SCALAZ

There's an applicative in my functor!

Photo Credit: http://www.flickr.com/photos/yannconz/2796312310

Agenda

- Option
- Monoid
- Applicative
- Validation
- Lens

Options

- Option functions & typeclass instances
- Option syntax extensions
- Using Option with other parts of Scalaz

Constructing Options

```
Some(42)
None
```

```
Some[Int]
None
```

Constructing Options

```
Some(42)                    Some[Int]
None                        None
```

```
import scalaz.std.option._
some(42)                    Option[Int]
none[Int]                   Option[Int]
```

Constructing Options

```
Some(42)                    Some[Int]
None                        None
```

Imports functions & typeclass instances for option

```
import scalaz.std.option._
some(42)                    Option[Int]
none[Int]                   Option[Int]
```

Constructing Options

```
val os = List(Some(42), None, Some(20))

os.foldLeft(None) { (acc, o) => acc orElse o }
```

Constructing Options

```
val os = List(Some(42), None, Some(20))

os.foldLeft(None) { (acc, o) => acc orElse o }
error: type mismatch;
 found   : Option[Int]
 required: None.type
os.foldLeft(None) { (acc, o) => acc orElse o }
                                ^
```

Constructing Options

```
val os = List(Some(42), None, Some(20))

os.foldLeft(None) { (acc, o) => acc orElse o }
error: type mismatch;
 found   : Option[Int]
 required: None.type
  os.foldLeft(None) { (acc, o) => acc orElse o }
                                  ^

os.foldLeft(None: Option[Int]) { (acc, o) => acc orElse o }
```

Some(42)

Constructing Options

```
val os = List(Some(42), None, Some(20))

os.foldLeft(None) { (acc, o) => acc orElse o }
error: type mismatch;
 found   : Option[Int]
 required: None.type
   os.foldLeft(None) { (acc, o) => acc orElse o }
                                        ^

os.foldLeft(None: Option[Int]) { (acc, o) => acc orElse o }
```

Some(42)

```
os.foldLeft(none[Int]) { (acc, o) => acc orElse o }
```

Some(42)

Option conditional

```
import scalaz.syntax.std.option._

def xget(opt: Option[Int]) =
  opt | Int.MaxValue

xget(Some(42))
xget(None)
```

Option conditional

Imports syntax extensions for option

```
import scalaz.syntax.std.option._
```

```
def xget(opt: Option[Int]) =
  opt | Int.MaxValue

xget(Some(42))
xget(None)
```

Option conditional

Alias for getOrElse

```
import scalaz.syntax.std.option._
```

```
def xget(opt: Option[Int]) =
  opt | Int.MaxValue

xget(Some(42))
xget(None)
```

Option conditional

```
import scalaz.syntax.std.option._

def xget(opt: Option[Int]) =
  opt | Int.MaxValue

xget(Some(42))     42
xget(None)         2147483647
```

Option unary ~

```
import scalaz.std.anyVal._
import scalaz.std.option._
import scalaz.syntax.std.option._

def zget(opt: Option[Int]) = ~opt
zget(Some(42))
zget(None)
```

Option unary ~

Imports functions & typeclass instances for int, float, etc.

```
import scalaz.std.anyVal._
import scalaz.std.option._
import scalaz.syntax.std.option._

def zget(opt: Option[Int]) = ~opt
zget(Some(42))
zget(None)
```

Option unary ~

Alias for getOrElse(zero)

```
import scalaz.std.anyVal._
import scalaz.std.option._
import scalaz.syntax.std.option._

def zget(opt: Option[Int]) = ~opt
zget(Some(42))
zget(None)
```

Option unary ~

Alias for getOrElse(zero)

```
import scalaz.std.anyVal._
import scalaz.std.option._
import scalaz.syntax.std.option._

def zget(opt: Option[Int]) = ~opt
zget(Some(42))    42
zget(None)         0
```

Option unary ~

Alias for getOrElse(zero)

```
import scalaz.std.anyVal._
import scalaz.std.option._
import scalaz.syntax.std.option._

def zget(opt: Option[Int]) = ~opt
zget(Some(42))    42
zget(None)         0
```

Where does zero come from?

Option unary ~

```
import scalaz.Monoid
import scalaz.std.string._
import scalaz.std.list._
```

```
def mzget[A : Monoid](opt: Option[A]) = ~opt
mzget(none[Int])          0
mzget(none[String])       ""
mzget(none[List[Int]])    List()
```

Aside: Context Bounds

```
def mzget[A : Monoid](opt: Option[A]) = ~opt
```

Aside: Context Bounds

```
def mzget[A : Monoid](opt: Option[A]) = ~opt
```

- Monoid is a *context bound* for type A

Aside: Context Bounds

```
def mzget[A : Monoid](opt: Option[A]) = ~opt
```

- Monoid is a *context bound* for type A
- Means there must be an implicit value of type Monoid[A] in implicit scope

Aside: Context Bounds

```
def mzget[A : Monoid](opt: Option[A]) = ~opt
```

- Monoid is a *context bound* for type A
- Means there must be an implicit value of type Monoid[A] in implicit scope
- Equivalent to:

```
def mzget[A](opt: Option[A])(
  implicit m: Monoid[A]) = ~opt
```

Aside: Context Bounds

```
def mzget[A : Monoid](opt: Option[A]) = ~opt
```

- Monoid is a *context bound* for type A
- Means there must be an implicit value of type Monoid[A] in implicit scope
- Equivalent to:

```
def mzget[A](opt: Option[A])(
  implicit m: Monoid[A]) = ~opt
```

- Even supports multiple bounds:

```
def other[A : Monoid : Functor] = ...
```

Option err

```
def knowBetter[A](opt: Option[A]) =
  opt err "You promised!"

knowBetter(some(42))
knowBetter(none)
```

Option err

Alias for getOrElse(sys.err(msg))

```
def knowBetter[A](opt: Option[A]) =
  opt err "You promised!"

knowBetter(some(42))   42
knowBetter(none)
```

throws RuntimException("You promised!")

