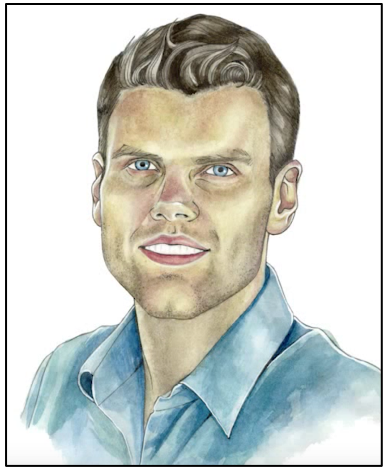
Sequence and Traverse

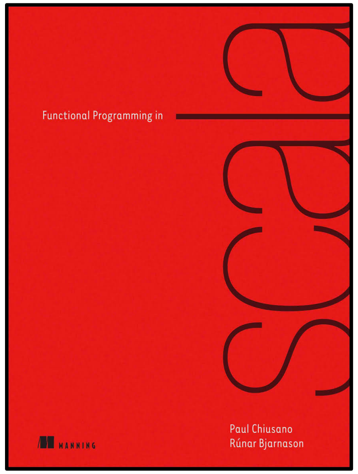
Part 3

learn about the sequence and traverse functions
through the work of



Runar Bjarnason

@runarorama

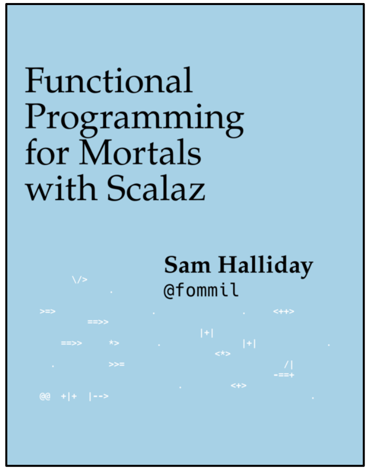


Functional Programming in
Scala
Paul Chiusano
Rúnar Bjarnason
MANNING

FP in Scala

Paul Chiusano

@pchiusano



Functional
Programming
for Mortals
with Scalaz

Sam Halliday
@fommil

Sam Halliday

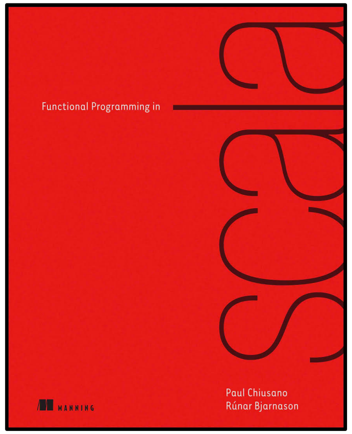
@fommil

slides by

@philip_schwarz

We now have instances of `Traverse` for `List`, `Option`, `Map`, and `Tree` . **What does this generalized `traverse` /`sequence` mean?** Let's just try plugging in some concrete type signatures for calls to `sequence`. We can speculate about what these functions do, just based on their signatures:

- `List[Option[A]] => Option[List[A]]` (a call to `Traverse[List].sequence` with `Option` as the `Applicative`) returns `None` if any of the input `List` is `None;` otherwise it returns the original `List` wrapped in `Some`.
- `Tree[Option[A]] => Option[Tree[A]]` (a call to `Traverse[Tree].sequence` with `Option` as the `Applicative`) returns `None` if any of the input `Tree` is `None`; otherwise it returns the original `Tree` wrapped in `Some`.
- `Map[K, Par[A]] => Par[Map[K,A]]` (a call to `Traverse[Map[K,_]].sequence` with `Par` as the `Applicative`) produces a parallel computation that evaluates all values of the map in parallel.



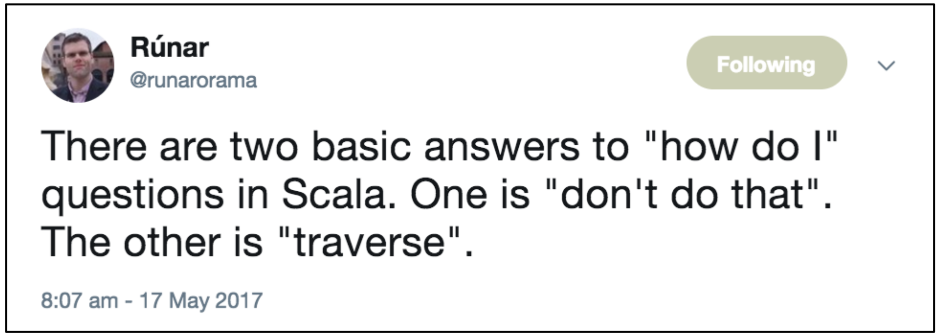
Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

There turns out to be a startling number of operations that can be defined in the most general possible way in terms of `sequence` and/or `traverse`


@runarorama


@pchiusano

```
trait Traverse[F[_]] {

  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
    sequence(map(fa)(f))

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```



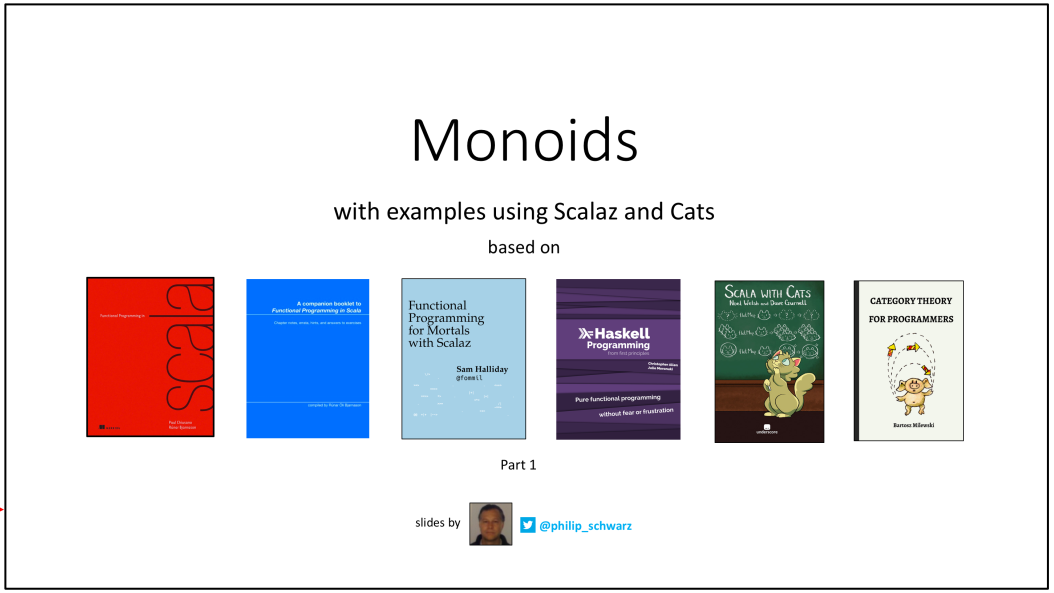
Rúnar
@runarorama

There are two basic answers to "how do I" questions in Scala. One is "don't do that". The other is "traverse".

8:07 am - 17 May 2017

In upcoming slides we are going to be referring to the **Foldable** trait and the **Monoid** trait. The next four slides are a minimal introduction to **Monoids**. The subsequent three slides are a minimal introduction to **Foldable**.

@philip_schwarz



Monoids

with examples using Scalaz and Cats

based on

Part 1

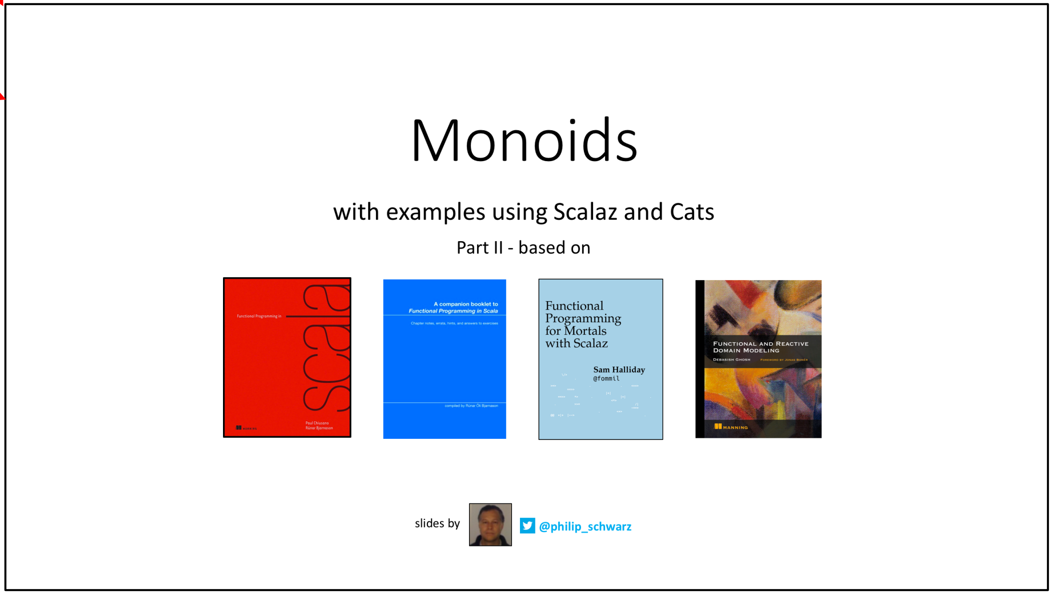
slides by @philip_schwarz

https://www.slideshare.net/pjschwarz/monoids-with-examples-using-scalaz-and-cats-part-1

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/monoids-with-examples-using-scalaz-and-cats-part-2

For more details on **Monoids**, see this and this. For more details on **Foldable** see slide 26 onwards of this (start from slide 16 for even more of an introduction to **folding**).



Monoids

with examples using Scalaz and Cats

Part II - based on

slides by @philip_schwarz

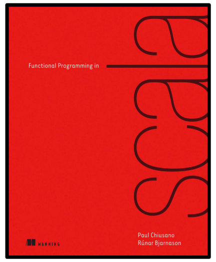
What is a monoid?

Let's consider the algebra of **string concatenation**. We can add `"foo" + "bar"` to get `"foobar"`, and the **empty string** is an **identity element** for that operation. That is, if we say `(s + "")` or `("" + s)`, the result is always `s`.

