and substitution.

From Wikipedia

Haskell Brooks Curry (1900-1982) was an American mathematician and logician. Curry is best known for his work in combinatory logic.

Haskell logo and name

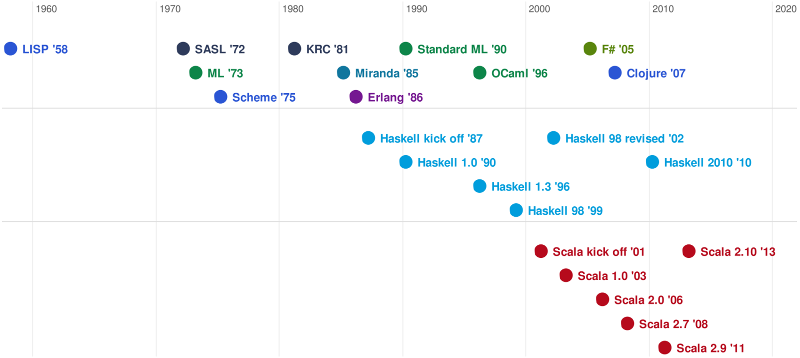
From Wikipedia

Lambda calculus (also written as λ-calculus or called "the lambda calculus") is a formal system in mathematical logic and computer science for expressing computation by way of variable binding and substitution.

From Wikipedia

Haskell Brooks Curry (1900-1982) was an American mathematician and logician. Curry is best known for his work in combinatory logic.

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

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

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

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

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

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

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

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

Who is this?

Referential transparency

The Polish Parliament meets in **the capital of Poland**.

The Polish Parliament meets in **Warsaw**.

Warsaw has been **the capital of Poland** since 1815.

Referential transparency

The Polish Parliament meets in **the capital of Poland**.

The Polish Parliament meets in **Warsaw**.

Warsaw has been **the capital of Poland** since 1815.

Referential transparency

The Polish Parliament meets in **the capital of Poland**.

The Polish Parliament meets in **Warsaw**.

Warsaw has been **the capital of Poland** since 1815.

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"
```

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"
```

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"
```

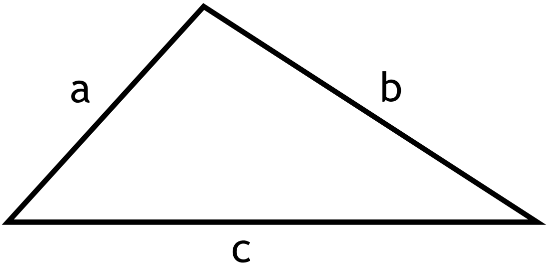
can be replaced with:

```
text = "murder"
```

Who is this?

Heron's formula

$T = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = \frac{a+b+c}{2}$



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

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

Functon definition

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

```
triangleArea a b c =
    sqrt (s * (s - a) * (s - b) * (s - c))
    where
        s = (a + b + c) / 2
```

Haskell

Currying


```haskell
add x y = x + y

add5 = add 5

print $ add5 10
```

Haskell

```scala
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

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
```

```haskell
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)
```

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

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

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

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
```

Haskell

Point-free notation

Standard notation:

```
double x = 2*x

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

Point-free notation:

```
double = (2*)

sum = foldr (+) 0
```

Haskell

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!

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!

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

Haskell

```haskell
lazyArgument = g (f x)
lazyArgument = g $ f x
strictArgument = g $! f x
```

Scala

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

Lazy values

Haskell

```haskell
lazyArgument = g (f x)

lazyArgument = g $ f x

strictArgument = g $! f x
```

Scala

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

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

Lazy values


```haskell
lazyArgument = g (f x)
lazyArgument = g $ f x
strictArgument = g $! f x
```

Haskell

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

Scala

Infinite streams

```
take 10 [1..]

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

Haskell

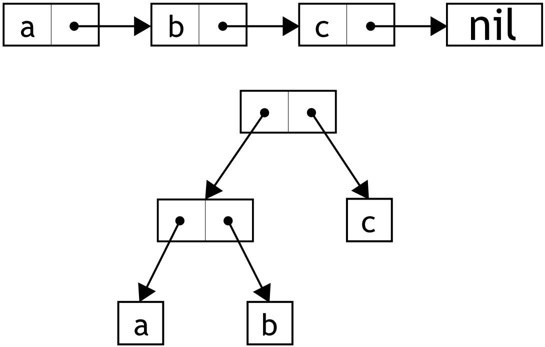
```
Stream.from(1).take(10).toList

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

Scala

Algebraic data types

From Wikipedia

Algebraic data type is a kind of composite type, i.e. a type formed by combining other types. Two common classes of algebraic type are product types, i.e. tuples and records, and sum types, also called tagged unions or variant types.



Algebraic data types


```haskell
data Boolean = True | False

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

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

Haskell

```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]
```

Scala

Algebraic data types


```haskell
data Boolean = True | False

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

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

Haskell

```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]
```

Scala

Algebraic data types

```haskell
data Boolean = True | False

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

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

Haskell

```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]
```

Scala

Algebraic data types


```haskell
data Boolean = True | False

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

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

Haskell

```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]
```

Scala

Algebraic data types

Note

Algebraic data type can be recursive and act as unions, structs and enums.

