Patrones de diseño inspirados por teoría de categorías

Universidad de Granada

Braulio Valdivielso Martínez

June 8, 2018

Abstract

Abstract goes here

Dedication

To mum and dad

Declaration

I declare that..

Acknowledgements

I want to thank...

Contents

Capítulo 1

Categorías y funtores

1.1 Categorías

1.1.1 Definición

Una categoría $\mathcal{C}$ queda determinada por dos colecciones:

1. $\mathcal{O}b(\mathcal{C})$ a cuyos elementos nos referiremos como *objetos* de $\mathcal{C}$.

2. $\mathcal{A}r(\mathcal{C})$ a la que nos referiremos como las *flechas* de $\mathcal{C}$. A cada flecha se le puede asignar un par de objetos: al primero de ellos le llamaremos *origen* (o dominio) y al otro *destino* (o *codominio*). Con la notación $f : A \longrightarrow B$ estaremos afirmando que f es una flecha que tiene como origen el objeto A y como destino el objeto B. No supondremos en general que $\mathcal{A}r(\mathcal{C})$ es un conjunto pero a lo largo de este texto sí que asumiremos que $Hom(A, B) = \{f : f : A \longrightarrow B\}$ lo es. A las categorías en las que $Hom(A, B)$ son conjuntos se les suele llamar categorías localmente pequeñas. En este texto solo trataremos categorías localmente pequeñas.

Para poder hablar de categorías necesitamos también una operación de composición $\circ$ que funcione de la siguiente forma:

1. Para cada terna de objetos A, B, C de la categoría $\mathcal{C}$ diremos que los pares de flechas de $Hom(B, C) \times Hom(A, B)$ se pueden componer y tendremos que $\circ : Hom(B, C) \times Hom(A, B) \to Hom(A, C)$. Dicho de otra forma si $f : A \longrightarrow B$ y $g : B \longrightarrow C$ tenemos que $g \circ f : A \longrightarrow C$.

2. $\circ$ es asociativa en el siguiente sentido: si $f : A \longrightarrow B, g : B \longrightarrow C$, $h : C \longrightarrow D$ tenemos que $(h \circ g) \circ f = h \circ (g \circ f)$.

3. Para cada objecto C de la categoría $\mathcal{C}$ existe una flecha a la que llamaremos $1_C : C \longrightarrow C$ tal que para cualquier flecha $f : X \longrightarrow C$ tenemos que $1_C \circ f = f$ y para cualquier flecha $g : C \longrightarrow Y$ se cumple $g \circ 1_C = g$ para cualquiera que sean los objetos X, Y de $\mathcal{C}$.

Voy a reescribirte esto, creo que la forma en que se introduce una categoría da una idea de la escuela que se sigue o de la filosofía con que se afronta esta teoría. Te voy a introducir en categorías como lo haría yo. TU ELIGES LA FORMA DEFINITIVA. No haré esto con el resto del trabajo pero el comienzo es importante.

Tradicionalmente las matemáticas están fundamentadas en *una teoría de conjuntos* y bajo esta fundamentación el concepto de *conjunto* es básico. Entendemos lo que es un conjunto pero no tratamos de dar una definición formal de este. Ocurre lo mismo con los conceptos de *elemento* y *pertenece* que son básicos en esta teoría. En la actualidad se puede utilizar la *teoría de categorías* para fundamentar las matemáticas y en este sentido los conceptos de *categoría, objeto, flecha* y *composición* serían los conceptos básicos que se intentan entender sin dar una definición formal de estos. Siguiendo esta idea podemos decir que tendremos una categoría $\mathcal{C}$ si:

- Conocemos sus objetos, que denotamos $A, B, C, \ldots$.

- Conocemos sus flechas, que denotamos $f, g, h, \ldots$.

- Para cada flecha f conocemos su dominio A y su codominio B, que serán objetos de $\mathcal{C}$, escribiremos $f : A \to B$ o bien $A \xrightarrow{f} B$.

- Para cada dos *flechas componibles* $A \xrightarrow{f} B \xrightarrow{g} C$ conocemos su composición $g \circ f : A \to C$.

Todos estos datos, que determinan una categoría $\mathcal{C}$, tienen que cumplir las siguientes propiedades o axiomas:

1. La composición es asociativa, en el siguiente sentido: si $f : A \longrightarrow B, g : B \longrightarrow C$ y $h : C \longrightarrow D$ se ha de cumplir $(h \circ g) \circ f = h \circ (g \circ f)$.

2. Existen identidades, esto es: para cada objecto C existe una flecha, a la que llamaremos identidad en C y que denotaremos $1_C : C \longrightarrow C$, que cumple que para cualquiera flechas $f : X \longrightarrow C$ y $g : C \longrightarrow Y$ se tiene $1_C \circ f = f$ y $g \circ 1_C = g$.

Con esta aproximación a la teoría de categorías no haríamos uso de la teoría de conjuntos. Sin embargo vamos a optar por una aproximación *no tan categórica*. En toda esta memoria vamos a asumir que para cada par de objetos A y B, de la categoría $\mathcal{C}$, las flechas de $\mathcal{C}$ con dominio A y codominio B forman un conjunto, que denotaremos $Hom_{\mathcal{C}}(A, B)$ o simplemente $Hom(A, B)$. De manera que la composición en $\mathcal{C}$ determina aplicaciones $\circ : Hom(B, C) \times Hom(A, B) \rightarrow Hom(A, C)$ para cada terna de objetos A, B, C de $\mathcal{C}$. En este sentido, en esta memoria trataremos sólo con categorías *localmente pequeñas*. Denotaremos $\mathcal{O}b(\mathcal{C})$ y $\mathcal{A}r(\mathcal{C})$ a las *clases* de todos los objetos y todas las flechas respectivamente de $\mathcal{C}$.

Mostramos a continuación algunos ejemplos de categorías.

1.1.2 Ejemplos

Conjuntos Uno de los más típicos ejemplos de categorías es `Set`, la categoría de los conjuntos. En esta categoría cada conjunto es un objeto (no se puede asumir que $\mathcal{O}b(\mathcal{C})$ es un conjunto por ejemplos como este: no tiene sentido hablar del conjunto de todos los conjuntos) y cada aplicación f entre conjuntos con dominio el conjunto A y como codominio el conjunto B es una flecha $f : A \longrightarrow B$. La composición es la composición habitual de aplicaciones y las identidades $1_C : C \longrightarrow C$ son las aplicaciones identidad en cada conjunto C. Comprobar que se cumplen los axiomas de las categorías es una tarea rutinaria.

Otras estructuras matemáticas Gran parte de las estructuras que se estudian en matemáticas forman categorías si consideramos sus morfismos como flechas. Podemos dar multitud de ejemplos de este tipo:

- `Grp`: la categoría en la que los objetos son grupos y las flechas son los homomorfismos de grupos.
- `Top`: la categoría en la que los objetos son espacios topológicos y las flechas son funciones continuas.

- `Ring`: la categoría en la que los objetos son anillos y las flechas son homomorfismos de anillos.

La lista sigue y sigue.

Monoides Proponemos este ejemplo para evitar la asumción de que en una categoría los objetos deben ser estructuras matemáticas y las flechas entre ellos aplicaciones que preservan la estructura. Definimos una categoría con un solo objeto al que llamaremos $*$. El conjunto de flechas será $Hom(*,*) = \mathbb{Z}$ y $\circ : Hom(*,*) \times Hom(*,*) \to Hom(*,*)$ quedará definido por $f \circ g = f + g$ donde la suma es la habitual de los enteros.

