深圳scala-meetup-20180902

Monadic programming - Reader Monad and MonadTransformer for dependency injection and DataAccess Result type

@雪川大虫

cnblogs.com/tiger-xc/

blog.csdn.net/tiger_xc/

github.com/bayakala/

- Monadic programming style - F[?]

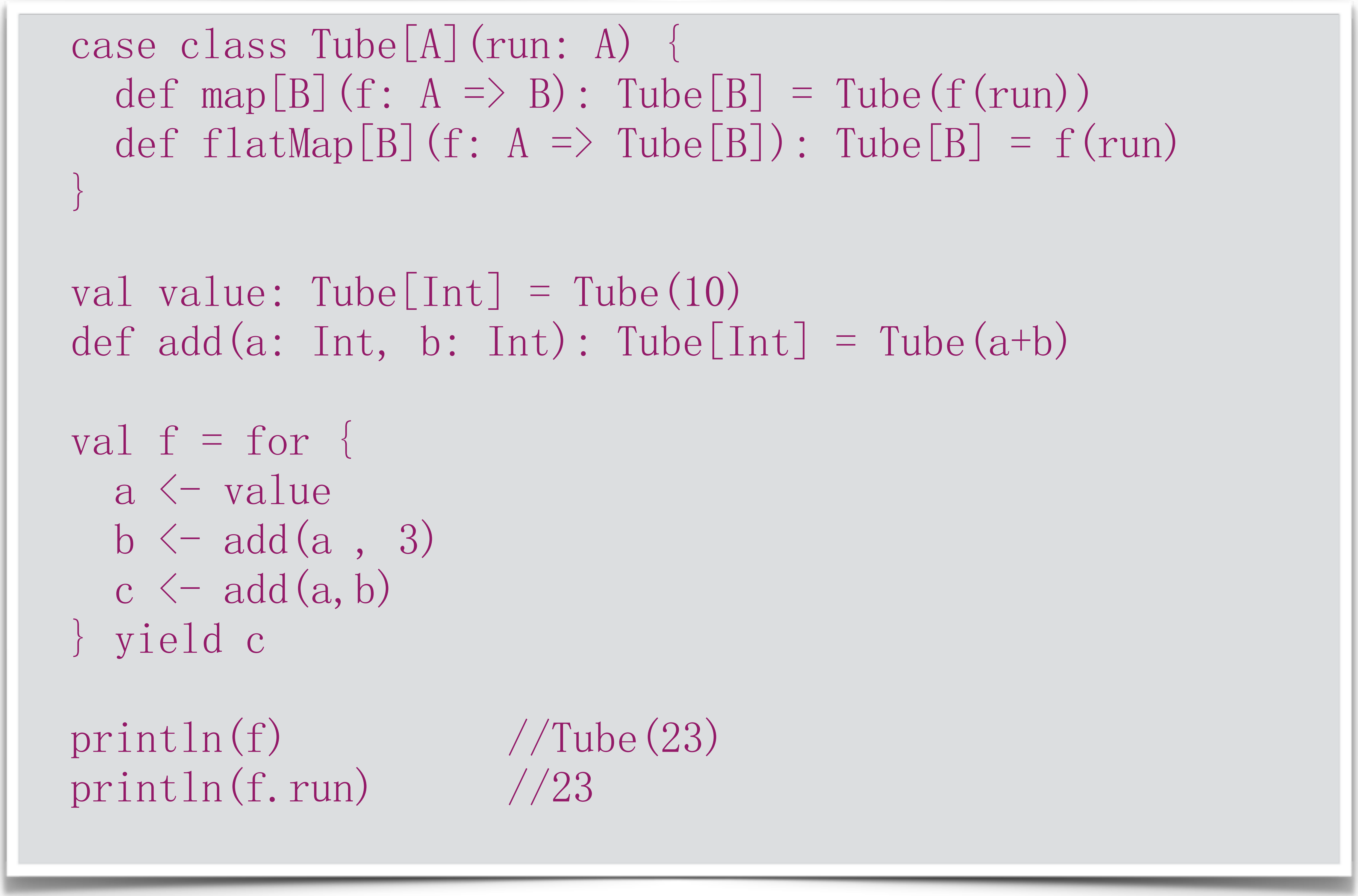
```
val sum: Option[Int] = Some(1).map( a => a + 2)   // Some(3)
```

```
行令编程模式（imperative programming)
def au(t:T): T      async update with result
val t2 = au(t1)
val t3 = au(t2)
val t4 = au(t2 + t3)           t4 = ???
```

```
monadic programming : program with monads
val fp3 = F[p1] ⊕ F[p1] ⊕ F[p1] = F[p1+p2+p3]
1、延迟运算 : val res = fp3.run
2、按序运算 : flatMap{a => flatMap{b => flatMap{c =>…
```

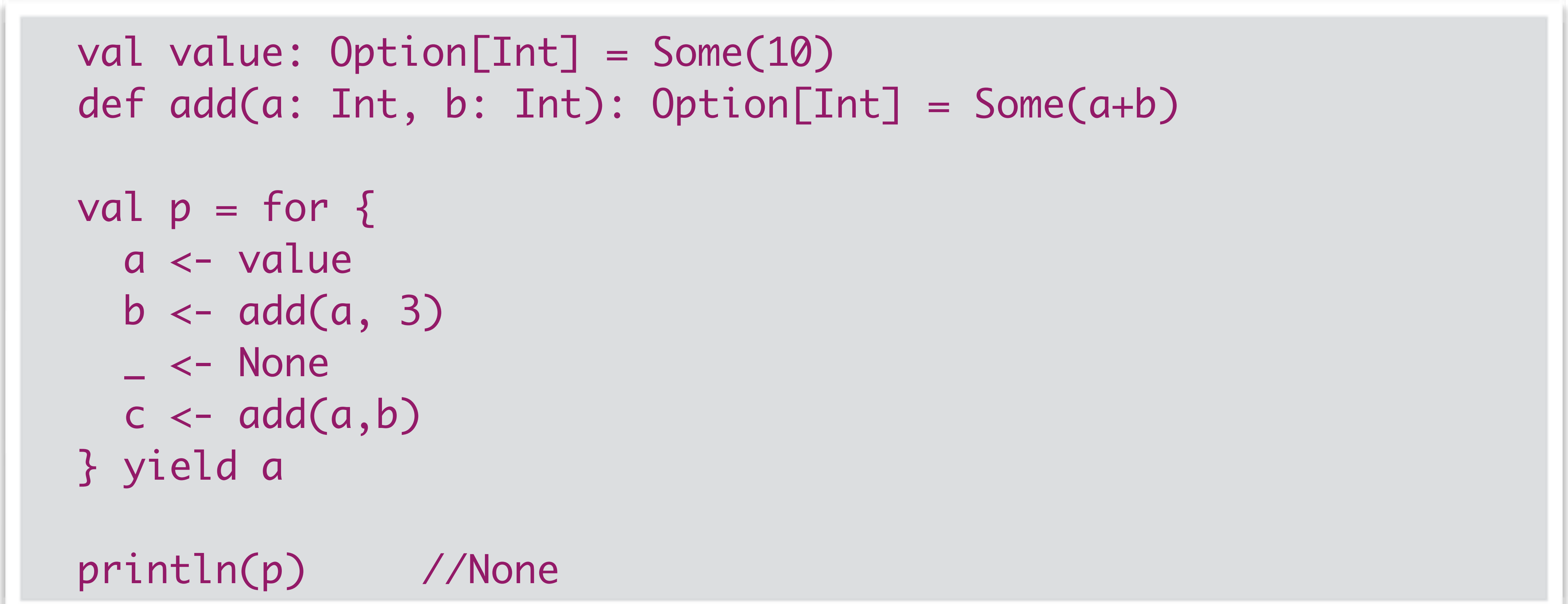
Functional programming is:
1、pure functions
2、function as first-class value

