a sighting of

traverse_

in

SECOND EDITION

PRACTICAL FP IN SCALA

A HANDS-ON APPROACH

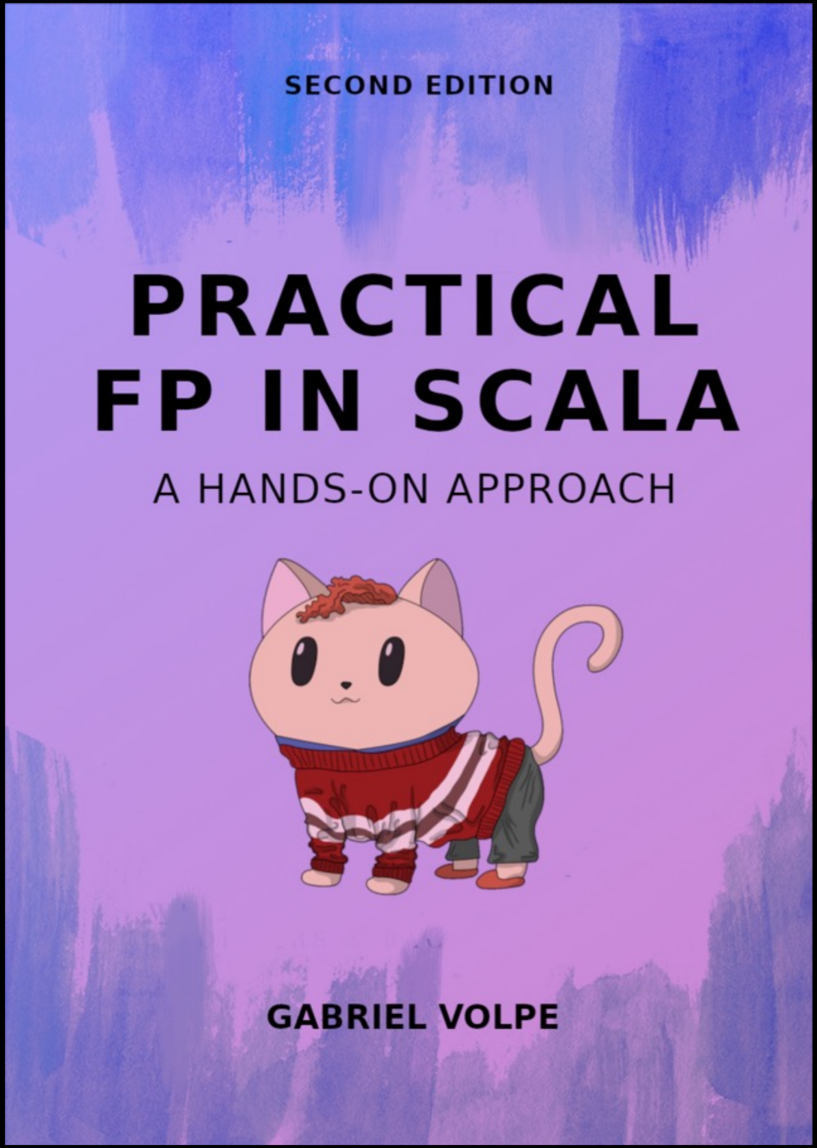
GABRIEL VOLPE

by

Gabrieλ Volpe
@volpegabriel87

slides by @philip_schwarz FP Iλλuminated http://fpilluminated.com/

```scala
trait ShoppingCart[F[_]] {
  def add(userId: UserId, itemId: ItemId, quantity: Quantity): F[Unit]
  def get(userId: UserId): F[CartTotal]
  def delete(userId: UserId): F[Unit]
  def removeItem(userId: UserId, itemId: ItemId): F[Unit]
  def update(userId: UserId, cart: Cart): F[Unit]
}

object ShoppingCart {
  def make[F[_]: GenUUID: MonadThrow](
    items: Items[F],
    redis: RedisCommands[F, String, String],
    exp: ShoppingCartExpiration
  ): ShoppingCart[F] = new ShoppingCart[F] {

    …

    override def update(userId: UserId, cart: Cart): F[Unit] =
      redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap =>
        itemIdToQuantityMap.toList.traverse_ { case (itemId, _) =>
          ID.read[F, ItemId](itemId).flatMap { id =>
            cart.items.get(id).traverse_ { quantity =>
              redis.hSet(userId.show, itemId, quantity.show)
            }
          }
        }
      } *> redis.expire(userId.show, exp.value).void

    …

  }
}
```



gvolpe / pfps-shopping-cart

pfps-shopping-cart / modules / core / src / main / scala / shop / services / ShoppingCart.scala

with some minor renaming, to ease comprehension for anyone lacking context – see repo for original

```scala
override def update(userId: UserId, cart: Cart): F[Unit] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap =>
    itemIdToQuantityMap.toList.traverse_ { case (itemId, _) =>
      ID.read[F, ItemId](itemId).flatMap { id =>
        cart.items.get(id).traverse_ { quantity =>
          redis.hSet(userId.show, itemId, quantity.show)
        }
      }
    }
  } *> redis.expire(userId.show, exp.value).void
```