```
scala> val s = "foo" + "bar"
s: String = foobar

scala> assert( s == s + "" )

scala> assert( s == "" + s )

scala>
```

Furthermore, if we combine three strings by saying `(r + s + t)`, the operation is **associative** —it doesn't matter whether we parenthesize it: `((r + s) + t)` or `(r + (s + t))`.

```
scala> val (r,s,t) = ("foo","bar","baz")
r: String = foo
s: String = bar
t: String = baz

scala> assert( ( ( r + s ) + t ) == ( r + ( s + t ) ) )

scala> assert( ( ( r + s ) + t ) == "foobarbaz" )

scala>
```

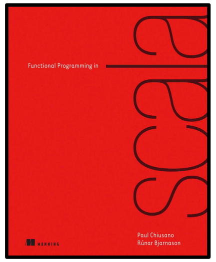


The exact same rules govern **integer addition**. It's **associative**, since `(x + y) + z` is always equal to `x + (y + z)`

```
scala> val (x,y,z) = (1,2,3)
x: Int = 1
y: Int = 2
z: Int = 3

scala> assert( ( ( x + y ) + z ) == ( x + ( y + z ) ) )

scala> assert( ( ( x + y ) + z ) == 6 )

scala>
```

and it has an **identity element**, `0` , which "does nothing" when added to another integer

```
scala> val s = 3
s: Int = 3

scala> assert( s == s + 0)

scala> assert( s == 0 + s)

scala>
```



Ditto for **integer multiplication**

```
scala> val (x,y,z) = (2,3,4)
x: Int = 2
y: Int = 3
z: Int = 4

scala> assert(( ( x * y ) * z ) == ( x * ( y * z ) ))

scala> assert(( ( x * y ) * z ) == 24)

scala>
```

whose **identity element** is **1**

```
scala> val s = 3
s: Int = 3

scala> assert( s == s * 1)

scala> assert( s == 1 * s)

scala>
```

The **Boolean** operators **`&&`** and **`||`** are likewise **associative**

```
scala> val (p,q,r) = (true,false,true)
p: Boolean = true
q: Boolean = false
r: Boolean = true

scala> assert(( ( p || q ) || r ) == ( p || ( q || r ) ))

scala> assert(( ( p || q ) || r ) == true )

scala> assert(( ( p && q ) && r ) == ( p && ( q && r ) ))

scala> assert(( ( p && q ) && r ) == false )
```

and they have **identity elements `true`** and **`false`**, respectively

```
scala> val s = true
s: Boolean = true

scala> assert( s == ( s && true ) )

scala> assert( s == ( true && s ) )

scala> assert( s == ( s || false ) )

scala> assert( s == ( false || s ) )
```

These are just a few simple examples, but **algebras like this are virtually everywhere**. The term for this kind of **algebra** is **monoid**.

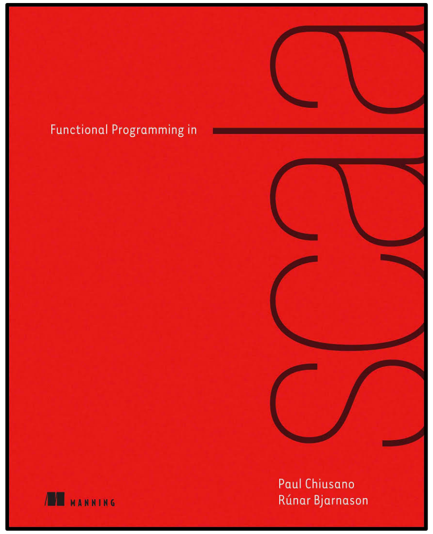
The **laws** of **associativity** and **identity** are collectively called the **monoid laws**.

A **monoid** consists of the following:

- Some type `A`
- An associative binary operation, `op`, that takes two values of type `A` and combines them into one: `op(op(x,y), z) == op(x, op(y,z))` for any choice of `x: A`, `y: A`, `z: A`
- A value, `zero: A`, that is an identity for that operation: `op(x, zero) == x` and `op(zero, x) == x` for any `x: A`

We can express this with a **Scala** trait:

```
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

An example instance of this trait is the `String` **monoid**:

```
val stringMonoid = new Monoid[String] {
  def op(a1: String, a2: String) = a1 + a2
  val zero = ""
}
```

String concatenation function

`List` concatenation also forms a **monoid**:

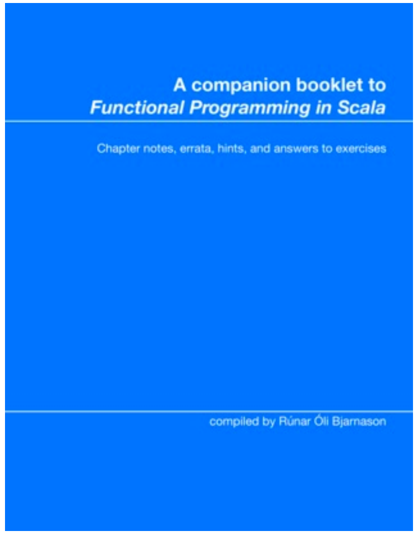
```
def listMonoid[A] = new Monoid[List[A]] {
  def op(a1: List[A], a2: List[A]) = a1 ++ a2
  val zero = Nil
}
```

List function returning a new list containing the elements from the left hand operand followed by the elements from the right hand operand

`Monoid` instances for **integer addition** and **multiplication** as well as the **Boolean operators**

```scala
implicit val intAdditionMonoid = new Monoid[Int] {
    def op(x: Int, y: Int) = x + y
    val zero = 0
}

implicit val intMultiplicationMonoid = new Monoid[Int] {
    def op(x: Int, y: Int) = x * y
    val zero = 1
}
```

```scala
implicit val booleanOr = new Monoid[Boolean] {
    def op(x: Boolean, y: Boolean) = x || y
    val zero = false
}

implicit val booleanAnd = new Monoid[Boolean] {
    def op(x: Boolean, y: Boolean) = x && y
    val zero = true
}
```

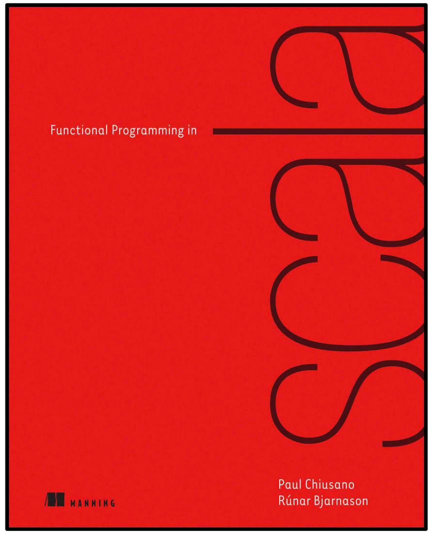

A companion booklet to
Functional Programming in Scala
Chapter notes, errata, hints, and answers to exercises
compiled by Rúnar Óli Bjarnason

(by Runar Bjarnason)
@runarorama

Just what is a **monoid**, then? It's simply a type `A` and **an implementation of `Monoid[A]`** that satisfies the **laws**.

