

# Scalaz

## Learn You Yet Another Real World Gentle Haskell (LYYARWGH) ((c) sproingie)

George Leontiev

deltamethod GmbH

April 17, 2013

( $\lambda x$ .folonexlambda-calcul.us)@  
folone.info

# Agenda

- Some hotness without context, to draw attention (Option, Boolean, Memo)
- Typeclasses
- Monoid
- Functor, Applicative, Monad
- Effects
- scalaz 6 vs seven
- typelevel.org

# What is scalaz

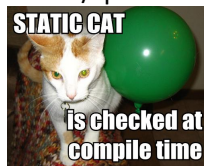
- Purely functional datatypes (Fingertree, HList, DList, Trees, Zippers, Nel, ImmutableArray)
- Typeclasses
- Effects
- Concurrency

## Examples -- typesafe equals

```
s> "" == 5  
res0: Boolean = false
```

```
s> "" === 5  
<console>:14: error: type mismatch;  
found    : Int(5)  
required: java.lang.String  
      "" === 5  
             ^
```

<spoiler> $\forall \text{stuff} \in \text{scalaz} \equiv \text{scala.stdlib} \mid \text{stuff is typesafe} \vee \text{stuff is strict}$ </spoiler>



## Examples -- options

```
s> some(5) getOrElse 0
res1: Int = 5
s> some(5) | 0
res2: Int = 5
s> some(1) getOrElse "ok"
res3: Any = 1
s> some(1) | "ok"
<console>:14: error: type mismatch;
 found   : java.lang.String("ok")
required: Int
    some(1) | "ok"
              ^
s> ~some(5) // Monoids
res4: Int = 5
s> ~none[Int] // NB: Beware of unary_~ on Validations
res5: Int = 0
```

## Examples -- options II

*// Smart constructors*

```
s> :t Some(1)
Some[Int]
```

```
s> :t None
```

```
None.type
```

```
s> :t some(1)
Option[Int]
```

```
s> :t none[Int]
```

```
Option[Int]
```

```
s> List(Some(1),None).foldLeft(None){(_, v) => v}
<console>:14: error: type mismatch;
 found   : v.type (with underlying type Option[Int])
 required: None.type
```

```
    List(Some(1),None).foldLeft(None){(_, v) => v}
                                     ^
```

```
s> List(Some(1),None).foldLeft(none[Int]){(_, v) => v}
res11: Option[Int] = None
```

## Examples -- booleans

```
scala> true ? println("true") | println("false")
true
```

```
scala> true ?? 5
res14: Int = 5
```

```
scala> true !? 5
res15: Int = 0
```

```
scala> false ?? 5
res15: Int = 0
```

```
scala> false !? 5
res17: Int = 5
```

## Examples -- function composition

```
val a = (_:Int) + 6
val b = (_:Int).toString
val c = (_:String).length
```

```
scala> 5 |> a |> b |> c
res18: Int = 2
```

```
scala> //(c . b . a) apply 5 // contramap
res19: Int = 2
```

```
scala> 5 |> //(a o b o c) // map
res20: Int = 2
```

```
// contramap == flip . map
```



## Examples -- Memo

```
def func(s: String) = // Expensive computation
scala> Memo.immutableHashMapMemo(func)
res11: String => java.lang.String = <function1>

// Different strategies
mutableHashMapMemo
arrayMemo // sized
immutableListMemo
immutableTreeMapMemo
doubleArrayMemo // memoizing Double results != sentinel
weakHashMapMemo // GC
```

## Examples -- Trampoline



```
def even(n: Int): Boolean =  
  if (n == 0) true  
  else odd(n - 1)  
def odd(n: Int): Boolean =  
  if (n == 0) false  
  else even(n - 1)
```

```
scala> even(30000)
```

# Examples -- Trampoline



```
def even(n: Int): Trampoline[Boolean] =  
  if (n == 0) done(true)  
  else suspend(odd(n - 1))  
def odd(n: Int): Trampoline[Boolean] =  
  if (n == 0) done(false)  
  else suspend(even(n - 1))
```

```
scala> even(30000).run
```

## Examples -- Trampoline

```
def fibRec(n: Int): Int =  
  if (n < 2) n else fibRec(n - 1) + fibRec(n - 2)  
  
def fibTailrec(n: Int) = {  
  def loop(n: Int, next: Int, result: Int) = n match {  
    case 0 => result  
    case _ => loop(n - 1, next + result, next)  
  }  
  loop(n, 1, 0)  
}
```