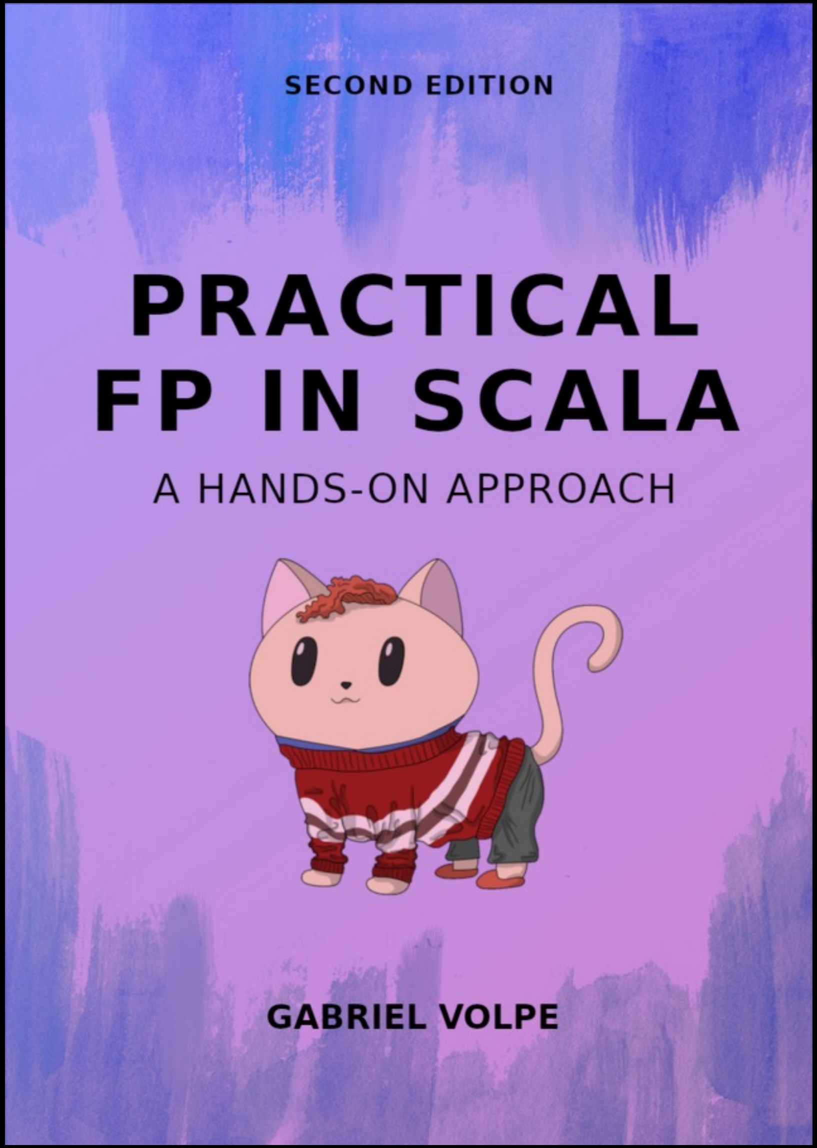
Option[Quantity]

F[Boolean]

^⇧P Type Info

```scala
override def update(userId: UserId, cart: Cart): F[Unit] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap =>
    itemIdToQuantityMap.toList.traverse_ { case (itemId, _) =>
      ID.read[F, ItemId](itemId).flatMap { id =>
        cart.items.get(id).traverse_ { quantity =>
          redis.hSet(userId.show, itemId, quantity.show)
        }
      }
    }
  } *> redis.expire(userId.show, exp.value).void
```