Good alternative to opt.get when you know it's defined!

Option fold/cata

```
import java.net.InetAddress
def ipAddress(opt: Option[InetAddress]) =
  opt.fold(_.getHostAddress, "0.0.0.0")

ipAddress(Some(InetAddress.getLocalHost))

ipAddress(None)
```

Option fold/cata

Replacement for matching on Some/None

```
import java.net.InetAddress
def ipAddress(opt: Option[InetAddress]) =
  opt.fold(_.getHostAddress, "0.0.0.0")

ipAddress(Some(InetAddress.getLocalHost))
                                    10.0.1.12
ipAddress(None)
                                    0.0.0.0
```

Option ifNone

```
var cnt = 0
some(42) ifNone { cnt += 1 }
none ifNone { cnt += 1 }
println(cnt)
```

Concatenate

Given a list of options, how do you sum them?

```
List(some(42), none, some(51)).sum
error: could not find implicit value for
parameter num: Numeric[Option[Int]]
```

Concatenate

Given a list of options, how do you sum them?

```
List(some(42), none, some(51)).sum
error: could not find implicit value for
parameter num: Numeric[Option[Int]]
```

```
import scalaz.syntax.traverse._
List(some(42), none, some(51)).concatenate
```

93

Sequence

Given a list of options, how do you get option of list?

```
List(some(42), none, some(51)).sequence
```

None

```
List(some(42), some(51)).sequence
```

Some(List(42, 51))

Semigroup & Monoid Review

Not Scala syntax!

- Semigroup

  `append: A ⇒ A ⇒ A`

- Monoid extends Semigroup

  `zero: A`

Syntax

```
import scalaz.std.anyVal._
import scalaz.syntax.monoid._
```

```
1 |+| 2
```

3

```
import scalaz.std.list._
```

```
List(1, 2) |+| List(3, 4)
```

List(1, 2, 3, 4)

Syntax

```
import scalaz.std.map._
import scalaz.std.set._

val low = Map(
  'odd -> Set(1, 3),
  'even -> Set(2, 4))
val hi = Map(
  'odd -> Set(5, 7),
  'even -> Set(6, 8))
low |+| hi
```

Syntax

```
import scalaz.std.map._
import scalaz.std.set._

val low = Map(
  'odd -> Set(1, 3),
  'even -> Set(2, 4))
val hi = Map(
  'odd -> Set(5, 7),
  'even -> Set(6, 8))
low |+| hi
```

Map('odd -> Set(1, 3, 5, 7),
'even -> Set(2, 4, 6, 8))

Option is Monoid

- There's a monoid of Option[A] if A is a Semigroup
  - zero = None
  - append = ???

Option is Monoid

```scala
implicit def optionMonoid[A: Semigroup]: Monoid[Option[A]] =
  new Monoid[Option[A]] {
    def append(f1: Option[A], f2: => Option[A]) = (f1, f2) match {
      case (Some(a1), Some(a2)) =>
        Some(Semigroup[A].append(a1, a2))
      case (Some(a1), None)      => f1
      case (None, Some(a2))      => f2
      case (None, None)          => None
    }

    def zero: Option[A] = None
  }
```

From Scalaz Seven source

https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/std/Option.scala

Option is Monoid

```
import scalaz.syntax.monoid._

some(20) |+| none |+| some(22)
```

Some(42)

Option alternative monoids

- Can we find other monoids for Option?
- Must implement zero and append
- But remember the monoid laws:
  - Closure
  - Associativity
  - Identity

See: http://en.wikipedia.org/wiki/Monoid

Option alternative monoids

- Keep everything the same as semigroup based monoid but focus on:

```
case (Some(a1), Some(a2)) =>
```

- Left

```
case (Some(a1), Some(a2)) => f1
```

- Right

```
case (Some(a1), Some(a2)) => f2
```

Option alternative monoids

```
some(20).first |+| none.first |+|
some(22).first
```

Some(20)

```
some(20).last |+| none.last |+|
some(22).last
```

Some(22)

Functor & Monad Review

Not Scala syntax!

- Functor

  `map: F[A] ⇒ (A ⇒ B) ⇒ F[B]`

- Monad extends Functor

  `point: A ⇒ M[A]`

  `flatMap: M[A] ⇒ (A ⇒ M[B]) ⇒ M[B]`

Functor & Monad Review

Not Scala syntax!

- Functor

  `map: F[A] ⇒ (A ⇒ B) ⇒ F[B]`

  Option[Int] ⇒ (Int ⇒ String) ⇒ Option[String]

- Monad extends Functor

  `point: A ⇒ M[A]`

  `flatMap: M[A] ⇒ (A ⇒ M[B]) ⇒ M[B]`

Functor & Monad Review

Not Scala syntax!

- Functor

  `map: F[A] ⇒ (A ⇒ B) ⇒ F[B]`

  Option[Int] ⇒ (Int ⇒ String) ⇒ Option[String]

- Monad extends Functor

  `point: A ⇒ M[A]`

  `flatMap: M[A] ⇒ (A ⇒ M[B]) ⇒ M[B]`

Functor & Monad Review

Not Scala syntax!

- Functor

  `map: F[A] ⇒ (A ⇒ B) ⇒ F[B]`

  Option[Int] ⇒ (Int ⇒ String) ⇒ Option[String]

- Monad extends Functor

  `point: A ⇒ M[A]`

  `flatMap: M[A] ⇒ (A ⇒ M[B]) ⇒ M[B]`

  String ⇒ Option[String]

  Option[String] ⇒ (String ⇒ Option[Int]) ⇒ Option[Int]

What's this?

