Aplicaciones de la teoría de categorías al diseño de software

Braulio Valdivielso Martínez

UGR

2018

Introducción

- El lenguaje de programación funcional haskell nace en la década de los 90 como un proyecto académico.
- Incorpora numerosas innovaciones en el campo de diseño de lenguajes de programación. Algunas de estas provienen de la teoría de categorías.
- Hoy haskell es utilizado en la industria en sectores diversos como las redes sociales o la banca.

Introducción a la teoría de categorías

1. Categorías y funtores.
2. Construcciones elementales.
3. Transformaciones naturales y Lema de Yoneda.
4. Adjunciones y Mónadas.

Patrones de Diseño

1. El patrón Categórico.
2. *Monad Transformers*.

Definición de Categoría

Definition

Una categoría $\mathcal{C}$ queda determinada por una colección de objetos $\mathcal{O}b(\mathcal{C})$, una colección de flechas $\mathcal{A}r(\mathcal{C})$ y una operación de composición $\circ$ que cumple las siguientes propiedades:

- La composición es asociativa. Dados $A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$ se cumple que $h \circ (g \circ f) = (h \circ g) \circ f)$.
- Existen identidades. Para cada objeto C en $\mathcal{C}$ existe una flecha $1_C : C \longrightarrow C$ tal que para cualquier flecha $f : X \longrightarrow C$ se cumple $1_C \circ f = f$ y para cualquier flecha $g : C \longrightarrow Y$ se cumple $g \circ 1_C = g$.

Ejemplos de Categorías

- `Set`: los objetos son los conjuntos, las flechas son las aplicaciones entre ellas y la composición es la composición habitual de aplicaciones.
- `Grp`: los objetos son los grupos, las flechas son los homomorfismos de grupos y la composición es la composición habitual de aplicaciones.
- Dado un monoide $(M, \cdot)$ podemos considerar la categoría con un solo objeto y en la que cada flecha es un elemento del monoide. La composición es la operación $\cdot$ del monoide.

Hask

En la categoría `Hask` los objetos son los tipos del lenguaje haskell y las flechas son las funciones entre esos tipos. La función `length :: String -> Int` es una flecha `String` $\longrightarrow$ `Int`.

La composición es:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) f g a = f (g a)
```

Definición de Funtor

Definition

Un funtor $F : \mathcal{C} \longrightarrow \mathcal{D}$ asigna a cada objeto C de $\mathcal{C}$ un objeto FC de $\mathcal{D}$ y a cada flecha $f : C \longrightarrow C'$ de $\mathcal{C}$ una flecha $Ff : FC \longrightarrow FC'$ sujeto a las siguientes condiciones.

- Se lleva bien con la composición, es decir dado el diagrama $A \xrightarrow{f} B \xrightarrow{g} C$ en $\mathcal{C}$ tenemos que $F(g \circ f) = Fg \circ Ff$.
- Lleva identidades en identidades. Es decir para cada objeto C de $\mathcal{C}$ se tiene que $F1_C = 1_{FC}$.

Ejemplos de Funtores

- Funtores de conjunto subyacente.
- Funtor grupo libre.
- Funtores Hom. Dado un objeto A de la categoría $\mathcal{C}$ podemos definir el funtor $\mathrm{Hom}(A, -) : \mathcal{C} \longrightarrow \mathtt{Set}$.

Funtores en Hask

En haskell tenemos la siguiente typeclass:

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

y según la documentación cualquier instancia de Functor debe cumplir:

```
fmap id == id
fmap (f . g) == fmap f . fmap g
```

Instancias de Functor: Either a

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

dividir_entre_el_primero
    :: Int -> [Int] -> Either String Int
dividir_entre_el_primero x [] =
    Left "La lista está vacía"
dividir_entre_el_primero x (0:xs) =
    Left "No se puede dividir entre 0"
dividir_entre_el_primero x (y:ys) =
    Right (x `div` ys)
```