- 自制Monadic运算类型

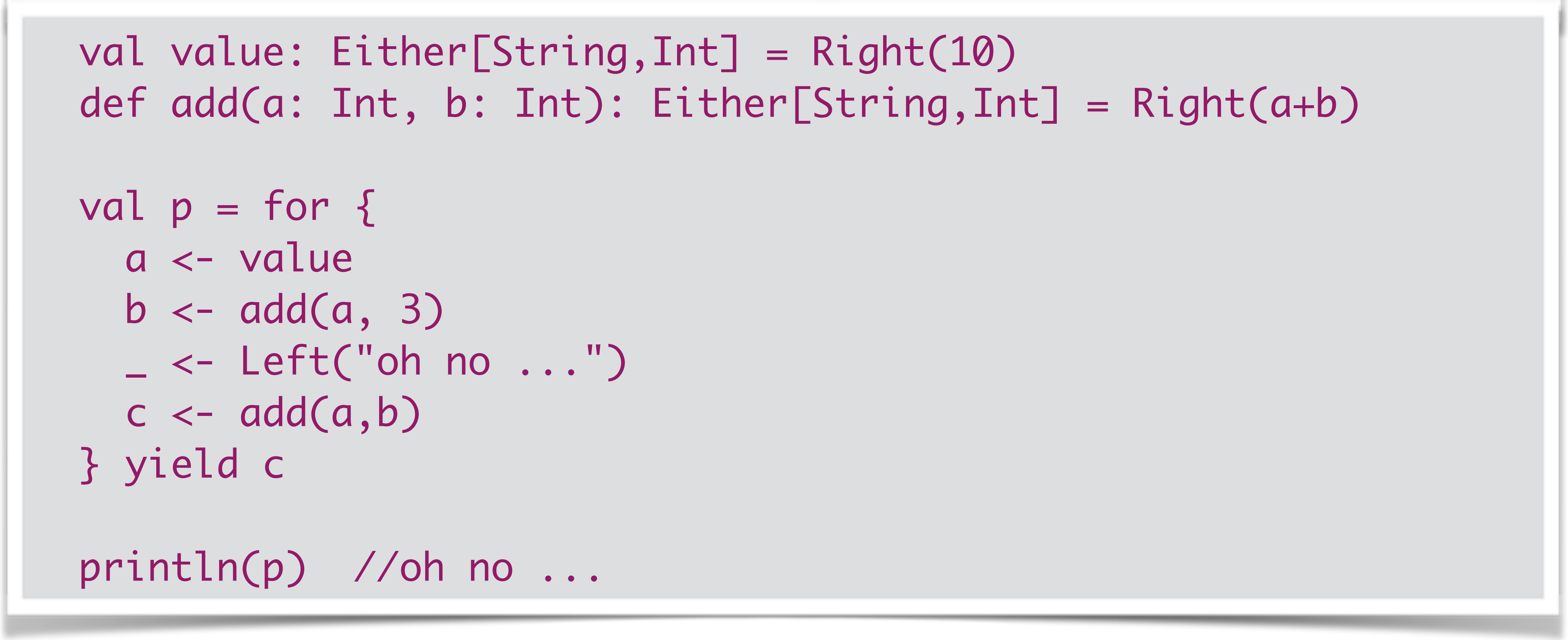


```
case class Tube[A](run: A) {
  def map[B](f: A => B): Tube[B] = Tube(f(run))
  def flatMap[B](f: A => Tube[B]): Tube[B] = f(run)
}

val value: Tube[Int] = Tube(10)
def add(a: Int, b: Int): Tube[Int] = Tube(a+b)

val f = for {
  a <- value
  b <- add(a , 3)
  c <- add(a,b)
} yield c

println(f)          //Tube(23)
println(f.run)      //23
```

- Monads: Option,Either示范

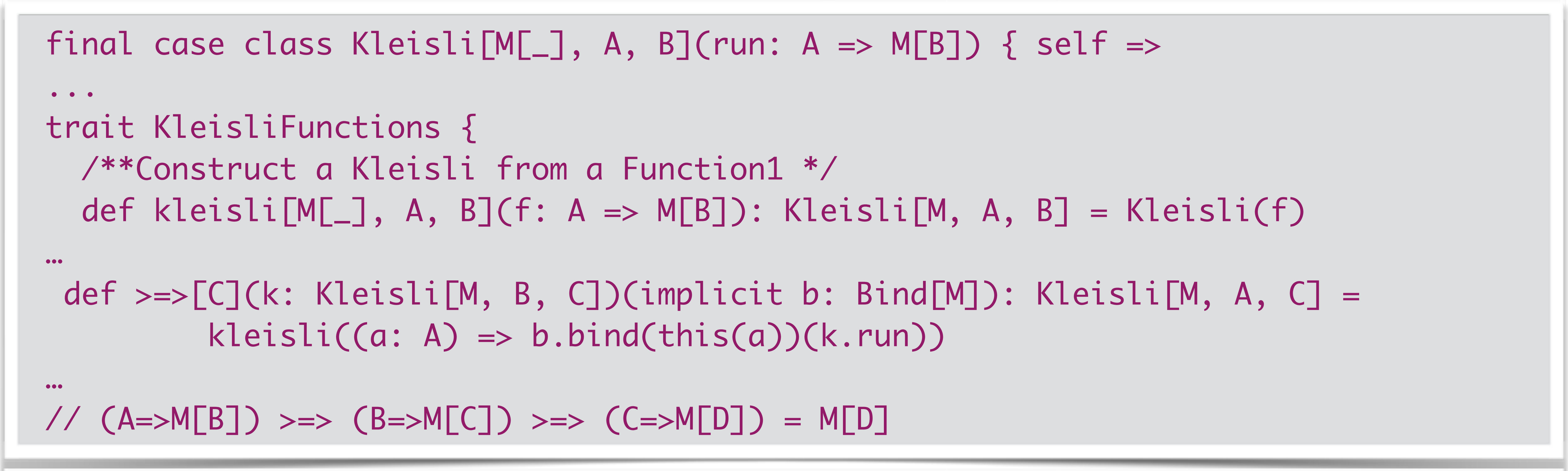


```
val value: Option[Int] = Some(10)
def add(a: Int, b: Int): Option[Int] = Some(a+b)

val p = for {
  a <- value
  b <- add(a, 3)
  _ <- None
  c <- add(a,b)
} yield a

println(p)     //None
```