Stated tersely, a **monoid** is a **type** together with a **binary operation** (**`op`**) over that type, satisfying **associativity** and having an **identity element** (**`zero`**).

What does this buy us? **Just like any abstraction, a monoid is useful to the extent that we can write useful generic code assuming only the capabilities provided by the abstraction. Can we write any interesting programs, knowing nothing about a type other than that it forms a monoid? Absolutely!**


Functional Programming in
Scala
Paul Chiusano
Rúnar Bjarnason
MANNING

Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

@philip_schwarz

That was the minimal introduction to **Monoid**. Next, we have three slides with a minimal introduction to **Foldable**.

Foldable data structures

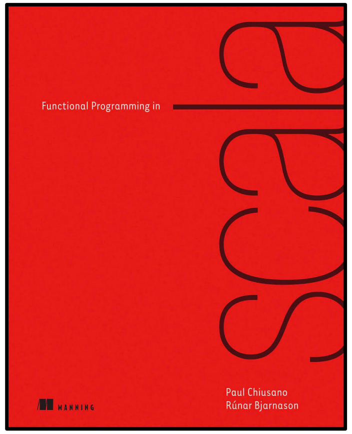
In chapter 3, we implemented the data structures `List` and `Tree`, both of which could be folded. In chapter 5, we wrote `Stream`, a lazy structure that also can be folded much like a `List` can, and now we've just written a fold for `IndexedSeq`.

When we're writing code that needs to process data contained in one of these structures, **we often don't care about** the shape of the structure (whether it's a tree or a list), or whether it's lazy or not, or provides efficient random access, and so forth.

For example, if we have a structure full of integers and want to calculate their sum, we can use `foldRight`:

```
ints.foldRight(0)(_ + _)
```

Looking at just this code snippet, **we shouldn't have to care about the type of `ints`**. It could be a `Vector`, a `Stream`, or a `List`, **or anything at all with a `foldRight` method**. **We can capture this commonality in a** `trait`:



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

```
trait Foldable[F[_]] {

   def foldRight[A,B](as: F[A])(z: B)(f: (A,B) => B): B

   def foldLeft[A,B](as: F[A])(z: B)(f: (B,A) => B): B

   def foldMap[A,B](as: F[A])(f: A => B)(mb: Monoid[B]): B

   def concatenate[A](as: F[A])(m: Monoid[A]): A =
     foldLeft(as)(m.zero)(m.op)
 }
```

Here we're abstracting over a type constructor `F`, much like we did with the `Parser` type in the previous chapter. We write it as `F[_]`, where the underscore indicates that `F` is not a type but a **type constructor** that takes one type argument. Just like functions that take other functions as arguments are called **higher-order functions**, something like `Foldable` is a **higher-order type constructor** or a higher-kinded type .[7]

[7] Just like values and functions have types, types and **type constructors** have kinds. Scala uses kinds to track how many type arguments a type constructor takes, whether it's co- or contravariant in those arguments, and what the kinds of those arguments are.

EXERCISE 10.12

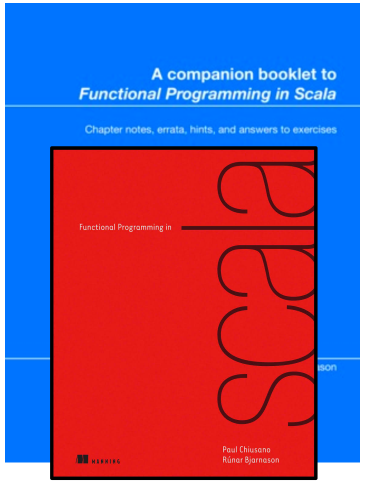
Implement **Foldable**`[List]`, **Foldable**`[IndexedSeq]`, and **Foldable**`[Stream]`. Remember that **foldRight**, **foldLeft**, and **foldMap** can all be implemented in terms of each other, but that might not be the most efficient implementation.

```scala
trait Foldable[F[_]] {

  def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
}

object ListFoldable extends Foldable[List] {
  override def foldRight[A, B](as:List[A])(z:B)(f:(A,B)=>B) =
    as.foldRight(z)(f)
  override def foldLeft[A, B](as:List[A])(z:B)(f:(B,A)=>B) =
    as.foldLeft(z)(f)
  override def foldMap[A, B](as:List[A])(f:A=>B)(mb:Monoid[B]):B =
    foldLeft(as)(mb.zero)((b, a) => mb.op(b, f(a)))
}

object IndexedSeqFoldable extends Foldable[IndexedSeq] {…}

object StreamFoldable extends Foldable[Stream] {
  override def foldRight[A, B](as:Stream[A])(z:B)(f:(A,B)=>B) =
    as.foldRight(z)(f)
  override def foldLeft[A, B](as:Stream[A])(z:B)(f:(B,A)=>B) =
    as.foldLeft(z)(f)
}
```

If you are new to **monoids**, don't worry about the implementation of **foldRight** and **foldLeft** except for the fact that it is possible to define them using **foldMap**.

Using the methods of **ListFoldable** and **StreamFoldable** to fold **Lists**/**Streams** of **Ints** and **Strings**.

A Companion booklet to FP in Scala



FP in Scala

```scala
assert( ListFoldable.foldLeft(List(1,2,3))(0)(_+_) == 6)
assert( ListFoldable.foldRight(List(1,2,3))(0)(_+_) == 6)

assert( ListFoldable.concatenate(List(1,2,3))(intAdditionMonoid) == 6)
assert( ListFoldable.foldMap(List("1","2","3"))(_ toInt)(intAdditionMonoid) == 6)

assert( StreamFoldable.foldLeft(Stream(1,2,3))(0)(_+_) == 6)
assert( StreamFoldable.foldRight(Stream(1,2,3))(0)(_+_) == 6)

assert( StreamFoldable.concatenate(Stream(1,2,3))(intAdditionMonoid) == 6)
assert( StreamFoldable.foldMap(Stream("1","2","3"))(_ toInt)(intAdditionMonoid) == 6)
```

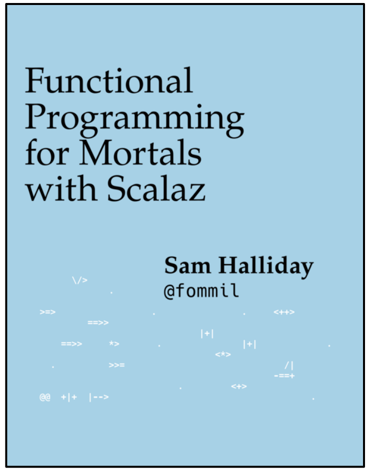
```scala
assert( ListFoldable.foldLeft(List("a","b","c"))("")(_+_) == "abc")
assert( ListFoldable.foldRight(List("a","b","c"))("")(_+_) == "abc")

assert( ListFoldable.concatenate(List("a","b","c"))(stringMonoid) == "abc")
assert( ListFoldable.foldMap(List(1,2,3))(_ toString)(stringMonoid) == "123")

assert( StreamFoldable.foldLeft(Stream("a","b","c"))("")(_+_) == "abc")
assert( StreamFoldable.foldRight(Stream("a","b","c"))("")(_+_) == "abc")

assert( StreamFoldable.concatenate(Stream("a","b","c"))(stringMonoid) == "abc")
assert( StreamFoldable.foldMap(Stream(1,2,3))(_ toString)(stringMonoid) == "123")
```

