Scalaz 102

Level Up Your Scalaz Foo

Colt Frederickson
@coltfred

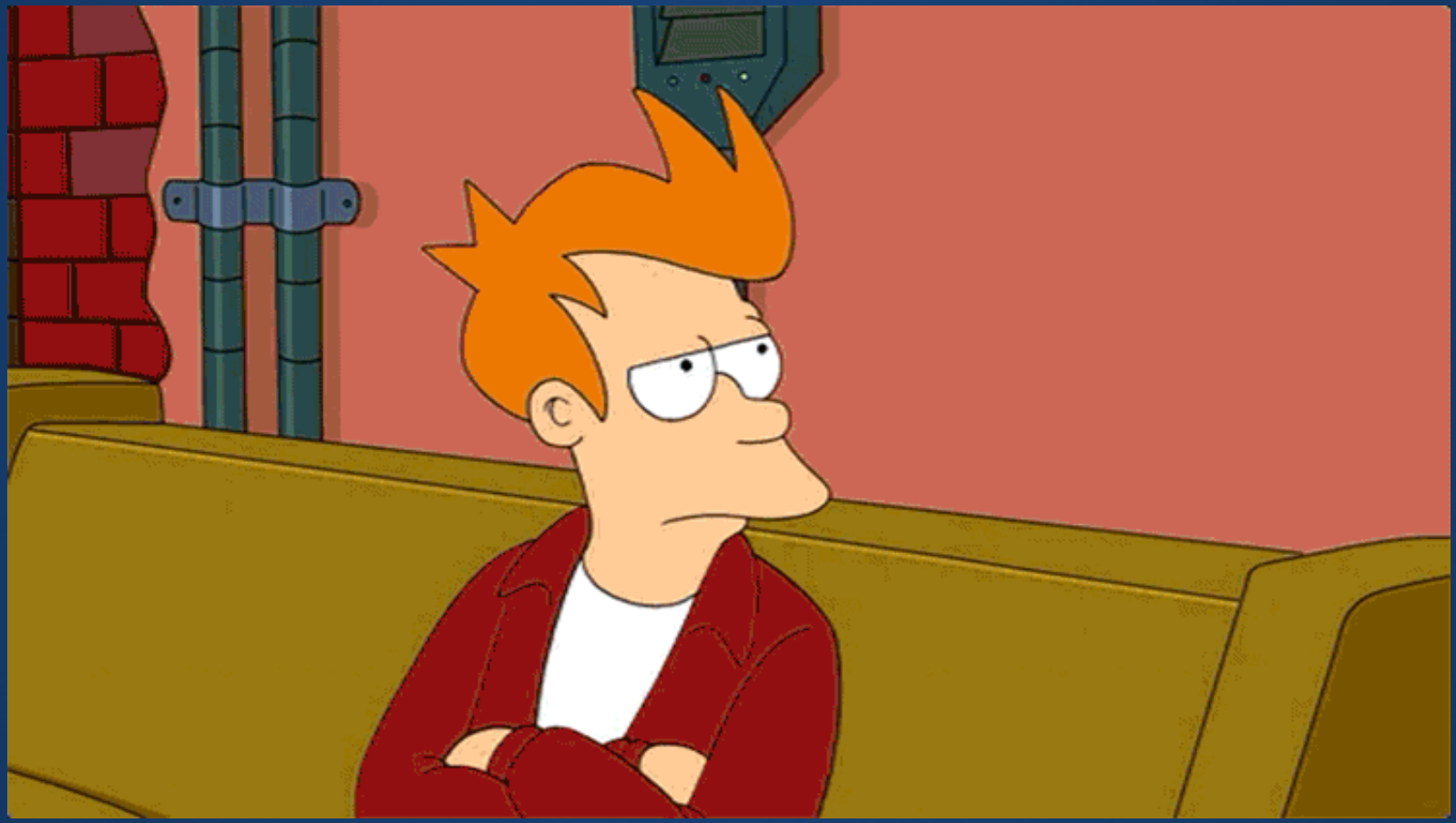
Who am I?