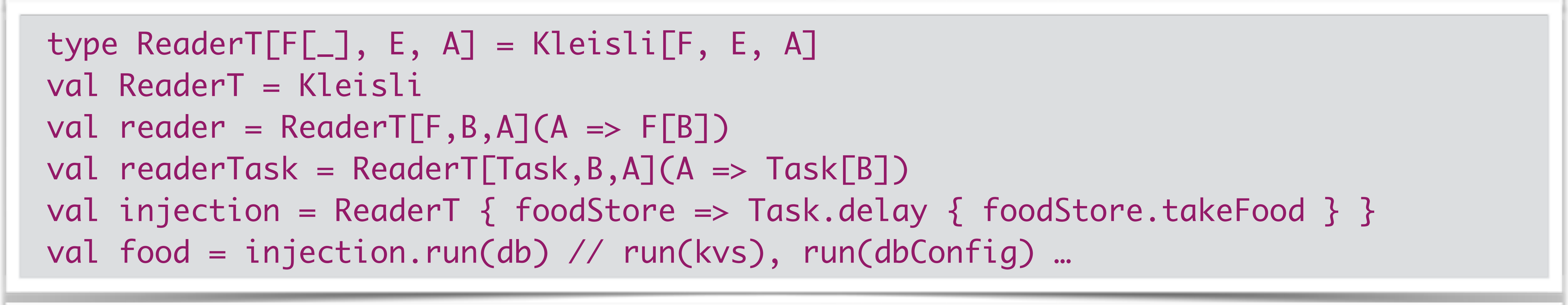


```
val value: Either[String,Int] = Right(10)
def add(a: Int, b: Int): Either[String,Int] = Right(a+b)

val p = for {
  a <- value
  b <- add(a, 3)
  _ <- Left("oh no ...")
  c <- add(a,b)
} yield c

println(p)  //oh no ...
```

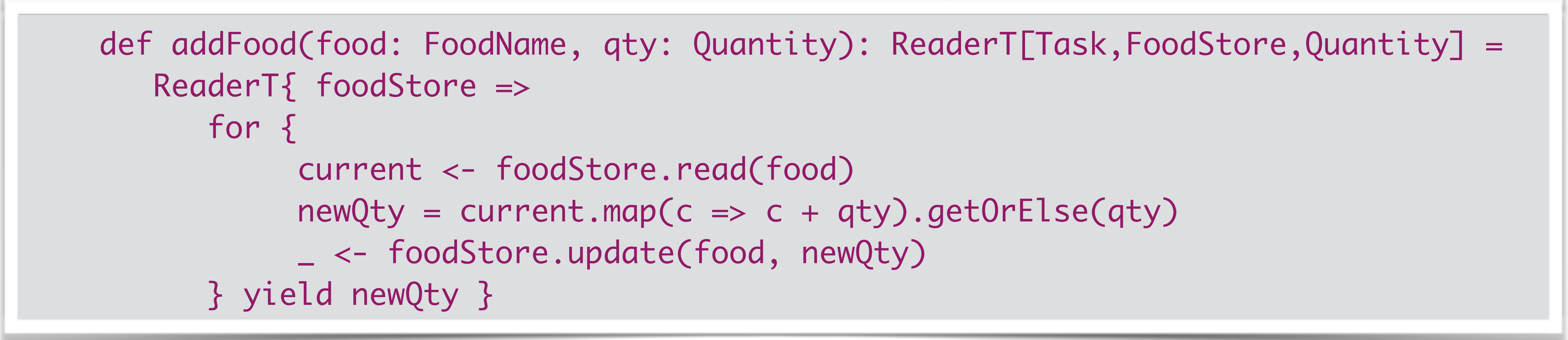
- Reader(Kleisli) for Dependency Injection



```
final case class Kleisli[M[_], A, B](run: A => M[B]) { self =>
...
trait KleisliFunctions {
  /**Construct a Kleisli from a Function1 */
  def kleisli[M[_], A, B](f: A => M[B]): Kleisli[M, A, B] = Kleisli(f)
…
 def >=>[C](k: Kleisli[M, B, C])(implicit b: Bind[M]): Kleisli[M, A, C] =
         kleisli((a: A) => b.bind(this(a))(k.run))
…
// (A=>M[B]) >=> (B=>M[C]) >=> (C=>M[D]) = M[D]
```

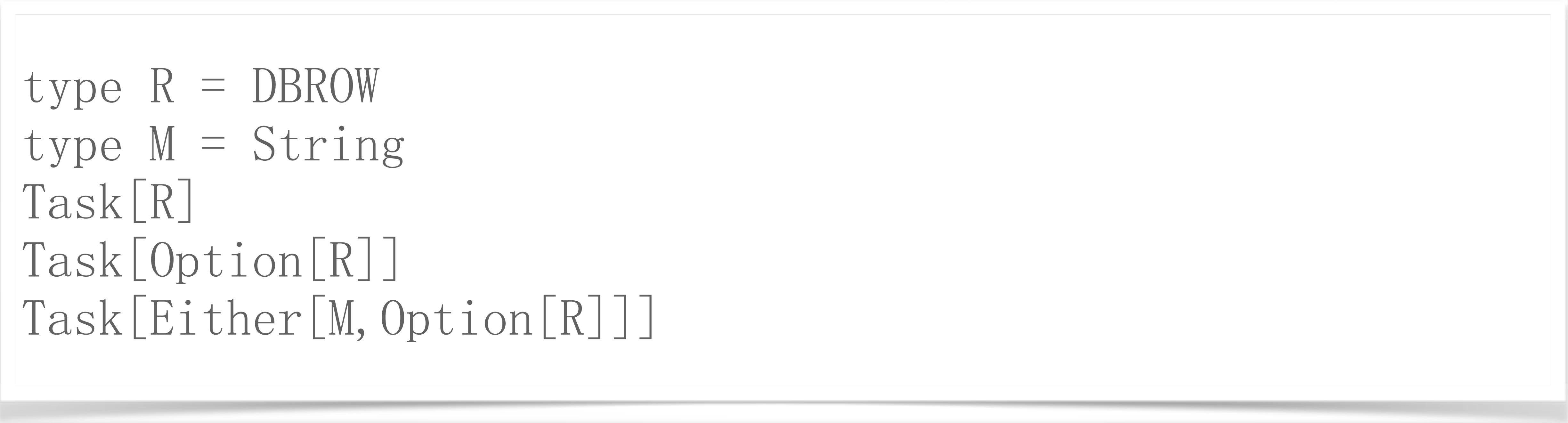


```
type ReaderT[F[_], E, A] = Kleisli[F, E, A]
val ReaderT = Kleisli
val reader = ReaderT[F,B,A](A => F[B])
val readerTask = ReaderT[Task,B,A](A => Task[B])
val injection = ReaderT { foodStore => Task.delay { foodStore.takeFood } }
val food = injection.run(db) // run(kvs), run(dbConfig) …
```