@fommil

Technically, **Foldable** is for data structures that can be walked to produce a summary value. However, this undersells the fact that it is **a one-typeclass army that can provide most of what you'd expect to see in a Collections API**.

```
@typeclass trait Foldable[F[_]] {
  def foldMap[A, B: Monoid](fa: F[A])(f: A => B): B
  def foldRight[A, B](fa: F[A], z: =>B)(f: (A, =>B) => B): B
  def foldLeft[A, B](fa: F[A], z: B)(f: (B, A) => B): B = ...
```

You might recognise **foldMap** by its marketing buzzword name, **MapReduce**. Given an `F[A]`, a function from `A` to `B`, and a way to **combine** `B` (provided by the **Monoid**, along with a **zero** `B`), we can **produce a summary value of type** `B`. There is no enforced operation order, allowing for parallel computation.



Functional
Programming
for Mortals
with Scalaz

Sam Halliday
@fommil

Sam Halliday

foldRight does not require its parameters to have a **Monoid**, meaning that it needs a starting value **z** and a way to **combine** each element of the data structure with the summary value. The order for traversing the elements is from right to left and therefore it cannot be parallelised.

foldLeft traverses elements from left to right. **foldLeft** can be implemented in terms of **foldMap**, but most instances choose to implement it because it is such a basic operation. Since it is usually implemented with **tail recursion**, there are no byname parameters.

The simplest thing to do with **foldMap** is to use the identity function, giving **fold** (the natural sum of the **monoidal** elements), with left/right variants to allow choosing based on performance criteria:

Sam Halliday

In **FPiS**, **fold** is called **concatenate**.

```
def fold[A: Monoid](t: F[A]): A = ...
 def sumr[A: Monoid](fa: F[A]): A = ...
 def suml[A: Monoid](fa: F[A]): A = ...
…
```

Anything you'd expect to find in a collection library is probably on **Foldable** and if it isn't already, it probably should be.

@philip_schwarz

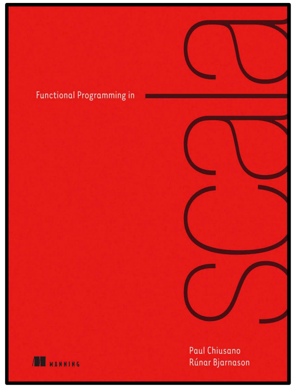
With that refresher on **Monoid** and **Foldable** out of the way, let's first compare **Traverse**.**traverse** with **Foldable**.**foldMap** and then see the connection between **Traverse**.**traverse** and **Functor**.**map**.

A **traversal** is similar to a **fold** in that both take some data structure and apply a function to the data within in order to produce a result.

The difference is that **`traverse`** **preserves the original structure** , whereas **`foldMap`** **discards the structure and replaces it with the operations of a** **monoid**.

Look at the signature `Tree[Option[A]] => Option[Tree[A]]`, for instance.

We're preserving the `Tree` structure, not merely collapsing the values using some monoid.



FP in Scala

```
trait Traverse[F[_]] {
  def traverse[M[_]:Applicative,A,B](as: F[A])(f: A => M[B]): M[F[B]]

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

```
trait Foldable[F[_]] {
  def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
}
```

To illustrate this difference between how **traverse** preserves the structure of a **`Tree`** and **foldMap** collapses the structure using some **monoid**, we are going to look at an example in which both **traverse** and **foldMap** convert the elements of the **`Tree`** from **`String`** values to **`Int`** values, but while **traverse** produces an **`Option[Tree[Int]]`**, **foldMap** produces an **`Int`**.

```
traverse(as: Tree[String])(f: String => Option[Int])(implicit G:  Applicative[Option]): Option[Tree[Int]]
 foldMap(as: Tree[String])(f: String => Int)        (implicit mb: Monoid[Int])        : Int
```

```
// traverse
//
//   "2"    -->     Some(2)    -->  Some( 2 )
//   / \             / \              / \
// "1" "3"    Some(1)  Some(3)        1   3
```

```
// foldMap using integer addition monoid
//
//   "2"    -->  2   -->  6
//   / \        / \
// "1" "3"     1   3
```

See the next slide for the full example.

@philip_schwarz

Traversing and folding the same tree.

```
case class Tree[+A](head: A, tail: List[Tree[A]])

val tree = Tree("2",List(Tree("1", Nil), Tree("3", Nil)))
```

```
//   "2"
//   /  \
// "1" "3"
```

```
implicit val optionApplicative = new Applicative[Option] {
  def map2[A, B, C](fa: Option[A], fb: Option[B])(f: (A, B) => C): Option[C] =
    (fa, fb) match {
      case (Some(a), Some(b)) => Some(f(a,b))
      case _ => None
  }
  def unit[A](a: => A): Option[A] = Some(a)
}
```

```
implicit val intMonoid = new Monoid[Int]{
  def op(a1: Int, a2: Int): Int = a1 + a2
  def zero: Int = 0
}
```

In **FPiS** the `Tree` type that **treeFoldable** operates on is a bit different from the one that **treeTraverse** operates on. Here I rewrote **treeFoldable** to operate on the same `Tree` type as **treeTraverse**, so that I can show an example of **traversing** and **folding** the same tree.

```
val listTraverse = new Traverse[List] {
  override def traverse[M[_],A,B](as:List[A])(f: A=>M[B])(implicit M:Applicative[M]):M[List[B]] =
    as.foldRight(M.unit(List[B]()))((a, fbs) => M.map2(f(a), fbs)(_ :: _))
}

val treeTraverse = new Traverse[Tree] {
  override def traverse[G[_],A,B](ta:Tree[A])(f: A=>G[B])(implicit G:Applicative[G]):G[Tree[B]] =
    G.map2(f(ta.head),
           listTraverse.traverse(ta.tail)(a => traverse(a)(f)))(Tree(_, _))
}
```

```
val treeFoldable = new Foldable[Tree] {
  override def foldMap[A,B](as:Tree[A])(f:A=>B)(mb:Monoid[B]):B =
    as match {
      case Tree(head,Nil) =>
        f(head)
      case Tree(head,tree::rest) =>
        mb.op(foldMap(Tree(head,rest))(f)(mb),
              foldMap(tree)(f)(mb))
    }
}
```

```
val parseInt: String => Option[Int] = x => Try{ x.toInt }.toOption

val traversed = Some(Tree(2,List(Tree(1,Nil), Tree(3,Nil))))

assert( treeTraverse.traverse(tree)(parseInt)(optionApplicative) == traversed )
```

```
val toInt: String => Int = _.toInt

val folded = 6

assert( treeFoldable.foldMap(tree)(toInt)(intMonoid) == folded )
```

```
// treeTraverse.traverse(tree)(parseInt)(optionApplicative)
//
//   "2"   -->    Some(2)   -->  Some( 2 )
//   /  \          /  \               / \
// "1" "3"   Some(1)  Some(3)        1  3
```

```
// treeFoldable.foldMap(tree)(toInt)(intMonoid)
//
//   "2"    -->  2   -->  6
//   /  \       / \
// "1" "3"     1   3
```

We were first introduced to the **Traverse** trait in **Part 2**

At that time we found it confusing that **Traverse**'s **traverse** function used a **map** function that was not defined anywhere

```
trait Traverse[F[_]] {

  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
    sequence(map(fa)(f))

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

We did however know, from **Part 1**, that just like **flatMap** is **map** and then **flatten**, **traverse** is **map** and then **sequence**.

