Programación Funcional Avanzada

Contenedores

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Monoides

Aparecen por todos lados

- `Data.Monoid` establece cualquier tipo monoide como

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

  - `mappend` – operador binario cerrado en el tipo.
  - `mempty` – elemento neutro para la operación.
  - `mconcat` – aplicar `mappend` en secuencia.

UNIVERSIDAD SIMÓN BOLÍVAR

Monoides

Aparecen por todos lados

- `Data.Monoid` establece cualquier tipo monoide como

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

  - `mappend` – operador binario cerrado en el tipo.
  - `mempty` – elemento neutro para la operación.
  - `mconcat` – aplicar `mappend` en secuencia.

- El programador debe proveer `mempty` y `mappend`

```
mconcat = foldr mappend mempty
```

`mempty` y `mappend` son genéricos –
`foldr` también según `Data.Foldable`

UNIVERSIDAD SIMÓN BOLÍVAR

Algunos monoides

- Las listas son los monoides más conocidos

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

- Si bien uno acostumbra escribir...

```
[1,2,3] ++ [4,5,6]
[] ++ "foo"
concat ["foo","bar","baz"]
```

- ...también puede escribir

```
mappend [1,2,3] [4,5,6]
mappend mempty "foo"
mconcat ["foo","bar","baz"]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Dos monoides para un mismo tipo

- Los números podrían ser un `Monoid` con el cero y la suma

```
instance Num a => Monoid a where
  mempty  = 0
  mappend = (+)
```

Dos monoides para un mismo tipo

- Los números podrían ser un `Monoid` con el cero y la suma

```
instance Num a => Monoid a where
  mempty  = 0
  mappend = (+)
```

- Los números podrían ser un `Monoid` con el uno y el producto

```
instance Num a => Monoid a where
  mempty  = 1
  mappend = (*)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Dos monoides para un mismo tipo

- Los números podrían ser un `Monoid` con el cero y la suma

```
instance Num a => Monoid a where
  mempty  = 0
  mappend = (+)
```

- Los números podrían ser un `Monoid` con el uno y el producto

```
instance Num a => Monoid a where
  mempty  = 1
  mappend = (*)
```

¿Cuál instancia aplicamos en
`mappend 42 mempty`?

UNIVERSIDAD SIMÓN BOLÍVAR

Envoltorios para discriminar

Números para sumar o números para multiplicar

- Envolvemos los números para sumar

```
newtype Sum a = Sum { getSum :: a }
    deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
    mempty                      = Sum 0
    mappend' (Sum x) (Sum y) = Sum (x + y)
```

Envoltorios para discriminar

Números para sumar o números para multiplicar

- Envolvemos los números para sumar

```
newtype Sum a = Sum { getSum :: a }
    deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
    mempty                      = Sum 0
    mappend' (Sum x) (Sum y) = Sum (x + y)
```

- Y nos permite escribir

```
> mappend (Sum 28) (Sum 14)
Sum {getSum = 42}
> getSum $ mappend (Sum 28) (Sum 14)
42
```

UNIVERSIDAD SIMÓN BOLÍVAR

Si, es incómodo

. . . porque sólo es una prueba de concepto

- En la práctica no vamos a sumar ni multiplicar con esa técnica – mucho menos para cosas como

```
> (getSum . mconcat . map Sum) [1..5]
15
```

- Con `Bool` hay un problema similar (¿cuál?) –
  busquen la solución antes de ver los detalles en `Data.Monoid`.

UNIVERSIDAD SIMÓN BOLÍVAR

Si, es incómodo

. . . porque sólo es una prueba de concepto

- En la práctica no vamos a sumar ni multiplicar con esa técnica – mucho menos para cosas como

```
> (getSum . mconcat . map Sum) [1..5]
15
```

- Con `Bool` hay un problema similar (¿cuál?) –
  busquen la solución antes de ver los detalles en `Data.Monoid`.
- Protip – Se acostumbra escribir (noten las comillas *invertidas*)

```
foo `mappend` bar
```

en lugar de

```
mappend foo bar
```

para transmitir la noción de "operador".