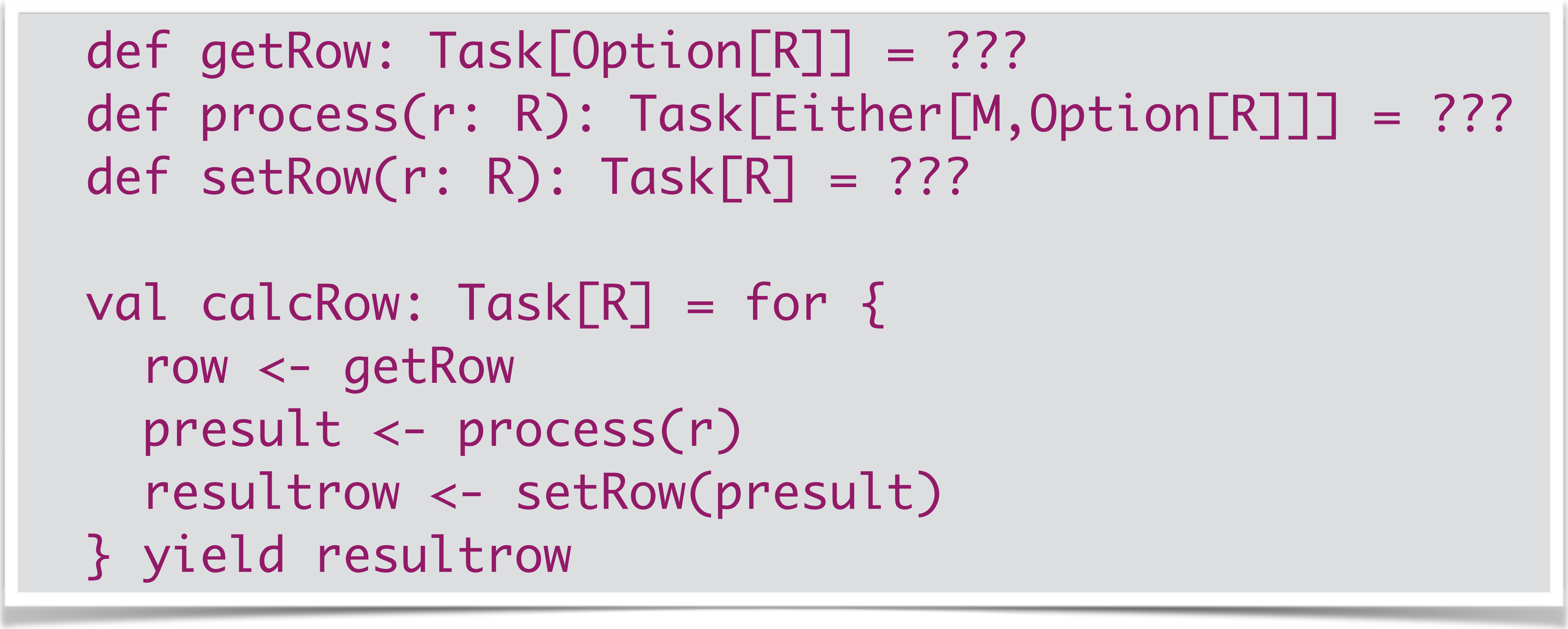


```
   def addFood(food: FoodName, qty: Quantity): ReaderT[Task,FoodStore,Quantity] =
      ReaderT{ foodStore =>
         for {
              current <- foodStore.read(food)
              newQty = current.map(c => c + qty).getOrElse(qty)
              _ <- foodStore.update(food, newQty)
         } yield newQty }
```

- Monad Transformer

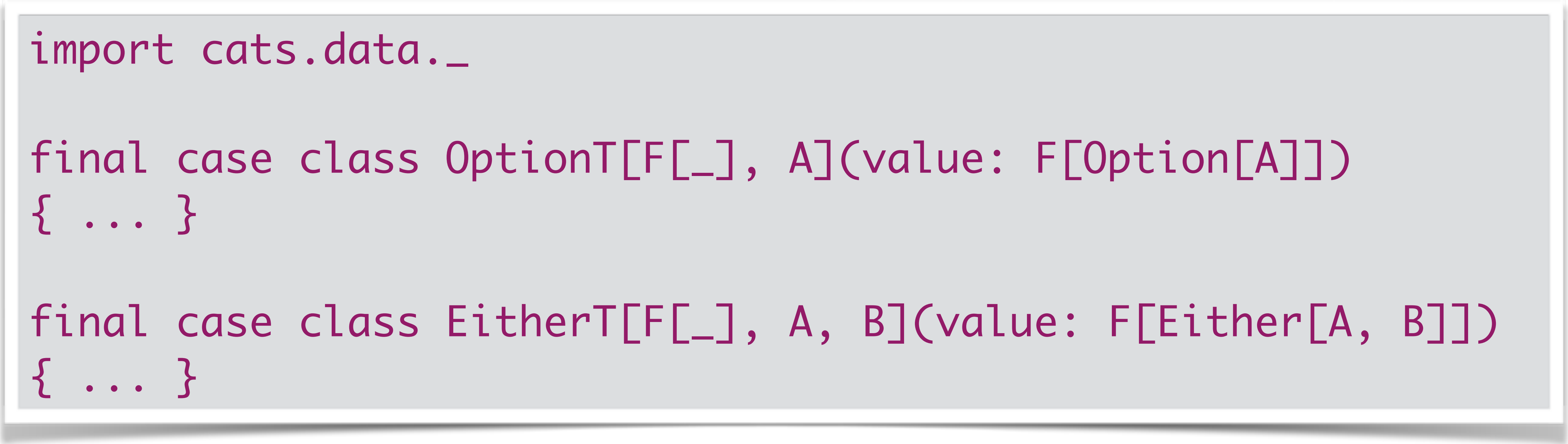


```
type R = DBROW
type M = String
Task[R]
Task[Option[R]]
Task[Either[M,Option[R]]]
```