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B] = flatten(ma map f)

def flatten[A](mma: F[F[A]]): F[A] = flatMap(mma)(x ⇒ x)

def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] = sequence(a map f)

def sequence[A](a: List[Option[A]]): Option[List[A]] = traverse(a)(x ⇒ x)
```

So at that time I suggested we think of **Traverse**'s **traverse** method as not having a body, i.e. being abstract. I reckoned the body was just there to point out that if **Traverse** did have a **map** function then it could be used to implement **traverse**.

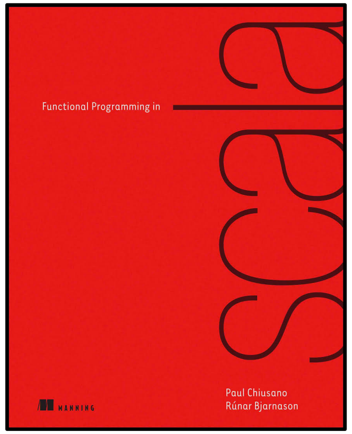
All the traverse instances we have looked at so far implemented **traverse** without using a **map** function.

In the next two slides we are finally going to see the connection between **Traverse** and **map**.

EXERCISE 12.14

Hard: Implement `map` in terms of `traverse` as a method on `Traverse[A]`. This establishes that `Traverse` is an extension of `Functor` and that the `traverse` function is a generalization of `map` (for this reason we sometimes call these **traversable functors**). Note that in implementing `map`, you can call `traverse` with your choice of `Applicative[G]`.

```
trait Traverse[F[_]] extends Functor[F] {

  def traverse[G[_],A,B](fa: F[A])(f: A => G[B])(implicit G: Applicative[G]): G[F[B]] =
    sequence(map(fa)(f))

  def sequence[G[_],A](fga: F[G[A]])(implicit G: Applicative[G]): G[F[A]] =
    traverse(fga)(ga => ga)

  def map[A,B](fa: F[A])(f: A => B): F[B] = ???
}
```



Functional Programming in Scala

(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

Answer to **EXERCISE 12.14**

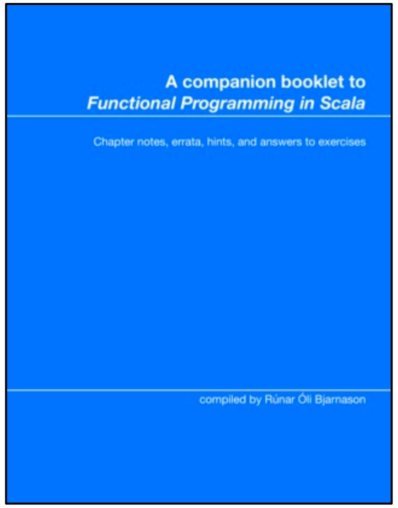
```
trait Traverse[F[_]] extends Functor[F] {

  def traverse[G[_] : Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]] =
    sequence(map(fa)(f))

  def sequence[G[_] : Applicative, A](fga: F[G[A]]): G[F[A]] =
    traverse(fga)(ga => ga)

  type Id[A] = A

  val idMonad = new Monad[Id] {
    def unit[A](a: => A) = a
    override def flatMap[A, B](a: A)(f: A => B): B = f(a)
  }

  def map[A, B](fa: F[A])(f: A => B): F[B] =
    traverse[Id, A, B](fa)(f)(idMonad)
}
```

The simplest possible `Applicative` we can use is `Id`.

We already know this forms a **Monad**, so it's also an **applicative functor**.

We can now implement **map** by calling **traverse**, picking `Id` as the `Applicative`.

Note that we can define **traverse** in terms of **sequence** and **map**, which means that a valid **Traverse** instance may define **sequence** and **map**, or just **traverse**.



by Runar Bjarnason @runarorama

Mind blown: **traverse** is a generalization of **map**. Conversely: **mapping** is a specialization of **traversing**. When we **traverse** using the degenerate **Applicative** that is the **Id Monad**, we are just **mapping**. **Mapping** is **traversing** with the **Id Monad** as an **Applicative**.

So that's why we found it confusing when **FPiS** first introduced **Traverse** (below) with a **traverse** implementation that referred to an undefined **map** function.

```scala
trait Traverse[F[_]] {

  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
    sequence(map(fa)(f))

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

We had not yet been told that **Traverse** is a **Functor**: it either simply defines a **traverse** function, in which case it gets free definitions of **sequence** and **map** based on **traverse**, or it defines both a **map** function and a **sequence** function and both are then used to implement **traverse**. See below for what was missing from the **Traverse** trait.

```scala
trait Traverse[F[_]] extends Functor[F] {

  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
    sequence(map(fa)(f))

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)

  def map[A,B](fa: F[A])(f: A => B): F[B] =
    traverse[Id, A, B](fa)(f)(idMonad)
}
```

In the next slide we have a go at using the **map** function of **listTraverse**, **optionTraverse** and **treeTraverse**.

@philip_schwarz

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

trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]

  override def map[A,B](m: F[A])(f: A => B): F[B] =
    flatMap(m)(a => unit(f(a)))

  override def map2[A,B,C](ma:F[A], mb:F[B])(f:(A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))
}

trait Applicative[F[_]] extends Functor[F] {

  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]
  def unit[A](a: => A): F[A]

  def apply[A,B](fab: F[A => B])(fa: F[A]): F[B] = map2(fab, fa)(_(_))
  def map[A,B](fa: F[A])(f: A => B): F[B] = apply(unit(f))(fa)
}
```

```
trait Traverse[F[_]] extends Functor[F] {

  def traverse[M[_]:Applicative,A,B](fa:F[A])(f:A=>M[B]):M[F[B]]

  def sequence[M[_] : Applicative, A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)

  type Id[A] = A
  val idMonad = new Monad[Id] {
    def unit[A](a: => A) = a
    override def flatMap[A, B](a: A)(f: A => B): B = f(a)
  }

  def map[A, B](fa: F[A])(f: A => B): F[B] =
    traverse[Id, A, B](fa)(f)(idMonad)
}
```

```
val listTraverse = new Traverse[List] {

  override def traverse[M[_], A, B](as: List[A])(f: A => M[B])
      (implicit M: Applicative[M]): M[List[B]] =
    as.foldRight(M.unit(List[B]()))
                 ((a, fbs) => M.map2(f(a), fbs)(_ :: _))
}

val optionTraverse = new Traverse[Option] {

  override def traverse[M[_],A,B](oa: Option[A])(f: A => M[B])
      (implicit M: Applicative[M]): M[Option[B]] =
    oa match {
      case Some(a) => M.map(f(a))(Some(_))
      case None    => M.unit(None)
    }
}

case class Tree[+A](head: A, tail: List[Tree[A]])

val treeTraverse = new Traverse[Tree] {

  override def traverse[M[_], A, B](ta: Tree[A])(f: A => M[B])
      (implicit M: Applicative[M]): M[Tree[B]] =
    M.map2(f(ta.head),
           listTraverse.traverse(ta.tail)(a => traverse(a)(f))
    )(Tree(_, _))
}
```

```
val double: Int => Int = _ * 2

assert(listTraverse.map(List(1,2,3))(double) == List(2,4,6))
assert(optionTraverse.map(Some(2))(double) == Some(4))

val tree = Tree(2,List(Tree(1, Nil), Tree(3, Nil)))
val doubledTree = Tree(4,List(Tree(2, Nil), Tree(6, Nil)))

assert(treeTraverse.map(tree)(double) == doubledTree)
```

And in the next slide we use another example to look a bit closer at how **mapping** is just **traversing** with the **Id Monad** **Applicative**.

```
implicit val optionApplicative = new Applicative[Option] {
  def map2[A, B, C](fa: Option[A], fb: Option[B])(f: (A, B) => C): Option[C] =
    (fa, fb) match {
      case (Some(a), Some(b)) => Some(f(a,b))
      case _ => None
  }
  def unit[A](a: => A): Option[A] = Some(a)
}
```

```
type Id[A] = A

val idMonad = new Monad[Id] {

  def unit[A](a: => A) = a

  override def flatMap[A, B](a: A)(f: A => B): B = f(a)
}
```

```
val parseInt: String => Option[Int] = x => scala.util.Try{ x.toInt }.toOption

assert( listTraverse.traverse(List("1","2","3"))(parseInt)(optionApplicative) == Some(List(1,2,3)))
```

```
List("1","2","3") → List(Some(1),Some(2),Some(3)) → Some(List(1,2,3) )

List[String] → List[Option[Int]] → Option[List[Int]]
```

```
val toInt: String => Id[Int] = _.toInt

assert( listTraverse.map(List("1","2","3"))(toInt) == List(1,2,3))
assert( listTraverse.traverse(List("1","2","3"))(toInt)(idMonad) == List(1,2,3))
```

Here we use Id(x) to represent x lifted into the Id Monad.