Es trivial ver que los axiomas se cumplen:

1. La composición es asociativa: dadas $n, m, k : * \longrightarrow *$ (el único tipo de flechas que se puede componer, el único tipo de flechas que hay) sabemos que $n \circ (m \circ k) = n + (m + k) = (n + m) + k = (n \circ m) \circ k$.

2. Existe la identidad para cada objeto: solo existe un objeto y a su identidad la llamaremos 0. Es trivial ver que $f \circ 0 = f$ y que $0 \circ g = g$ en este contexto.

En general esta construcción que acabamos de aplicar a $(\mathbb{Z}, +)$ se puede aplicar a cualquier monoide. Toda categoría con un solo objeto se puede interpretar como un monoide (y viceversa): la asociatividad de la composición garantiza la asociatividad de la operación monoidal y la existencia de la flecha identidad garantiza la existencia del elemento neutro del monoide. En este sentido podemos considerar que las categorías son una generalización de los monoides.

Categoría producto Dado un par de categorías $\mathcal{C}$ y $\mathcal{D}$ podemos construir la categoría producto $\mathcal{C} \times \mathcal{D}$ de la siguiente forma:

- Los objetos serán de la forma (C, D) donde C es un objeto de $\mathcal{C}$ y D es un objeto de $\mathcal{D}$.

- Las flechas serán de la forma $(f, g) : (C, D) \longrightarrow (C', D')$ donde $f : C \longrightarrow C'$ es una flecha de $\mathcal{C}$ y $g : D \longrightarrow D'$ es una flecha de $\mathcal{D}$.

- La composición actúa componente a componente $(f, g) \circ (f', g') = (f \circ f', g \circ g')$ siempre que f y f', y g y g' se puedan componer.

Las identidad del objeto (C, D) es claramente la flecha $(1_C, 1_D)$. Es trivial comprobar que $\mathcal{C} \times \mathcal{D}$ cumple los axiomas de una categoría.

1.2 Funtores

1.2.1 Definición

De la misma manera que para los grupos se definen los homomorfismos de grupos, para los anillos los homomorfismos de anillos y para los espacios topológicos las funciones continuas, también podemos asociar a las categorías una noción de morfismos que preserva su estructura. A estos morfismos de categorías les llamaremos *funtores*. Un funtor F de una categoría $\mathcal{C}$ en una categoría $\mathcal{D}$ (que notaremos por $F : \mathcal{C} \longrightarrow \mathcal{D}$) tendrá que llevar objetos de $\mathcal{C}$ en objetos de $\mathcal{D}$ y flechas de $\mathcal{C}$ en flechas de $\mathcal{D}$ preservando la estructura de la categoría en el siguiente sentido:

1. F respeta los dominios y los codominios: si $f : A \longrightarrow B$ una flecha de la categoría $\mathcal{C}$ entonces $Ff : FA \longrightarrow FB$ es la correspondente flecha asociada en $\mathcal{D}$.

2. F preserva las identidades. Dicho de otra forma si C es un objeto de $\mathcal{C}$ entonces $F1_C = 1_{FC}$.

3. F respeta la composición: si tenemos $f : A \longrightarrow B$ y $g : B \longrightarrow C$ tenemos que $F(g \circ f) = Fg \circ Ff$.

Aunque la acción sobre los objetos no determina por completo a un funtor, habrá ocasiones en las que el lector podrá completar por sí mismo fácilmente la acción de este sobre las flechas. En tales casos nos limitaremos a referirnos al funtor describiendo su acción sobre los objetos.

La ley de la composición se puede interpretar como que F lleva diagramas conmutativos a diagramas conmutativos. para hacer este comentario aquí tendrías que decir lo que es un diagrama conmutativo ?¿?¿ Sí, diré qué es un diagrama conmutativo (antes de esta parte para que se entienda el comentario).

1.2.2 Ejemplos

Funtores Identidad Para cada categoría $\mathcal{C}$ podemos definir el funtor identidad $1_{\mathcal{C}} : C \longrightarrow C$ que deja invariante tanto a los objetos como a las flechas.

Comprobar las propiedades de los funtores es trivial.

Funtores subyacentes Podemos considerar el funtor $U : \mathtt{Grp} \longrightarrow \mathtt{Set}$ que asigna a cada grupo su conjunto subyacente y a cada flecha la aplicación entre los conjuntos subyacentes (cada homomorfismo de grupos es también una aplicación entre ambos conjuntos). Es rutinario comprobar que se respetan los axiomas de funtores. Existen también funtores subyacentes desde la categoría de anillos, espacios topológicos o retículos por ejemplo.

Grupos libres Podemos definir un funtor $F : \mathtt{Set} \longrightarrow \mathtt{Grp}$ de la siguiente forma: a cada conjunto X le asignamos el grupo libre sobre X (al que llamaremos FX) y a cada aplicación $f : X \longrightarrow Y$ entre conjuntos le asignamos el único homomorfismo de grupos $Ff : FX \longrightarrow FY$ que extiende a f. Comprobar que F es en efecto un funtor es sencillo.

Funtores Hom Ya hemos dicho que para cada par de objetos A, B de una categoría $\mathcal{C}$ tenemos que $\mathrm{Hom}(A, B)$ es un conjunto. Fijado un objeto A de $\mathcal{C}$ tenemos que $\mathrm{Hom}(A, -)$ nos permite asociar a cada objeto B de la categoría $\mathcal{C}$ un conjunto $\mathrm{Hom}(A, B)$.

Veamos que $\mathrm{Hom}(A, -)$ es un funtor $\mathrm{Hom}(A, -) : \mathcal{C} \to \mathtt{Set}$. Ya conocemos la acción sobre los objetos ahora tenemos que encontrar como actúa el funtor sobre las flechas. Supongamos que tenemos $f : B \longrightarrow C$ una flecha de $\mathcal{C}$. Definimos $\mathrm{Hom}(A, f) : \mathrm{Hom}(A, B) \longrightarrow \mathrm{Hom}(A, C)$ ($\mathrm{Hom}(A, f)$ es una aplicación entre los conjuntos $\mathrm{Hom}(A, B)$ y $\mathrm{Hom}(A, C)$ es decir a cada función $A \longrightarrow B$ le asigna una función $A \longrightarrow C$) por

$$\mathrm{Hom}(A, f)(g) = f \circ g$$

.

Probamos a modo de ejemplo que se cumplen los axiomas de los funtores. En primer lugar supongamos que $g : C \longrightarrow D$ y $h : D \longrightarrow E$ entonces tenemos que probar

$$\mathrm{Hom}(A, h \circ g) = \mathrm{Hom}(A, h) \circ \mathrm{Hom}(A, g) : \mathrm{Hom}(A, B) \longrightarrow \mathrm{Hom}(A, C)$$

Para probar tal cosa suponemos $f \in \mathrm{Hom}(A, B)$ y entonces:

$$\begin{aligned}\mathrm{Hom}(A, h \circ g)(f) &= (h \circ g) \circ f = h \circ (g \circ f) \\ &= h \circ \mathrm{Hom}(A, g)(f) = \mathrm{Hom}(A, h)(\mathrm{Hom}(A, g)(f)) \\ &= (\mathrm{Hom}(A, h) \circ \mathrm{Hom}(A, g))(f)\end{aligned}$$

Y por tanto se comporta bien respecto a la composición. Veamos que se comporta bien respecto a la identidad. Sea $f \in \mathrm{Hom}(A, B)$ entonces

$$\mathrm{Hom}(A, 1_B)(f) = 1_B \circ f = f$$

y por tanto $\mathrm{Hom}(A, 1_B) = 1_{\mathrm{Hom}(A,B)}$.