- Working on a small "Big Data" team at Oracle
- Writing Scala since 2011
- Functional Programming Convert
- Haskell Dabbler

Ask Questions!

What is Scalaz?

What is Scalaz?



What is Scalaz?

```
trait MonadTrans[F[_[_], _]] {
  ...
}
```

What is Scalaz?

```
type ->[A, B] = (({type λ[α]=A})#λ) ~> (({type λ[α]=B})#λ)
```

No Really...

- Data Types
  - IO
  - Task
  - \/
  - NonEmptyList
  - etc.
- Better abstractions (type classes)
  - Functor
  - Applicative
  - Monad
  - Foldable
  - Traverse
  - etc.

Scalaz Package Structure

Data Types

```
//Data types are under scalaz
import scalaz.NonEmptyList
import scalaz.Kleisli
import scalaz.Zipper
import scalaz.Monad
import scalaz.Functor
//etc

//Or get them all
import scalaz._
```

Scalaz Package Structure

Syntax

```
// Adds .right and .left, etc to everything.
import scalaz.syntax.either._

// Adds =/= and ===, etc to things that have an Equal typeclass.
import scalaz.syntax.equal._

// Adds .map, strengthL, strengthR, etc to things with Functor typeclass.
import scalaz.syntax.functor._

//Or get them all!
import scalaz.syntax.all._
```

Scalaz Package Structure

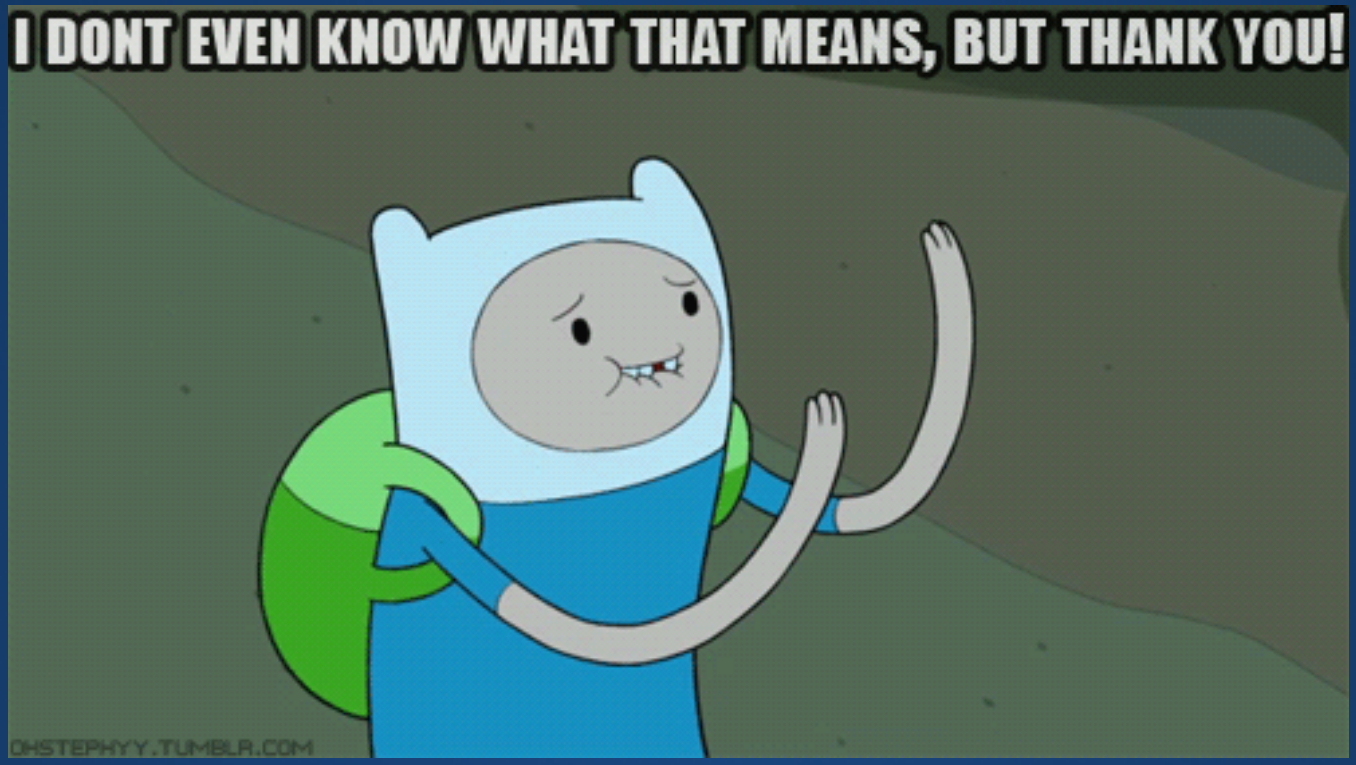
Syntax

```
// Adds .intersperse, .filterM, etc.
import scalaz.syntax.std.list._

// Adds .toRightDisjunction, etc.
import scalaz.syntax.std.option._

//Or get them all!
import scalaz.syntax.std.all._
```