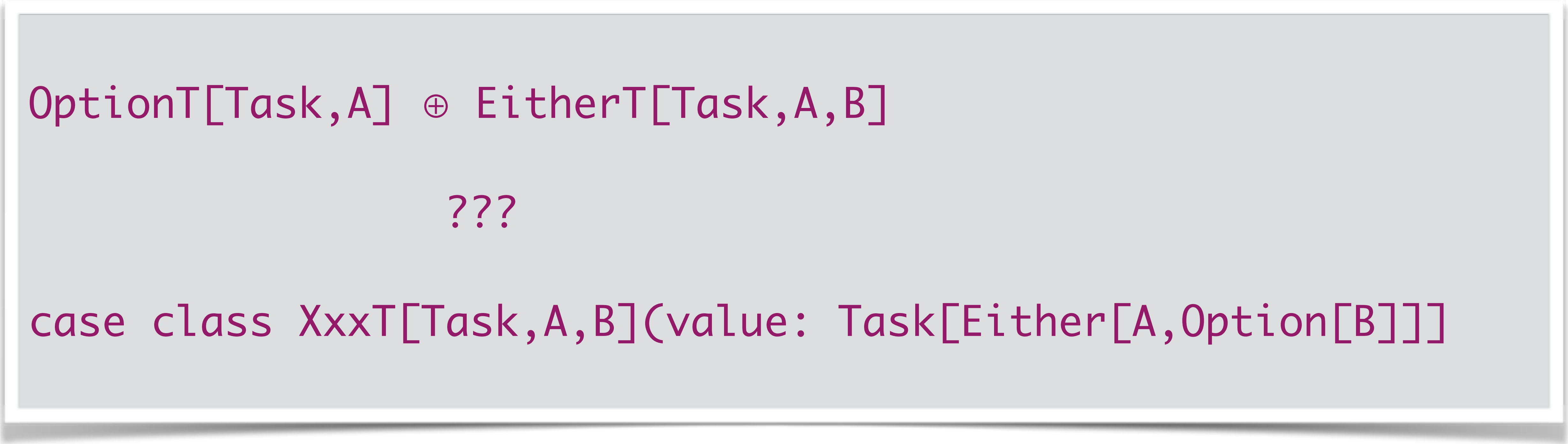


```scala
def getRow: Task[Option[R]] = ???
def process(r: R): Task[Either[M,Option[R]]] = ???
def setRow(r: R): Task[R] = ???

val calcRow: Task[R] = for {
  row <- getRow
  presult <- process(r)
  resultrow <- setRow(presult)
} yield resultrow
```

- Monad Transformer: OptionT, EitherT

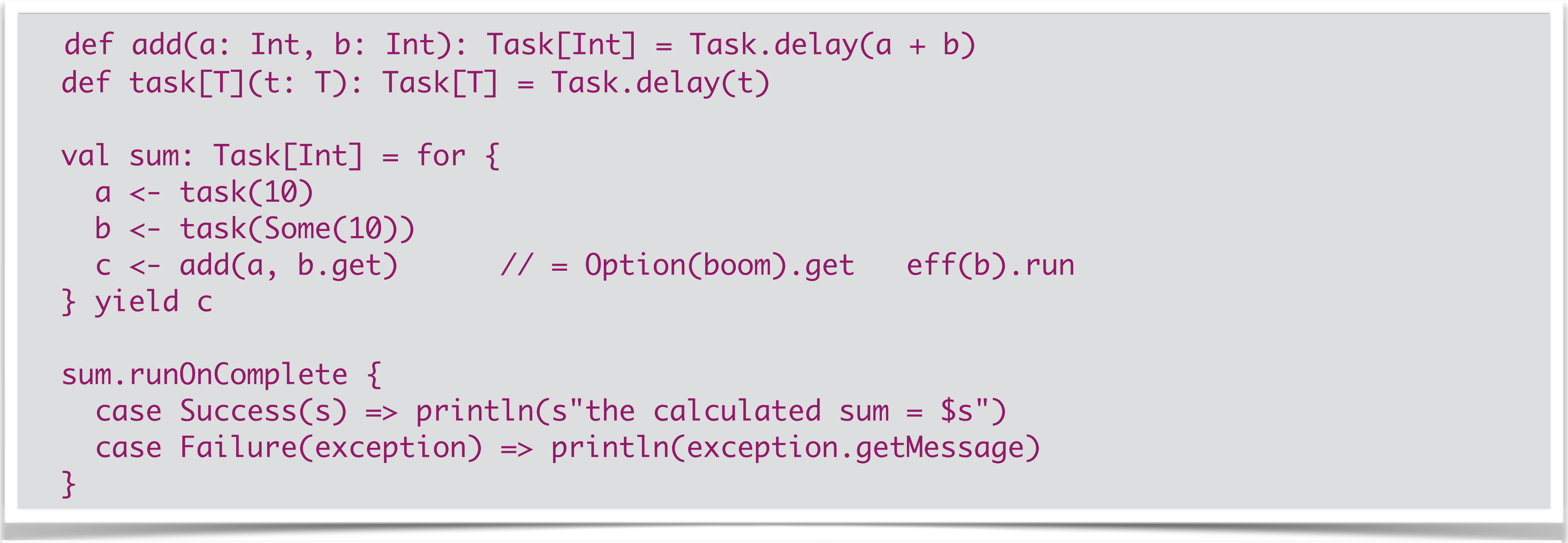


```
import cats.data._

final case class OptionT[F[_], A](value: F[Option[A]])
{ ... }

final case class EitherT[F[_], A, B](value: F[Either[A, B]])
{ ... }
```

