

Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Functores Aplicativos

Un **functor** puede entenderse como un constructor de tipo $f :: * \rightarrow *$ junto a una función de tipo

$$(a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

que permite mapear/reemplazar los valores de tipo a contenidos en una estructura de tipo $f\ a$ por valores de tipo b .

Functor

En Haskell el concepto de functor es capturado por una clase:

```
class Functor (f :: * → *) where  
  fmap :: (a → b) → f a → f b
```

Functor

En Haskell el concepto de functor es capturado por una clase:

```
class Functor (f :: * → *) where  
    fmap :: (a → b) → f a → f b
```

Para ser efectivamente un functor la función *fmap* debe satisfacer las siguientes propiedades:

$$\begin{aligned} \text{fmap id} &= \text{id} \\ \text{fmap } (f.g) &= \text{fmap } f . \text{fmap } g \end{aligned}$$

que deberian ser chequeadas al definir cada instancia de la clase.

Ejemplos

```
instance Functor [] where  
  fmap = map
```

Ejemplos

```
instance Functor [] where  
    fmap = map
```

```
instance Functor Maybe where  
    fmap f Nothing = Nothing  
    fmap f (Just a) = Just (f a)
```

Ejemplos

```
instance Functor [] where  
    fmap = map
```

```
instance Functor Maybe where  
    fmap f Nothing = Nothing  
    fmap f (Just a) = Just (f a)
```

```
instance Functor (Either a) where  
    fmap f (Right x) = Right (f x)  
    fmap f (Left x)  = Left x
```


Ejemplos

```
instance Functor [] where  
    fmap = map
```

```
instance Functor Maybe where  
    fmap f Nothing = Nothing  
    fmap f (Just a) = Just (f a)
```

```
instance Functor (Either a) where  
    fmap f (Right x) = Right (f x)  
    fmap f (Left x)  = Left x
```

```
instance Functor (( $\rightarrow$ ) r) where  
    fmap f h =  $\lambda r \rightarrow f (h r)$   -- o sea,  $f.h$ 
```

Modelando Error con Maybe

División segura

$$\begin{array}{lcl} \text{div} M \times y & | & y \neq 0 \\ & | & \text{otherwise} \end{array} = \begin{array}{l} \text{Just } (x \text{ 'div' } y) \\ \text{Nothing} \end{array}$$

Modelando Error con Maybe

División segura

$$\begin{array}{lcl} \text{divM } x \ y & | & y \neq 0 \\ & | & \text{otherwise} \end{array} = \begin{array}{l} \text{Just } (x \text{ 'div' } y) \\ \text{Nothing} \end{array}$$

Con *fmap* puede aplicar una función *pura* al resultado de una división

$$\text{foo } x \ y = \text{fmap } (+2) (\text{divM } x \ y)$$

Modelando Error con Maybe

División segura

$$\begin{array}{l|l} \text{divM } x \ y & y \neq 0 \\ & \text{otherwise} \end{array} \begin{array}{l} = \text{Just } (x \text{ 'div' } y) \\ = \text{Nothing} \end{array}$$

Con *fmap* puede aplicar una función *pura* al resultado de una división

$$\text{foo } x \ y = \text{fmap } (+2) (\text{divM } x \ y)$$

en lugar de hacer:

$$\begin{array}{l} \text{foo } x \ y = \text{case divM } x \ y \text{ of} \\ \quad \text{Just } r \quad \rightarrow \text{Just } (r + 2) \\ \quad \text{Nothing} \rightarrow \text{Nothing} \end{array}$$

Funtores Aplicativos

Los **funtores aplicativos** son funtores que permiten modelar efectos y aplicar funciones dentro del functor (lo que les da el mote de *aplicativos*).

```
class Functor f ⇒ Applicative f where  
  pure  :: a → f a  
  (<*>) :: f (a → b) → f a → f b
```

Funtores Aplicativos

Los **funtores aplicativos** son funtores que permiten modelar efectos y aplicar funciones dentro del functor (lo que les da el mote de *aplicativos*).

```
class Functor f ⇒ Applicative f where  
  pure  :: a → f a  
  (<*>) :: f (a → b) → f a → f b
```

Se debe cumplir que:

$$fmap\ f\ x = pure\ f\ <*>\ x$$

Sinónimo en *Applicative*:

```
(<$>) :: Functor f ⇒ (a → b) → f a → f b  
f <$> t = fmap f t
```

Ejemplo: Maybe

```
instance Applicative Maybe where  
  pure = Just  
  (Just f) <*> (Just x) = Just (f x)  
  _ <*> _ = Nothing
```

Ejemplo: Maybe

```
instance Applicative Maybe where
  pure = Just
  (Just f) <*> (Just x) = Just (f x)
  _ <*> _ = Nothing
```

Puedo por ejemplo modelar expresiones con errores:

```
type Expr = Maybe Int
valE x    = pure x
addE x y  = (+) <$> x <*> y
divE x y  = case (x, y) of
  (Just vx, Just vy) → divM vx vy
  _                  → Nothing
```


Leyes de funtores aplicativos

Identidad:

$$\text{pure } id \langle * \rangle u \equiv u$$

Composición:

$$\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w \equiv u \langle * \rangle (v \langle * \rangle w)$$