Teniendo en cuenta $\mathrm{Hom}(A, -)$ se comporta bien respecto a la composición y lleva identidades en identidades concluimos que es un funtor.

Bifuntores Llamamos bifuntor a un funtor de la forma $F : \mathcal{C}_1 \times \mathcal{C}_2 \longrightarrow \mathcal{D}$. Un ejemplo de bifuntor sería $- \times - : \mathtt{Set} \times \mathtt{Set} \longrightarrow \mathtt{Set}$ que a cada par de conjuntos le asigna su producto cartesiano.

Composición de funtores Dados dos funtores $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{D} \longrightarrow \mathcal{E}$ podemos definir $F \circ G : \mathcal{C} \longrightarrow \mathcal{E}$ tal que $(F \circ G)C = F(G(C))$ y $(F \circ G)(f) = F(Gf) : FGC \longrightarrow FGC'$ donde C, C' son objetos de $\mathcal{C}$ y $f : C \longrightarrow C'$ una flecha de esta categoría. $F \circ G$ es un funtor. Además esta composición de funtores es asociativa y los funtores identidad se comportan como identidades frente a esta composición.

Producto de funtores *Hablar de esto*

1.3 En programación

1.3.1 Categorías

`Hask` En el contexto del lenguaje de programación Haskell (aunque esta construcción es análoga en otros lenguajes de programación fuertemente tipados) se suele hablar de la categoría `Hask` en la que los objetos son los tipos del lenguaje (por ejemplo `Int`, `String` o `Double`) y las flechas son las funciones entre esos tipos. Por ejemplo la función `length :: String -> Int` vista en `Hask` sería una flecha `length` $\in \mathrm{Hom}_{\mathtt{Hask}}(\mathtt{String}, \mathtt{Int})$. Como operador de composición tenemos la composición habitual de funciones, que en haskell se nota con `.` (un punto) y se define de la siguiente forma:

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

Además tenemos la función `id` que nos define las flechas identidad en `Hask`:

```
id :: a -> a
id x = x
```

Nótese que aun teniendo esta colección de objetos, de flechas, la operación de composición (que es asociativa) y las identidades tenemos que `Hask` no es una categoría. Esto se debe entre otras cosas a algunas peculiaridades del comportamiento del valor especial `undefined` de Haskell. Salvando el uso de este valor, la teoría de categorías es un modelo ampliamente aceptado para el estudio de `Hask`. No presumimos en este trabajo de que `Hask` cumpla bajo toda circunstancia los axiomas de las categorías pero eso no nos impedirá utilizar la teoría de categorías para analizar y razonar sobre construcciones hechas sobre Haskell (siendo conscientes de la imperfección del modelo). Una justificación de que `Hask` es indistinguible de una categoría restringiéndose a un subconjunto del lenguaje lo encontramos en [1].

A lo largo del trabajo veremos como especializando construcciones categóricas a `Hask` obtendremos aplicaciones de la teoría de la categorías a la programación.

`Pipes` Hablar de este otro ejemplo de categorías

1.3.2 Funtores

Endofuntores en `Hask` Llamamos endofuntor a un funtor que va de una categoría $\mathcal{C}$ en sí misma. Sabiendo esto podemos comenzar a entender en qué consiste un endofuntor en `Hask`: una forma de asignar tipos a otros tipos (los objetos de `Hask`) y transformar funciones en otras funciones (las flechas de `Hask`) de manera que se preserven algunas relaciones.

Es habitual encontrar mecanismos que permiten asignar tipos a otros tipos en los lenguajes de programación usados hoy día. En `C++` tenemos como ejemplo concreto `vector` que a cada tipo (por ejemplo `int`, un entero) le asigna otro tipo (`vector<int>`, un vector de enteros). En general las templates de `C++` permiten realizar este tipo de construcciones. En `java` los Generics cumplen una función similar.

También es habitual en los lenguajes de programación modernos que las funciones sean *ciudadanos de primera clase* (*first class citizens*), es decir, se puede operar sobre las funciones como se opera sobre cualquier otro valor del lenguaje. En este contexto es natural que surjan funciones que reciben funciones como parámetro y devuelven otras funciones. Python es un ejemplo de lenguaje en el que se encuentran estas *higher order functions* (se usan tan habitualmente que hasta se incorporó en el lenguaje sintaxis específica para ellas).

En la biblioteca estándar de Haskell existe una `typeclass` (el mecanismo de polimorfismo de Haskell, que para los propósitos de este trabajo podemos suponer similar a las interfaces de java) que sirve para dotar de comportamiento funtorial a los constructores de tipo que implementemos. La typeclass se llama, convenientemente, `Functor` y se define de la siguiente manera:

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

Si queremos que nuestro constructor de tipo sea un `Functor` tendremos que implementar sobre él una función llamada `fmap` que reciba una función de tipo `a -> b` y nos devuelva una función de tipo `F a -> F b` donde `F` es nuestro constructor de tipo.

Veremos ejemplos a continuación que aclararán la situación pero si tuviéramos que trazar un paralelismo con `C++` podríamos decir que `vector` (que sería `F` del código en Haskell) sería una instancia de la typeclass `Functor` si implementáramos una función llamada `fmap` que recibe como parámetro una función que va de un tipo cualquiera `A` a un tipo cualquiera `B` y devuelve una función que va del tipo `vector<A>` (análogo a `F a` en el código en Haskell) a `vector<B>`.

Haskell no comprueba que tu implementación de `fmap` verifica los axiomas de los funtores. Esa tarea se delega al desarrollador. Los axiomas de los funtores en haskell teniendo en cuenta los operadores de composición `.` y la función identidad son:

```
; f :: a -> b
; g :: b -> c

fmap (g . f) = (fmap g) . (fmap f) :: F a -> F c

fmap id = id :: F a -> F a
```

Proponemos algunos ejemplos de instancias de la typeclass `Functor` que se encuentran en la biblioteca estándar de haskell.

Maybe La definición de `Maybe` es la siguiente:

```
data Maybe a = Just a | Nothing
```

Este tipo se usa constantemente en Haskell. Representa el resultado de computaciones que podrían fallar o podrían no tener solución en casos concretos. Podemos poner un ejemplo de utilización de este tipo: `head_safe`, una función que devuelve el primer elemento de una lista:

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

Decidimos que el tipo de retorno de `head_safe` sea `Maybe a` puesto que este cómputo puede no tener solución en caso de que la lista no tenga elementos. En otros lenguajes la función `head` lanzaría una excepción si se le pasara una lista vacía, pero en Haskell se puede codificar la naturaleza propensa a fallos del resultado en el sistema de tipos.

Resulta que se puede dotar al constructor de tipos `Maybe` de un comportamiento funtorial. Mostramos a continuación su implementación de la typeclass `Functor`

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

Lo que hace esta función `fmap` es extender funciones de tipo `a -> b` a una función de tipo `Maybe a -> Maybe b`. Esta función no hace nada si recibe un `Nothing` (representante del fallo en el cómputo) y aplica la función al contenido del valor `Just x`.