- ??? extends Functor

```
point: A ⇒ F[A]

???: F[A] ⇒ F[A ⇒ B] ⇒ F[B]
```

What's this?

- ??? extends Functor

  ```
  point: A ⇒ F[A]

  ???: F[A] ⇒ F[A ⇒ B] ⇒ F[B]
  ```

- Read as...

  Given a function from A to B in a context F and a value of A in a context F, produce a value of B in context F.

What's this?

- ??? extends Functor

  ```
  point: A ⇒ F[A]

  ???: F[A] ⇒ F[A ⇒ B] ⇒ F[B]
  ```

- Read as...
  Given a function from A to B in a context F and a value of A in a context F, produce a value of B in context F.

- Represents *function application in a context*

What's this?

- ??? extends Functor

  ```
  point: A ⇒ F[A]

  ???: F[A] ⇒ F[A ⇒ B] ⇒ F[B]
  ```

- Read as...
  Given a function from A to B in a context F and a value of A in a context F, produce a value of B in context F.
- Represents *function application in a context*
- Defined in http://www.soi.city.ac.uk/~ross/papers/Applicative.html

Applicative

- Applicative extends Functor

```
point: A ⇒ F[A]

ap: F[A] ⇒ F[A ⇒ B] ⇒ F[B]
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Pointed.scala
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Applicative.scala

Applicative

- Applicative extends Functor

```
point: A ⇒ F[A]

ap: F[A] ⇒ F[A ⇒ B] ⇒ F[B]
```

- Or in proper Scala:

```
trait Pointed[F[_]] extends Functor[F] {
  def point[A](a: => A): F[A]
}

trait Applicative[F[_]] extends Functor[F] with Pointed[F] {
  def ap[A,B](fa: => F[A])(f: => F[A => B]): F[B]
}
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Pointed.scala
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Applicative.scala

Option Is Applicative

Let's define an applicative instance for Option

```
implicit val optionInstance = new Applicative[Option] {
  override def point[A](a: => A) = ???
  override def ap[A, B](
    fa: => Option[A])(f: => Option[A => B]) = ???
}
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Option.scala

Option Is Applicative

How can we lift a value of A to Option[A]?

```
implicit val optionInstance = new Applicative[Option] {
  override def point[A](a: => A) = ???
  override def ap[A, B](
    fa: => Option[A])(f: => Option[A => B]) = ???
}
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Option.scala

Option Is Applicative

```
implicit val optionInstance = new Applicative[Option] {
  override def point[A](a: => A) = Some(a)
  override def ap[A, B](
    fa: => Option[A])(f: => Option[A => B]) = ???
}
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Option.scala

Option Is Applicative

How can we apply the function in Option[A=>B] to Option[A]?

```
implicit val optionInstance = new Applicative[Option] {
  override def point[A](a: => A) = Some(a)
  override def ap[A, B](
    fa: => Option[A])(f: => Option[A => B]) = ???
}
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Option.scala

Option Is Applicative

```
implicit val optionInstance = new Applicative[Option] {
  override def point[A](a: => A) = Some(a)
  override def ap[A, B](
    fa: => Option[A])(f: => Option[A => B]) = f match {
      case Some(f) => fa match {
        case Some(x) => Some(f(x))
        case None    => None
      }
      case None    => None
  }
}
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Option.scala

Option Is Applicative

```
implicit val optionInstance = new Applicative[Option] {
  override def point[A](a: => A) = Some(a)
  override def ap[A, B](
    fa: => Option[A])(f: => Option[A => B]) = f match {
      case Some(f) => fa match {
        case Some(x) => Some(f(x))
        case None    => None
      }
      case None    => None
  }
}
```

What about map[A](f: A => B): Option[B]?

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Option.scala

Functor.map on Applicative

Can we define map in terms of pointed and ap?

```
trait Applicative[F[_]] extends Functor[F] with Pointed[F] {
  def ap[A,B](fa: => F[A])(f: => F[A => B]): F[B]
  override def map[A, B](fa: F[A])(f: A => B): F[B] =
    ???
}
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Option.scala

Functor.map on Applicative

```
trait Applicative[F[_]] extends Functor[F] with Pointed[F] {
  def ap[A,B](fa: => F[A])(f: => F[A => B]): F[B]
  override def map[A, B](fa: F[A])(f: A => B): F[B] =
    ap(fa)(point(f))
}
```

Summarized from Scalaz Seven source
https://github.com/scalaz/scalaz/blob/scalaz-seven/core/src/main/scala/scalaz/Option.scala

Applicative Usage

```
import scalaz.std.option._
import scalaz.syntax.applicative._

val oa5: Option[Int => Int] = Some((_: Int) + 5)
some(15) <*> oa5
```

Applicative Usage

```
import scalaz.std.option._
import scalaz.syntax.applicative._

val oa5: Option[Int => Int] = Some((_: Int) + 5)
some(15) <*> oa5
```

Alias for ap

Applicative Usage

```
import scalaz.std.option._
import scalaz.syntax.applicative._

val oa5: Option[Int => Int] = Some((_: Int) + 5)
some(15) <*> oa5
```

Some(20)

Alias for ap

Applicative Usage

```
import scalaz.std.option._
import scalaz.syntax.applicative._

some(15).<**>(some(5)) { _ + _ }

some(15).<***>(some(5), some(6)) { _ + _ + _ }

some(15).<****>(some(5), some(6), some(7)) {
  _ + _ + _ + _ }
```

Applicative Usage

```
import scalaz.std.option._
import scalaz.syntax.applicative._

some(15).<**>(some(5)) { _ + _ }
```

Some(20)

```
some(15).<***>(some(5), some(6)) { _ + _ + _ }
```

Some(26)

```
some(15).<****>(some(5), some(6), some(7)) {
  _ + _ + _ + _ }
```

Some(33)

