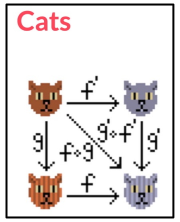


a sighting of traverseFilter and foldMap in

by

Gabrieλ Volpe
@volpegabriel87

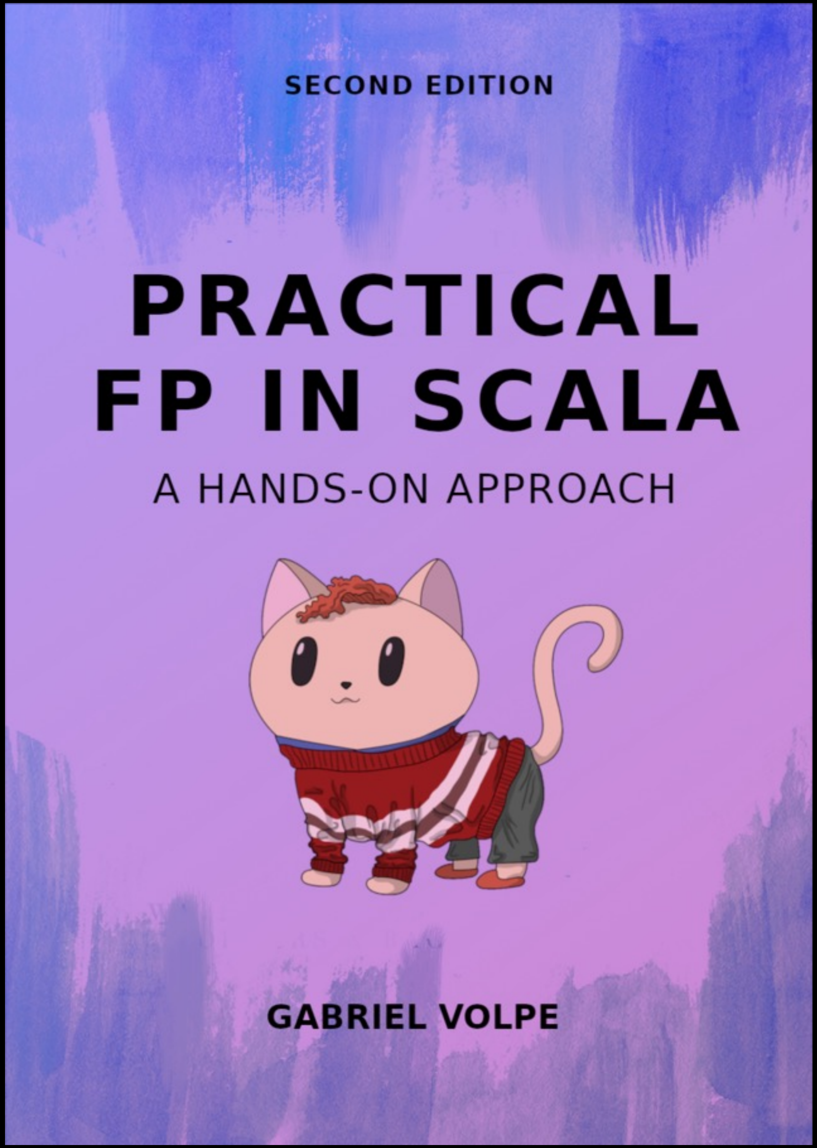
slides by

@philip_schwarz

FP Iλλuminated

http://fpilluminated.com/

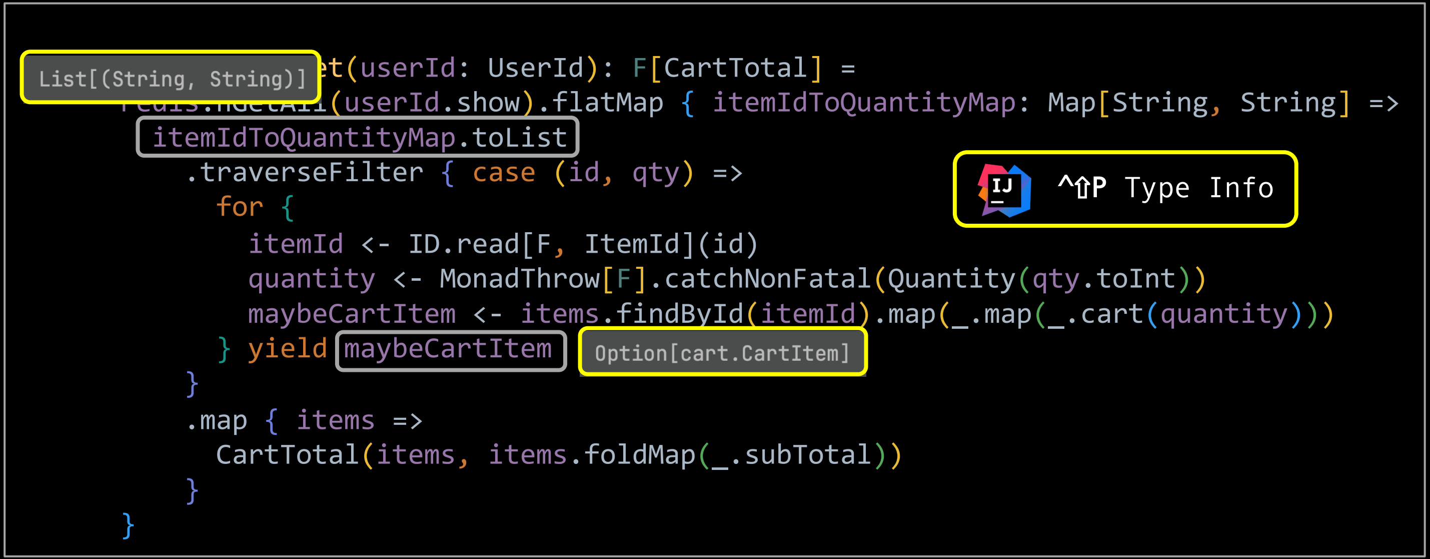
```scala
trait ShoppingCart[F[_]] {
  def add(userId: UserId, itemId: ItemId, quantity: Quantity): F[Unit]
  def get(userId: UserId): F[CartTotal]
  def delete(userId: UserId): F[Unit]
  def removeItem(userId: UserId, itemId: ItemId): F[Unit]
  def update(userId: UserId, cart: Cart): F[Unit]
}

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

    override def add(userId: UserId, itemId: ItemId, quantity: Quantity): F[Unit] =
      redis.hSet(userId.show, itemId.show, quantity.show) *>
        redis.expire(userId.show, exp.value).void

    override def get(userId: UserId): F[CartTotal] =
      redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap: Map[String, String] =>
        itemIdToQuantityMap.toList
          .traverseFilter { case (id, qty) =>
            for {
              itemId <- ID.read[F, ItemId](id)
              quantity <- MonadThrow[F].catchNonFatal(Quantity(qty.toInt))
              maybeCartItem <- items.findById(itemId).map(_.map(_.cart(quantity)))
            } yield maybeCartItem
          }
          .map { items =>
            CartTotal(items, items.foldMap(_.subTotal))
          }
      }
    …
  }
}
```