Scalaz Package Structure

Typeclass Instances

```
//Bring in instances for std lib types
import scalaz.std.option._
import scalaz.std.list._
//etc
```



https://www.gliffy.com/go/publish/4950560

Let's Dig In!

- Types
  - NonEmptyList
  - Validation
- Typeclasses
  - Semigroup
  - Monoid
  - Foldable
  - Traverse

NonEmptyList

Motivation:

```
type Error = Throwable //Not important

def validateUsers(l:List[User]): Error \/ List[User] = ... //Is Nil really valid???
```

OneAnd

Definition:

```
final case class OneAnd[F[_], A](head: A, tail: F[A])

//More concretely
type NonEmptyList[A] = OneAnd[List,A]

//Scalaz provides an actual type for NEL though!
final class NonEmptyList[+A] private[scalaz](val head: A, val tail: List[A])
```

NonEmptyList

Use the types! Don't be a jerk!

NonEmptyList

Constructors:

```
def nel[A](h: A, t: List[A]): NonEmptyList[A] = new NonEmptyList(h, t)

def nels[A](h: A, t: A*): NonEmptyList[A] = nel(h, t.toList)
```

Operations:

```
//Roughly the same as List, except...
val head: A

//In case of emergency you can just call
def list: List[A] = head :: tail
```

NonEmptyList

Validation:

```
type Error = Throwable //Not important

def validateUsers(l:List[User]): Error \/ List[User] = ...

//OR

def validateUsers(l:List[User]): Error \/ NonEmptyList[User] = ...
```

NonEmptyList

Parsers:

```
//Shamelessly stolen from Atto

//0 to N matches
def many[A](p: => Parser[A]): Parser[List[A]]

//1 to N matches
def many1[A](p: => Parser[A]): Parser[NonEmptyList[A]]
```

NonEmptyList

BE PRECISE!!!

Validation

Validation:

```
type Error = Throwable //Not important

//But what if we had more than 1 error?
def validateUsers(l:List[User]): Error \/ NonEmptyList[User] = ...
```

Validation

Definition:

```
sealed abstract class Validation[+E, +A]

final case class Success[A](a: A) extends Validation[Nothing, A]
final case class Failure[E](e: E) extends Validation[E, Nothing]
```

Validation

Operations:

```
/** Convert to a disjunction. */
def disjunction: (E \/ A)

/** Throw away the error and return Nil or return the Success in a List */
def toList: List[A]

/** Catamorphism. Run the first given function if failure, otherwise, the second given functio
def fold[X](fail: E => X, succ: A => X): X

/** Wraps failures in a NonEmptyList */
def toValidationNel: ValidationNel[E, A]

//Many many more!
```

Validation

ValidationNel:

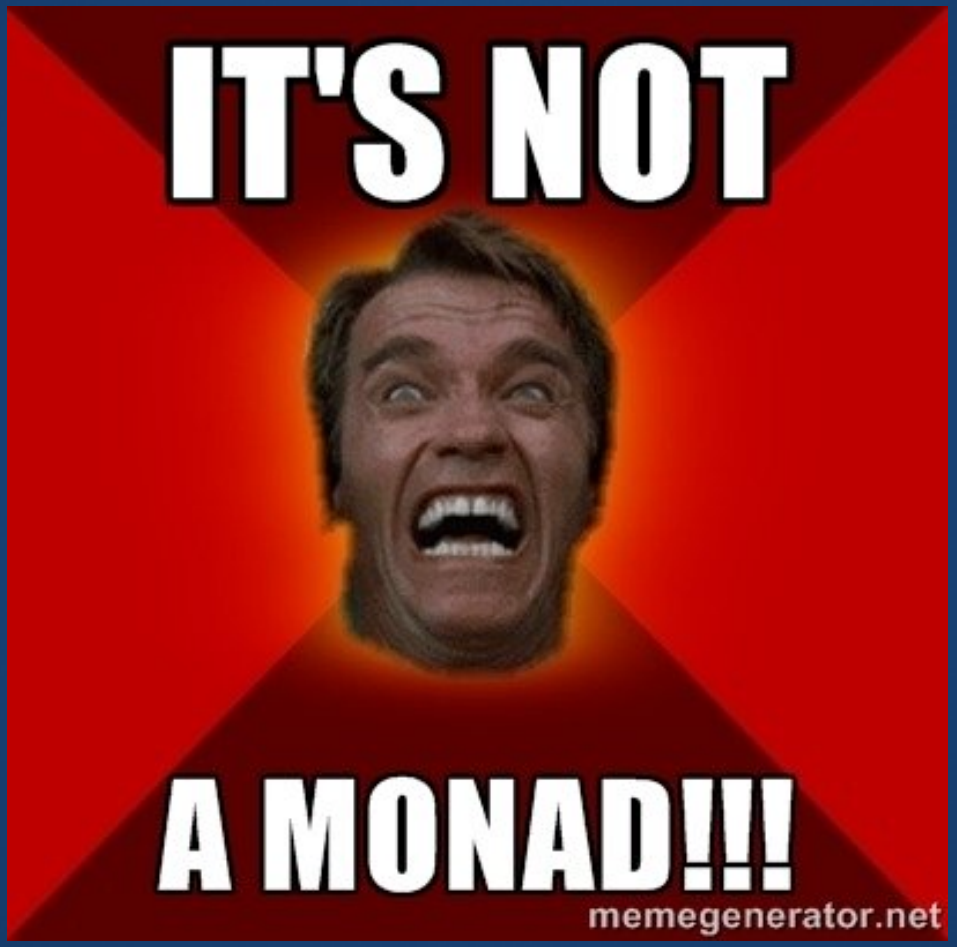
```
type ValidationNel[E, A] = Validation[NonEmptyList[E], A]

//So maybe we really want
def validateUsers(l:List[User]): ValidationNel[E, NonEmptyList[User]] = ...
```

Validation

Examples:

```
//The type in [] is the *error* type on success.
(1.success[String] |@| 2.success[String]) {_ + _}
// Success(3)

(1.success[String] |@| "error".failure[Int]) {_ + _}
// Failure(error)

("error2".failure[Int] |@| "error".failure[Int]) {_ * _}
// Failure(error2error)

("error2".failureNel[Int] |@| "error".failureNel[Int]) {_ * _}
// Failure(NonEmptyList(error2,error))
```

Validation




FYS
And that's the end of that chapter.