F[Unit]



gvolpe / pfps-shopping-cart

pfps-shopping-cart / modules / core / src / main / scala / shop / services / ShoppingCart.scala

with some minor renaming, to ease comprehension for anyone lacking context – see repo for original

```scala
override def update(userId: UserId, cart: Cart): F[Unit] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap =>
    itemIdToQuantityMap.toList.traverse_ { case (itemId, _) =>
      ID.read[F, ItemId](itemId).flatMap { id =>
        cart.items.get(id).traverse_ { quantity =>
          redis.hSet(userId.show, itemId, quantity.show)
        }
      }
    }
  } *> redis.expire(userId.show, exp.value).void
```

Option[Quantity]

F[Boolean]

^⇧P Type Info

```scala
override def update(userId: UserId, cart: Cart): F[Unit] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap =>
    itemIdToQuantityMap.toList.traverse_ { case (itemId, _) =>
      ID.read[F, ItemId](itemId).flatMap { id =>
        cart.items.get(id).traverse_ { quantity =>
          redis.hSet(userId.show, itemId, quantity.show)
        }
      }
    }
  } *> redis.expire(userId.show, exp.value).void
```

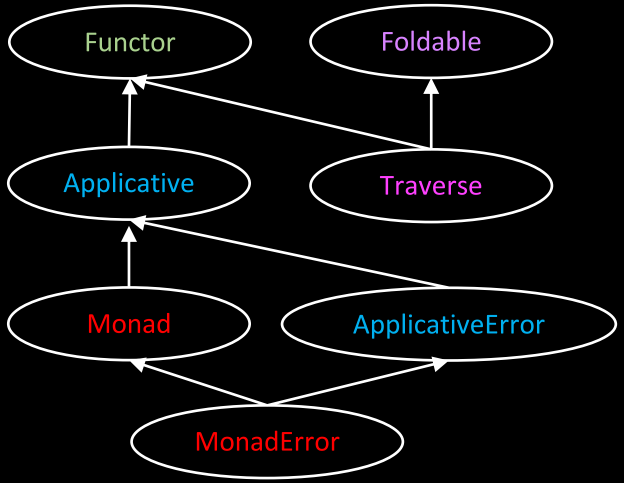
F[Unit]

```scala
def make[F[_]: GenUUID: MonadThrow]
```

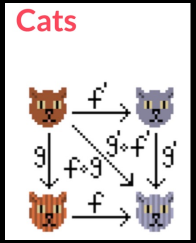
```scala
Option[Quantity] => (Quantity => F[Boolean]) => F[Unit]
```

```scala
@typeclass(excludeParents = List("FoldableNFunctions"))
trait Foldable[F[_]] extends UnorderedFoldable[F] with FoldableNFunctions[F] {
  …
  Traverse F[A] using Applicative[G]. A values will be mapped into G[B] and combined using Applicative#map2.
  This method is primarily useful when G[_] represents an action or effect, and the specific A aspect of
  G[A] is not otherwise needed.
def traverse_[G[_], A, B](fa: F[A])(f: A => G[B])(implicit G: Applicative[G]): G[Unit] =
```

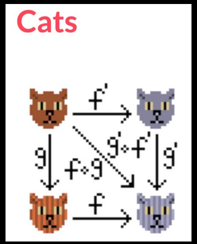
```scala
An applicative that also allows you to raise and or handle an error value.
This type class allows one to abstract over error-handling applicatives.
trait ApplicativeError[F[_], E] extends Applicative[F] { …

This type class allows one to abstract over error-handling monads.
trait MonadError[F[_], E] extends ApplicativeError[F, E] with Monad[F] { …

type MonadThrow[F[_]] = MonadError[F, Throwable]
```

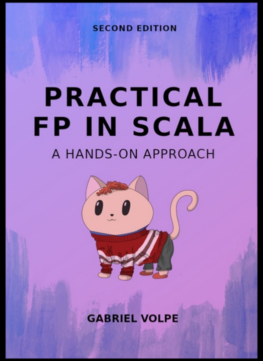


$$\begin{tikzcd}
\text{Functor} & \text{Foldable} \\
\text{Applicative} \arrow[u] & \text{Traverse} \arrow[ul] \arrow[u] \\
\text{Monad} \arrow[u] & \text{ApplicativeError} \arrow[ul] \\
\text{MonadError} \arrow[u] \arrow[ur] &
\end{tikzcd}$$



```
scala> import cats.implicits._, cats.effect.IO, cats.effect.unsafe.implicits.global

scala> :type Option("snap").traverse_(IO.println)
cats.effect.IO[Unit]

scala> Option("snap").traverse_(IO.println).unsafeRunSync
snap
```





```
override def update(userId: UserId, cart: Cart): F[Unit] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap =>
    itemIdToQuantityMap.toList.traverse_ { case (itemId, _) =>
      ID.read[F, ItemId](itemId).flatMap { id =>
        cart.items.get(id).traverse_ { quantity =>
          redis.hSet(userId.show, itemId, quantity.show)
        }
      }
    }
  } *> redis.expire(userId.show, exp.value).void
```