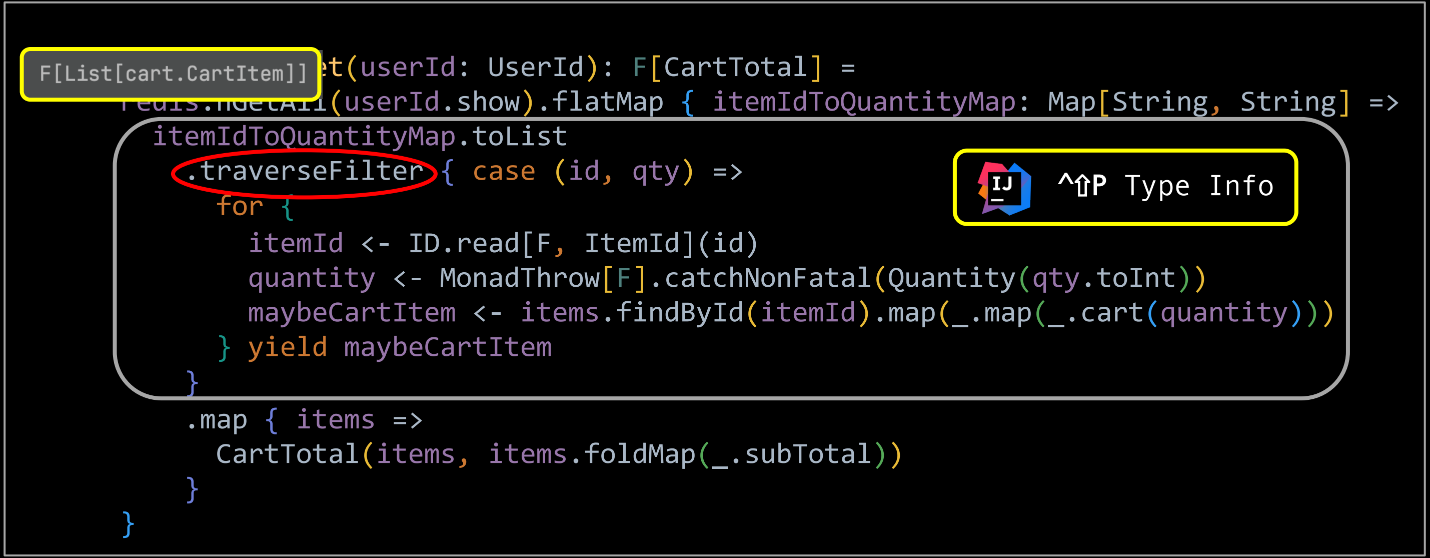
gvolpe / pfps-shopping-cart

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

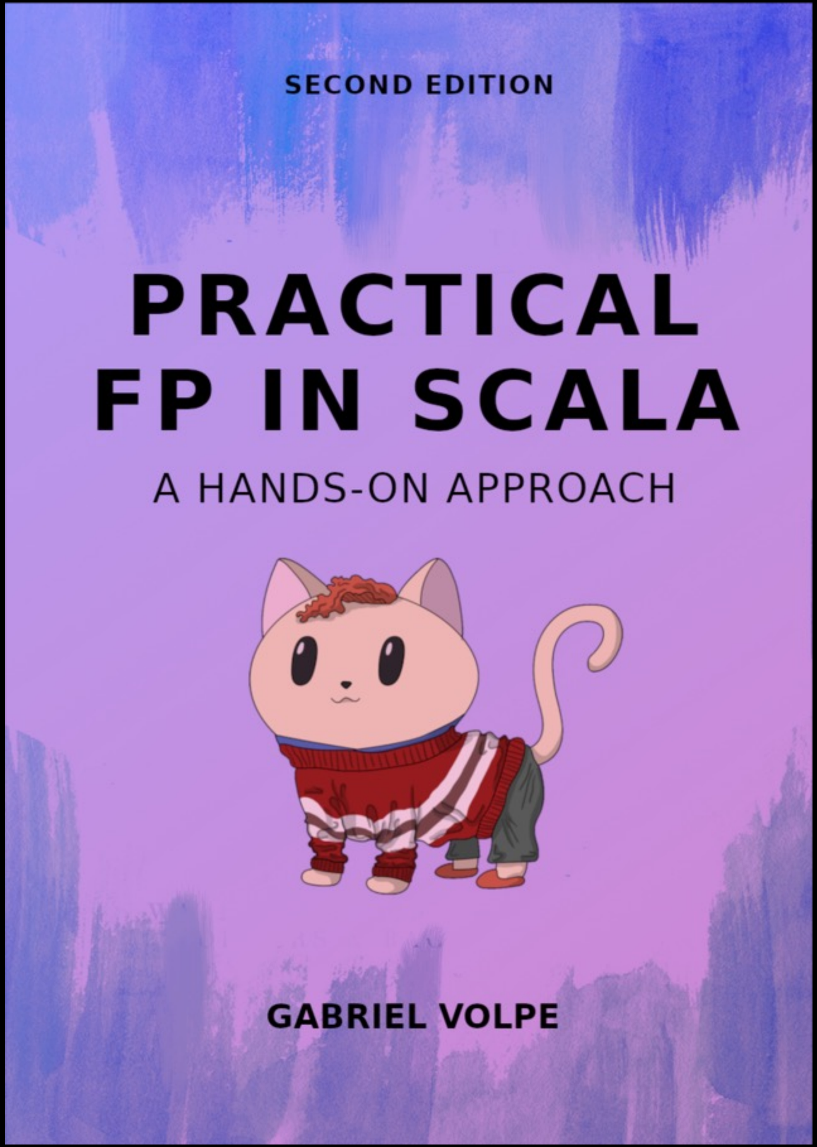
with some minor renaming, to ease comprehension for anyone lacking context – see repo for original



```scala
                t(userId: UserId): F[CartTotal] =
    redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap: Map[String, String] =>
      itemIdToQuantityMap.toList
        .traverseFilter { case (id, qty) =>
          for {
            itemId <- ID.read[F, ItemId](id)
            quantity <- MonadThrow[F].catchNonFatal(Quantity(qty.toInt))
            maybeCartItem <- items.findById(itemId).map(_.map(_.cart(quantity)))
          } yield maybeCartItem
        }
        .map { items =>
          CartTotal(items, items.foldMap(_.subTotal))
        }
    }
```



```scala
                t(userId: UserId): F[CartTotal] =
    redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap: Map[String, String] =>
      itemIdToQuantityMap.toList
        .traverseFilter { case (id, qty) =>
          for {
            itemId <- ID.read[F, ItemId](id)
            quantity <- MonadThrow[F].catchNonFatal(Quantity(qty.toInt))
            maybeCartItem <- items.findById(itemId).map(_.map(_.cart(quantity)))
          } yield maybeCartItem
        }
        .map { items =>
          CartTotal(items, items.foldMap(_.subTotal))
        }
    }
```