Applicative Builder

Most common and convenient usage

```
import scalaz.std.option._
import scalaz.syntax.applicative._

(some(15) |@| some(5)) apply { _ + _ }

(some(15) |@| some(5) |@| some(6)) apply { _ + _ + _ }

(some(15) |@| some(5)).tupled
```

Applicative Builder

Most common and convenient usage

```
import scalaz.std.option._
import scalaz.syntax.applicative._

(some(15) |@| some(5)) apply { _ + _ }
```

Some(20)

```
(some(15) |@| some(5) |@| some(6)) apply { _ + _ + _ }
```

Some(26)

```
(some(15) |@| some(5)).tupled
```

Some((15, 5))

Applicative Builder

Most common and convenient usage

```
import scalaz.std.option._
import scalaz.syntax.applicative._

(some(15) |@| some(5)) apply { _ + _ }
```

Some(20)

```
(some(15) |@| some(5) |@| some(6)) apply { _ + _ + _ }
```

Some(26)

```
(some(15) |@| some(5)).tupled
```

Some((15, 5))

Supports up to 12 arguments

Applicative Builder

```
import scalaz.std.option._
import scalaz.syntax.applicative._

case class Album(name: String, artist: String)

(some("Wilco") ⊛ some("Sky Blue Sky")) apply Album.apply
```

Applicative Builder

```
import scalaz.std.option._
import scalaz.syntax.applicative._

case class Album(name: String, artist: String)

(some("Wilco") ⊛ some("Sky Blue Sky")) apply Album.apply
```

Companion has an apply method automatically created

Applicative Builder

```
import scalaz.std.option._
import scalaz.syntax.applicative._

case class Album(name: String, artist: String)

(some("Wilco") ⊛ some("Sky Blue Sky")) apply Album.apply
```

Companion has an apply method automatically created

Alias for |@|

Applicative Builder

```
import scalaz.std.option._
import scalaz.syntax.applicative._

case class Album(name: String, artist: String)

(some("Wilco") ⊛ some("Sky Blue Sky")) apply Album.apply
```

Companion has an apply method automatically created

Alias for |@|

Some(Album(Wilco, Sky Blue Sky))

Why Applicatives?

- Less restrictive than Monads, and thus, more general (powerful)
- Composable
  http://blog.tmorris.net/monads-do-not-compose/
- Support parallel computation
  More on this point later...

A word on typeclasses

- So far we've seen Semigroup, Monoid, Functor, Pointed, Applicative, Monad

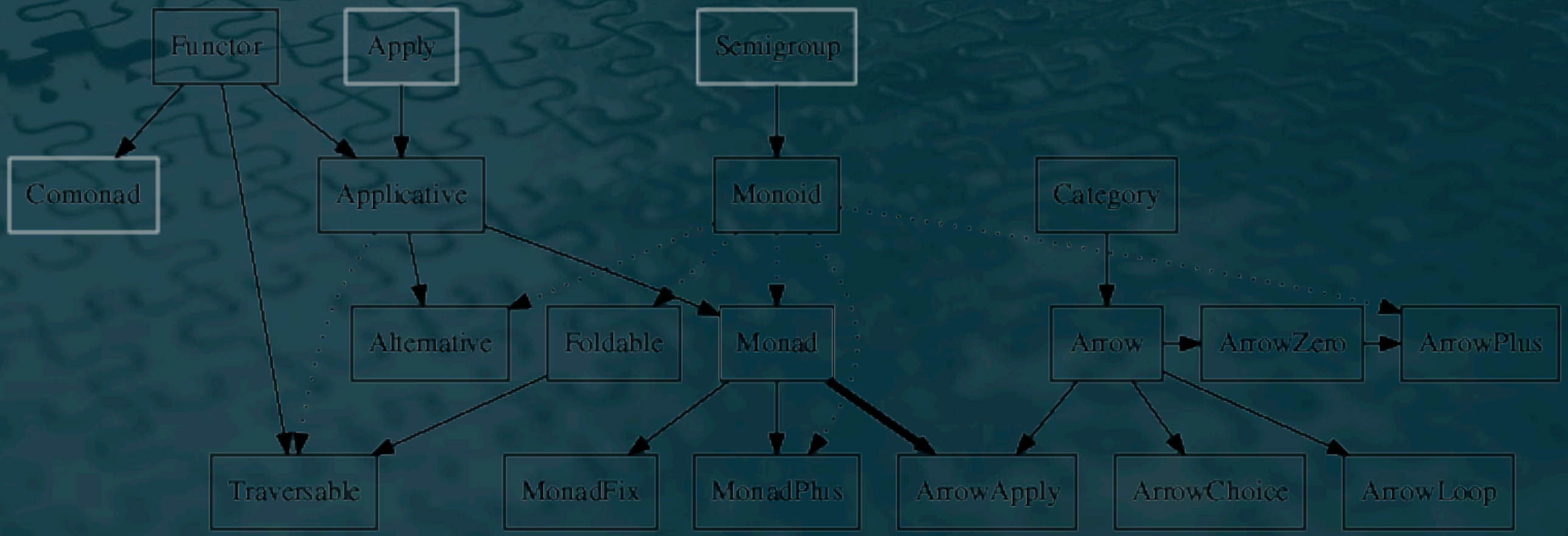
A word on typeclasses

- So far we've seen Semigroup, Monoid, Functor, Pointed, Applicative, Monad
- There are many others -- ApplicativePlus, MonadPlus, Comonad, Category, Arrow, ArrowPlus, Foldable, Traversable, Monad Transformers, Reader, Writer, State, Identity, and more

A word on typeclasses

- So far we've seen Semigroup, Monoid, Functor, Pointed, Applicative, Monad
- There are many others -- ApplicativePlus, MonadPlus, Comonad, Category, Arrow, ArrowPlus, Foldable, Traversable, Monad Transformers, Reader, Writer, State, Identity, and more
- These are functional design patterns
  - Simpler than most OO design patterns

