Аппликативы

Контакты

Лекторы

- Платонова Наталья; **n.platonova@tinkoff.ru**
- Кобенко Михаил; **m.kobenko@tinkoff.ru**

Группа в Telegram

- https://goo.gl/Aq3Ntx

Цели занятия

- вспомнить функторы
- класс типов которого нет
- аппликативные функторы
- traversable функторы

F[_]

```scala
def f[T[_], A, B, C]: T[A] => T[B] => C
```

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

```scala
sealed trait List[+E]
case object Nil extends List[Nothing]
case class Cons[E](h:E, t:List[E]) extends List[E]

implicit val listFunctor = new Functor[List] {
 def map [A, B](f: A=>B)(fa:List[A]): List[B] = fa match {
   case Nil => Nil
   case Cons(x, t) => Cons( f ( x ), map ( f )( t ))
 }
}
```

```scala
implicit val optionFunctor = new Functor[Option] {
 def map [A, B](f:A => B)(fa:Option[A]): Option[B] = fa match {
   case None => None
   case Some(x) => Some(f(x))
 }
}
```

- Composition:
  - `fa.map(f).map(g) = fa.map(f.andThen(g))`
- Identity:
  - `fa.map(x => x) = fa`

```scala
implicit val listFunctor = new Functor[List] {
 def map [A, B](f: A=>B)(fa:List[A]): List[B] = fa match {
   case Nil => Nil
   case x :: xs => f(x) :: f(x) :: map(f(xs))
 }
}
```

```
    //aka Unit, Pointed, Return и т.д.
trait Singleton[F[_]] extends Functor[F] {
 def unit[A](a: A): F[A]
}
```

```scala
implicit val optionSingleton = new Singleton[Option] {
 def unit[A](a:A): Option[A] = Some(a)
}

implicit val listSingleton = new Singleton[List] {
 def unit[A](a:A): List[A] = List(a)
}
```

```scala
implicit val treeSingleton = new Singleton[Tree] {
  def unit[A](a: A) = Leaf(a)
}
```

```
implicit val treeSingleton = new Singleton[Tree] {
 def unit[A](a: A) = Branch(Leaf(a), a(Leaf(a)))
}
```

```
implicit def tupleSingleton[X](implicit x: ???) =
 new Singleton[Tuple] {
   def pure[A](a: A): (X, A) = (???, a)
 }
```

```
    implicit   def   tupleSingleton[X](implicit   x:   Monoid)   =   new
Singleton[Tuple] {
      def pure[A](a: A): (X, A) = (x.empty, a)
     }
```

- `unit.andThen(map(g)) === g.andThen(unit)`

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

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

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

trait Applicative[F[_]] extends Functor[F] {
 def product[A, B](ff: F[A => B])(fa: F[A]): F[B]

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

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

```
implicit def forEither[Err]: Applicative[({type L[A] = Either[Err,
A]})#L] =
   new Applicative[({type L[A] = Either[Err, A]})#L] {
     override def unit[A](x: => A): Either[Err, A] =
       Right(x)

      override def product[A, B](fa: Either[Err, A => B], x: =>
Either[Err, A]): Either[Err, B] =
       fa.flatMap(f => x.map(f))
   }
```

```scala
val F: Applicative[Option] = ...
val depts: Map[String,String] = ...
val salaries: Map[String,Double] = ...
val o: Option[String] =
 F.map2(depts.get("Alice"), salaries.get("Alice"))(
   (dept, salary) => s"Alice in $dept makes $salary per year"
 )
```

```scala
val idsByName: Map[String,Int]
val depts: Map[Int,String] = ...
val salaries: Map[Int,Double] = ...
val o: Option[String] =
 idsByName.get("Bob").flatMap { id =>
   F.map2(depts.get(id), salaries.get(id))(
     (dept, salary) => s"Bob in $dept makes $salary per year"
 }
```