```haskell
data DaysOfWeek = Monday | Tuesday | Wednesday | Thursday
    | Friday | Saturday | Sunday

data Account = Account
    { number    :: Int
    , firstName :: String
    , lastName  :: String
    , balance   :: Float }

data Account = Account Int String String Float

data Tree = Branch { left  :: Tree
                   , value :: Int
                   , right :: Tree }
          | Leaf { value :: Int }

data Tree = Branch Tree Int Tree | Leaf Int
```

Haskell

Algebraic data types

Note

Algebraic data type can be recursive and act as unions, structs and enums.

```
data DaysOfWeek = Monday | Tuesday | Wednesday | Thursday
    | Friday | Saturday | Sunday

data Account = Account
    { number    :: Int
    , firstName :: String
    , lastName  :: String
    , balance   :: Float }

data Account = Account Int String String Float

data Tree = Branch { left  :: Tree
                   , value :: Int
                   , right :: Tree }
          | Leaf { value :: Int }

data Tree = Branch Tree Int Tree | Leaf Int
```

Haskell

Algebraic data types

Note

Algebraic data type can be recursive and act as unions, structs and enums.

```
data DaysOfWeek = Monday | Tuesday | Wednesday | Thursday
    | Friday | Saturday | Sunday

data Account = Account
    { number    :: Int
    , firstName :: String
    , lastName  :: String
    , balance   :: Float }

data Account = Account Int String String Float

data Tree = Branch { left  :: Tree
                   , value :: Int
                   , right :: Tree }
          | Leaf { value :: Int }

data Tree = Branch Tree Int Tree | Leaf Int
```

Haskell

Algebraic data types

Note

Algebraic data type can be recursive and act as unions, structs and enums.

```
data DaysOfWeek = Monday | Tuesday | Wednesday | Thursday
    | Friday | Saturday | Sunday

data Account = Account
    { number    :: Int
    , firstName :: String
    , lastName  :: String
    , balance   :: Float }

data Account = Account Int String String Float

data Tree = Branch { left  :: Tree
                   , value :: Int
                   , right :: Tree }
          | Leaf { value :: Int }

data Tree = Branch Tree Int Tree | Leaf Int
```

Haskell

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

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

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

Pattern matching


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

```haskell
treeToString t = case t of
    Empty -> "empty"
    Leaf a -> "leaf " ++ show a
    Branch l r -> "branch[" ++ treeToString l ++
                      " " ++ treeToString r ++ "]"
```

```haskell
treeToString Empty = "empty"
treeToString (Leaf a) = "leaf " ++ show a
treeToString (Branch l r) =
    "branch[" ++ treeToString l ++ " " ++ treeToString r ++ "]"
```

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

```
"branch[branch[leaf 2 leaf 3] leaf 4]"
```

Haskell

Default implementations


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

Haskell

```scala
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
```

Haskell

```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
  }
}
```

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
```

Haskell

```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
  }
}
```

Scala

Who is this?

List comprehensions


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

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

Works in any monadic context.

Haskell

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

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

Works for any type implementing map/flatMap/filter.

Scala

List comprehensions


```haskell
[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.

```scala
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.

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
```

Works in any monadic context.

Haskell

```
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
```

Works for any type implementing map/flatMap/filter.

Scala

List comprehensions

```haskell
[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.

```scala
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
    y <- fun1 x
    z <- fun2 y
    return z
```

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

```
do
    x <- Just 8
    y <- fun1 x
    fun2 y
```

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

Scala

```
for {
  x <- Some(8)
  y <- fun1(x)
  z <- fun2(y)
} yield z
```

```
Some(8).flatMap (x =>
fun1(x).flatMap (y =>
fun2(y).map     (z =>
     z)))
```

Monadic notation


```haskell
do
    x <- Just 8
    y <- fun1 x
    z <- fun2 y
    return z
```

```haskell
Just 8 >>= \x ->
fun1 x >>= \y ->
fun2 y >>= return
```

```haskell
do
    x <- Just 8
    y <- fun1 x
    fun2 y
```

```haskell
Just 8 >>= \x ->
fun1 x >>= \y ->
fun2 y
```

Haskell

```scala
for {
  x <- Some(8)
  y <- fun1(x)
  z <- fun2(y)
} yield z
```

```scala
Some(8).flatMap (x =>
fun1(x).flatMap (y =>
fun2(y).map     (z =>
     z)))
```

