Programación Funcional Avanzada

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

Functores Aplicativos

La clase Monad

En las versiones actuales de GHC tenemos que la clase *Monad* es una subclase de *Applicative*:

$$
\begin{aligned}
&\text{class } \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m \text{ where} \\
&\quad \mathit{return} :: a \to m\ a \\
&\quad (\gg\!=) \ :: m\ a \to (a \to m\ b) \to m\ b \\
&\quad (\gg) \ \ :: m\ a \to m\ b \to m\ b
\end{aligned}
$$

Analicemos entonces qué son los **functores aplicativos**.

$$
\begin{aligned}
&\text{class } \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f \text{ where} \\
&\quad \mathit{pure} \ \ :: a \to f\ a \\
&\quad (<\!*\!>) :: f\ (a \to b) \to f\ a \to f\ b
\end{aligned}
$$

Mejor primero analicemos qué son los **functores**.

Functor

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

$$(a \to b) \to f\ a \to f\ b$$

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:

$$\begin{aligned}&\text{class } \mathit{Functor}\ (f :: * \to *) \text{ where}\\ &\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b\end{aligned}$$

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

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

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

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 ((->) r) where
  fmap f h = \r -> f (h r)   -- o sea, f . h

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

Functores Aplicativos

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

$$\begin{aligned}&\text{class } \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f \text{ where}\\ &\quad \mathit{pure}\ \ :: a \to f\ a\\ &\quad (\texttt{<*>}) :: f\ (a \to b) \to f\ a \to f\ b\end{aligned}$$

Se debe cumplir que:

$$\mathit{fmap}\ f\ x = \mathit{pure}\ f\ \texttt{<*>}\ x$$

Sinónimo en *Applicative*:

$$\begin{aligned}&(\texttt{<\$>}) :: \mathit{Functor}\ f \Rightarrow (a \to b) \to f\ a \to f\ b\\ &f\ \texttt{<\$>}\ t = \mathit{fmap}\ f\ t\end{aligned}$$

Ejemplo: *Maybe*

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

Ejemplo: *Maybe*

Con functores aplicativos, el mapeo de múltiples valores *Maybe*:

```
fmap2 f (Just x) (Just y)  = Just (f x y)
fmap2 _ _        _         = Nothing
fmap3 f (Just x) (Just y) (Just z) = Just (f x y z)
fmap3 _ _        _         _       = Nothing
...
```

se puede definir de manera uniforme:

```
fmap2 f a b   = f <$> a <*> b
fmap3 f a b c = f <$> a <*> b <*> c
...
```

Diferencia entre functores aplicativos y mónadas

- **Mónadas:** las acciones subsecuentes pueden depender de los resultados de los acciones anteriores.

$$(\gg\!=) :: (Monad\ m) \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

- **Functores aplicativos:** los efectos son establecidos estáticamente.

$$(\texttt{<*>}) :: (Applicative\ f) \Rightarrow f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

La operación <*> es como *ap*:

$$ap :: Monad\ m \Rightarrow m\ (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$$
$$ap = liftM2\ id$$

- Toda mónada es un functor aplicativo
- No todo functor aplicativo es una mónada

Diferencia entre functores aplicativos y mónadas

La diferencia entre mónadas y functores aplicativos se puede apreciar en los siguientes operadores condicionales:

```
ifM :: Monad m => m Bool -> m a -> m a -> m a
ifM mb mt me = do b <- mb
                  if b then mt else me
```

No todas computaciones se ejecutan (se elije entre *mt* y *me*)

```
ifA :: Applicative f => f Bool -> f a -> f a -> f a
ifA fb ft fe = cond <$> fb <*> ft <*> fe
  where
    cond b t e = if b then t else e
```

Las tres computaciones (*fb*, *ft* y *fe*) se ejecutan y finalmente se elije uno de los resultados.

Leyes de functores aplicativos

Identidad:

$$pure\ id\ \texttt{<*>}\ u \equiv u$$

Composición:

$$pure\ (\ .\ )\ \texttt{<*>}\ u\ \texttt{<*>}\ v\ \texttt{<*>}\ w \equiv u\ \texttt{<*>}\ (v\ \texttt{<*>}\ w)$$

Homomorfismo:

$$pure\ f\ \texttt{<*>}\ pure\ x \equiv pure\ (f\ x)$$

Intercambio:

$$u\ \texttt{<*>}\ pure\ x \equiv pure\ (\lambda f \rightarrow f\ x)\ \texttt{<*>}\ u$$

Funciones sobre functores aplicativos

$$
\begin{aligned}
&\mathit{sequenceA} :: \mathit{Applicative}\ f \Rightarrow [f\ a] \to f\ [a] \\
&\mathit{sequenceA}\ [\,] \qquad\ = \mathit{pure}\ [\,] \\
&\mathit{sequenceA}\ (a : as) = (:) \texttt{<\$>}\ a \texttt{<*>}\ \mathit{sequenceA}\ as
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{traverse} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to [a] \to f\ [b] \\
&\mathit{traverse}\ f = \mathit{sequenceA} \mathbin{.} \mathit{fmap}\ f
\end{aligned}
$$

que equivale a:

$$
\begin{aligned}
&\mathit{traverse}\ f\ [\,] \qquad\ = \mathit{pure}\ [\,] \\
&\mathit{traverse}\ f\ (x : xs) = (:) \texttt{<\$>}\ f\ x \texttt{<*>}\ \mathit{traverse}\ f\ xs
\end{aligned}
$$