```
List("1","2","3") → List(1,2,3) → List(1,2,3)
List("1","2","3") → List(Id(1), Id(2), Id(3)) → Id( List(1,2,3) )
List("1","2","3") → List(idMonad.unit(1),idMonad.unit(2),idMonad.unit(3)) → idMonad.unit( List(1,2,3) )

List[String] → List[Id[Int]] → Id[List[Int]]
List[String] → List[Id[Int]] → Id[List[Int]]
List[String] → List[   Int ] →    List[Int]
```

@philip_schwarz

Having compared **Traverse**.**traverse** with **Foldable**.**foldMap** and seen the connection between **Traverse**.**traverse** and **Functor**.**map**, we know go back to looking at the connection between **Traverse** and **Foldable**.

12.7.1 From monoids to applicative functors

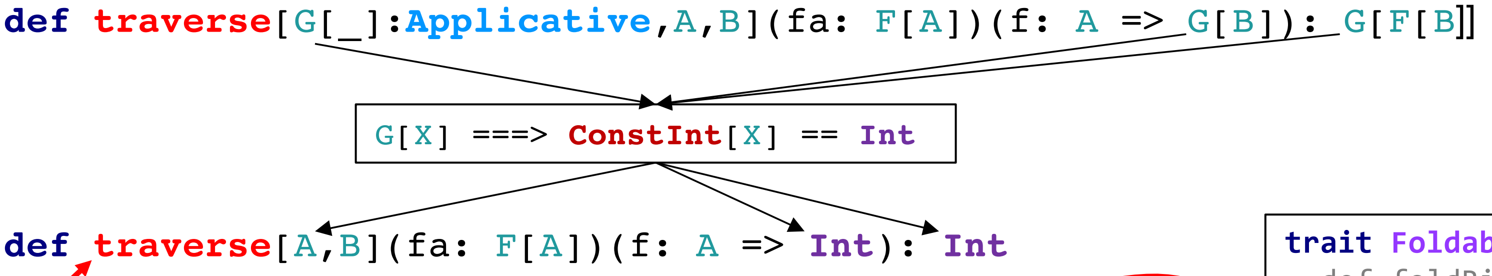
We've just learned that **`traverse`** **is more general than** **`map`**. Next we'll learn that **`traverse`** **can also express** **`foldMap`** **and by extension** **`foldLeft`** **and** **`foldRight`**! Take another look at the signature of **`traverse`**:

```scala
def traverse[G[_]:Applicative,A,B](fa: F[A])(f: A => G[B]): G[F[B]]
```

Suppose that our `G` were a **type constructor** **`ConstInt`** that takes any type to **`Int`**, so that **`ConstInt[A]`** **throws away** its type argument `A` and just gives us **`Int`**:

```scala
type ConstInt[A] = Int
```

Then in the type signature for **`traverse`**, if we instantiate `G` to be **`ConstInt`**, it becomes



```scala
def traverse[G[_]:Applicative,A,B](fa: F[A])(f: A => G[B]): G[F[B]]
```

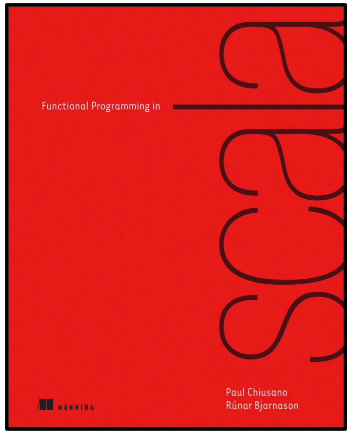
```
G[X] ===> ConstInt[X] == Int
```

```scala
def traverse[A,B](fa: F[A])(f: A => Int): Int
```

This looks a lot like **`foldMap`** from **`Foldable`**. Indeed, if `F` is something like **`List`**, then what we need to implement this signature is a way of combining the **`Int`** values returned by `f` for each element of the list, and a **"starting" value** for handling the **empty list**. In other words, we only need a **`Monoid[Int]`**. And that's easy to come by.

In fact, **given a constant functor** like we have here, **we can turn any** **`Monoid`** **into an** **`Applicative`**. (see next slide)

```scala
trait Foldable[F[_]] {
  def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
}
```



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Listing 12.8 Turning a **Monoid** into an **Applicative**

```
type Const[M, B] = M

implicit def monoidApplicative[M](M: Monoid[M]) =

  new Applicative[({ type f[x] = Const[M, x] })#f] {

    def unit[A](a: => A): M = M.zero

    def map2[A,B,C](m1: M, m2: M)(f: (A,B) => C): M = M.op(m1,m2)

  }
```

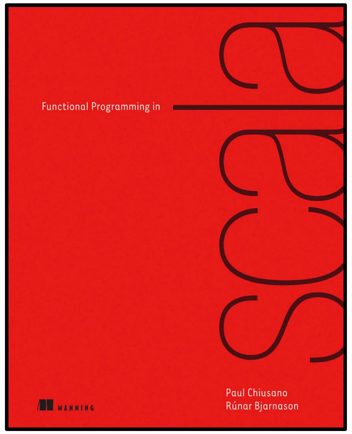
This is `ConstInt` generalized to any `M`, not just `Int`.

Scala can't infer the partially applied `Const` type alias here, so we have to provide an annotation.

This means that **Traverse** can extend **Foldable** and we can give a default implementation of **foldMap** in terms of **traverse**:

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {
  …
  override def foldMap[A,M](as: F[A])(f: A => M)(mb: Monoid[M]): M =
    traverse[({type f[x] = Const[M,x]})#f,A,Nothing](as)(f)(monoidApplicative(mb))
}
```

Note that **Traverse** now extends both **Foldable** and **Functor** ! Importantly, **Foldable** itself can't extend **Functor**. Even though it's possible to write **map** in terms of a **fold** for most foldable data structures like **List**, it's not possible in general.



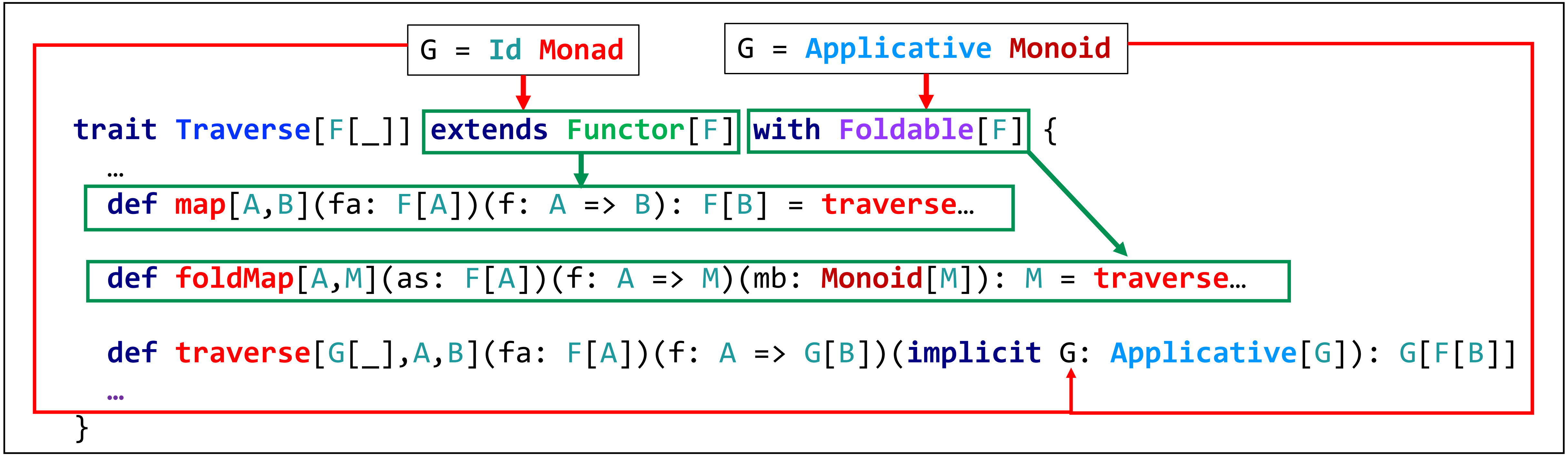
Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Earlier I had a **Mind Blown** moment because **traverse** is a generalization of **map**. **Mind Blown again:** **traverse** can also express **foldMap**.