Semigroup

Motivation:

```
def updateAppendIntMap[A](m: Map[A,Int])(k: A, v: Int) = {
  val newValue = m.get(k).map(_ + v).getOrElse(v)
  m + (k -> newValue)
}

updateAppendMap(Map("A" -> 1))("A", 2)
// Yields Map(A -> 3)
```

Semigroup

Definition:

```
trait Semigroup[F]  {
  // |+| is also used for this.
  def append(f1: F, f2: => F): F
}
```

Laws:

```
// Associativity
(a |+| b) |+| c == a |+| (b |+| c)

// Or...
append(append(a, b), c) == append(a, append(b,c))
```

Semigroup

Example:

```
def updateAppendMap[A, B: Semigroup](m: Map[A,B])(k: A, v: B) = {
  val newValue = m.get(k).map(_ |+| v).getOrElse(v)
  m + (k -> newValue)
}

updateAppendMap(Map("A" -> List(1)))("A", List(2))
// Yields Map(A -> List(1, 2))

updateAppendMap(Map("A" -> List(1)))("B", List(2))
// Yields Map(A -> List(1), B -> List(2))
```

Monoid

Motivation:

```
List(1,2,3).sum
// Yields 6

List[Int]().sum
// Yields 0

List(Some(1), Some(2), None).sum
// <console>:14: error: could not find implicit value for parameter num: Numeric[Option[Int]]


//BUT I KNOW HOW TO APPEND THEM!!!
```

Monoid

Definition:

```
trait Monoid[F] extend Semigroup[F] {
  def zero: F
}
```

Laws:

```
// Associativity
(a |+| b) |+| c == a |+| (b |+| c)

append(append(a, b), c) == append(a, append(b,c))

// Left identity
a == append(zero, a)

// Right identity
a == append(a, zero)
```

Monoid

Gives us:

```
def multiply(value: F, n: Int): F =
  Stream.fill(n)(value).foldLeft(zero)((a,b) => append(a,b))

/** Whether `a` == `zero`. */
def isMZero(a: F)(implicit eq: Equal[F]): Boolean = eq.equal(a, zero)

def ifEmpty[B](a: F)(t: => B)(f: => B)(implicit eq: Equal[F]): B =
  if (isMZero(a)) { t } else { f }
```

Monoid

Instances:

- Int (Product)
- Int (Sum)
- List
- String
- Option (First)
- Option (Last)
- Tuples
- Map

Monoid

Example :

```
import scalaz.Monoid
import scalaz.syntax.monoid._ // for |+|

def betterSum[A: Monoid](l: TraversableOnce[A]): A =
  l.foldLeft(Monoid[Option[Int]].zero)((acc, a) => acc |+| a)

betterSum(List("a","b","c"))
// Yields "abc"

betterSum(List(Some(1), Some(2), None))
// Yields Some(3)
```

Monoid

Example :

```
import scalaz.Monoid
import scalaz.syntax.monoid._ // for |+|
import scalaz.std.map._ // For Map instances

Map("A" -> 2) |+| Map("A" -> 3)
// Yields Map("A" -> 5)

Map("A" -> 2) |+| Map("B" -> 3)
// Yields Map("A" -> 2, "B" -> 3)
```