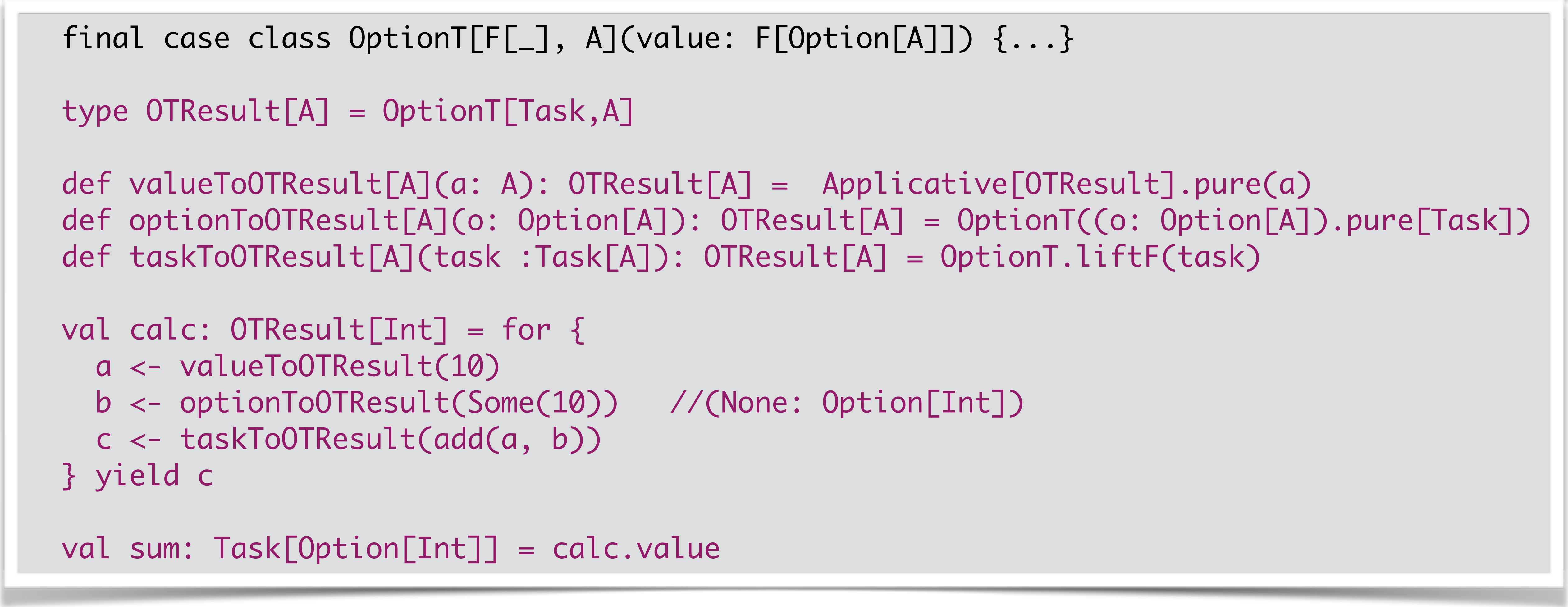


```
OptionT[Task,A] ⊕ EitherT[Task,A,B]

                 ???

case class XxxT[Task,A,B](value: Task[Either[A,Option[B]]]
```

- MonadTransformer: OptionT,EitherT示范

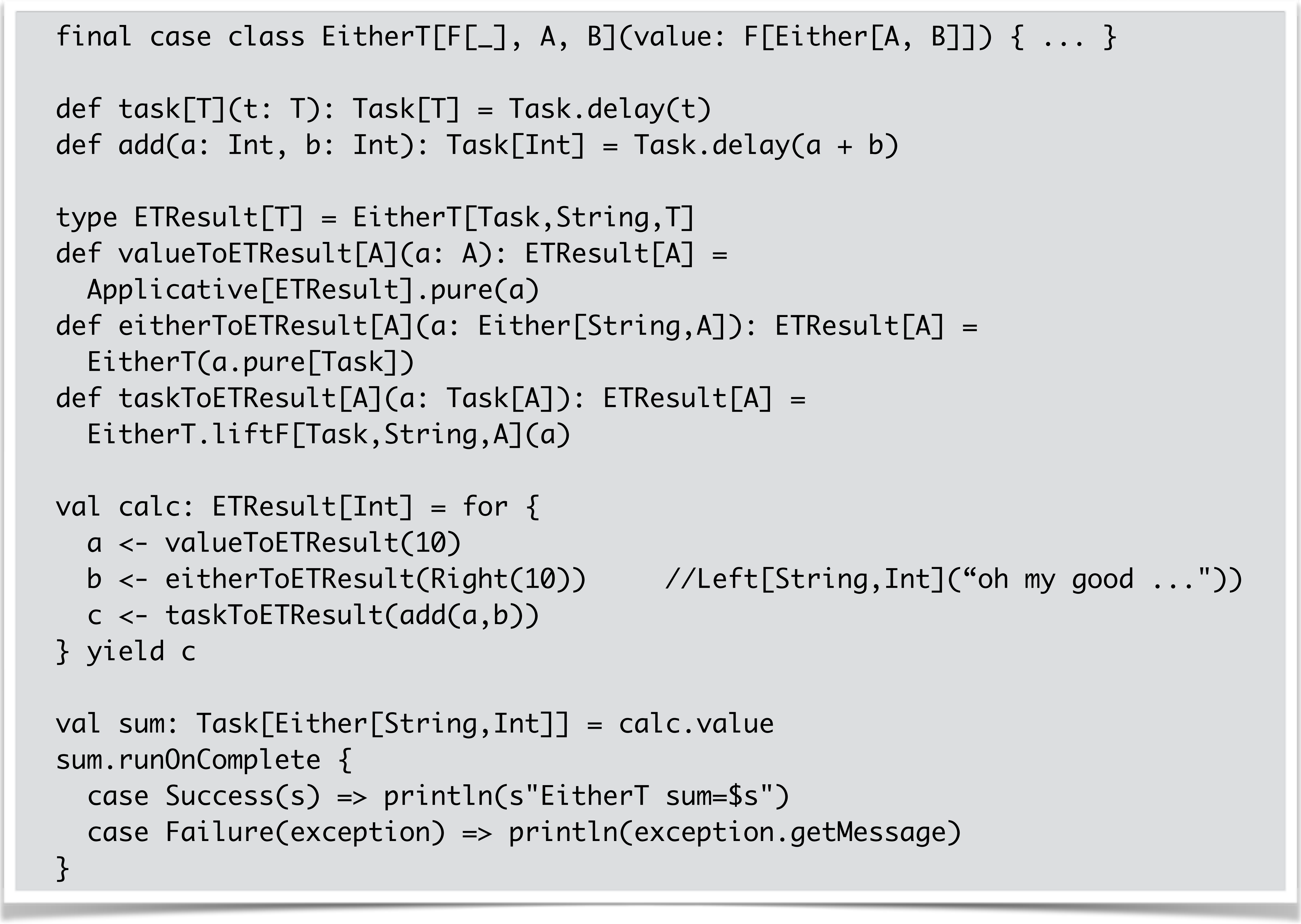


```
def add(a: Int, b: Int): Task[Int] = Task.delay(a + b)
def task[T](t: T): Task[T] = Task.delay(t)

val sum: Task[Int] = for {
  a <- task(10)
  b <- task(Some(10))
  c <- add(a, b.get)      // = Option(boom).get   eff(b).run
} yield c

sum.runOnComplete {
  case Success(s) => println(s"the calculated sum = $s")
  case Failure(exception) => println(exception.getMessage)
}
```



```
final case class OptionT[F[_], A](value: F[Option[A]]) {...}

type OTResult[A] = OptionT[Task,A]

def valueToOTResult[A](a: A): OTResult[A] =  Applicative[OTResult].pure(a)
def optionToOTResult[A](o: Option[A]): OTResult[A] = OptionT((o: Option[A]).pure[Task])
def taskToOTResult[A](task :Task[A]): OTResult[A] = OptionT.liftF(task)

val calc: OTResult[Int] = for {
  a <- valueToOTResult(10)
  b <- optionToOTResult(Some(10))   //(None: Option[Int])
  c <- taskToOTResult(add(a, b))
} yield c

val sum: Task[Option[Int]] = calc.value
```