## Examples -- Trampoline

```
def fibTramp(n: Int): Trampoline[Int] =  
  if (n < 2) done(n) else suspend {  
    for {  
      i <- fibTramp(n - 1)  
      j <- fibTramp(n - 2)  
    } yield i + j  
  }  
// Continuation monad magic
```



Consult @runarorama's paper "Stackless Scala with Free Monads"

# Typeclasses



<http://www.haskell.org/haskellwiki/Typeclassopedia>  
<http://typeclassopedia.bitbucket.org/>

# Monoids

$$\begin{aligned} & (S, \otimes, 1) \\ & \forall a, b \in S : a \otimes b \in S \\ & \forall a, b, c \in S : (a \otimes b) \otimes c = a \otimes (b \otimes c) \\ & \forall a \in S : 1 \otimes a = a \otimes 1 = a \end{aligned}$$

```
trait Semigroup[F] {  
  def append(a1: F, a2: F): F  
}  
trait Monoid[F] extends Semigroup[F] {  
  def zero: F  
}  
// scalacheck-binding  
import scalaz.scalacheck.ScalazProperties._  
semigroup.laws[Int]  
monoid.laws[String]
```

# Monoids

```
scala> 1 |+| 5  
res2: Int = 6
```

```
scala> Multiplication(2) |+| Multiplication(3)  
res4: Int @@ Multiplication = 6
```

```
scala> some(1) |+| some(5)  
res5: Option[Int] = Some(6)
```

*// Monoids beget monoids*

```
scala> some(some((1, "OH ", 1 + (_:Int)))) |+|  
      some(some((4, "HAI", 2 * (_:Int))))  
res6: Option[Option[(Int, java.lang.String,  
    Int => Int)]] = Some(Some((5, OH HAI,  
    <function1>)))
```

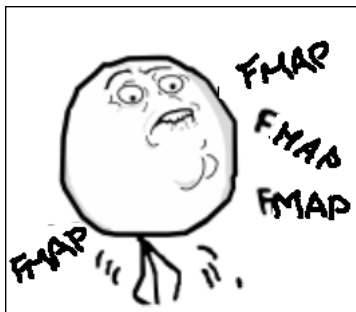


# Monoids

```
scala> List(1,2,3).sum
res16: Int = 6
```

```
scala> List("OH ", "HAI", "!").sum
res17: java.lang.String = OH HAI!
```

# Functors



```
trait Functor[F[_]] {  
  def fmap[A, B](f: A => B): F[A] => F[B]  
}
```

```
scala> some(3) map(_.toString)  
res13: Option[java.lang.String] = Some(3)
```

# Applicatives

```
trait Applicative[T[_]] extends Functor[T] {  
  def pure[A](a: A): T[A]  
  def <*>[A, B](tf: T[A => B])(ta: T[A]): T[B]  
}
```

```
scala> some(1) <*> some((_:Int) + 2) <*> some((_:Int) * 5)  
res10: Option[Int] = Some(15)
```

```
scala> List(1,2) <*> List((_:Int) * 5, (_: Int) + 2)  
res12: List[Int] = List(5, 10, 3, 4)
```

# Applicatives

```
scala> List(some(1), some(2), some(3))  
res21: List[Option[Int]] = List(Some(1), Some(2),  
                                Some(3))
```

```
scala> .sequence  
res22: Option[List[Int]] = Some(List(1, 2, 3))
```

```
scala> res21.traverse(x => some(x))  
res23: Option[List[Int]] = Some(List(1, 2, 3))
```

# Monads

```
trait Monad[M[_]] extends Applicative[M]{  
  def >>=[A, B](ma: M[A])(f: A => M[B]): M[B]  
}
```

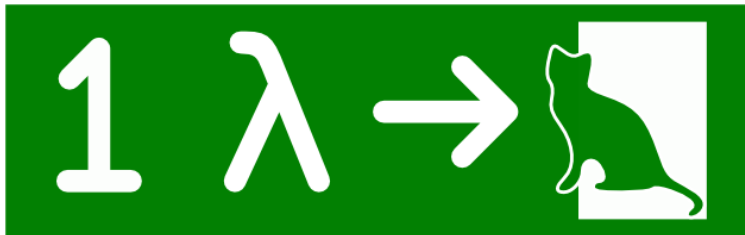
```
scala> for {  
  | i <- List(1,2,3)  
  | j <- List(4,5,6)  
  | } yield i*j  
res15: List[Int] = List(4, 5, 6, 8, 10, 12, 12, 15, 18)
```

# DEMO

trapd in IO monad



That's it



Questions?