Representa cómputos que pueden fallar.

```haskell
instance Functor (Either a) where
    -- fmap :: (b -> c) -> Either a b -> Maybe a c
    fmap f Nothing = Nothing
    fmap f (Just b) = Just (f b)

fmap (3+) (dividir_entre_el_primero 5 [])
-- Left "La lista esta vacia"

fmap (5*) (dividir_entre_el_primero 4 [4, 3])
-- Right 5
```

Instancias de Functor: Reader a

```
data Reader a b = Reader (a -> b)

data Config = Config { width :: Int, height :: Int}

area :: Reader Config Int
area = Reader (\(Config width height) ->
    width * height)

runReader :: Reader a b -> a -> b
runReader (Reader f) a = f a

runReader area (Config 3 4) -- 12
```

Reader a b envuelve funciones de tipo a -> b.

```haskell
instance Functor (Reader a) where
    fmap :: (b -> c) -> Reader a b -> Reader a c
    fmap f (Reader g) = Reader (f . g)

areaMayorQue10 :: Reader Config Bool
areaMayorQue10 = fmap (> 10) area
```

Transformaciones naturales

Definition

Sean $F, G : \mathcal{C} \longrightarrow \mathcal{D}$ dos funtores. Una transformación natural $\tau : F \Rightarrow G$ asigna a cada objeto C de $\mathcal{C}$ una flecha $\tau_C : FC \longrightarrow GC$ tal que para cada flecha $f : C \longrightarrow C'$ tenemos que el siguiente diagrama es conmutativo:

$$
\begin{tikzcd}
FC \arrow[r, "\tau_C"] \arrow[d, "Ff"'] & GC \arrow[d, "Gf"] \\
FC' \arrow[r, "\tau'_C"] & GC'
\end{tikzcd}
$$

Ejemplos de transformaciones naturales

- $\phi : 1_{\mathtt{Vect}\text{-}K} \Rightarrow (-)^{**} \quad \phi_V : V \longrightarrow V^{**}.$
- $\pi : 1_{\mathtt{Grp}} \Rightarrow (-)^{ab}, \quad \pi_G : G \longrightarrow G^{ab}.$

Transformaciones naturales en haskell

Dados dos funtores `F` y `G` en haskell cualquier función con tipo:

```
f :: F a -> G a
```

Es una transformación natural entre ambos funtores.

Ejemplos

```haskell
discardLeft :: Either l a -> Maybe a
discardLeft (Left _) = Nothing
discardLeft (Right a) = Just a

addErrorContext :: b -> (Maybe a -> Either b a)
addErrorContext _ (Just a) = Right a
addErrorContext b Nothing  = Left b

maybeToList :: Maybe a -> [a]
maybeToList (Just x) = [x]
maybeToList Nothing = []
```

Adjunciones

Definition

Sean $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{D} \longrightarrow \mathcal{C}$ dos funtores. Diremos que F y G son funtores adjuntos si existe un isomorfismo natural entre los funtores

$$\mathrm{Hom}_{\mathcal{D}}(F-, -) \cong \mathrm{Hom}_{\mathcal{C}}(-, G-)$$

Ejemplos de adjunciones

- Funtor conjunto subyacente y funtor grupo libre.
- Currificación.

Mónadas

Definition

Una mónada es una terna (T, η, μ) donde $T : \mathcal{C} \longrightarrow \mathcal{C}$ es un endofuntor, $\eta : 1_{\mathcal{C}} \Rightarrow T$ una transformación natural y $\mu : T^2 \Rightarrow T$ otra transformación satisfaciendo:

$$\begin{tikzcd} T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"] & T \end{tikzcd} \tag{1}$$

$$\begin{tikzcd} T \arrow[r, "T\eta"] \arrow[dr, equal] & T^2 \arrow[d, "\mu"] & T \arrow[l, "\eta T"'] \arrow[dl, equal] \\ & T & \end{tikzcd} \tag{2}$$