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
pSuceed   :: 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: *IO*

```
instance Applicative IO where
  pure x      = return x
  m <*> m'    = do f ← m
                   x ← m'
                   return (f x)
```

Ejemplo:

```
lectura = do xs ← getLine
             ys ← getLine
             return (xs ++ ys)
```

o en forma aplicativa:

```
lecturaA = (++) <$> getLine <*> getLine
```

Ejemplo: listas

instance *Applicative* [ ] where
  pure $x = [x]$
  fs <*> $xs = [f\ x \mid f \leftarrow fs, x \leftarrow xs]$

Ejemplo:

$$[(+1),(+2)] \text{ <*> } [1,2,3]$$

$$\rightsquigarrow$$

$$[2,3,4,3,4,5]$$

Ejemplo: *ZipList*

```
newtype ZipList a = ZipList {getZipList :: [a]}

instance Functor ZipList where
  fmap f (ZipList as) = ZipList $ map f as

instance Applicative ZipList where
  pure x = ZipList $ repeat x
  ZipList fs <*> ZipList xs = ZipList $ zipWith ($) fs xs
```

Ejemplo:

```
transpose :: [[a]] → [[a]]
transpose = getZipList . sequenceA . map ZipList
```

Ejemplo: *Failing*

```
data Failing e a = Success a | Failure e

instance Functor (Failing e) where
  fmap f (Success a) = Success (f a)
  fmap f (Failure e) = Failure e

instance Monoid e => Applicative (Failing e) where
  pure a = Success a
  Success f <*> Success a = Success (f a)
  Failure e <*> Success _ = Failure e
  Success _ <*> Failure e = Failure e
  Failure e <*> Failure e' = Failure (e `mappend` e')
```

Ejemplo: *Failing* (2)

La instancia anterior de *Applicative* para *Failing* **no** es una mónada.

$$\begin{aligned}&\text{instance } \mathit{Monoid}\ e \Rightarrow \mathit{Monad}\ (\mathit{Failing}\ e) \text{ where}\\ &\quad \mathit{return} = \mathit{Success}\\ &\quad \mathit{Failure}\ e \gg\!= f = ??\\ &\quad \ldots\end{aligned}$$

- No podemos aplicar f en este caso porque sólo se aplica cuando la primera computación retorna un valor (*Success a*).
- Esto no ocurre en la instancia de *Applicative*.
- Con esta instancia, <*> y *ap* son distintos:

$$\begin{aligned}&\mathit{Failure}\ a \mathbin{<\!*\!>} \mathit{Failure}\ b = \mathit{Failure}\ (a \mathbin{`\mathit{mappend}`} b)\\ &\mathit{Failure}\ a \mathbin{`\mathit{ap}`} \mathit{Failure}\ b = \mathit{Failure}\ a\end{aligned}$$

Ejemplo: *Failing* (3)

La siguiente instancia de Applicative iguala <*> y *ap*:

```
instance Applicative (Failing e) where
  pure = Success
  Success f <*> Success a = Success (f a)
  Success f <*> Failure e = Failure e
  Failure e <*> _         = Failure e
```

Notación propuesta para functores aplicativos

Las operaciones de functores aplicativos pueden llevarse a la forma:

$$\begin{aligned} &\mathit{pure}\ f\ \texttt{<*>}\ x_1\ \texttt{<*>}\ \ldots\ \texttt{<*>}\ x_n \\ &\qquad f\ \texttt{<\$>}\ x_1\ \texttt{<*>}\ \ldots\ \texttt{<*>}\ x_n \end{aligned}$$

McBride y Paterson proponen anotarlo:

$$[\!|\ f\ x_1\ \ldots\ x_n\ |\!]$$

Con un poco de "type-class hacking" se puede escribir:

$$\mathit{iI}\ f\ x_1\ \ldots\ x_n\ \mathit{Ii}$$

Composición

La clase de functores aplicativos es cerrada bajo la composición.

$$\text{newtype } (f \star g)\ a = \mathit{Compose}\ \{\mathit{getCompose} :: f\ (g\ a)\}$$

$$\text{instance } (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (f \star g) \text{ where}$$
$$\quad \mathit{fmap}\ f\ (\mathit{Compose}\ x) = \mathit{Compose}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ x)$$

$$\text{instance } (\mathit{Applicative}\ f, \mathit{Applicative}\ g)$$
$$\qquad \Rightarrow \mathit{Applicative}\ (f \star g) \text{ where}$$
$$\quad \mathit{pure}\ x = \mathit{Compose}\ (\mathit{pure}\ (\mathit{pure}\ x))$$
$$\quad \mathit{Compose}\ f \texttt{<*>} \mathit{Compose}\ x = \mathit{Compose}\ ((\texttt{<*>})\ \texttt{<\$>}\ f\ \texttt{<*>}\ x)$$

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