SECOND EDITION

PRACTICAL FP IN SCALA

A HANDS-ON APPROACH

GABRIEL VOLPE

```
List[(String, String)] et(userId: UserId): F[CartTotal] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap: Map[String, String] =>
    itemIdToQuantityMap.toList
      .traverseFilter { case (id, qty) =>
        for {
          itemId <- ID.read[F, ItemId](id)
          quantity <- MonadThrow[F].catchNonFatal(Quantity(qty.toInt))
          maybeCartItem <- items.findById(itemId).map(_.map(_.cart(quantity)))
        } yield maybeCartItem   Option[cart.CartItem]
      }
      .map { items: List[cart.CartItem] =>
        CartTotal(items, items.foldMap(_.subTotal))
      }
  }
```

^⇧P Type Info

```
F[List[cart.CartItem]] et(userId: UserId): F[CartTotal] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap: Map[String, String] =>
    itemIdToQuantityMap.toList
      .traverseFilter { case (id, qty) =>
        for {
          itemId <- ID.read[F, ItemId](id)
          quantity <- MonadThrow[F].catchNonFatal(Quantity(qty.toInt))
          maybeCartItem <- items.findById(itemId).map(_.map(_.cart(quantity)))
        } yield maybeCartItem
      }
      .map { items: List[cart.CartItem] =>
        CartTotal(items, items.foldMap(_.subTotal))
      }
  }
```

^⇧P Type Info

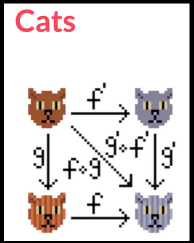
```
def make[F[_]: GenUUID: MonadThrow]
```

```
List[(String,String)] => ((String,String) => F[Option[CartItem]]) => F[List[CartItem]]
```

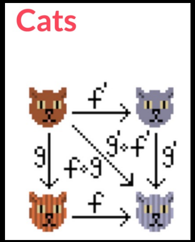
```
@typeclass trait TraverseFilter[F[_]] extends FunctorFilter[F] {
  …
  A combined traverse and filter. Filtering is handled via Option instead of Boolean such that the
  output type B can be different than the input type A.
  def traverseFilter[G[_], A, B](fa: F[A])(f: A => G[Option[B]])(implicit G: Applicative[G]): G[F[B]]
```

```
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]
```