So although earlier we said that the difference between **traverse** and **foldMap** is that **traverse** **preserves the original structure** whereas **foldMap** **discards the structure**, if we pass **traverse** a **monoid** **applicative** then it behaves like **foldMap** and **discards the structure** (replacing it with the operations of a **monoid**).

The next slide is a final recap of how, by using `Id` and `Const`, we can get **Traverse** to implement **map** and **foldMap**, which means that **Traverse** is both a **Functor** and a **Foldable**.

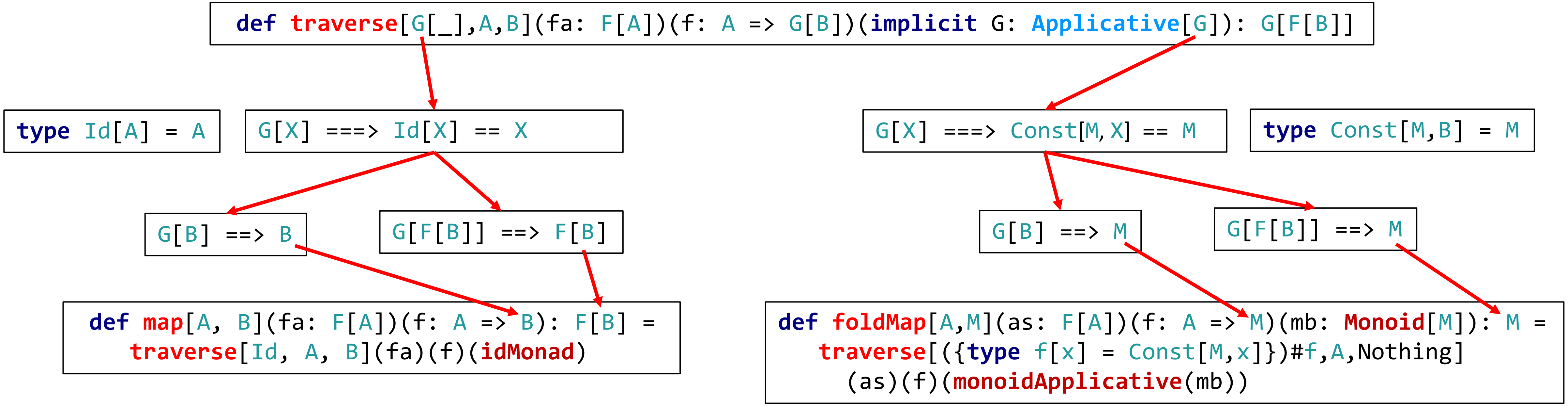
@philip_schwarz



G = Id Monad

G = Applicative Monoid

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {
  …
  def map[A,B](fa: F[A])(f: A => B): F[B] = traverse…

  def foldMap[A,M](as: F[A])(f: A => M)(mb: Monoid[M]): M = traverse…

  def traverse[G[_],A,B](fa: F[A])(f: A => G[B])(implicit G: Applicative[G]): G[F[B]]
  …
}
```



```
def traverse[G[_],A,B](fa: F[A])(f: A => G[B])(implicit G: Applicative[G]): G[F[B]]
```

```
type Id[A] = A
```

```
G[X] ===> Id[X] == X
```

```
G[B] ==> B
```

```
G[F[B]] ==> F[B]
```

```
def map[A, B](fa: F[A])(f: A => B): F[B] =
  traverse[Id, A, B](fa)(f)(idMonad)
```

```
G[X] ===> Const[M, X] == M
```

```
type Const[M,B] = M
```

```
G[B] ==> M
```

```
G[F[B]] ==> M
```

```
def foldMap[A,M](as: F[A])(f: A => M)(mb: Monoid[M]): M =
  traverse[({type f[x] = Const[M,x]})#f,A,Nothing]
    (as)(f)(monoidApplicative(mb))
```

5. Scalaz Typeclasses

Before we introduce the typeclass hierarchy, we will peek at **the four most important methods from a control flow perspective**, **the methods we will use the most in typical FP applications**:

…

Typeclass	Method	From	Given	To
`Functor`	`map`	`F[A]`	`A => B`	`F[B]`
`Applicative`	`pure`	`A`		`F[A]`
`Monad`	`flatMap`	`F[A]`	`A => F[B]`	`F[B]`
`Traverse`	`traverse`	`F[A]`	`A => G[B]`	`G[F[B]]`

…

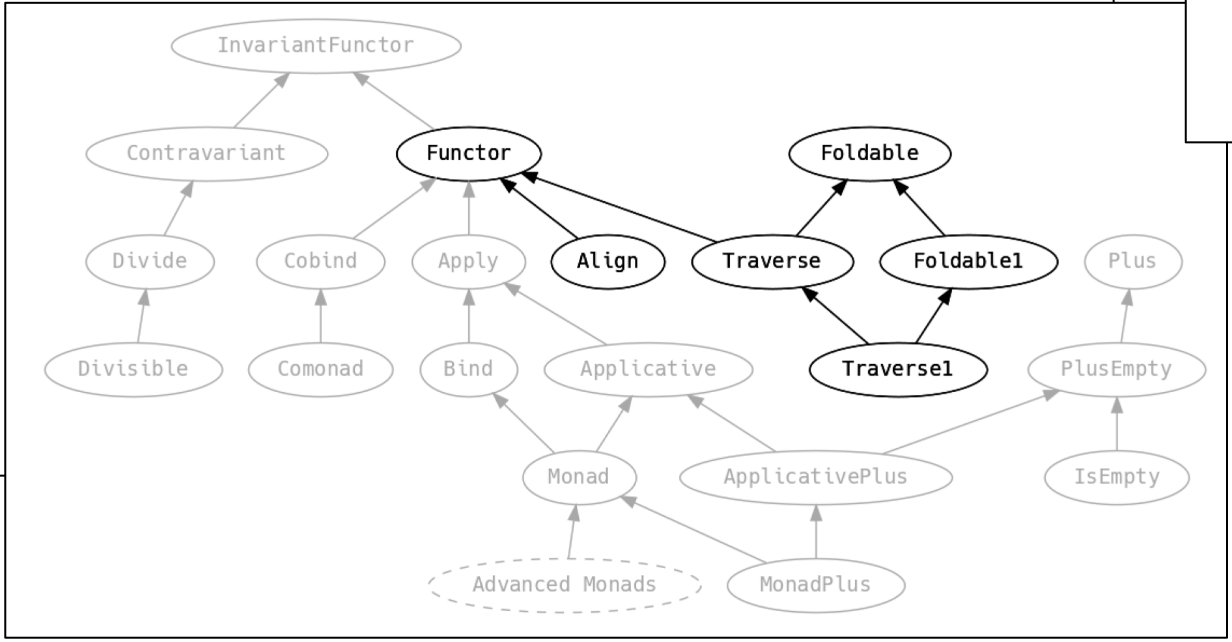
traverse is useful for rearranging type constructors. If you find yourself with an `F[G[_]]` but you really need a `G[F[_]]` then you need Traverse. For example, say you have a `List[Future[Int]]` but you need it to be a `Future[List[Int]]`, just call .**traverse**(**identity**), or its simpler sibling .**sequence**.

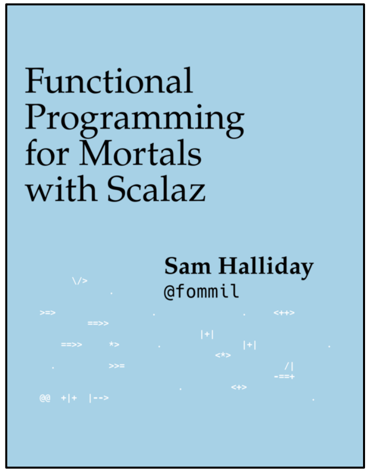
…

5.4 Mappable Things

We're focusing on things that can be **mapped over**, or **traversed**, in some sense…

…




Functional
Programming
for Mortals
with Scalaz

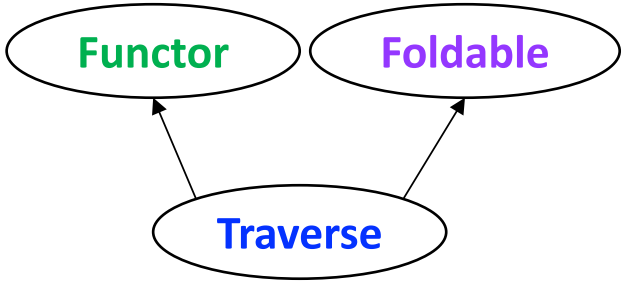
Sam Halliday
@fommil

Sam Halliday

@fommil

Sam Halliday

Traverse is what happens when you cross a **Functor** with a **Foldable**.



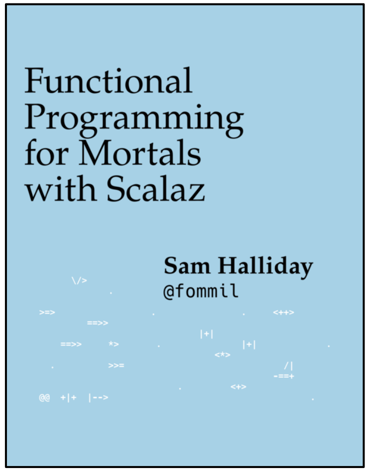
Functor

Foldable

Traverse

You will use these methods (**sequence** and **traverse**) more than you could possibly imagine.

@fommil



Functional
Programming
for Mortals
with Scalaz

Sam Halliday
@fommil

Sam Halliday

5.4.3 Traverse

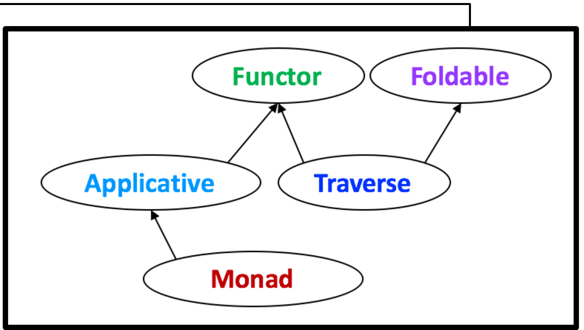
Traverse is what happens when you cross a **Functor** with a **Foldable**

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] {
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
  def sequence[G[_]: Applicative, A](fga: F[G[A]]): G[F[A]] = ...
  def reverse[A](fa: F[A]): F[A] = ...
  def zipL[A, B](fa: F[A], fb: F[B]): F[(A, Option[B])] = ...
  def zipR[A, B](fa: F[A], fb: F[B]): F[(Option[A], B)] = ...
  def indexed[A](fa: F[A]): F[(Int, A)] = ...
  def zipWithL[A, B, C](fa: F[A], fb: F[B])(f: (A, Option[B]) => C): F[C] = ...
  def zipWithR[A, B, C](fa: F[A], fb: F[B])(f: (Option[A], B) => C): F[C] = ...
  def mapAccumL[S, A, B](fa: F[A], z: S)(f: (S, A) => (S, B)): (S, F[B]) = ...
  def mapAccumR[S, A, B](fa: F[A], z: S)(f: (S, A) => (S, B)): (S, F[B]) = ...
}
```