UNIVERSIDAD SIMÓN BOLÍVAR

The fast `Monoid`

This is the monoid you're looking for

- Muchos algoritmos requieren coleccionar cosas – el principiante usa `[a]` y eso es suficiente.
  - Evitar `(++)` en lo posible – costoso.
  - Coleccionar por el principio con `(:)`.
  - Aplicar `reverse` si el orden importa.
- Listas y pilas van muy bien así – colas y *deques* no.

UNIVERSIDAD SIMÓN BOLÍVAR

The fast `Monoid`

This is the monoid you're looking for

- Muchos algoritmos requieren coleccionar cosas –
  el principiante usa `[a]` y eso es suficiente.
  - Evitar `(++)` en lo posible – costoso.
  - Coleccionar por el principio con `(:)`.
  - Aplicar `reverse` si el orden importa.
- Listas y pilas van muy bien así – colas y *deques* no.
- `Data.Sequence` provee el ADT `Seq`
  - Modela secuencias **finitas** generalizadas.
  - Altamente eficientes en tiempo – 2-3 *finger trees* con valores.
  - Más operaciones que `[a]`.
  - La mayoría de las operaciones son *estrictas* – eficientes en espacio.

Si la colección es **finita**,
`Data.Sequence` es la estructura

UNIVERSIDAD SIMÓN BOLÍVAR

The fast `Monoid`

- ADT `Seq a` – implantación opaca para el programador.
- Constructores $O(1)$

```
empty     :: Seq a
singleton :: a -> Seq a
```

The fast `Monoid`

- ADT `Seq a` – implantación opaca para el programador.
- Constructores $O(1)$

```
empty     :: Seq a
singleton :: a -> Seq a
```

- Operadores de agregado $O(1)$ – por *ambos* extremos

```
(<|) :: a -> Seq a -> Seq a
(|>) :: Seq a -> a -> Seq a
```

UNIVERSIDAD SIMÓN BOLÍVAR

The fast `Monoid`

- ADT `Seq a` – implantación opaca para el programador.
- Constructores $O(1)$

```
empty     :: Seq a
singleton :: a -> Seq a
```

- Operadores de agregado $O(1)$ – por *ambos* extremos

```
(<|) :: a -> Seq a -> Seq a
(|>) :: Seq a -> a -> Seq a
```

- Operador de concatenación $O(log(min(n_1, n_2)))$

```
(><) :: Seq a -> Seq a -> Seq a
```

UNIVERSIDAD SIMÓN BOLÍVAR

The fast `Monoid`

- ADT `Seq a` – implantación opaca para el programador.
- Constructores $O(1)$

```
empty     :: Seq a
singleton :: a -> Seq a
```

- Operadores de agregado $O(1)$ – por *ambos* extremos

```
(<|) :: a -> Seq a -> Seq a
(|>) :: Seq a -> a -> Seq a
```

- Operador de concatenación $O(log(min(n_1, n_2)))$

```
(><) :: Seq a -> Seq a -> Seq a
```

- Upgrade your lists in $O(n)$!

```
fromList :: [a] -> Seq a
```

UNIVERSIDAD SIMÓN BOLÍVAR

The fast `Monoid`

- La instancia de `Monoid` es directa

```
instance Monoid (Seq a) where
  mempty  = empty
  mappend = (><)
```

- Escriba sus programas genéricos para `Monoid a`
- Use `[a]` para prototipos – reemplaze con `Seq a` para producción.

Preservando la forma

`map`, de nuevo

```
map :: (a -> b) -> [a] -> [b]
```

- Aplica una función a cada elemento de la lista.
- Produce una lista con los resultados – quizás de tipo diferente.
- El resultado sigue siendo una lista de la misma longitud.

`map` transformó los contenidos,
pero **preservó** la estructura.

UNIVERSIDAD SIMÓN BOLÍVAR

Podemos imitar el concepto convenientemente

- Si tenemos un árbol

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Podemos escribir

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf a)     = Leaf (f a)
treeMap f (Node l r) = Node (treeMap f l)
                            (treeMap f r)
```

- `treeMap` aplica `f` preservando la estructura de árbol.

UNIVERSIDAD SIMÓN BOLÍVAR

Podemos imitar el concepto convenientemente

- Si tenemos un árbol

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Podemos escribir

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf a)     = Leaf (f a)
treeMap f (Node l r) = Node (treeMap f l)
                            (treeMap f r)
```

- `treeMap` aplica `f` preservando la estructura de árbol.