Some Haskell Typeclasses



Credit: Typeclassopedia
http://www.haskell.org/haskellwiki/Typeclassopedia#Introduction

Typeclass References

- Haskell Typeclassopedia is a great reference for learning these
  http://www.haskell.org/haskellwiki/Typeclassopedia
- Scala Typeclassopedia describes the big 3 (Functor, Applicative, Monad)
  http://typeclassopedia.bitbucket.org/

Validation

```
sealed trait Validation[+E, +A] {
  ...
}

final case class Success[E, A](a: A)
  extends Validation[E, A]

final case class Failure[E, A](e: E)
  extends Validation[E, A]
```

Validation

- Represents either success or failure
- On success, provides the successful value
- On failure, provides the failure value
- Sound familiar?

Isomorphic to Either

$$\mathbb{1} + \mathbb{1} \cong$$

Isomorphic to Either

`Validation`		`Either`
`Success`	≅	`Right`
`Failure`		`Left`

Isomorphic to Either

`Validation`		`Either`
`Success`	≅	`Right`
`Failure`		`Left`

Q: So why not just use Either?

Isomorphic to Either

Validation		Either
Success	≅	Right
Failure		Left

Q: So why not just use Either?

A: When Validation is constructed with an error type that has a Semigroup, there exists an Applicative Functor for Validation that *accumulates errors*

Constructing Validations

```
import scalaz.{Success, Failure}
Success(42)
Failure("boom")
```

Success[Nothing, Int]

Failure[String, Nothing]

Constructing Validations

```
import scalaz.{Success, Failure}
Success(42)                          Success[Nothing, Int]
Failure("boom")                      Failure[String, Nothing]

import scalaz.Validation
Validation.success(42)               Validation[Nothing, Int]
Validation.failure("boom")           Validation[String, Nothing]
```

Constructing Validations

```
import scalaz.{Success, Failure}
Success(42)                          Success[Nothing, Int]
Failure("boom")                      Failure[String, Nothing]

import scalaz.Validation
Validation.success(42)               Validation[Nothing, Int]
Validation.failure("boom")           Validation[String, Nothing]

import scalaz.syntax.validation._
42.success                           Validation[Nothing, Int]
"boom".fail                          Validation[String, Nothing]
```

Constructing Validations

```
import scalaz.{Success, Failure}
Success(42)                          Success[Nothing, Int]
Failure("boom")                      Failure[String, Nothing]

import scalaz.Validation
Validation.success(42)               Validation[Nothing, Int]
Validation.failure("boom")           Validation[String, Nothing]

import scalaz.syntax.validation._
42.success                           Validation[Nothing, Int]
"boom".fail                          Validation[String, Nothing]

42.success[String]                   Validation[String, Int]
"boom".fail[Int]                     Validation[String, Int]
```

Option to Validation

```
import scalaz.std.option._
import scalaz.syntax.std.option._

some(42).toSuccess("boom")
none[Int].toSuccess("boom")

some(42).toFailure("boom")
none[Int].toFailure("boom")
```

Option to Validation

```
import scalaz.std.option._
import scalaz.syntax.std.option._

some(42).toSuccess("boom")
none[Int].toSuccess("boom")

some(42).toFailure("boom")
none[Int].toFailure("boom")
```

Success(42)

Failure("boom")

Option to Validation

```
import scalaz.std.option._
import scalaz.syntax.std.option._

some(42).toSuccess("boom")          Success(42)
none[Int].toSuccess("boom")         Failure("boom")

some(42).toFailure("boom")          Failure(42)
none[Int].toFailure("boom")         Success("boom")
```

Either to Validation

```
import scalaz.syntax.id._
import scalaz.Validation.fromEither

fromEither(42.right)
fromEither("boom".left)
```

Either to Validation

```
import scalaz.syntax.id._
import scalaz.Validation.fromEither

fromEither(42.right)
fromEither("boom".left)
```

Success(42)

Failure("boom")

Throwing to Validation

```
import scalaz.Validation.fromTryCatch

fromTryCatch("42".toInt)
fromTryCatch("notAnInt".toInt)
```

Throwing to Validation

```
import scalaz.Validation.fromTryCatch

fromTryCatch("42".toInt)
fromTryCatch("notAnInt".toInt)
```

```
Success(42)
Failure(java.lang.NumberFormatException: For input string: "notAnInt")
```

Throwing to Validation

```
import scalaz.Validation.fromTryCatch

fromTryCatch("42".toInt)
fromTryCatch("notAnInt".toInt)
```

Success(42)
Failure(java.lang.NumberFormatException: For input string: "notAnInt")

```
fromTryCatch("notAnInt".toInt)
  .fail.map { _.getMessage }.validation
```

Throwing to Validation

```
import scalaz.Validation.fromTryCatch

fromTryCatch("42".toInt)
fromTryCatch("notAnInt".toInt)
```

Success(42)
Failure(java.lang.NumberFormatException: For input string: "notAnInt")

```
fromTryCatch("notAnInt".toInt)
  .fail.map { _.getMessage }.validation
```

Failure(For input string: "notAnInt")

Mapping via Bifunctor

```
import scalaz.syntax.bifunctor._
```

Bifunctor = functor of 2 arguments

```
fromTryCatch("notAnInt".toInt).
  <-: { _.getMessage }
```

<-: = Map left value

Mapping via Bifunctor

```
import scalaz.syntax.bifunctor._
```

Bifunctor = functor of 2 arguments

```
fromTryCatch("notAnInt".toInt).
  <-: { _.getMessage }
```

<-: = Map left value

Failure(For input string: "notAnInt")