At the beginning of the chapter we showed **the importance of traverse and sequence for swapping around type constructors to fit a requirement** (e.g. `List[Future[_]]` to `Future[List[_]]`). **You will use these methods more than you could possibly imagine**.

In the next two slides we have a go at using a **Traverse** instance (now that **Traverse** is both a **Functor** and a **Foldable**).

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

trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]

  override def map[A,B](m: F[A])(f: A => B): F[B] =
    flatMap(m)(a => unit(f(a)))

  override def map2[A,B,C](ma:F[A], mb:F[B])(f:(A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))
}
```



```
trait Foldable[F[_]] {
  import Monoid._

  def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def foldMap[A,B](as:F[A])(f:A=>B)(implicit mb: Monoid[B]):B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

  def concatenate[A](as: F[A])(implicit m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
}
```

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] { self =>

  def traverse[M[_]:Applicative,A,B](fa:F[A])(f:A=>M[B]):M[F[B]]

  def sequence[M[_] : Applicative, A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)

  type Id[A] = A
  val idMonad = new Monad[Id] {
    def unit[A](a: => A) = a
    override def flatMap[A, B](a: A)(f: A => B): B = f(a)
  }

  def map[A, B](fa: F[A])(f: A => B): F[B] =
    traverse[Id, A, B](fa)(f)(idMonad)

  import Applicative._

  override def foldMap[A,B](as: F[A])(f: A => B)
    (implicit mb: Monoid[B]): B =
    traverse[({type f[x] = Const[B,x]})#f,A,Nothing](
      as)(f)(monoidApplicative(mb))
 …
}
```

```
trait Monoid[A] {
  def op(x: A, y: A): A
  def zero: A
}
```

```
type Const[M, B] = M
implicit def monoidApplicative[M](M: Monoid[M]) =
  new Applicative[({ type f[x] = Const[M, x] })#f] {
    def unit[A](a: => A): M = M.zero
    def map2[A,B,C](m1: M, m2: M)(f: (A,B) => C): M = M.op(m1,m2)
  }
```

```
trait Applicative[F[_]] extends Functor[F] {

  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]

  def unit[A](a: => A): F[A]

  def map[B](fa: F[A])(f: A => B): F[B] =
    map2(fa, unit(()))((a, _) => f(a))

  def sequence[A](fas: List[F[A]]): F[List[A]] =
    traverse(fas)(fa => fa)

  def traverse[A,B](as: List[A])(f: A => F[B]): F[List[B]]
    as.foldRight(unit(List[B]()))((a, fbs) => map2(f(a), fbs)(_ :: _))
  …
}
```

```scala
implicit val optionApplicative = new Applicative[Option] {

  def map2[A, B, C](fa: Option[A], fb: Option[B])(f: (A, B) => C): Option[C] =
    (fa, fb) match {
      case (Some(a), Some(b)) => Some(f(a,b))
      case _ => None
    }

  def unit[A](a: => A): Option[A] = Some(a)
}
```

Creating a **Traverse[List]** instance and exercising its functions, including its **Functor** and **Foldable** functions.

```scala
implicit val integerAdditionMonoid = new Monoid[Int] {
  def op(x: Int, y: Int): Int = x + y
  def zero: Int = 0
}
```

```scala
val parseInt: String => Option[Int] = (s:String) => scala.util.Try(s.toInt).toOption
```

```scala
val listTraverse = new Traverse[List] {
  override def traverse[M[_], A, B](as: List[A])(f: A => M[B])(implicit M: Applicative[M]): M[List[B]] =
    as.foldRight(M.unit(List[B]()))((a, fbs) => M.map2(f(a), fbs)(_ :: _))
}
```

`implicit optionApplicative`

`implicit integerAdditionMonoid`

```scala
// Traverse
assert( listTraverse.traverse(List("1","2","3"))(parseInt) == Some(List(1, 2, 3)) )
assert( listTraverse.sequence(List(Option(1),Option(2),Option(3))) == Some(List(1, 2, 3)) )
// Functor                          uses IdMonad
assert( listTraverse.map(List(1,2,3))(_ + 1) == List(2,3,4))
// Foldable                             uses monoidApplicative
assert( listTraverse.foldMap(List("1","2","3"))(_ toInt) == 6 )
assert( listTraverse.concatenate(List(1,2,3)) == 6 )
assert( listTraverse.foldRight(List(1,2,3))(0)(_ + _) == 6 )
assert( listTraverse.foldLeft(List(1,2,3))(0)(_ + _) == 6 )
```

To be continued in part IV