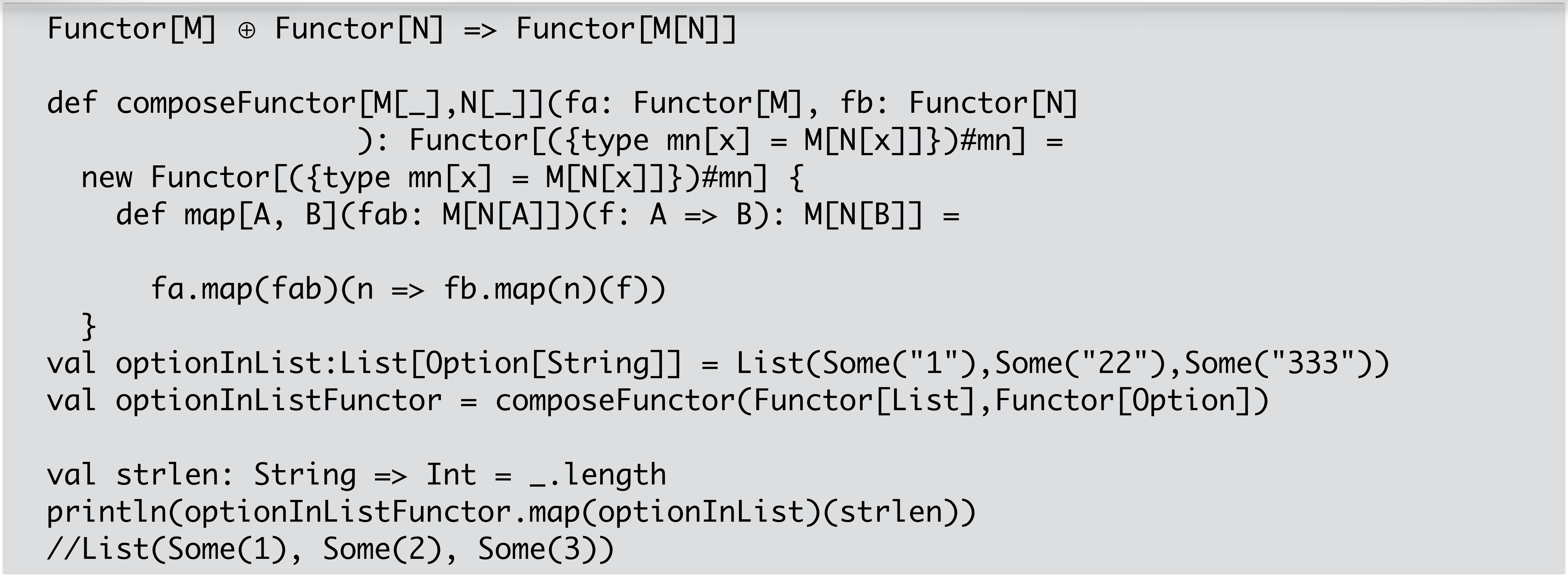
- MonadTransformer: OptionT,EitherT示范



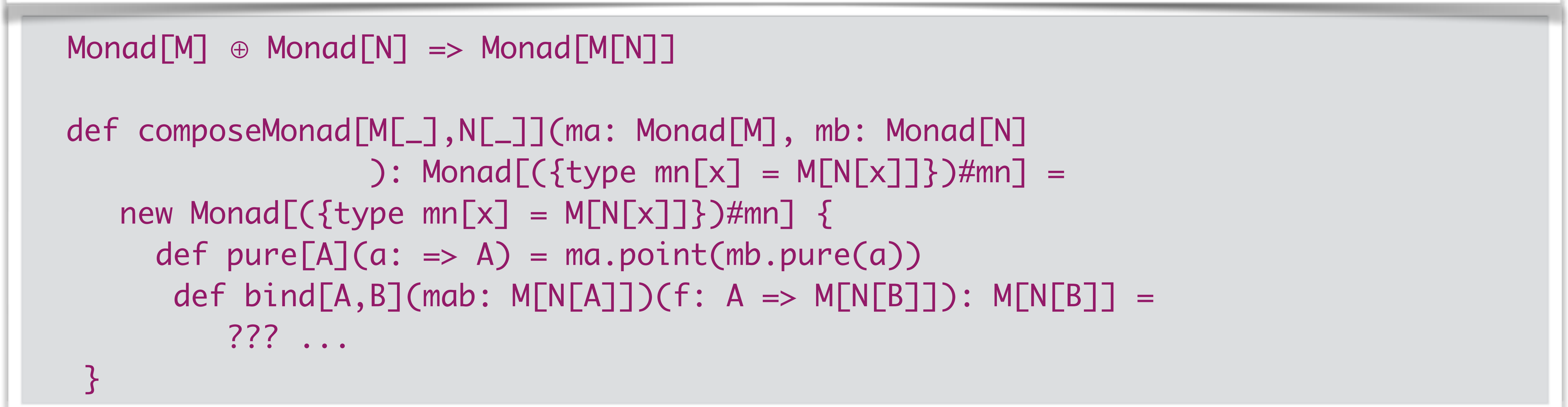
```
final case class EitherT[F[_], A, B](value: F[Either[A, B]]) { ... }

def task[T](t: T): Task[T] = Task.delay(t)
def add(a: Int, b: Int): Task[Int] = Task.delay(a + b)

type ETResult[T] = EitherT[Task,String,T]
def valueToETResult[A](a: A): ETResult[A] =
  Applicative[ETResult].pure(a)
def eitherToETResult[A](a: Either[String,A]): ETResult[A] =
  EitherT(a.pure[Task])
def taskToETResult[A](a: Task[A]): ETResult[A] =
  EitherT.liftF[Task,String,A](a)

val calc: ETResult[Int] = for {
  a <- valueToETResult(10)
  b <- eitherToETResult(Right(10))     //Left[String,Int](“oh my good ..."))
  c <- taskToETResult(add(a,b))
} yield c

val sum: Task[Either[String,Int]] = calc.value
sum.runOnComplete {
  case Success(s) => println(s"EitherT sum=$s")
  case Failure(exception) => println(exception.getMessage)
}
```

- Composing MonadTransformers - no, no, no!

```
val optEitherT = OptionT[Task,T] ⊕ EitherT[Task,String,T]
optEitherT.run = Task[Either[String,Option[T]]]
```