¿Cómo generalizar el concepto?

UNIVERSIDAD SIMÓN BOLÍVAR

Functores

- `Data.Functor` establece cualquier tipo functor como

```
class Functor f where
  fmap  :: (a -> b) -> f a -> f b
  (<$)  :: a -> f b -> f a
  (<$>) :: Functor f => (a -> b) -> f a -> f b
```

  - El functor es `f` – el que *contiene* datos.
  - `fmap` – aplicador de función preservando estructura de `f`.
  - `<$` – inyector de contenidos en `f`.
  - `<$>` – igual a `fmap` pero como operador infijo no asociativo.

UNIVERSIDAD SIMÓN BOLÍVAR

Functores

- `Data.Functor` establece cualquier tipo functor como

```
class Functor f where
  fmap  :: (a -> b) -> f a -> f b
  (<$)  :: a -> f b -> f a
  (<$>) :: Functor f => (a -> b) -> f a -> f b
```

  - El functor es `f` – el que *contiene* datos.
  - `fmap` – aplicador de función preservando estructura de `f`.
  - `<$` – inyector de contenidos en `f`.
  - `<$>` – igual a `fmap` pero como operador infijo no asociativo.
- `fmap` es una función de *lifting*
  - "Levanta" el dato del contenedor.
  - Permite la aplicación del cómputo.
  - "Baja" el resultado al contenedor.

"Contenedor" es una simplificación de
Contexto de Cómputo

UNIVERSIDAD SIMÓN BOLÍVAR

Las listas son functores

No debería sorprender a nadie...

- `Prelude` define la instancia

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

- ...y ahora podemos escribir

```
> fmap (*2) [1,2,3]
[2,4,6]
> (*2) `fmap` [1,2,3]
[2,4,6]
```

- Y podemos aprovechar <$> para comodidad

```
> (*2) <$> [1,2,3]
[2,4,6]
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y el inyector?

```
(<$) :: a -> f b -> f a
```

- Los functores *preservan* la estructura ante las operaciones de *lifting*.
- El inyector no está exento de esa responsabilidad
  - Su primer argumento es *un valor*.
  - El segundo argumento es un functor conteniendo "algo".
  - El functor no puede cambiar su forma – sólo sus contenidos.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y el inyector?

```
(<$) :: a -> f b -> f a
```

- Los functores *preservan* la estructura ante las operaciones de *lifting*.
- El inyector no está exento de esa responsabilidad
  - Su primer argumento es *un valor*.
  - El segundo argumento es un functor conteniendo "algo".
  - El functor no puede cambiar su forma – sólo sus contenidos.
- ¡Inyecta tantos valores como hagan falta!

```
> 42 <$ [1,2,3]
[42,42,42]
> [1,2,3] <$ "quux"
[[1,2,3],[1,2,3],[1,2,3],[1,2,3]]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Nuestro árbol puede ser un functor

- Basta definir la instancia

```
instance Functor Tree where
  fmap = treeMap
```

usando `treeMap` que definimos "manualmente"

Nuestro árbol puede ser un functor

- Basta definir la instancia

```
instance Functor Tree where
  fmap = treeMap
```

usando `treeMap` que definimos "manualmente"

- ...y ahora podemos escribir

```
> fmap (*2) Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
Node (Leaf 2) (Node (Leaf 4) (Leaf 6))
```

UNIVERSIDAD SIMÓN BOLÍVAR

Nuestro árbol puede ser un functor

- Basta definir la instancia

```
instance Functor Tree where
  fmap = treeMap
```

usando `treeMap` que definimos "manualmente"

- ...y ahora podemos escribir

```
> fmap (*2) Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
Node (Leaf 2) (Node (Leaf 4) (Leaf 6))
```

- Incluso inyectar

```
> "foo" <$ Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
Node (Leaf "foo") (Node (Leaf "foo") (Leaf "foo"))
```

UNIVERSIDAD SIMÓN BOLÍVAR

Meanwhile in GHC

. . . a partir de 6.12

```
{-# LANGUAGE DeriveFunctor #-}

import qualified Data.Functor as DF

data Tree a = Leaf a
            | Node (Tree a) (Tree a)
            deriving (Show,DF.Functor)
```

- GHC es capaz de *generar* la instancia `Functor` – suficiente en el caso general y sin esfuerzo de programación.