Podemos comprobar que se cumplen los axiomas de los funtores.

```
; veamos fmap id = id :: Maybe a -> Maybe a
; fmap id x = id x = x
; si x = (Just y) entonces
(fmap id) x = fmap id (Just y) = Just (id y) = (Just y) = x

; si x = Nothing
(fmap id) Nothing = Nothing
```

```
; veamos que se comporta bien con la composición.
; una vez más supongamos que x = (Just y)
(fmap (f . g)) x = fmap (f . g) (Just y)
                 = Just ( (f . g) y )
                 = Just (f (g y))
                 = (fmap f) (Just (g y))
                 = ( (fmap f) . (fmap g) ) (Just y)
                 = ( (fmap f) . (fmap g) ) x

; si x = Nothing
(fmap (f . g)) x = (fmap (f . g)) Nothing
                 = Nothing
                 = (fmap f) Nothing
                 = (fmap f) ((fmap g) Nothing)
                 = ((fmap f) . (fmap g)) Nothing
```

Either La definición de `Either` es la siguiente:

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

El tipo `Either` en haskell se usa para representar cómputos que pueden devolver valores de dos tipos distintos. Un ejemplo muy habitual de `Either` es representar cómputos que, al igual que `Maybe`, podrían ser erróneos, pero dando detalles sobre el error en caso de error.

Proponemos un ejemplo artificial que aun así muestra para qué se podría usar este tipo. Supongamos que tenemos un sistema con usuarios registrados y en nuestra empresa queremos premiar la fidelidad de nuestros usuarios en edad laboral. Supongamos además que tenemos dos tipos de premio: uno para adultos jóvenes y otro para el resto de personas en edad laboral.

Utilizando `Maybe` para resolver el problema nos quedaría un código de la siguiente forma:

```
data PremiosTrabajadores = PremiosJovenes | PremiosMayores

dar_premio :: Int -> Maybe PremiosAdultos
dar_premio age
  | age < 16 = Nothing
```

```
  | 16 <= age < 40 = Just PremiosJovenes
  | 40 <= age < 65 = Just PremiosMayores
  | 65 <= age = Nothing
```

Este código cumple su propósito de decirnos qué premio le corresponde al usuario en caso de que efectivamente le toque un premio. Sin embargo, lo que no devuelve la función es el motivo por el que el cliente no es elegible para este. Para conseguir que la función devuelva ese tipo de información podemos usar `Either`.

```
data PremiosTrabajadores = PremiosJovenes | PremiosMayores

dar_premio :: Int -> Either String PremiosAdultos
dar_premio edad
  | edad < 16 = Left "Demasiado joven para estar en edad laboral"
  | 16 <= edad < 40 = Right PremiosJovenes
  | 40 <= edad < 65 = Right PremiosMayores
  | 65 <= edad = Left "Demasiado mayor para estar en edad laboral"
```

¿Es `Either` un funtor? La respuesta es que no, porque de entrada `Either` es un constructor de tipo con dos parametros y para que un constructor de tipo sea un funtor necesitamos que solo tenga un parámetro. Entonces `Either` no es un funtor, pero resulta que `Either a` donde `a` es algún (cualquier) tipo fijo de Haskell sí es un funtor. Es decir si consideramos fijo el primer tipo `(Either a)` es un constructor de tipos que admite un tipo como parámetro y además se puede implementar una instancia de `Functor` sobre él de la siguiente forma:

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

Esta instancia de `Functor` es similar a la de `Maybe`: si el valor es de los de *error* no se hace nada con él. Si es de los valores *buenos* se transforma mediante la función `f`. Veamos que efectivamente esta instancia de `Functor` cumple con las leyes:

```
; la identidad va a la identidad:
; supongamos x = (Left y)
```

```
fmap id x = fmap id (Left y) = Left y = x

; supongamos x = Right y
fmap id x = fmap id (Right y) = Right (id y) = Right y = x

; probemos ahora que se lleva bien con la composición
fmap (f . g) (Left y) = Left y = (fmap f) (Left y)
                      = (fmap f) (fmap g (Left y))
                      = (fmap f) . (fmap g) (Left y)

fmap (f . g) (Right y) = Right ( (f . g) y )
                       = fmap f (Right (g y))
                       = (fmap f . fmap g) (Right y)
```

Veamos un ejemplo de utilización de la instancia de `Functor` de `Either a` siguiendo con el ejemplo que utilizamos antes. Imaginemos que tenemos una función que asocia los distintos premios a sus títulos. Por ejemplo:

```
titulos_premios :: PremiosTrabajadores -> String
titulos_premios PremioJovenes = "Semana de senderismo"
titulos_premios PremioMayores = "Cata de Vinos"
```

Entonces si quisiéramos una función que a partir de la edad de un usuario nos devolviera qué mensaje mostrarle en la interfaz con respecto al premio podríamos hacer lo siguiente:

```
mensaje_premio :: Int -> String
mensaje_premio edad =
  case resultado of
    (Left mensajeError) -> "Error: " ++ mensajeError
    (Right tituloPremio) -> "Enhorabuena has conseguido una " ++ tituloPremio

  where
    resultado :: Either String String
    resultado = fmap titulos_premios (dar_premio edad)
```

List

Reader Definimos `Reader` de la siguiente forma:

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

Esta definición quiere decir que un valor de *tipo* `Reader a b` es de la forma `Reader g` donde `g` es una función que recibe un valor de tipo `a` como parámetro y devuelve un valor de tipo `b`. De la misma manera que hicimos con `Either` podemos fijar la primera variable de tipo e implementar una instancia de `Functor` para `(Reader a)`:

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

Podemos probar que esta instancia de `Functor` cumple las leyes de los funtores:

```
; f :: b -> c
; g :: c -> d
; a  = (Reader h) :: (Reader a b) y entonces
; h :: a -> b

fmap (g . f) a = fmap (g. f) (Reader h)
               = Reader ( (g . f) . h)
               = Reader ( g . (f . h))
               = fmap g (Reader (f . h))
               = fmap g (fmap f (Reader h))
               = (fmap g . fmap f) (Reader h)
               = (fmap g . fmap f) a

; y por tanto fmap (g . f) = fmap g . fmap f
; en las mismas condiciones:

fmap id a = fmap id (Reader h)
          = Reader (id . h)
          = Reader h
          = a

;; y por tanto fmap id = id
```

Veremos más adelante en el trabajo las aplicaciones de `Reader`. Creemos también importante resaltar las similitudes entre `Reader` y los funtores Hom descritos en los ejemplos matemáticos de funtores. Formalizaremos esta relación más adelante en el trabajo cuando hablemos de objetos Hom internos.

Capítulo 2

Construcciones elementales

Dedicamos este capítulo al tratamiento de construcciones elementales sobre categorías. En este capítulo podremos ver cómo un marco tan general como el de la teoría de categorías permite realizar definiciones *Continuar introducción de construcciones elementales luego.*

2.1 Elementos

Cuando hablamos de conjuntos es común hablar de sus elementos. Muchas definiciones que se hacen sobre conjuntos y aplicaciones entre ellos se hacen en base a los elementos de los conjuntos involucrados. Dos ejemplos de este tipo de definiciones serían las definiciones de aplicación inyectiva y de producto cartesiano.

Definición. *Una aplicación* $f : A \longrightarrow B$ *es inyectiva si dados* $a, a' \in A$ *tenemos que* $f(a) = f(a') \implies a = a'$.

Definición. *Dados dos conjuntos* A *y* B *definimos su producto cartesiano como:*

$$A \times B = \{(a, b) : a \in A \quad b \in B\}$$

Si intentamos trasladar las construcciones que hacemos sobre los conjuntos y sus aplicaciones a categorías arbitrarias con sus objetos y sus flechas tenemos que saber cómo trasladar la noción de elemento.