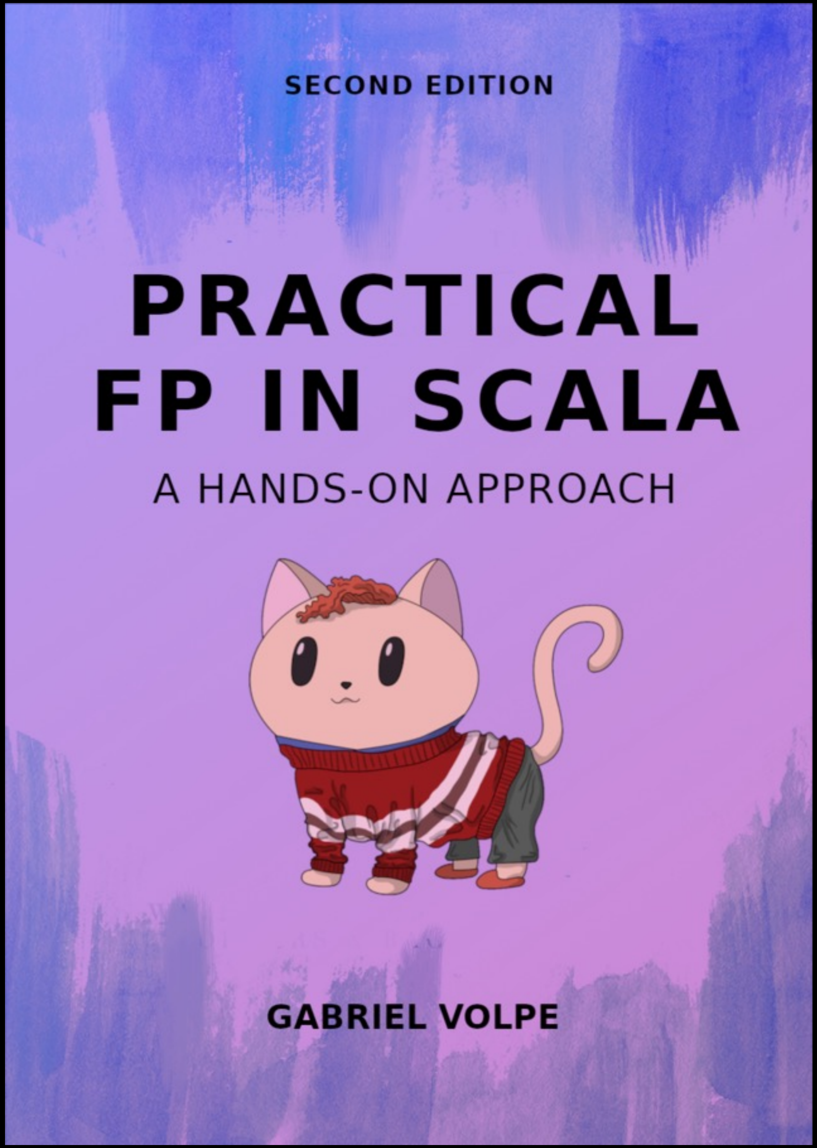
$$\begin{tikzcd}
\text{Functor} & \text{Foldable} & \text{FunctorFilter} \\
\text{Applicative} \arrow[u] & \text{Traverse} \arrow[ul] \arrow[u] & \text{TraverseFilter} \arrow[u] \\
\text{Monad} \arrow[u] & \text{ApplicativeError} \arrow[ul] & \\
& \text{MonadError} \arrow[ul] \arrow[u] &
\end{tikzcd}$$



```
FunctorFilter[F] allows you to map and filter out elements simultaneously.
@typeclass trait FunctorFilter[F[_]] extends Serializable

TraverseFilter, also known as Witherable, represents list-like structures
that can essentially have a traverse and a filter applied as a single
combined operation (traverseFilter).
@typeclass trait TraverseFilter[F[_]] extends FunctorFilter[F] {
```



```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 add(userId: UserId, itemId: ItemId, quantity: Quantity): F[Unit] =
      redis.hSet(userId.show, itemId.show, quantity.show) *>
        redis.expire(userId.show, exp.value).void

    override def get(userId: UserId): F[CartTotal] =
      redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap: Map[String, String] =>
        itemIdToQuantityMap.toList
          .traverseFilter { case (id, qty) =>
            for {
              itemId <- ID.read[F, ItemId](id)
              quantity <- MonadThrow[F].catchNonFatal(Quantity(qty.toInt))
              maybeCartItem <- items.findById(itemId).map(_.map(_.cart(quantity)))
            } yield maybeCartItem
          }
          .map { items: List[cart.CartItem] =>
            CartTotal(items, items.foldMap(_.subTotal))
          }
      }
    …
  }

}
```



```scala
override def get(userId: UserId): F[CartTotal] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap: Map[String, String] =>
    itemIdToQuantityMap.toList
      .traverseFilter { case (id, qty) =>
        for {
          itemId <- ID.read[F, ItemId](id)
          quantity <- MonadThrow[F].catchNonFatal(Quantity(qty.toInt))
          maybeCartItem <- items.findById(itemId).map(_.map(_.cart(quantity)))
    List[cart.CartItem] ybeCartItem
      }
      .map { items =>
        CartTotal(items, items.foldMap(_.subTotal))
      }
  }
```

```
Money
```

```
Implicit arguments:
moneyMonoid: cats.Monoid[Money]  (trait OrphanInstances)
```

```
(cart.CartItem => B) => cats.Monoid[B] => B
```

```
[B](f: cart.CartItem => B)(implicit B: cats.Monoid[B])
```