List[(String, String)]

F[Unit]

^⇧P Type Info

```
override def update(userId: UserId, cart: Cart): F[Unit] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap =>
    itemIdToQuantityMap.toList.traverse_ { case (itemId, _) =>
      ID.read[F, ItemId](itemId).flatMap { id =>
        cart.items.get(id).traverse_ { quantity =>
          redis.hSet(userId.show, itemId, quantity.show)
        }
      }
    }
  } *> redis.expire(userId.show, exp.value).void
```

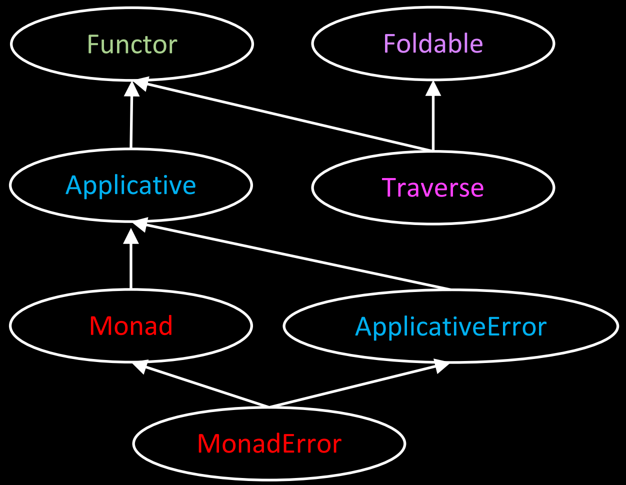
F[Unit]

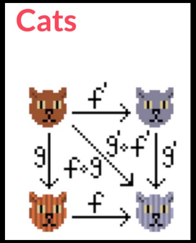
```
def make[F[_]: GenUUID: MonadThrow]
```

```
List[(String,String)] => ((String,String) => F[Unit]) => F[Unit]
```

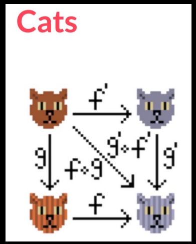
```
@typeclass(excludeParents = List("FoldableNFunctions"))
trait Foldable[F[_]] extends UnorderedFoldable[F] with FoldableNFunctions[F] {
  …
  Traverse F[A] using Applicative[G]. A values will be mapped into G[B] and combined using Applicative#map2.
  This method is primarily useful when G[_] represents an action or effect, and the specific A aspect of
  G[A] is not otherwise needed.
def traverse_[G[_], A, B](fa: F[A])(f: A => G[B])(implicit G: Applicative[G]): G[Unit] =
```

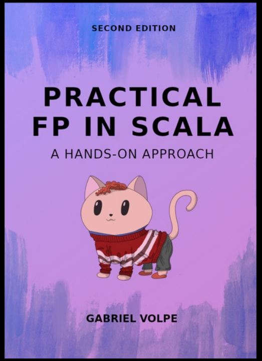
```
An applicative that also allows you to raise and or handle an error value.
This type class allows one to abstract over error-handling applicatives.
trait ApplicativeError[F[_], E] extends Applicative[F] { …

This type class allows one to abstract over error-handling monads.
trait MonadError[F[_], E] extends ApplicativeError[F, E] with Monad[F] { …

type MonadThrow[F[_]] = MonadError[F, Throwable]
```


$$\begin{tikzcd}
\text{Functor} & \text{Foldable} \\
\text{Applicative} \arrow[u] & \text{Traverse} \arrow[u] \arrow[ul] \\
\text{Monad} \arrow[u] & \text{ApplicativeError} \arrow[ul] \\
& \text{MonadError} \arrow[ul] \arrow[u]
\end{tikzcd}$$



```
scala> import cats.implicits._, cats.effect.IO, cats.effect.unsafe.implicits.global

scala> :type List("snap", "crackle", "pop").traverse_(IO.println)
cats.effect.IO[Unit]

scala> List("snap", "crackle", "pop").traverse_(IO.println).unsafeRunSync
snap
crackle
pop
```




SECOND EDITION
PRACTICAL FP IN SCALA
A HANDS-ON APPROACH
GABRIEL VOLPE