UNIVERSIDAD SIMÓN BOLÍVAR

Una instancia reveladora

Sólo contiene cuando hay espacio

- `Data.Maybe` es un contenedor

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

  - Si el contenedor está "vacío", continuará vacío.
  - Si el contenedor tiene algo, se opera sobre su contenido.

UNIVERSIDAD SIMÓN BOLÍVAR

Una instancia reveladora

Sólo contiene cuando hay espacio

- Data.Maybe es un contenedor

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

  - Si el contenedor está "vacío", continuará vacío.
  - Si el contenedor tiene algo, se opera sobre su contenido.

- ...y podemos escribir

```
> (*2) <$> Nothing
Nothing
> (*2) <$> Just 21
Just 42
> "Ponies!" <$ Just 7
Just "Ponies!"
```

UNIVERSIDAD SIMÓN BOLÍVAR

El IO es un Functor

Wait. . . what?

- Haskell separa las operaciones puras de las impuras – se modelan efectos de borde con las acciones de tipo `IO a`
  - El típico programa opera en el *contexto* `IO`.
  - Los valores deben ser "levantados" del contexto `IO`.
  - Las funciones puras operan sobre ellos.
  - Los resultados deben ser "bajados" al contexto `IO`.

UNIVERSIDAD SIMÓN BOLÍVAR

El IO es un Functor

Wait. . . what?

- Haskell separa las operaciones puras de las impuras – se modelan efectos de borde con las acciones de tipo `IO a`
  - El típico programa opera en el *contexto* `IO`.
  - Los valores deben ser "levantados" del contexto `IO`.
  - Las funciones puras operan sobre ellos.
  - Los resultados deben ser "bajados" al contexto `IO`.
- No debería sorprender

```
instance Functor IO where
  fmap f action  = do
                     result <- action
                     return (f result)
```

UNIVERSIDAD SIMÓN BOLÍVAR

`fmap` funciones dentro del `IO`

- En lugar de...

```
import Data.Char

main = do
  in <- getLine
  let out = (reverse . map toUpper) in
  putStrLn out
```

`fmap` funciones dentro del `IO`

- En lugar de...

```
import Data.Char

main = do
  in <- getLine
  let out = (reverse . map toUpper) in
  putStrLn out
```

- ...es más compacto y elegante...

```
import Data.Char

main = do
 out <- fmap (reverse . map toUpper) getLine
 putStrLn out
```

- Protip – usar <$> en modo operador.

UNIVERSIDAD SIMÓN BOLÍVAR

The fast `Monoid`

- También es un `Functor` – `fmap` cualquier `Seq`

The fast `Monoid`

- También es un `Functor` – `fmap` cualquier `Seq`
- Dos `Functor` interesantes y muy útiles

```
viewl :: Seq a -> ViewL a
viewr :: Seq a -> ViewR a
```

  - Inspeccionar extremos de la secuencia *sin* eliminarlos – $O(1)$
  - `fmap` sobre el extremo **y** el resto.

UNIVERSIDAD SIMÓN BOLÍVAR

The fast Monoid

- También es un `Functor` – `fmap` cualquier `Seq`
- Dos `Functor` interesantes y muy útiles

```
viewl :: Seq a -> ViewL a
viewr :: Seq a -> ViewR a
```

  - Inspeccionar extremos de la secuencia *sin* eliminarlos – $O(1)$
  - `fmap` sobre el extremo **y** el resto.

- Son análogos al `(:)` de las listas

```
> let x = fromList [1,2,3]
> viewl x
1 :< fromList [2,3]
> viewr x
fromList [1,2] >: 3
> fmap (2*) $ viewl x
2 :< fromList [4,6]
```

- Protip – usar `viewr`/`viewl` en *pattern matching*.