Mapping via Bifunctor

```
import scalaz.syntax.bifunctor._
```

Bifunctor = functor of 2 arguments

```
fromTryCatch("notAnInt".toInt).
  <-: { _.getMessage }
```

<-: = Map left value

Failure(For input string: "notAnInt")

```
fromTryCatch("notAnInt".toInt).
  bimap(_.getMessage, identity)
```

Mapping via Bifunctor

```
import scalaz.syntax.bifunctor._
```

Bifunctor = functor of 2 arguments

```
fromTryCatch("notAnInt".toInt).
  <-: { _.getMessage }
```

<-: = Map left value

Failure(For input string: "notAnInt")

```
fromTryCatch("notAnInt".toInt).
  bimap(_.getMessage, identity)
```

Failure(For input string: "notAnInt")

Validation is Monad

```
def justMessage[S](
  v: Validation[Throwable, S]): Validation[String, S] =
  v.<-: { _.getMessage }

def extractId(
  metadata: Map[String, String]): Validation[String, UUID] =
  for {
    str <- metadata.get("id").toSuccess("No id property")
    id <- justMessage(fromTryCatch(UUID.fromString(str)))
  } yield id

extractId(Map.empty)
extractId(Map("id" -> "notUuid"))
extractId(Map("id" -> UUID.randomUUID.toString))
```

Validation is Monad

```
def justMessage[S](
  v: Validation[Throwable, S]): Validation[String, S] =
  v.<-: { _.getMessage }

def extractId(
  metadata: Map[String, String]): Validation[String, UUID] =
  for {
    str <- metadata.get("id").toSuccess("No id property")
    id <- justMessage(fromTryCatch(UUID.fromString(str)))
  } yield id

extractId(Map.empty)
extractId(Map("id" -> "notUuid"))
extractId(Map("id" -> UUID.randomUUID.toString))
```

Failure(No id property)

Failure(Invalid UUID string: notUuid)

Success(f6af15db-feeb-4895-8c67-a70a6f947d69)

Aside: Monadic DSL

```
def justMessage[S](
  v: Validation[Throwable, S]): Validation[String, S] =
  v.<-: { _.getMessage }

def extractId(
  metadata: Map[String, String]): Validation[String, UUID] =
  for {
    str <- metadata.get("id").toSuccess("No id property")
    id <- justMessage(fromTryCatch(UUID.fromString(str)))
  } yield id

extractId(Map.empty)
extractId(Map("id" -> "notUuid"))
extractId(Map("id" -> UUID.randomUUID.toString))
```

This reads pretty naturally!

Aside: Monadic DSL

```
def justMessage[S](
  v: Validation[Throwable, S]): Validation[String, S] =
  v.<-: { _.getMessage }

def extractId(
  metadata: Map[String, String]): Validation[String, UUID] =
  for {
    str <- metadata.get("id").toSuccess("No id property")
    id <- justMessage(fromTryCatch(UUID.fromString(str)))
  } yield id

extractId(Map.empty)
extractId(Map("id" -> "notUuid"))
extractId(Map("id" -> UUID.randomUUID.toString))
```

This doesn't! Notation is inside-out!

Aside: Monadic DSL

```
import scalaz.syntax.id._

def justMessage[S](
  v: Validation[Throwable, S]): Validation[String, S] =
  v.<-: { _.getMessage }

def parseUUID(s: String): Validation[Throwable, UUID] =
  fromTryCatch(UUID.fromString(s))
```

```
def extractId(
  metadata: Map[String, String]): Validation[String, UUID] =
  for {
    str <- metadata.get("id").toSuccess("No id property")
    id <- str |> parseUUID |> justMessage
  } yield id
```

Pipe-forward operator reverses order of function application (from F#)

For more info on pipe-forward, see: http://stackoverflow.com/questions/1457140/haskell-composition-vs-fs-pipe-forward-operator

Validation

- Monadic usage of Validation is incredibly useful
  - As useful as monadic option but provides *why* failures have occurred
- Can it get better?

Validation is Applicative...

... if the error type is Semigroup

Validation is Applicative...

... if the error type is Semigroup

Which of these are applicative?

```
Validation[Int, Any]

Validation[Any, Int]

Validation[List[String], String]

Validation[String, Int]
```

Validation is Applicative...

... if the error type is Semigroup

Which of these are applicative?

✔ `Validation[Int, Any]`

`Validation[Any, Int]`

`Validation[List[String], String]`

`Validation[String, Int]`

Validation is Applicative...

... if the error type is Semigroup

Which of these are applicative?

✔ `Validation[Int, Any]`

✘ `Validation[Any, Int]`

`Validation[List[String], String]`

`Validation[String, Int]`

Validation is Applicative...

... if the error type is Semigroup

Which of these are applicative?

✔ `Validation[Int, Any]`

✘ `Validation[Any, Int]`

✔ `Validation[List[String], String]`

`Validation[String, Int]`

Validation is Applicative...

... if the error type is Semigroup

Which of these are applicative?

✔ `Validation[Int, Any]`

✘ `Validation[Any, Int]`

✔ `Validation[List[String], String]`

✔ `Validation[String, Int]`

Validation[List[_], _]

```
Validation[List[String], Int]
```

Validation[List[_], _]

```
Validation[List[String], Int]

  Success(42)
```

Validation[List[_], _]

```
Validation[List[String], Int]

  Success(42)

  Failure(List(“Tigers”, “Lions”, “Bears”))
```

Validation[List[_], _]

```
Validation[List[String], Int]

  Success(42)

  Failure(List(“Tigers”, “Lions”, “Bears”))

  Failure(List())
```

Validation[List[_], _]

```
Validation[List[String], Int]

  Success(42)

  Failure(List(“Tigers”, “Lions”, “Bears”))

  Failure(List())
```

Failure of no errors?

Can we do better?

Use the type system?