1/1/2010, 25
2/1/2010, 28
3/1/2010, 42
4/1/2010, 53
...

```
val F: Applicative[Parser] = ...
val d: Parser[Date] = ...
val temp: Parser[Double] = ...
val row: Parser[Row] = F.map2(d,
temp)(Row(_, _))
val rows: Parser[List[Row]] = row.sep("\n")
```

Temperature, Date
25, 1/1/2010
28, 2/1/2010
42, 3/1/2010
53, 4/1/2010
...

```
case class Row(date: Date,
temperature: Double)
val F: Monad[Parser] = ...
val d: Parser[Date] = ...
val temp: Parser[Double] = ...
val header: Parser[Parser[Row]] = ...
val rows: Parser[List[Row]] =
 F.flatMap (header) { row =>
row.sep("\n") }
```

```
def eitherMonad[E]: Monad[({type f[x] = Either[E, x]})#f] =
 new Monad[({type f[x] = Either[E, x]})#f] {
   def unit[A](a: => A): Either[E, A] = Right(a)

    override  def  flatMap[A,B](eea:  Either[E,  A])(f:  A  =>
Either[E, B]) = eea match {
     case Right(a) => f(a)
     case Left(b) => Left(b)
   } }
```

```
validName(field1) flatMap (f1 =>
 validBirthdate(field2) flatMap (f2 =>
    validPhone(field3) map (f3 => WebForm(f1, f2,
f3))
```

```
map3(
 validName(field1),
 validBirthdate(field2),
 validPhone(field3))(
 WebForm(_,_,_))
```

```scala
sealed trait Validation[+E, +A]
case class Failure[E](head: E, tail: Vector[E] = Vector())
 extends Validation[E, Nothing]
case class Success[A](a: A) extends Validation[Nothing, A]
```

```
def validationApplicative[E]: Applicative[({type f[x] = Validation[E,x]})#f] =
 new Applicative[({type f[x] = Validation[E,x]})#f] {

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

override def map2[A,B,C](fa: Validation[E,A], fb: Validation[E,B])(f: (A, B) =>
C) =
     (fa, fb) match {
       case (Success(a), Success(b)) => Success(f(a, b))
       case (Failure(h1, t1), Failure(h2, t2)) =>
         Failure(h1, t1 ++ Vector(h2) ++ t2)
       case (e@Failure(_, _), _) => e
       case (_, e@Failure(_, _)) => e
     }
 }
```

```
case class WebForm(name: String, birthdate: Date, phoneNumber: String)
```

```
def validName(name: String): Validation[String, String] =
 if (name != "") Success(name)
 else Failure("Name cannot be empty")
def validBirthdate(birthdate: String): Validation[String, Date] =
 try {
   import java.text._
   Success((new SimpleDateFormat("yyyy-MM-dd")).parse(birthdate))
 } catch {
   Failure("Birthdate must be in the form yyyy-MM-dd")
 }
def validPhone(phoneNumber: String): Validation[String, String] =
 if (phoneNumber.matches("[0-9]{10}"))
   Success(phoneNumber)
 else Failure("Phone number must be 10 digits")
```

```
def validWebForm(name: String,
                 birthdate: String,
                 phone: String): Validation[String, WebForm] =
 map3(
   validName(name),
   validBirthdate(birthdate),
   validPhone(phone))(
   WebForm(_,_,_))
```

identity = ???
homomorphism = ???
interchage = ???
map = ???

```
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)(ma => ma)
}
```

```
object Traverse {
 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)
     }
 }
```

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

```scala
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(_, _))
```

- Identity law: `sequence[Id,A](xs) = xs`.
- Fusion law: `sequence[({type f[x] = F[G[x]]})#f, A](xs) = map(sequence[F,G[A]](xs))(sequence[G,A])`.

```
List(1, 2, 3) traverse { x: Int => Some(x + 1):
Option[Int] }
//res0: Option[List[Int]] = Some(List(2, 3, 4))

List(1, 2, 3) traverse { (x: Int) => None }
//res2: Option[List[Nothing]] = None
```