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

- ¿Cuál es el significado de `fmap`? – la intuición hasta ahora:
  1. Liberar un valor puro de un "entorno".
  2. Aplicarle una función pura para obtener un resultado.
  3. Hacer que el "entorno" envuelva el resultado.

Si están pensando "caja", "paquete" o "tobo"
no los culpo – es lo más parecido.

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

Entonces es un "contenedor"

¿Cuál es el contenedor más simple?

The meaning of `fmap`, `Functor` and everything...

Entonces es un "contenedor"

¿Cuál es el contenedor más simple?

```
newtype Tobo a = Tobo a

instance Functor Tobo where
  fmap f (Tobo a) = Tobo (f a)
```

- Es un `newtype` – ¡no tiene representación "especial" en memoria!
- En otras palabras, `Tobo` es una *ilusión* –
  lo que importa es que define un contexto de cómputo.
- `fmap` lo único que hace es mantener ese contexto.

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

Entonces es un "contenedor"

¿Cuál es el contenedor más simple?

```
newtype Tobo a = Tobo a

instance Functor Tobo where
  fmap f (Tobo a) = Tobo (f a)
```

- Es un `newtype` – ¡no tiene representación "especial" en memoria!
- En otras palabras, `Tobo` es una *ilusión* –
  lo que importa es que define un contexto de cómputo.
- `fmap` lo único que hace es mantener ese contexto.

Now for something completely different...

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

Pick a function, any function

- Consideremos una función polimórfica simple

```
foo :: d -> r
```

The meaning of `fmap`, `Functor` and everything...

Pick a function, any function

- Consideremos una función polimórfica simple

```
foo :: d -> r
```

- `->` no es "mágico" – es un constructor de tipos infijo.

```
foo :: (->) d r
```

  - `(->)`
    tipo de las funciones.
  - `(->) d r`
    tipo de las funciones que operan sobre `d` produciendo `r`

The meaning of `fmap`, `Functor` and everything...

Pick a function, any function

- Consideremos una función polimórfica simple

```
foo :: d -> r
```

- `->` no es "mágico" – es un constructor de tipos infijo.

```
foo :: (->) d r
```

  - `(->)`
    tipo de las funciones.
  - `(->) d r`
    tipo de las funciones que operan sobre `d` produciendo `r`

¿Qué es `((->) d)`?

The meaning of `fmap`, `Functor` and everything...

Let's go meta...

```
((->) d)
```

The meaning of `fmap`, `Functor` and everything...

Let's go meta...

```
((->) d)
```

- No "parece" un tobo...
  - ...pero "envuelve" cosas tipo `r`.
  - ...y eso lo hace polimórfico.

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

Let's go meta...

```
((->) d)
```

- No "parece" un tobo...
  - ...pero "envuelve" cosas tipo `r`.
  - ...y eso lo hace polimórfico.
- Eso sugiere recorrer la ruta...

```
instance Functor ((->) d) where
  fmap f {- ... WTF? -}
```

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

La deducción es simple

- Veamos la firma de `fmap`

```
fmap :: (a -> b) -> f a -> f b
```

The meaning of `fmap`, `Functor` and everything...

La deducción es simple

- Veamos la firma de `fmap`

```
fmap :: (a -> b) -> f a -> f b
```

- En nuestro ejemplo, el "tobo" `f` es `((->) d)`

```
fmap :: (a -> b) -> ((->) d) a -> ((->) d) b
```

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

La deducción es simple

- Veamos la firma de `fmap`

```
fmap :: (a -> b) -> f a -> f b
```

- En nuestro ejemplo, el "tobo" `f` es `((->) d)`

```
fmap :: (a -> b) -> ((->) d) a -> ((->) d) b
```

- Regresemos las funciones a su notación habitual

```
fmap :: (a -> b) -> (d -> a) -> (d -> b)
```

¡`fmap` de una función sobre una función
debe producir una función para preservar el "tobo"!

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

When you see it...

- ¿Cuál es la conexión? – La respuesta está en los tipos.

```
fmap :: (a -> b) -> (d -> a) -> (d -> b)
```

  - El "contexto" es una función `(d -> a)`
  - Después de hacerle `fmap` de una función `(a -> b)` el nuevo "contexto" *tiene* que ser `(d -> b)`

UNIVERSIDAD SIMÓN BOLÍVAR

The meaning of `fmap`, `Functor` and everything...