Si consideramos el conjunto con un solo elemento, al que llamaremos $*$, y las aplicaciones que salen de él nos damos cuentas de que podemos identificar las aplicaciones entre el conjunto $* = \{1\}$ y un conjunto A (estas aplicaciones son de la forma $1 \mapsto a \in A$) arbitrario con los elementos (en el sentido de teoría de conjuntos) de A. Dicho de otra forma $\mathrm{Hom}(*, A)$ son lo mismo A como conjuntos. pero $\mathrm{Hom}(*, A)$ está formado por flechas y eso es algo de lo que sí podemos hablar dentro de una categoría. Sin embargo, para realizar este procedimiento de identificar los elementos de un conjunto con un conjunto de flechas de la categoría hemos tenido que acudir a un objeto especial de la categoría de conjuntos: $*$. Este procedimiento es algo que quizá no podamos realizar en otras categorías pero nos motiva a hacer una definición más general de *elemento categórico*:

Definición 1. *En una categoría $\mathcal{C}$ llamaremos elemento de un objeto A a cualquier flecha $x : T \longrightarrow A$ (sea cual sea el objeto T). De forma paralela a como se hace con conjuntos, utilizaremos la notación $x \in^T A$.*

Con esta noción de elemento notamos que $f : A \longrightarrow B$ lleva elementos de A a elementos de B mediante la composición: si $x \in^T A$ entonces $f \circ x \in^T B$. Esto motiva que en lo que sigue omitamos a veces el signo de composición (fx en lugar de $f \circ x$) si buscamos que se interpreten algunas flechas como elementos categóricos y la acción de otras flechas sobre estos.

Veamos como esta noción nos permite llevar a teoría de categorías algunos conceptos originados sobre conjuntos.

2.2 Monomorfismos

Definición 2. *Consideremos una categoría $\mathcal{C}$ y una flecha $f : A \longrightarrow B$ de esta. Diremos que f es un monomorfismo si dados dos elementos de A $x, y \in^T A$ tenemos que $fx = fy \implies x = y$.*

Nótese cómo esta definición es casi idéntica a la definición de inyectividad sobre aplicaciones entre conjuntos. De hecho es sencillo demostrar que la noción de monomorfismo sobre `Set` se corresponde efectivamente con las aplicaciones inyectivas.

Ejemplos Esta noción se puede aplicar sobre otras categorías. En general cuando consideramos categorías en la que los objetos son estructuras matemáticas y las flechas son sus morfismos, los monomorfismos son aquellas flechas tales que las aplicaciones entre conjuntos subyacentes son inyectivas. Ejemplos de esto son las categorías `Grp`, `Ring`...

2.3 Isomorfismos

Definición 3. *Consideremos una categoría $\mathcal{C}$ y una flecha $f : A \longrightarrow B$ de esta. Diremos que f es un isomorfismo si f es una biyección entre los elementos de A y los elementos de B definido sobre T para cualquier objeto T de $\mathcal{C}$.*

Esta definición es similar a la definición habitual de biyección sobre conjuntos pero generalizando los elementos de los conjuntos sobre elementos de los objeto sobre el resto los elementos de la categoría.

Nótese que de este concepto podemos dar una definición equivalente que no requiere del uso de elementos categóricos.

Proposición 1. *Dada la categoría $\mathcal{C}$ y y una flecha $f : A \longrightarrow B$ tenemos que f es un isomorfismo sí y solo sí existe una flecha $g : B \longrightarrow A$ tal que $f \circ g = 1_B$ y $g \circ f = 1_A$.*

Esta caracterización de isomorfismo nos permite ver rápidamente que los isomorfismos de la categoría `Set` se corresponden con las biyecciones y que los isomorfismos sobre `Grp`, `Ring`, `Top`... se corresponden con los isomorfismos de grupos, isomorfismos de anillos y homeomorfismos de espacios topológicos.

2.4 Productos

Otra de los ejemplos que dimos anteriormente de construcción que se realiza sobre conjuntos era el producto cartesiano. Esta definición podemos llevarla a categorías arbitrarias de la siguiente forma:

Definición 4. *Sea $\mathcal{C}$ una categoría y A y B dos objetos de esta. Diremos que existe el producto de A y B si existe una terna $(A \times B, \pi_1, \pi_2)$ donde $A \times B$ es un objeto de $\mathcal{C}$ y $\pi_1 : A \times B \longrightarrow A, \pi_2 : A \times B \longrightarrow B$ son dos flechas tales que para cualquiera elementos $x : T \longrightarrow A$ de A e $y : T \longrightarrow B$ de B, existe un único elemento $(x, y) : T \longrightarrow A \times B$ de $A \times B$ tal que $\pi_1(x, y) = x$ y $\pi_2(x, y) = y$.*

Expresaremos esto diciendo que el siguiente diagrama es conmutativo:

$$\begin{tikzcd} & T \arrow[dl, "x"'] \arrow[d, dotted, "\exists!(x,y)"] \arrow[dr, "y"] & \\ A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B \end{tikzcd}$$

Debemos resaltar dos aspectos importantes de esta definición:

- El producto de dos objetos en una categoría dada no tiene por qué existir.
- De existir un producto, este solo queda determinado salvo isomorfismo. De hecho si $(A \times B, \pi_1, \pi_2)$ es un producto de A y B y $\phi : P \longrightarrow A \times B$ es un isomorfismo entonces $(P, \pi_1 \circ \phi, \pi_2 \circ \phi)$ es otro producto de A y B.

2.4.1 Ejemplos

`Set` Dados dos conjuntos A y B siempre existe el producto categórico y además coincide (salvo el isomorfismo que comentamos antes) con el producto cartesiano de ambos conjuntos y las proyecciones canónicas. Lo demostramos a continuación:

Sea $a : T \longrightarrow A, b : T \longrightarrow B$ un par de funciones. Podemos definir la función $(a, b) : X \longrightarrow A \times B$ tal que $f(x) = (a(x), b(x))$. Esta función cumple en efecto que $\pi_1 \circ f = a$ y $\pi_2 \circ f = b$. Pero además es la única que cumple esto pues $f(x) = (\pi_1 \circ f(x), \pi_2 \circ f(x)) = (f_1(x), f_2(x))$.

Otras estructuras matemáticas A continuación mostramos ejemplos de productos categóricos en categorías conocidas:

- En `Grp` el producto categórico se corresponde con el producto directo de grupos.
- En `Top` se corresponde con el producto de espacios topológicos.
- En `Ring` se corresponde con el producto de anillos.

Categoría de cuerpos En la categoría de cuerpos no existen los productos. *Demostrar*

2.5 Objetos iniciales

Al principio de la sección le dimos un papel especial al conjunto de un solo elemento $*$. Nos gustaría caracterizar a ese objeto de la categoría `Set` con alguna propiedad estrictamente categórica. Esto es posible:

Definición 5. *Sea $\mathcal{C}$ una categoría y I un objeto. Diremos que I es un objeto inicial si dado cualquier objeto T de la categoría solo existe un elemento de I definido sobre T.*

Esto es cierto sobre $* = \{1\}$: la única flecha que llega a a $*$ desde cualquier conjunto es la que vale constantemente 1, pero una vez más esta definición se puede aplicar a otras categorías.

2.5.1 Ejemplos

En `Grp` El objeto inicial en `Grp` es el grupo trivial. Solo existe una función que vaya del grupo trivial a cualquier otro grupo y es la aplicación nula. Resulta, además, que el objeto final de la categoría es también el grupo trivial. Si en una categoría tenemos que un objeto O es a la vez inicial y final diremos que O es un objeto nulo. Esto es importante porque nos permite definir morfismos nulos.