Monoid

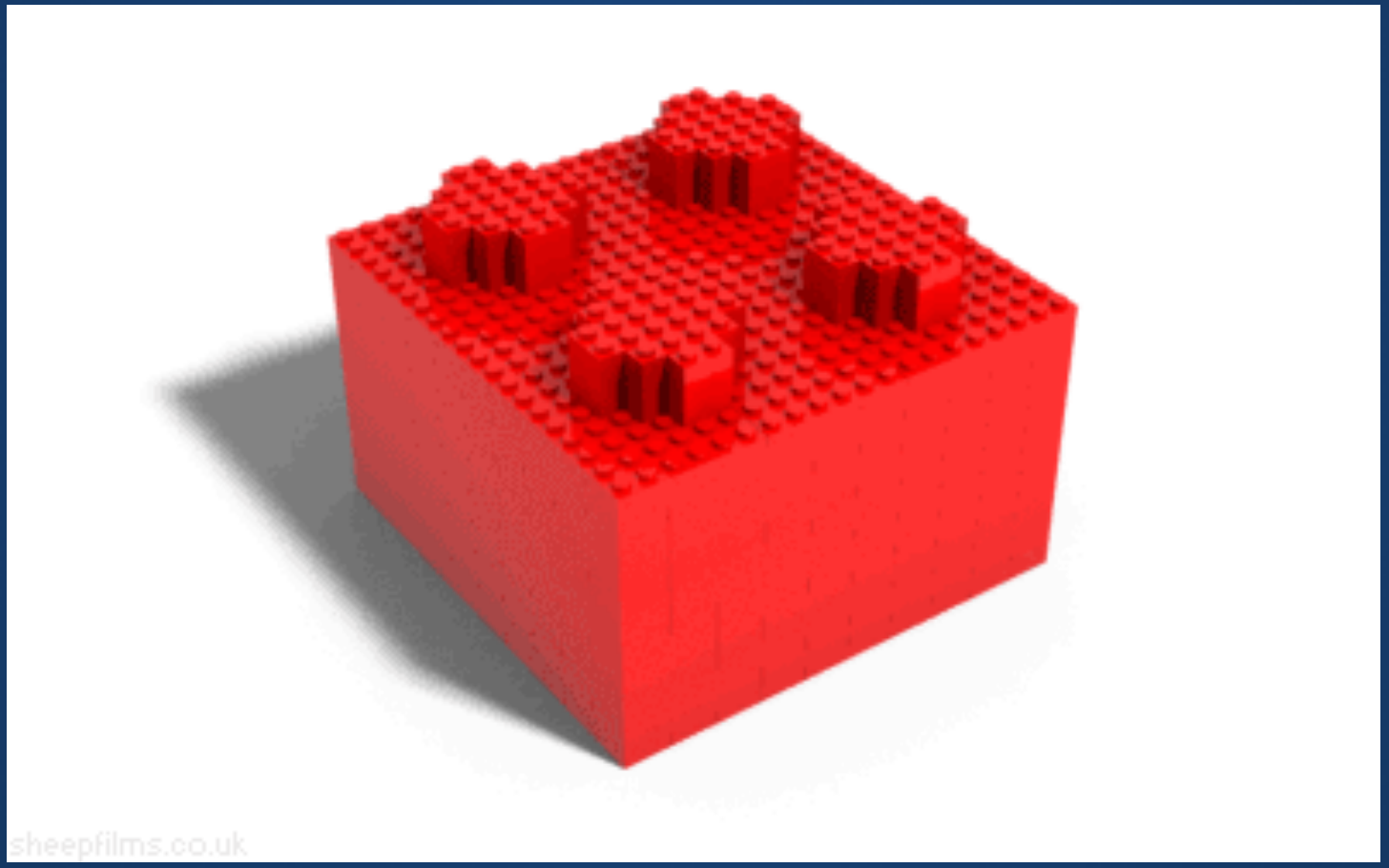
Example :

```
import scalaz.Monoid
import scalaz.syntax.monoid._ // for |+|
import scalaz.std.map._ // For Map instances

Map("A" -> 2) |+| Map("A" -> 3)
// Yields Map("A" -> 5)

Map("A" -> 2) |+| Map("B" -> 3)
// Yields Map("A" -> 2, "B" -> 3)
```

Map is a Monoid if the values can be appended!

(if there is Semigroup for them)

Monoid



Foldable

Motivation:

```
Map("A" -> 1, "B" -> 2).sum
//<console>:14: error: could not find implicit value for parameter num: Numeric[(String,Int)

//UGH!!!!!
```

Foldable

Definition:

```
trait Foldable[F[_]]  {
  /** Map each element of the structure to a [[scalaz.Monoid]], and combine the results. */
  def foldMap[A,B](fa: F[A])(f: A => B)(implicit F: Monoid[B]): B

  /**Right-associative fold of a structure. */
  def foldRight[A, B](fa: F[A], z: => B)(f: (A, => B) => B): B
}
```

Laws:

```
//foldLeft where you append the value is equal to foldMap
foldMap(fa)(Vector(_)) == foldLeft(fa, Vector.empty[A])(_ :+ _)

//foldRight where you prepend the value is equal to foldMap
foldMap(fa)(Vector(_)) == foldRight(fa, Vector.empty[A])(_ +: _)
```