Homomorfismo:

$$\text{pure } f \langle * \rangle \text{pure } x \equiv \text{pure } (f \ x)$$

Intercambio:

$$u \langle * \rangle \text{pure } x \equiv \text{pure } (\lambda f \rightarrow f \ x) \langle * \rangle u$$

Leyes de funtores aplicativos

Identidad:

$$\text{pure } id \langle * \rangle u \equiv u$$

Composición:

$$\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w \equiv u \langle * \rangle (v \langle * \rangle w)$$

Homomorfismo:

$$\text{pure } f \langle * \rangle \text{pure } x \equiv \text{pure } (f \ x)$$

Intercambio:

$$u \langle * \rangle \text{pure } x \equiv \text{pure } (\lambda f \rightarrow f \ x) \langle * \rangle u$$

Si se cumplen, entonces se cumple:

$$fmap \ f \ x = \text{pure } f \langle * \rangle x$$

Funciones sobre funtores aplicativos

$sequenceA :: Applicative\ f \Rightarrow [f\ a] \rightarrow f\ [a]$
 $sequenceA\ [] = pure\ []$
 $sequenceA\ (a : as) = (:) <\$> a <*> sequenceA\ as$

$traverse :: Applicative\ f \Rightarrow (a \rightarrow f\ b) \rightarrow [a] \rightarrow f\ [b]$
 $traverse\ f = sequenceA.fmap\ f$

que equivale a:

$traverse\ f\ [] = pure\ []$
 $traverse\ f\ (x : xs) = (:) <\$> f\ x <*> traverse\ f\ xs$

Alternative

En *Control.Applicative* también se define:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some  :: f a -> f [a]  -- one or more
  many  :: f a -> f [a]  -- zero or more
```

Ejemplo de Alternative: Parsers

```
instance Applicative (Parser s) where  
  pure = pSucceed  
  <*> = <*>
```

```
instance Alternative (Parser s) where  
  empty = pFail  
  <|> = <|>  
  many = pList  
  some p = (:) <$> p <*> pList p
```

donde

```
pFail   :: Parser s a  
pSucceed :: a → Parser s a  
<*>     :: Parser s (a → b) → Parser s a → Parser s b  
<|>     :: Parser s a → Parser s a → Parser s a  
pList   :: Parser s a → Parser s [a]
```

Ejemplo: listas

```
instance Applicative [] where  
  pure x = [x]  
  fs <*> xs = [f x | f ← fs, x ← xs]
```

Ejemplo:

$[(+1), (+2)] <*> [1, 2, 3]$

Ejemplo: listas

```
instance Applicative [] where  
  pure x = [x]  
  fs <*> xs = [f x | f ← fs, x ← xs]
```

Ejemplo:

```
leadsto  
  [(+1), (+2)] <*> [1, 2, 3]  
  [2, 3, 4, 3, 4, 5]
```

Ejemplo: Either

```
data Either a b = Left a | Right b
```

```
instance Functor (Either e) where  
  fmap f (Right a) = Right (f a)  
  fmap f (Left e)  = Left  e
```


Ejemplo: Either

```
data Either a b = Left a | Right b
```

```
instance Functor (Either e) where  
  fmap f (Right a) = Right (f a)  
  fmap f (Left e)  = Left  e
```

Una posible instancia de *Applicative*:

```
instance Applicative (Either e) where  
  pure = Right  
  Right f <*> Right a = Right (f a)  
  Right f <*> Left e  = Left e  
  Left e  <*> _      = Left e
```

Ejemplo: Either

```
data Either a b = Left a | Right b
```

```
instance Functor (Either e) where  
  fmap f (Right a) = Right (f a)  
  fmap f (Left e)  = Left  e
```

Una posible instancia de *Applicative*:

```
instance Applicative (Either e) where  
  pure = Right  
  Right f <*> Right a = Right (f a)  
  Right f <*> Left e  = Left e  
  Left e  <*> _       = Left e
```

Otra:

```
instance Monoid e => Applicative (Either e) where  
  pure = Right  
  Right f <*> Right a = Right (f a)  
  Left e  <*> Right _ = Left e  
  Right _ <*> Left e  = Left e  
  Left e  <*> Left e' = Left (e 'mappend' e')
```

Composición

La clase de funtores aplicativos es cerrada bajo la composición.

```
newtype (f :. g) a = Compose { getCompose :: f (g a) }
```

```
instance (Functor f, Functor g) => Functor (f :. g) where  
    fmap f (Compose x) = Compose (fmap (fmap f) x)
```

```
instance (Applicative f, Applicative g)  
    => Applicative (f :. g) where  
    pure x = Compose (pure (pure x))  
    Compose f <*> Compose x = Compose ((<*>) <$> f <*> x)
```

Composición

La clase de funtores aplicativos es cerrada bajo la composición.

```
newtype (f :. g) a = Compose { getCompose :: f (g a) }
```

```
instance (Functor f, Functor g) => Functor (f :. g) where  
    fmap f (Compose x) = Compose (fmap (fmap f) x)
```

```
instance (Applicative f, Applicative g)  
    => Applicative (f :. g) where  
    pure x = Compose (pure (pure x))  
    Compose f <*> Compose x = Compose ((<*>) <$> f <*> x)
```

La composición de dos mónadas puede no ser una mónada, pero es un aplicativo.