En `Ring` En la categoría de los anillos con unidad el objeto inicial es $\mathbb{Z}$ y el único morfismo que existe de él en cualquier otro anillo R es el que lleva $1_{\mathbb{Z}}$ a 1_R. No existe objeto final en la categoría de anillos (se razona considerando la característica del supuesto anillo final).

2.6 Dualidad

El concepto de dualidad que se encuentra frecuentemente en las matemáticas puede ser analizado de forma muy general desde el punto de vista categórico. Para introducir esta noción presentamos a continuación la definición de categoría opuesta.

2.6.1 La categoría opuesta

Dada una categoría $\mathcal{C}$ podemos construir una categoría $\mathcal{C}^{op}$ en la que definimos $\mathcal{O}b(\mathcal{C}^{op}) = \mathcal{O}b(\mathcal{C})$, $\mathrm{Hom}_{\mathcal{C}^{op}}(A,B) = \mathrm{Hom}_{\mathcal{C}}(B,A)$ y una operación de composición dad por $f \circ_{op} g = g \circ f$. Veamos la construcción realizada describe en efecto una categoría:

- La composición es asociativa: sean $f \in \mathrm{Hom}_{\mathcal{C}^{op}}(A,B)$, $g \in \mathrm{Hom}_{\mathcal{C}^{op}}(B,C)$ y $h \in \mathrm{Hom}_{\mathcal{C}^{op}}(C,D)$. Tenemos que

$$\begin{aligned}(h \circ_{op} g) \circ_{op} f &= (g \circ h) \circ_{op} f \\ &= f \circ (g \circ h) = (f \circ g) \circ h = (g \circ_{op} f) \circ h = h \circ_{op} (g \circ_{op} f)\end{aligned}$$

- Existen las identidades. Las identidades siguen siendo las mismas flechas que son identidad en $\mathcal{C}$. La demostración es inmediata.

La categoría opuesta nos otorga una herramienta para reaprovechar definiciones realizadas anteriormente. Veremos ejemplos de propiedades duales en las siguientes secciones.

2.6.2 Epimorfismos

En una sección anterior especificamos cuando una flecha $f : A \longrightarrow B$ de una categoría $\mathcal{C}$ era un monomorfismo. La categoría opuesta nos permite dualizar esta definición de la siguiente manera:

Definición 6. *Dada una categoría $\mathcal{C}$ y una flecha $f : A \longrightarrow B$, diremos que f es un epimorfismo si y solo si f es un monomorfismo en $\mathcal{C}^{op}$.*

En este sentido decimos que las propiedades "f es un monomorfismo" y "f es un epimorfismo" son propiedades duales. Esta propiedad se puede enunciar de forma sencilla también sin recurrir a $\mathcal{C}^{op}$.

Proposición 2. *f es un epimorfismo si y solo si dado cualquier objeto X y cualquier par de flechas $g_1, g_2 : B, X \longrightarrow$ tenemos que $g_1 \circ f = g_1 \circ f \implies g_1 = g_2$. Decimos también en este caso que f se puede cancelar por la derecha.*

2.6.2.1 Ejemplos

En `Set` En la categoría de los conjuntos los epimorfismos coinciden con las aplicaciones sobreyectivas.

Isomorfismos Todo isomorfismo es un epimorfismo y un monomorfismo. El recíproco es cierto en algunas categorías (como en `Set` o `Grp`) pero no lo es en general.

En `Ring` Consideremos la categoría de anillos con unidad (en la que los objetos son anillos con unidad y las flechas son homomorfismos de anillos). En esta categoría que $f : R \longrightarrow S$ es equivalente también a que f sea una aplicación inyectiva.

Sin embargo consideremos la inclusión $i : \mathbb{Z} \longrightarrow \mathbb{Q}$, un anillo R y un par de aplicaciones de anillos $g_1, g_2 : \mathbb{Q} \longrightarrow R$. Supongamos que $g_1 \circ f = g_2 \circ f$. Ahora $\forall x \in \mathbb{Q}$ tenemos que $\exists a, b \in \mathbb{Z} : x = \frac{a}{b}$ y entonces:

$$g_1(x) = g_1(\frac{a}{b}) = g_1(a)g_1(b)^{-1}$$
$$= (g_1 \circ i)(a)(g_1 \circ i)(b)^{-1} = (g_2 \circ i)(a)(g_2 \circ i)(b)^{-1} = g_2(a)g_2(b)^{-1} = g_2(x)$$

Con lo que $g_1 = g_2$. Esto prueba que $i : \mathbb{Z} \longrightarrow \mathbb{Q}$ es epimorfismo y sin embargo no es sobreyectiva como aplicación. Un isomorfismo en la categoría de anillos se corresponde con un isomorfismo de anillos (que en particular es una biyección de conjuntos) y por tanto en la categoría de anillos se pueden encontrar flechas que son monomorfismos y epimorfismos pero no son isomorfismos.

2.6.3 Objetos finales

Existe también una propiedad dual a la de ser objeto inicial.

Definición 7. *Dado un objeto T de una categoría C decimos que T es un objeto final si T es un objeto inicial en la categoría C^{op}.*

Podemos probar una caracterización que no hace referencia a $\mathcal{C}^{op}$:

Proposición 3. *El objeto T es final si y solo si para cualquier otro objeto X existe una sola flecha tiene a X como dominio y a T como codominio.*

2.6.3.1 Ejemplos

En `Set` En `Set` el conjunto vacío $\emptyset$ es un objeto final.

En `Grp` El objeto final en `Grp` es el grupo de un solo elemento. Notemos que coincide con el objeto inicial. Cuando en una categoría $\mathcal{C}$ coinciden el objeto inicial y el final T llamamos a T objeto nulo. El objeto nulo nos permite definir una aplicación nula (*explicar esto mejor*).

Otro ejemplo de categoría en la que ocurre esto es en la de espacios vectoriales sobre un cuerpo K.

En `Ring` *Probar que no existen objetos finales en la categoría de anillos*

2.6.4 Coproducto

Podemos repetir la definición de producto categórico sobre la categoría opuesta y *dando la vuelta* a las flechas para volver a la categoría inicial $\mathcal{C}$ llegaríamos a la siguiente definición de lo que llamaremos **coproducto**:

Definición 8. *Sea $\mathcal{C}$ una categoría y A y B dos objetos de esta. Diremos que existe el coproducto de A y B si existe una terna $(A+B, i_1, i_2)$ donde P es un objeto de $\mathcal{C}$ y $i_1 : P \longrightarrow A, i_2 : P \longrightarrow B$ son dos flechas tales que para cualquier objeto Y y sendas flechas $f_1 : A \longrightarrow Y$, $f_2 : B \longrightarrow Y$ existe un morfismo $f : P \longrightarrow Y$ tal que el siguiente diagrama es conmutativo:*

$$\begin{tikzcd}
& Y \arrow[dr, "f_2"] & \\
A \arrow[ur, "f_1"] \arrow[r, "i_1"] & P \arrow[u, dotted, "\exists! f"'] & B \arrow[l, "i_2"]
\end{tikzcd}$$

Retocar esta sección para que sea más consistente con las otras Hablar de que es único salvo isomorfismo Arreglar el diagrama

2.6.5 Funtores $\mathrm{Hom}(-, A)$

Sea $\mathcal{C}$ una categoría y A un objeto de esta. Sea $f \in \mathrm{Hom}_{\mathcal{C}^{op}}(C, C')$. Definimos:

$$\mathrm{Hom}_{\mathcal{C}}(f, A) : \mathrm{Hom}_{\mathcal{C}}(C, A) \longrightarrow \mathrm{Hom}_{\mathcal{C}}(C', A)$$
$$\mathrm{Hom}_{\mathcal{C}}(f, A)(g) = g \circ f$$

(recordando que $g : C \longrightarrow A$ y $f : C' \longrightarrow C$ como flechas de $\mathcal{C}$ y por tanto se pueden componer)

Se cumple además que:

- $\mathrm{Hom}_{\mathcal{C}}(1_C, A)(g) = g \circ 1_C = g$ para toda flecha $g \in \mathrm{Hom}_{\mathcal{C}}(C, A)$ y por tanto $\mathrm{Hom}_{\mathcal{C}}(1_C, A) = 1_{\mathrm{Hom}_{\mathcal{C}}(C,A)}$ para todo objeto C de $\mathcal{C}^{op}$.