When you see it...

- ¿Cuál es la conexión? – La respuesta está en los tipos.

```
fmap :: (a -> b) -> (d -> a) -> (d -> b)
```

  - El "contexto" es una función `(d -> a)`
  - Después de hacerle `fmap` de una función `(a -> b)` el nuevo "contexto" *tiene* que ser `(d -> b)`

- ¡Composición de funciones!

```
instance Functor ((->) d) where
  fmap = (.)
```

The meaning of `fmap`, `Functor` and everything...

- La aplicación funcional **es** un `Functor`.
  - El "contexto" es una función existente.
  - Transformar el contexto vía una función, es componerla.
  - El nuevo "contexto" es la combinación.

The meaning of `fmap`, `Functor` and everything...

- La aplicación funcional **es** un `Functor`.
  - El "contexto" es una función existente.
  - Transformar el contexto vía una función, es componerla.
  - El nuevo "contexto" es la combinación.
- `Control.Monad.Instances` tiene la instancia `Functor`

```
> :t fmap (*2) (+14)
fmap (*2) (+14) :: (Num a) => a -> a
> fmap (*2) (+14) 7
42
```

...y si combinamos con el operador de `Data.Functor`

```
> (*2) <$> (+14) $ 7
42
```

Looks legit to me...

Contextos en contexto

- En definitiva, un `Functor` terminó siendo . . .
  - Un tipo polimórfico `f` que pone un *valor en contexto*.
  - Una función que permite tomar cualquier cómputo *ajeno* al contexto y operar con el contenido del contexto.
  - El *contexto* es cualquier valor – y como las funciones son valores, llegamos hasta aquí.
- `fmap` toma un cómputo *ajeno* al contexto...

```
fmap :: (a -> b) -> f a -> f b
```

- Pero sabemos que es posible tener *funciones* dentro del contexto. . .

UNIVERSIDAD SIMÓN BOLÍVAR

Cómputos, valores y resultados en contexto

- En lugar de traer los cómputos desde afuera, ¿por qué no mantenerlos en el mismo contexto?
  - El `Functor` daría contexto al operando.
  - El `Functor` daría contexto al resultado.
  - El `Functor` daría contexto a la **operación**.

UNIVERSIDAD SIMÓN BOLÍVAR

Cómputos, valores y resultados en contexto

- En lugar de traer los cómputos desde afuera, ¿por qué no mantenerlos en el mismo contexto?
  - El `Functor` daría contexto al operando.
  - El `Functor` daría contexto al resultado.
  - El `Functor` daría contexto a la **operación**.
- Sería un `Functor` que se puede *aplicar*

```
class (Functor f) => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

  - `pure` – pone cualquier valor puro en contexto.
  - `<*>` – aplica una operación en contexto a su operando.
  - Sigue siendo un `Functor` – ¡podemos aprovechar cómputos ajenos al contexto vía `fmap`!

UNIVERSIDAD SIMÓN BOLÍVAR

Cómputos, valores y resultados en contexto

- En lugar de traer los cómputos desde afuera, ¿por qué no mantenerlos en el mismo contexto?
  - El `Functor` daría contexto al operando.
  - El `Functor` daría contexto al resultado.
  - El `Functor` daría contexto a la **operación**.
- Sería un `Functor` que se puede *aplicar*

```
class (Functor f) => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

  - `pure` – pone cualquier valor puro en contexto.
  - `<*>` – aplica una operación en contexto a su operando.
  - Sigue siendo un `Functor` – ¡podemos aprovechar cómputos ajenos al contexto vía `fmap`!
- Protip – escribir `f <$> x` en lugar de `pure f <*> x`

UNIVERSIDAD SIMÓN BOLÍVAR

Functores aplicables

Maybe otra vez

- `Control.Applicative` provee la instancia

```
instance Applicative Maybe where
  pure                      = Just

  Nothing <*> _             = Nothing
  (Just f) <*> something = fmap f something
```

- ¿Cómo pongo un valor puro en contexto `Maybe`? – `Just` do it.
- ¿Cómo aplico una operación en contexto `Maybe`?
  - Si no hay nada que aplicar, pues no hay resultados en contexto.
  - Si hay alguna función que aplicar, `fmap` se encarga de preservar el contexto en el cual se aplica.