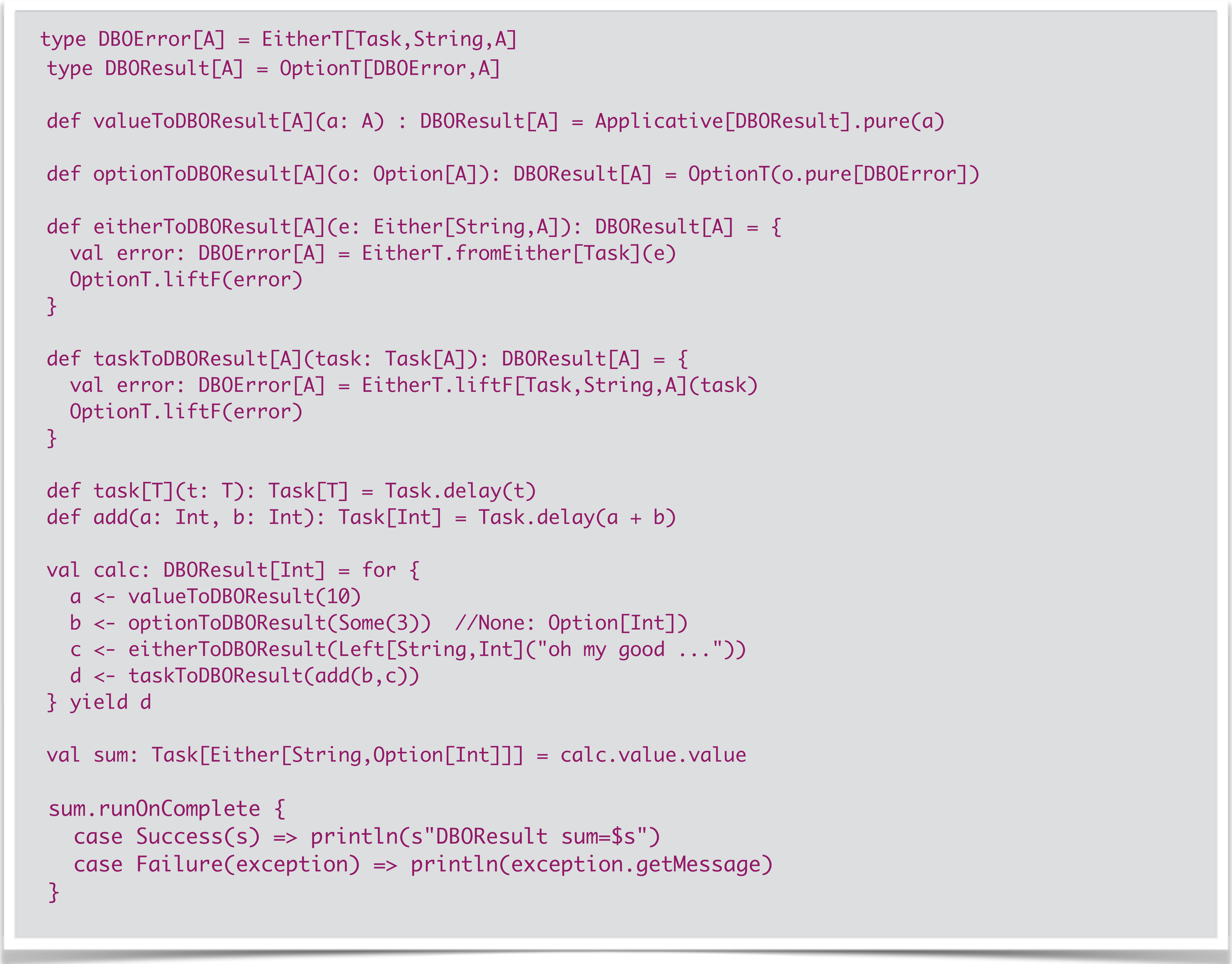


```
Functor[M] ⊕ Functor[N] => Functor[M[N]]

def composeFunctor[M[_],N[_]](fa: Functor[M], fb: Functor[N]
                  ): Functor[({type mn[x] = M[N[x]]})#mn] =
  new Functor[({type mn[x] = M[N[x]]})#mn] {
    def map[A, B](fab: M[N[A]])(f: A => B): M[N[B]] =

      fa.map(fab)(n => fb.map(n)(f))
  }
val optionInList:List[Option[String]] = List(Some("1"),Some("22"),Some("333"))
val optionInListFunctor = composeFunctor(Functor[List],Functor[Option])

val strlen: String => Int = _.length
println(optionInListFunctor.map(optionInList)(strlen))
//List(Some(1), Some(2), Some(3))
```



```
Monad[M] ⊕ Monad[N] => Monad[M[N]]

def composeMonad[M[_],N[_]](ma: Monad[M], mb: Monad[N]
                 ): Monad[({type mn[x] = M[N[x]]})#mn] =
   new Monad[({type mn[x] = M[N[x]]})#mn] {
     def pure[A](a: => A) = ma.point(mb.pure(a))
      def bind[A,B](mab: M[N[A]])(f: A => M[N[B]]): M[N[B]] =
         ??? ...
 }
```

- Combine MonadTransformers - embedding



```
type DBOError[A] = EitherT[Task,String,A]
type DBOResult[A] = OptionT[DBOError,A]

def valueToDBOResult[A](a: A) : DBOResult[A] = Applicative[DBOResult].pure(a)

def optionToDBOResult[A](o: Option[A]): DBOResult[A] = OptionT(o.pure[DBOError])

def eitherToDBOResult[A](e: Either[String,A]): DBOResult[A] = {
  val error: DBOError[A] = EitherT.fromEither[Task](e)
  OptionT.liftF(error)
}

def taskToDBOResult[A](task: Task[A]): DBOResult[A] = {
  val error: DBOError[A] = EitherT.liftF[Task,String,A](task)
  OptionT.liftF(error)
}

def task[T](t: T): Task[T] = Task.delay(t)
def add(a: Int, b: Int): Task[Int] = Task.delay(a + b)

val calc: DBOResult[Int] = for {
  a <- valueToDBOResult(10)
  b <- optionToDBOResult(Some(3))  //None: Option[Int])
  c <- eitherToDBOResult(Left[String,Int]("oh my good ..."))
  d <- taskToDBOResult(add(b,c))
} yield d

val sum: Task[Either[String,Option[Int]]] = calc.value.value

sum.runOnComplete {
  case Success(s) => println(s"DBOResult sum=$s")
  case Failure(exception) => println(exception.getMessage)
}
```

Thank you !

谢谢！

github.com/bayakala/scala-meetup-20180902

github.com/sz-scala-meetup/scala-meetup-180630