Ejemplos de mónadas

- Dado un monoide $(M, \cdot)$ y $T(X) = M \times X$, $\eta_X(x) = (e, x)$ y $\mu(m, n, x) = (m \cdot n, x)$ tenemos que (T, η, μ).
- Sea A un conjunto. Definimos:
  - el endofuntor $T = \mathrm{Hom}(A, -) : \mathtt{Set} \longrightarrow \mathtt{Set}$;
  - la transformación natural $\eta : 1_{\mathtt{Set}} \Rightarrow T$, dada por $\eta_X : X \longrightarrow \mathrm{Hom}(A, X)$ con $\eta_X(x)(a) = x$;
  - la transformación natural $\mu : T^2 \Rightarrow T$, dada por $\mu_X : \mathrm{Hom}(A, \mathrm{Hom}(A, X)) \longrightarrow \mathrm{Hom}(A, X)$ con $\mu_X(f)(a) = f(a)(a)$.

  La terna (T, η, μ) es una mónada.

Relación entre adjunciones y mónadas

- Un par funtores adjuntos inducen una mónada.
- Toda mónada es inducida por un par de funtores adjuntos (aunque no de forma única).

```haskell
class Functor m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b

    -- sujeto a
    return a >>= k = k a
    m >>= return = m
    m >>= (\x -> (k x >>= h)) = m >>= k >>= h
```

```haskell
instance Monad (Either b) where
    return x = Right x
    (Left x) >>= f = Left x
    (Right a) >>= f = f a
```

```haskell
sum_three_heads :: [Int] -> [Int] -> [Int] -> Maybe Int
sum_three_heads xs ys zs = case (head_safe xs) of
  Just x -> case (head_safe ys) of
      Just y -> case (head_safe zs) of
          Just z -> Just (x + y + z)
          Nothing -> Nothing
      Nothing -> Nothing
  Nothing -> Nothing
```

```haskell
sum_three_heads :: [Int] -> [Int] -> [Int] -> Maybe Int
sum_three_heads xs ys zs =
    head_safe xs >>= (\x ->
        head_safe ys >>= (\y ->
            head_safe zs >>= (\z ->
                return (x + y + z)
            )
        )
    )
```

```haskell
sum_three_heads :: [Int] -> [Int] -> [Int] -> Maybe Int
sum_three_heads xs ys zs = do
    x <- head_safe xs
    y <- head_safe ys
    z <- head_safe zs
    return (x + y + z)
```

Leyes de los mónadas expresadas con do notation

```
(do
  a <- return v
  k a)
= k v

(do
    a <- f x
    return a
) = f x
```

Ley 3

```
do
    x <- f a
    y <- g b
    h y
```

Ley 3

```
acc a = do
    x <- f a
    g b

do
    y <- acc a
    h y
```

Filosofía Unix

- escribe programas que hagan una sola cosa pero que la hagan bien;
- escribe programas para que puedan funcionar conjuntamente;
- escribe programas que trabajen con texto, es un formato universal.

Patrón categórico

Organizar tu librería entorno a primitivas básicas cuya composición se puede interpretar como la composición en una categoría.

- La composición permite obtener programas elaborados a partir de la combinación de primitivas sencillas.
- La asociatividad de la composición nos permite analizar un programa analizando partes por separado.
- La existencia de identidades es un caso trivial de nuestra operación de composición que la hace más fácil de entender.

`Pipes`. Tres abstracciones para el manejo de streams:

- Producers
- Consumers
- Pipes

```haskell
threeNumbers :: Producer Int IO ()
threeNumbers = do
    yield 3
    yield 4
    lift (putStrLn "Accion de IO")
    yield 5
    return ()
```

```haskell
dup :: Int -> Producer Int IO ()
dup x = do
    yield x
    yield x
```

```haskell
do
    yield 3
    yield 3
    yield 4
    yield 4
    lift (putStrLn "Accion de IO")
    yield 5
    yield 5
    return ()
```

```haskell
(~>) :: Monad m
     => (a -> Producer b m r)
     -> (b -> Producer c m r)
     -> (a -> Producer c m r)
 (f ~> g) x = for (f x) g
```

```
(dup2 ~> leng) ~> numberAndSquare
-- y
dup2 ~> (leng ~> numberAndSquare)
```