Foldable

Gives Us:

```
//Count how long the foldable structure is
def length[A](fa: F[A]): Int

//Convert to collections
def toList[A](fa: F[A]): List[A]
def toIndexedSeq[A](fa: F[A]): IndexedSeq[A]
def toSet[A](fa: F[A]): Set[A]

//Get Max, Min, etc
def minimumBy[A, B: Order](fa: F[A])(f: A => B): Option[A]
def maximumBy[A, B: Order](fa: F[A])(f: A => B): Option[A]

//foldMap when the value is a Monad.
def foldMapM[G[_], A, B](fa: F[A])(f: A => G[B])(implicit B: Monoid[B], G: Monad[G]): G[B]

//Plus *many* more
```

Foldable

Example:

```
List(1,2,3,4).foldMap(identity)
//yields 10

//This is the same as
List(1,2,3,4).foldLeft(Monoid[Int].zero){(acc,i) => acc |+| i}
```

Foldable

Example:

```
List(Map("A" -> List(1,2,3)), Map("A" -> List(4,5,6)), Map("B" -> List(1))).foldMap(identity)

// Yields List(Map("A" -> List(1,2,3,4,5,6), Map("B" -> List(1)))
```

Foldable

Example:

```
val listOfMaps = List(Map("A" -> 1), Map("A" -> 2), Map("B" -> 3))

//Compose a nested Foldable
Foldable[List].compose[Map[String,?]].foldMap(listOfMaps)(_ + 100)
//Result
306
```

Traverse

Motivation:

```
List("colt", "bob", "%%%").map(validate(_))
// Yields List(Success("colt"), Success("bob"), Failure("Special characters aren't allowed"))

// This is a *pain* to work with!
```

Traverse

Definition:

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {
  /** Transform `fa` using `f`, collecting all the `G`s with `ap`. */
  def traverseImpl[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
}
```

Laws:

```
/**
 *
 * See Traverse.scala, it's verbose...
 *
 */
```

Traverse

Gives Us:

```
/** A version of `traverse` that infers the type constructor `G`. */
def traverseU[A, GB](fa: F[A])(f: A => GB)(implicit G: Unapply[Applicative, GB]): G.M[F[G.A]]

def sequenceU[A](self: F[A])(implicit G: Unapply[Applicative, A]): G.M[F[G.A]]

/**The composition of Traverses `F` and `G`, `[x]F[G[x]]`, is a Traverse */
def compose[G[_]](implicit G0: Traverse[G]): Traverse[({type λ[α] = F[G[α]]})#λ]

//And more!
```

Traverse

Traverse

Examples:

```
List(1,2,3).map(i => Option(i + 100)).sequenceU
// yields Some(List(101,102,103))

List(1,2,3).map(i => if(i == 1) None else Some(i)).sequenceU
// yields None
```

Traverse

Examples:

```
List(1,2,3).map(i => Option(i + 100)).sequenceU
// yields Some(List(101,102,103))

List(1,2,3).map(i => if(i == 1) None else Some(i)).sequenceU
// yields None
```

.map + sequenceU == traverseU

Traverse

Examples:

```
List(1,2,3).traverseU(i => Option(i + 100))
// yields Some(List(101,102,103))

List(1,2,3).traverseU(i => if(i == 1) None else Some(i))
// yields None
```

Traverse

Examples:

```
type MyValidation[A] = Validation[NonEmptyList[String], A]

def validateUser(u: String): MyValidaton[User] = ...

List("coltfred", "bob", "%%%%%%%", "$$$$$").traverseU(validateUser(_))

// yields Failure(NonEmptyList("'%%%%%%%' isn't a valid user","'$$$$$' isn't a valid user"))
```

Traverse

Examples:

```
type MyValidation[A] = Validation[NonEmptyList[String], A]

def validateUser(u: String): MyValidaton[User] = ...

List("coltfred", "bob").traverseU(validateUser(_))

// yields Success(List(User("coltfred"), User("bob")))
```

Thank you!

Questions?

Ask now or come to #scalaz IRC!