- Sean $f \in \mathrm{Hom}_{\mathcal{C}^{op}}(C, D)$ y $g : \mathrm{Hom}_{\mathcal{C}^{op}}(D, E)$ entonces

$$\begin{aligned}\mathrm{Hom}_{\mathcal{C}}(g \circ_{op} f, A)(h) &= h \circ (g \circ_{op} f) \\ h \circ (f \circ g) &= (h \circ f) \circ g = \mathrm{Hom}_{\mathcal{C}}(g, A)(h \circ f) \\ &= \mathrm{Hom}_{\mathcal{C}}(g, A)(\mathrm{Hom}_{\mathcal{C}}(f, A)(h))\end{aligned}$$

  para toda $h \in \mathrm{Hom}_{\mathcal{C}}(C, A)$ y por tanto $\mathrm{Hom}_{\mathcal{C}}(g \circ_{op} f, A) = \mathrm{Hom}_{\mathcal{C}}(g, A) \circ \mathrm{Hom}_{\mathcal{C}}(f, A)$.

Esto nos muestra que $\mathrm{Hom}_{\mathcal{C}}(-, A)$ es un funtor $\mathrm{Hom}_{\mathcal{C}}(-, A) : \mathcal{C}^{op} \longrightarrow$ `Set`, lo que no resulta nada sorprendente teniendo en cuenta que $\mathrm{Hom}_{\mathcal{C}}(-, A) = \mathrm{Hom}_{\mathcal{C}^{op}}(A, -)$ (que ya vimos que era un funtor) tanto sobre los objetos como sobre las flechas. Más aun, se puede demostrar que $\mathrm{Hom}_{\mathcal{C}}(-, -) : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow$ `Set` es un bifuntor.

Cuando queremos hacer énfasis a que un funtor está actuando sobre la categoría opuesta a una particular, decimos que es un funtor contravariante. Por ejemplo, diríamos que $\mathrm{Hom}(-, A)$ es un funtor contravariante.

2.7 En programación

A lo largo de esta sección aplicamos las construcciones que hemos visto a la categoría `Hask`.

2.7.1 Objetos iniciales y finales

`Hask` tiene tanto objetos iniciales como objetos finales. El objeto final es el tipo con un solo valor, al que se le suele llamar `()` (pronunciado `Unit`). `()` es un tipo cuyo único valor es `()` y para cada tipo `a` podemos definir la función:

```
constUnit :: a -> ()
constUnit x = ()
```

Por otro lado tenemos que el objeto inicial de `Hask` es el tipo llamado `Void`, que no tiene ningún valor. En la biblioteca estándar se encuentra una función llamada `absurd` que tiene como tipo `Void -> a` y es la única función con este tipo. En cualquier caso, la función no puede ser llamada puesto que no hay ningún valor con el que llamarla. La situación tanto como para el objeto inicial como para el final es muy similar a la que se daba en `Set`.

2.7.2 Productos

`Hask` tiene productos. Dado dos tipos `A` y `B` podemos construir el tipo `(A, B)` que tiene como valores pares de valores de `A` y de `B`. Las proyecciones se implementan en la biblioteca estándar de Haskell de la siguiente forma:

```
fst :: (a, b) -> a
fst (x, y) = x

snd :: (a, b) -> b
fst (x, y) = y
```

Si tenemos un par de funciones `f1 :: X -> A` y `f2 :: X -> B` podemos definir la función `f`:

```
f :: X -> (A, B)
f x = (f1 x, f2 x)
```

2.7.3 Coproductos

En la sección sobre funtores introdujimos el tipo `Either`:

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

Resulta que `Either` es el coproducto en `Hask`. Si `A` y `B` son dos tipos de haskell entonces su coproducto es `Either A B` y las inyecciones canónicas son por un lado `Left :: A -> Either A B` y por otro lado `Right :: B -> Either A B`. Si tenemos un par de funciones `f1 :: A -> Y` y `f2 :: B -> Y` tenemos que la única función `f :: Either A B -> Y` que se lleva bien con las inyecciones es:

```
f :: Either A B -> Y
f (Left a) = f1 a
f (Right b) = f2 b
```

Veamos que se lleva bien con las inyecciones:

```
(f . Left) a = f (Left a) = f1 a
;; por tanto f . Left = f1

(f . Right) b = f (Right b) = f2 b
;; por tanto f . Right = f2
```

Capítulo 3

Transformaciones Naturales y el lema de Yoneda

3.1 Transformaciones Naturales

En el primer capítulo introdujimos los funtores como los morfismos entre las categorías. Ahora ha llegado el momento de dar un paso más e introducir las transformaciones naturales como morfismos entre funtores. Esta es una de las nociones que motivó la creación de la teoría de categorías: se encuentran ejemplos de esta en múltiples ramas de las matemáticas.

Procedemos con la definición:

Definición 9. *Dados dos funtores $F : \mathcal{C} \longrightarrow \mathcal{D}, G : \mathcal{C} \longrightarrow \mathcal{D}$ decimos que $\lambda : F \longrightarrow G$ es una transformación natural si λ asigna a cada objeto C de $\mathcal{C}$ una flecha $\lambda_C : FC \longrightarrow GC$ de manera que para cualquier flecha $g : C \longrightarrow C'$ el siguiente diagrama es conmutativo:*

$$\begin{tikzcd}
FC \arrow[r, "\lambda_C"] \arrow[d, "Fg"'] & GC \arrow[d, "Gg"] \\
FC' \arrow[r, "\lambda_{C'}"] & GC'
\end{tikzcd}$$

Los siguientes resultados nos permitirán considerar categorías en las que los objetos son funtores entre dos categorías $\mathcal{C}$ y $\mathcal{D}$ y las flechas son las transformaciones naturales entre los funtores:

Proposición 4. 1. *Dado cualquier funtor $F : \mathcal{C} \longrightarrow \mathcal{D}$ podemos definir $(1_F)_C = 1_{(FC)} : FC \longrightarrow FC$. 1_F es una transformación natural entre $1_F : F \longrightarrow F$.*

2. *Podemos componer transformaciones naturales de la siguiente forma: dados funtores $F, G, H : \mathcal{C} \longrightarrow \mathcal{D}$ y transformaciones naturales $\lambda : F \longrightarrow G$, $\sigma : G \longrightarrow H$ podemos definir $\sigma \circ \lambda$ dada por sus componentes $(\sigma \circ \lambda)_C = \sigma_C \circ \lambda_C$. $\sigma \circ \lambda$ es una transformación natural $\sigma \circ \lambda : F \longrightarrow H$.*

3. *La composición de transformaciones naturales es asociativa en el siguiente sentido: dados $F \xrightarrow{\lambda} G \xrightarrow{\sigma} H \xrightarrow{\tau} I$ tenemos que $(\tau \circ \sigma) \circ \lambda = \tau \circ (\sigma \lambda)$*

4. *Dado cualquier par de funtores $F, G : \mathcal{C} \longrightarrow \mathcal{D}$ y transformaciones naturales $\tau : F \longrightarrow G$, $\sigma : G \longrightarrow F$ tenemos que $1_F \circ \sigma = \sigma$ y $\tau \circ 1_F = \tau$.*

En definitiva los funtores $\mathcal{C} \longrightarrow \mathcal{D}$ y las transformaciones naturales entre estos funtores forman una categoría. A esa categoría la llamaremos $[\mathcal{C}, \mathcal{D}]$.