ValidationNEL

```
Validation[NonEmptyList[String], Int]

   Success(42)

   Failure(NonEmptyList(
      “Tigers”, “Lions”, “Bears”))
```

ValidationNEL

```
Validation[NonEmptyList[String], Int]
    Success(42)

    Failure(NonEmptyList(
        “Tigers”, “Lions”, “Bears”))
```

This is so common, there's an alias for it:

```
ValidationNEL[String, Int]
```

ValidationNEL

```
Validation[NonEmptyList[String], Int]
```

```
  Success(42)
```

```
  Failure(NonEmptyList(
    “Tigers”, “Lions”, “Bears”))
```

This is so common, there's an alias for it:

```
ValidationNEL[String, Int]
```

And any Validation[E, S] can be converted to ValidationNEL[E, S]:

```
v.toValidationNEL
```

Putting together the pieces

- Given a map of string to string containing keys *name*, *level*, *manager*, representing the name of an employee, their numeric level, and *true* if they are an manager and *false* otherwise, create an Employee containing those values if:
  - Name is non-empty
  - Level is 1 to 15
- Otherwise, report all errors in the map

Putting together the pieces

```
case class Employee(
  name: String,
  level: Int,
  manager: Boolean)

type Metadata = Map[String, String]
```

Putting together the pieces

```
def justMessage[S](
  v: Validation[Throwable, S]): Validation[String, S] =
  v.<-: { _.getMessage }
```

```
def extractString(
  metadata: Metadata,
  key: String): Validation[String, String] =
  metadata.
    get(key).
    toSuccess("Missing required property %s".format(key))
```

Putting together the pieces

```
def extractName(
metadata: Metadata): Validation[String, String] =
extractString(metadata, "name") flatMap { name =>
  if (name.isEmpty) "Must have a non-empty name!".fail
  else name.success
}
```

Putting together the pieces

```
def extractLevel(
  metadata: Metadata): Validation[String, Int] =
  extractString(metadata, "level").
    flatMap { _.parseInt |> justMessage }.
    flatMap { level =>
      if (level < 1) "Level must be at least 1".fail
      else if (level > 15) "Really?".fail
      else level.success
    }
```

Putting together the pieces

```
def extractManager(
  metadata: Metadata): Validation[String, Boolean] =
  extractString(metadata, "manager").
    flatMap { _.parseBoolean |> justMessage }
```

Putting together the pieces

```
def extractEmployee(
  metadata: Metadata): ValidationNEL[String, Employee] = {
  extractName(metadata).toValidationNel |@|
  extractLevel(metadata).toValidationNel |@|
  extractManager(metadata).toValidationNel
} apply Employee.apply
```

Putting together the pieces

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "15",
  "manager" -> "false"))
```

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "0",
  "manager" -> "true"))
```

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "17",
  "manager" -> "true"))
```

Putting together the pieces

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "15",
  "manager" -> "false"))
```

Success(Employee(Turing,15,false))

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "0",
  "manager" -> "true"))
```

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "17",
  "manager" -> "true"))
```

Putting together the pieces

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "15",
  "manager" -> "false"))
```

Success(Employee(Turing,15,false))

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "0",
  "manager" -> "true"))
```

Failure(NonEmptyList(Level must be at least 1))

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "17",
  "manager" -> "true"))
```

Putting together the pieces

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "15",
  "manager" -> "false"))
```

Success(Employee(Turing,15,false))

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "0",
  "manager" -> "true"))
```

Failure(NonEmptyList(Level must be at least 1))

```
extractEmployee(Map(
  "name" -> "Turing",
  "level" -> "17",
  "manager" -> "true"))
```

Failure(NonEmptyList(Really?))

Putting together the pieces

```
extractEmployee(Map(
  "name" -> "Turing"))
```

```
extractEmployee(Map(
  "name" -> "",
  "level" -> "17",
  "manager" -> "notBool"))
```

Putting together the pieces

```
extractEmployee(Map(
  "name" -> "Turing"))
```

Failure(NonEmptyList(
Missing required property level,
Missing required property manager))

```
extractEmployee(Map(
  "name" -> "",
  "level" -> "17",
  "manager" -> "notBool"))
```

Putting together the pieces

```
extractEmployee(Map(
  "name" -> "Turing"))
```

```
Failure(NonEmptyList(
  Missing required property level,
  Missing required property manager))
```

```
extractEmployee(Map(
  "name" -> "",
  "level" -> "17",
  "manager" -> "notBool"))
```

```
Failure(NonEmptyList(
  Must have a non-empty name!,
  Really?,
  For input string: "notBool"))
```

Putting together the pieces

- Building upon previous example, given a list of metadata, produce a list of employees or a list of errors

Putting together the pieces

```
def extractEmployees(
  metadata: List[Metadata]
): ValidationNEL[String, List[Employee]] = ???
```

Putting together the pieces

```
def extractEmployees(
  metadata: List[Metadata]
): ValidationNEL[String, List[Employee]] = {


  val vEmps: List[ValidationNEL[String, Employee]] =
    metadata map extractEmployee

  ???
}
```

Putting together the pieces

```
def extractEmployees(
  metadata: List[Metadata]
): ValidationNEL[String, List[Employee]] = {

  val vEmps: List[ValidationNEL[String, Employee]] =
    metadata map extractEmployee

  ???
}
```

How can we swap order of ValidationNEL and List?

Putting together the pieces

```
def extractEmployees(
  metadata: List[Metadata]
): ValidationNEL[String, List[Employee]] = {

  val vEmps: List[ValidationNEL[String, Employee]] =
    metadata map extractEmployee

  vEmps.sequence[
    ({type λ[a] = ValidationNEL[String, a]})#λ, Employee]
}
```

Type annotation necessary due to limitations in Scala type inferencing

Type Lambdas

