Symmetry in the interrelation of flatMap/foldMap/traverse and flatten/fold/sequence

to flatMap, first map and then flatten
to foldMap, first map and then fold ‡
to traverse, first map and then sequence

to flatten, just flatMap identity
to fold, just foldMap identity
to sequence, just traverse with identity

@philip_schwarz

While this is just a simple observation, I think it is a nice example of the **symmetries** that help us reason about **functional programs**.

Of course I am not suggesting that flatMap, traverse and foldMap are actually implemented this way, just that this definition helps us understand them.

‡ see next slide for a caveat.

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]                    =  flatten(map(ma)(f))
def foldMap[A,B:Monoid](fa: F[A])(f: A ⇒ B): B                   =     fold(map(fa)(f))  ‡
def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A ⇒ M[B]): M[F[B]] = sequence(map(fa)(f))

def flatten[A](mma: F[F[A]]): F[A]                               =  flatMap(mma)(x ⇒ x)
def fold[A:Monoid](fa: F[A]): A                                  =   foldMap(fa)(x ⇒ x)
def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]]          = traverse(fma)(x ⇒ x)
```

flatMapping is mapping and then flattening - **flattening** is just flatMapping identity

foldMapping is mapping and then folding – **folding** is just foldMapping identity ‡

traversing is mapping and then sequencing - **sequencing** is just traversing with identity

```scala
trait Monad[F[_]] extends Functor[F] { self ⇒

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

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

  …
}
```

to **flatMap**, first **map** and then **flatten**
to **foldMap**, first **map** and then **fold**
to **traverse**, first **map** and then **sequence**

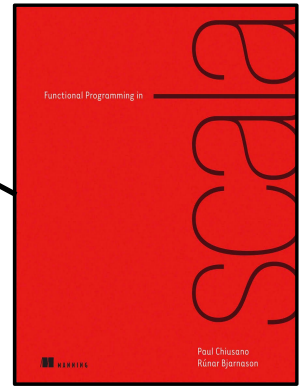
to **flatten**, just **flatMap identity**
to **fold**, just **foldMap identity**
to **sequence**, just **traverse** with **identity**

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

  def foldMap[A,B:Monoid](fa: F[A])(f: A ⇒ B): B = fold(map(fa)(f)) ‡

  def fold[A:Monoid](fa: F[A]): A = foldMap(fa)(x ⇒ x)

  …
}
```

@philip_schwarz

‡ While in most cases it is possible for **Foldable**'s **foldMap** to be defined in terms of **map**, it is not always possible. **Traverse**'s **foldMap** can instead always be defined in terms of **map**.

Note that **Traverse** 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.



FP in Scala

```scala
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]] = sequence(map(fa)(f))

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

  override def foldMap[A,B:Monoid](fa: F[A])(f: A ⇒ B): B = fold(map(fa)(f))

  …
}
```