Scala

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

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

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

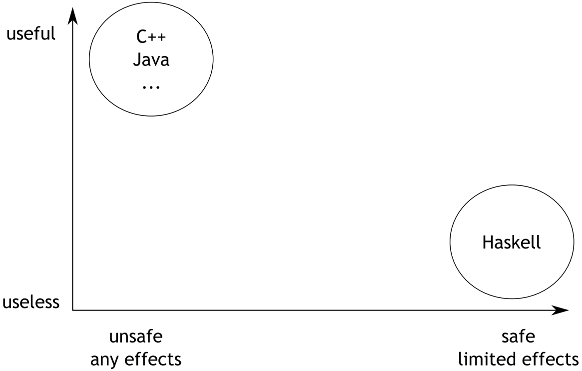
	Haskell	Scala	Java
strong static typing	yes	yes	yes
type inference	yes	yes	no
higher order types	yes	yes	yes**
algebraic data types	yes	yes (verbose)	no
infinite streams	yes	yes	no*
strict semantics	optional	default	yes
lazy evaluation	default	optional	no
currying	default	optional	no
lambda expressions	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

Who is this?

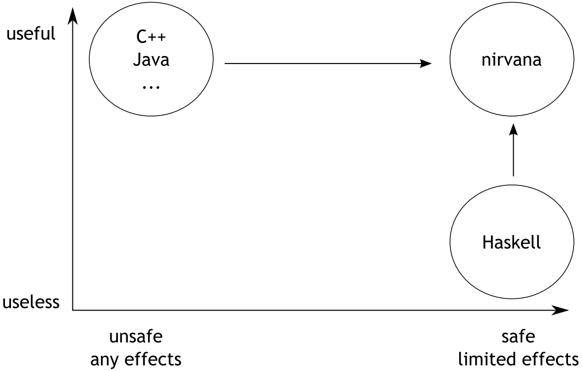
Haskell is useless

Simon Peyton Jones



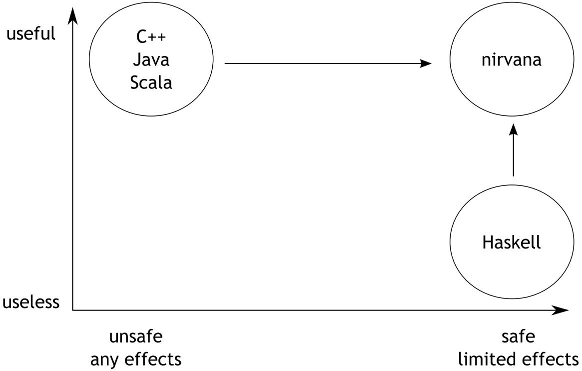
Haskell is useless

Simon Peyton Jones



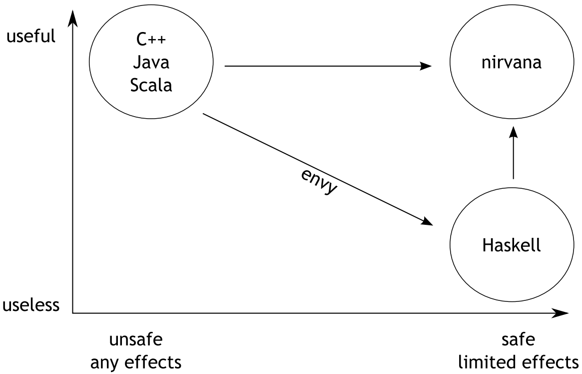
Haskell is useless

Simon Peyton Jones



Haskell is useless

Simon Peyton Jones



Libraries and tools inspired by Haskell

- Type classes library Scalaz
  (Haskell standard library)
- Combinator parser
  (Haskell: Parsec, attoparsec, polyparse)
- Automated specification-based testing ScalaCheck
  (Haskell: QuickCheck)

Libraries and tools inspired by Haskell

- Type classes library Scalaz
  (Haskell standard library)
- Combinator parser
  (Haskell: Parsec, attoparsec, polyparse)
- Automated specification-based testing ScalaCheck
  (Haskell: QuickCheck)

Libraries and tools inspired by Haskell

- Type classes library Scalaz
  (Haskell standard library)
- Combinator parser
  (Haskell: Parsec, attoparsec, polyparse)
- Automated specification-based testing ScalaCheck
  (Haskell: QuickCheck)

Resources

Links

- http://808Fabrik.com/scala
- http://hyperpolyglot.org/ml
- http://downgra.de
- http://hseeberger.wordpress.com
- http://code.google.com/p/scalaz/
- http://code.google.com/p/scalacheck/
- http://www.artima.com/pins1ed/combinator-parsing.html
- http://www.haskell.org/haskellwiki/Typeclassopedia
- http://typeclassopedia.bitbucket.org

Books and papers

- Eugenio Moggi, “Notions of computation and monads”
- Philip Wadler, “Comprehending Monads”
- Philip Wadler, “Monads for functional programming”
- Conor McBride, Ross Paterson, “Applicative programming with effects”
- Ross Paterson, “Arrows and computation”
- Jeff Newbern, “All About Monads”
- Brent Yorgey, “The Typeclassopedia” in “The Monad.Reader Issue 13”