3.1.1 Ejemplos

El doble dual Consideremos la categoría de espacios vectoriales sobre un cuerpo K a la que llamaremos $\texttt{Vect-}K$. Podemos considerar los endofuntores identidad $1_{\texttt{Vect-}K} : \texttt{Vect-}K \longrightarrow \texttt{Vect-}K$ y el funtor doble dual: $(-)^{**} : \texttt{Vect-}K \longrightarrow \texttt{Vect-}K$. El último de estos funtores actúa así sobre los morfismos:

$$(-)^{**} : \mathrm{Hom}(V, W) \longrightarrow \mathrm{Hom}(V^{**}, W^{**})$$
$$f^{**}(g)(\phi) = g(\phi \circ f)$$

Para cada espacio vectorial sobre K V podemos además definir un monomorfismo de espacios vectoriales (isomorfismo en el caso finito-dimensional) en su doble dual:

$$i_V : V \longrightarrow V^{**}$$
$$i_V(v)(\phi) = \phi(v)$$

Pues i es una transformación natural entre 1_K y $(-)^{**}$. Más aun, si nos restringimos al caso finito dimensional, debido a que todas las componentes de la transformación natural son isomorfismos, decimos que ambos funtores son naturalmente isomorfos.

Abelianización de grupos Dado un grupo G definimos su abelianización G^{ab} como $G^{ab} = \frac{G}{[G,G]}$. Llamemos $\pi_G : G \longrightarrow G^{ab}$ a la proyección sobre el cociente. Para cualquier morfismo de grupos $f : G \longrightarrow H$ tenemos que la aplicación $\pi_H \circ f$ contiene en su nucleo a $[G, G]$ (ya que H^{ab} es abeliano y un morfismo de un grupo a un grupo abeliano lleva los conmutadores al 0) y por tanto factoriza a través de G^{ab} permitiéndonos definir una aplicación $f^{ab} : G^{ab} \longrightarrow H^{ab}$ y comprobar que $(-)^{ab}$ es un funtor. La proyección de π es una transformación natural $\pi : 1_{\mathtt{Grp}} \longrightarrow (-)^{ab}$.

3.2 Lema de Yoneda

El lema de Yoneda es uno de los primeros resultados inesperados que surgen a raíz de la teoría de categorías. Una de las consecuencias más importantes de este resultado es el *embebimiento* (cambiar) de Yoneda, que nos permite ver una categoría $\mathcal{C}$ cualquiera dentro de la categoría de funtores $[\mathcal{C}, \mathtt{Set}]$. Necesitaremos algunos resultados previos para enunciar el lema:

Proposición. *Dada una categoría $\mathcal{C}$ y un objeto suyo A,*

$$\mathrm{Nat}(\mathrm{Hom}(A, -), -) : \mathcal{C} \times [\mathcal{C}, \mathtt{Set}] \longrightarrow \mathtt{Set}$$

es un bifuntor donde $\mathrm{Nat}(F, G)$ *es el conjunto de transformaciones naturales entre los funtores F y G.*

Proof. Consideramos el funtor identidad $1_{[\mathcal{C},\mathtt{Set}]} : [\mathcal{C}, \mathtt{Set}] \longrightarrow [\mathcal{C}, \mathtt{Set}]$ y el funtor $\mathrm{Hom}(*, -) : C \longrightarrow [\mathcal{C}, \mathtt{Set}]^{op}$ (demostrar) el producto de estos nos define un funtor:

$$\mathrm{Hom}(*, -) \times 1_{[\mathcal{C},\mathtt{Set}]} : \mathcal{C} \times [\mathcal{C}, \mathtt{Set}] \longrightarrow [\mathcal{C}, \mathtt{Set}]^{op} \times [\mathcal{C}, \mathtt{Set}]$$

pero sabemos que

$$\mathrm{Nat}(-, -) = \mathrm{Hom}_{[\mathcal{C},\mathtt{Set}]}(-, -) : [\mathcal{C}, \mathtt{Set}]^{op} \times [\mathcal{C}, \mathtt{Set}] \longrightarrow \mathtt{Set}$$

es un funtor. La composición de estos dos funtores que hemos dicho es el funtor que buscamos. □

Necesitaremos probar que podemos definir el siguiente funtor:

Proposición. *Sea $\mathcal{C}$ una categoría y $Ev(-, -) : \mathcal{C} \times [\mathcal{C}, \mathtt{Set}] \longrightarrow \mathtt{Set}$ dado por $Ev(C, F) = FC$. E es un funtor y lo llamamos* ***funtor de evaluación.***

Proof. Sean $\sigma : F \longrightarrow G$ y $\tau : G \longrightarrow H$ transformaciones naturales entre funtores $F, G, H : \mathcal{C} \longrightarrow \texttt{Set}$. Sean además $f : C \longrightarrow D$, $g : D \longrightarrow E$ flechas de $\mathcal{C}$. Podemos definir la acción de Ev sobre las flechas de $\mathcal{C} \times [\mathcal{C}, \texttt{Set}] \longrightarrow \texttt{Set}$ como $Ev(f, \sigma) = \sigma_D \circ Ff : FC \longrightarrow GD$. Veamos que E se comporta bien respecto a las composiciones:

$$\begin{aligned} Ev(g \circ f, \tau \circ \sigma) = (\tau \circ \sigma)_E \circ F(g \circ f) = \tau_E \circ \sigma_E \circ Fg \circ Ff = \\ \tau_E \circ Gg \circ \sigma_D \circ Ff = \tau_E \circ Gg \circ Ev(f, \sigma) = \\ Ev(g, \tau) \circ Ev(f, \sigma) \end{aligned}$$

El hecho de que $Ev(1_C, 1_F) = 1_{FC}$ se prueba de forma sencilla y por tanto $Ev : \mathcal{C} \times [\mathcal{C}, \texttt{Set}] \longrightarrow \texttt{Set}$ es un funtor

Explicar mejor la demostración □

Teorema 1 (Lema de Yoneda)**.** *Sea $\mathcal{C}$ una categoría, A un objeto de esta. Los funtores*

$$(A, F) \mapsto \mathrm{Nat}(\mathrm{Hom}(A, -), F)$$

$$(A, F) \mapsto FA$$

son naturalmente isomorfos.

Proof. Sea $a \in FA$. Definimos la transformación natural entre $\lambda_a : \mathrm{Hom}(A, -) \longrightarrow F$ componente a componente de la siguiente forma: $(\lambda_a)_C(f) = f(a)$. *Continuar mañana* □

Capítulo 4

Chapter Four Title

Capítulo 5

Conclusion

Bibliography

[1] Patrik Jansson Nils Anders Danielsson John Hughes and Jeremy Gibbons. “Fast and Loose Reasoning is Morally Correct”. In: (POPL ’06 Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages).

Appendix A

Appendix Title