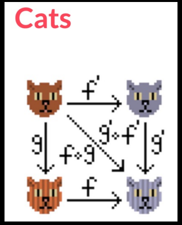
```
^⇧P Type Info
⌘P Parameter Info
⇧⌘P Show implicit arguments
```

squants

```scala
final class Money private
  (val amount: BigDecimal)
  (val currency: Currency)
  extends Quantity[Money] {
```

Cats

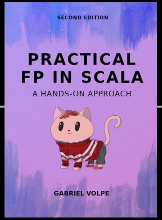
$$\begin{tikzcd}
\text{Semigroup} \\
\text{Monoid} \arrow[u]
\end{tikzcd}$$



```
A semigroup is any set A with an associative operation (combine).
trait Semigroup[@sp(Int, Long, Float, Double) A] extends Any with Serializable {

A monoid is a semigroup with an identity. A monoid is a specialization of a
semigroup, so its operation must be associative. Additionally, combine(x,
empty) == combine(empty, x) == x. For example, if we have Monoid[String],
with combine as string concatenation, then empty = "".
trait Monoid[@sp(Int, Long, Float, Double) A] extends Any with Semigroup[A] {
```

```scala
override def get(userId: UserId): F[CartTotal] =
  redis.hGetAll(userId.show).flatMap { itemIdToQuantityMap: Map[String, String] =>
    itemIdToQuantityMap.toList
      .traverseFilter { case (id, qty) =>
        for {
          itemId <- ID.read[F, ItemId](id)
          quantity <- MonadThrow[F].catchNonFatal(Quantity(qty.toInt))
          maybeCartItem <- items.findById(itemId).map(_.map(_.cart(quantity)))
        } yield maybeCartItem
      }
      .map { items =>
        CartTotal(items, items.foldMap(_.subTotal))(moneyMonoid)
      }
  }
```

```
List[CartItem] => (CartItem => Money) => Monoid[Money] => Money
```



```scala
implicit val moneyMonoid: Monoid[Money] =
  new Monoid[Money] {
    override def empty: Money = USD(0)
    override def combine(x: Money, y: Money): Money = x + y
  }

case class CartItem(item: Item, quantity: Quantity) {
  def subTotal: Money = USD(item.price.amount * quantity.value)
}
```

```scala
trait Foldable[F[_]] extends UnorderedFoldable[F] with FoldableNFunctions[F] {
 …
 Fold implemented by mapping A values into B and then
 combining them using the given Monoid[B] instance.
 def foldMap[A, B](fa: F[A])(f: A => B)(implicit B: Monoid[B]): B =
   foldLeft(fa, B.empty)((b, a) => B.combine(b, f(a)))
```

Cats