UNIVERSIDAD SIMÓN BOLÍVAR

Functores aplicables

- Ahora podemos escribir cosas como

```
> Just (*2) <*> Nothing
Nothing
> pure (*2) <*> Nothing
Nothing
> pure (*2) <*> Just 21
Just 42
> Nothing <*> Just 42
Nothing
```

UNIVERSIDAD SIMÓN BOLÍVAR

Functores aplicables

- Ahora podemos escribir cosas como

```
> Just (*2) <*> Nothing
Nothing
> pure (*2) <*> Nothing
Nothing
> pure (*2) <*> Just 21
Just 42
> Nothing <*> Just 42
Nothing
```

- El verdadero poder está en hacer aplicaciones *parciales*

```
> pure (*) <*> Just 2 <*> Just 21
Just 42
```

¡Construir cómputos **puros**
por partes y en un contexto específico!

UNIVERSIDAD SIMÓN BOLÍVAR

¡Las listas también!

- Colección de funciones y resultados – no-determinismo "ordenado"

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

¡Las listas también!

- Colección de funciones y resultados – no-determinismo "ordenado"

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

- Aplicar una lista de funciones a una lista de argumentos – `map` en esteroides

```
> [length] <*> ["I","grok","applicative"]
[1,4,11]
> pure (++) <*> ["wtf","huh"] <*> ["?","!",""]
["wtf?","wtf!","wtf","huh?","huh!","huh"]
```

UNIVERSIDAD SIMÓN BOLÍVAR

El IO es un functor aplicable

- `Control.Applicative` provee la instancia

```
instance Applicative IO where
  pure = return
  a <*> b = do
              f <- a
              x <- b
            return (f x)
```

El IO es un functor aplicable

- `Control.Applicative` provee la instancia

```
instance Applicative IO where
  pure = return
  a <*> b = do
              f <- a
              x <- b
            return (f x)
```

- Permite establecer una *secuencia* de operaciones

```
main = do
  a <- pure (++) <*> getLine <*> getLine
  putStrLn $ a
```

UNIVERSIDAD SIMÓN BOLÍVAR

El golpe de gracia

lifting generalizado

- `Functor` – *lift* de una función *unaria* hacia el contexto via `fmap`
- `Applicative` – igual, con la función fuera o en contexto.

UNIVERSIDAD SIMÓN BOLÍVAR

El golpe de gracia

lifting generalizado

- `Functor` – *lift* de una función *unaria* hacia el contexto via `fmap`
- `Applicative` – igual, con la función fuera o en contexto.
- ¿Y si mi función es $n-$aria y no puedo ir por partes?

```
liftA2 :: Applicative f =>
          (a -> b -> c)
          -> f a -> f b -> f c
liftA3 :: Applicative f =>
          (a -> b -> c -> d)
          -> f a -> f b -> f c -> f d
```

UNIVERSIDAD SIMÓN BOLÍVAR

El golpe de gracia

lifting generalizado

- `Functor` – *lift* de una función *unaria* hacia el contexto via `fmap`
- `Applicative` – igual, con la función fuera o en contexto.
- ¿Y si mi función es $n-$aria y no puedo ir por partes?

```
liftA2 :: Applicative f =>
          (a -> b -> c)
          -> f a -> f b -> f c
liftA3 :: Applicative f =>
          (a -> b -> c -> d)
          -> f a -> f b -> f c -> f d
```

- "Interracial functoring"

```
> liftA2 (:) (Just 2) (Just [3])
Just [2,3]
```

Quiero saber más...

- Applicative Functors in Haskell
  Rüegg
- Applicative programming with effects
  McBride & Patterson
- Documentación de `Data.Monoid`
- Documentación de `Data.Sequence`
- Documentación de `Data.Functor`
- Documentación de `Control.Applicative`

UNIVERSIDAD SIMÓN BOLÍVAR