```
sequence[({type λ[a] = ValidationNEL[String, a]})#λ, Employee]
```

Type Lambdas

```
sequence[({type λ[a] = ValidationNEL[String, a]})#λ, Employee]
```

Defines an anonymous structural type with one member, λ[a]

Type Lambdas

```
sequence[({type λ[a] = ValidationNEL[String, a]})#λ, Employee]
```

Defines type λ[a] as an alias for ValidationNEL[String, a]

Type Lambdas

```
sequence[({type λ[a] = ValidationNEL[String, a]})#λ, Employee]
```

References the λ member of the structural type

Type Lambdas

```
sequence[λ[a] = ValidationNEL[String, a], Employee]
```

Not valid Scala syntax, but IntelliJ will render type lambdas this way

Type Lambdas

```
sequence[ValidationNEL[String, _], Employee]
```

Can be thought of this way (sort of)

Lens: Problem Statement

```
case class Contact(
  name: Name,
  phone: Phone)

case class Name(
  salutation: String,
  first: String,
  last: String)

case class Phone(
  digits: String)

val seth = Contact(
  Name("Mr.", "Seth", "Avett"), Phone("555-5555"))
```

Lens: Problem Statement

```
case class Contact(
  name: Name,
  phone: Phone)

case class Name(
  salutation: String,
  first: String,
  last: String)

case class Phone(
  digits: String)

val seth = Contact(
  Name("Mr.", "Seth", "Avett"), Phone("555-5555"))
```

We need a copy of seth with first name set to "Scott"

Lens: Problem Statement

```
val seth = Contact(
  Name("Mr.", "Seth", "Avett"),
  Phone("555-5555"))
```

```
val scott = seth.copy(
  name = seth.name.copy(first = "Scott"))
```

Lens: Problem Statement

```
val seth = Contact(
  Name("Mr.", "Seth", "Avett"),
  Phone("555-5555"))
```

```
val scott = seth.copy(
  name = seth.name.copy(first = "Scott"))
```

Doesn't scale - each layer in record structure results in another layer of copies and duplication

Lens

- Given a record type and a field, a *lens* provides the ability to focus on the specified field, which allows accessing the field value and setting the field value (immutably)

Lens

- Given a record type and a field, a *lens* provides the ability to focus on the specified field, which allows accessing the field value and setting the field value (immutably)
- Translation: Lens = immutable getter/setter

Lens

```
case class Contact(
  name: Name,
  phone: Phone)
```

Record Type Field Type

```
val contactNameLens = Lens.lensu[Contact, Name](
  (c, n) => c.copy(name = n), _.name)
```

Setter Getter

```
val contactPhoneLens = Lens.lensu[Contact, Phone](
  (c, p) => c.copy(phone = p), _.phone)
```

Setter Getter

Lens

- Case class lenses usually use copy in setter

```
case class Contact(
  name: Name,
  phone: Phone)
```

```
val contactNameLens = Lens.lensu[Contact, Name](
  (c, n) => c.copy(name = n), _.name)
```

Setter — Getter

```
val contactPhoneLens = Lens.lensu[Contact, Phone](
  (c, p) => c.copy(phone = p), _.phone)
```

Setter — Getter

Lens

- Case class lenses usually use copy in setter
- Case class lenses are purely mechanical to author (there's even a compiler plugin that does it!)

```
case class Contact(
  name: Name,
  phone: Phone)
```

```
val contactNameLens = Lens.lensu[Contact, Name](
  (c, n) => c.copy(name = n), _.name)
```

Setter Getter

```
val contactPhoneLens = Lens.lensu[Contact, Phone](
  (c, p) => c.copy(phone = p), _.phone)
```

Setter Getter

Lens

```scala
case class Name(
  salutation: String,
  first: String,
  last: String)

val nameSalutationLens = Lens.lensu[Name, String](
  (n, s) => n.copy(salutation = s), _.salutation)

val nameFirstLens = Lens.lensu[Name, String](
  (n, f) => n.copy(first = f), _.first)

val nameLastLens = Lens.lensu[Name, String](
  (n, l) => n.copy(last = l), _.last)
```

Lens

```
case class Phone(
  digits: String)

val phoneDigitsLens = Lens.lensu[Phone, String](
  (p, d) => p.copy(digits = d), _.digits)
```

Lens

Lenses compose!!

```
val contactFirstNameLens =
  contactNameLens >=> nameFirstLens
```

Composes 2 lenses in to a Contact to First Name lens via andThen

Lens

Lenses compose!!

```
val contactFirstNameLens =
  contactNameLens >=> nameFirstLens
```

Composes 2 lenses in to a Contact to First Name lens via andThen

```
val seth = Contact(
  Name("Mr.", "Seth", "Avett"), Phone("555-5555"))
```

Lens

Lenses compose!!

```
val contactFirstNameLens =
  contactNameLens >=> nameFirstLens
```

Composes 2 lenses in to a Contact to First Name lens via andThen

```
val seth = Contact(
  Name("Mr.", "Seth", "Avett"), Phone("555-5555"))

contactFirstNameLens.get(seth)
```

"Seth"

Lens

Lenses compose!!

```
val contactFirstNameLens =
  contactNameLens >=> nameFirstLens
```

Composes 2 lenses in to a Contact to First Name lens via andThen

```
val seth = Contact(
  Name("Mr.", "Seth", "Avett"), Phone("555-5555"))

contactFirstNameLens.get(seth)
```

"Seth"

```
contactFirstNameLens.set(seth, "Scott")
```

Contact(Name(Mr.,Scott,Avett),Phone(555-5555))

Future Topics

- Zipper
- Reader
- Writer
- State
- Monad Transformers
- Arrow
- Kleisli
- Category
- Iteratees

...and much more!

Questions?

Photo Credit: http://www.flickr.com/photos/yannconz/2795462449