UNIVERSIDAD DE GRANADA

Trabajo Fin de Grado

Grado en Matemáticas

Patrones de diseño inspirados por teoría de categorías

Universidad de Granada

Autor
Braulio Valdivielso Martínez

Tutor: María Burgos Navarro
Didáctica de la Matemática

Pedro A. García Sánchez
Álgebra

Junio 2018

Índice general

Abstract

Abstract goes here

Dedication

To mum and dad

Declaration

I declare that..

Acknowledgements

I want to thank...

Índice general

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 $\circ : Hom(B, C) \times Hom(A, B) \to Hom(A, C)$ 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.

1.1.1.1. 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.

1.1.2. 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 $f : A \longrightarrow B$ es 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.

A lo largo del trabajo mostraremos diagramas en la que los nodos son objetos de una categoría y las aristas dirigidas que los unen son flechas. Diremos que el diagrama es conmutativo cuando para cada par de objetos C y C' del diagrama la composición de las flechas que conforman un camino entre C y C' es independiente del camino utilizado. Un ejemplo sería el siguiente diagrama:

$$\begin{tikzcd} FA \arrow[r, "Ff"] \arrow[dr, "F(g\circ f)"'] & FB \arrow[d, "Fg"] \\ & FC \end{tikzcd}$$

Donde afirmar que el diagrama es conmutativo se traduce en que $Fg \circ Ff = F(g \circ f)$. Según McLane en [2]

> Una parte considerable de la efectividad de los métodos categóricos reside en el hecho de que los diagramas conmutativos en representan muy fielmente las acciones de las flechas en un contexto determinado.

1.2.1.1. Ejemplos

Funtores subyacentes a `Set` Podemos considerar el funtor $U : \texttt{Grp} \longrightarrow \texttt{Set}$ 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 $F : \texttt{Set} \longrightarrow \texttt{Grp}$ 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 $\mathrm{Hom}(A, -) : \mathcal{C} \to \texttt{Set}$. 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.

1.2.2. Funtores fieles y plenos

Dado un funtor $F : \mathcal{C} \longrightarrow \mathcal{D}$, diremos que es fiel (resp. pleno) si para cada par de objetos C y C' de $\mathcal{C}$ tenemos que la aplicación inducida $F : \mathrm{Hom}_{\mathcal{C}}(C,C') \longrightarrow \mathrm{Hom}_{\mathcal{D}}(FC,FC')$ es inyectiva (resp. sobreyectiva).

1.2.3. Funtores Identidad

Dada una categoría $\mathcal{C}$ definimos el funtor $1_{\mathcal{C}} : \mathcal{C} \longrightarrow \mathcal{C}$ donde $1_{\mathcal{C}}C = C$ para todo objeto C de $\mathcal{C}$ y $1_{\mathcal{C}}f = f$ para cada flecha $f : C \longrightarrow C'$ de $\mathcal{C}$.

1.2.4. Composición de funtores

Dados dos funtores $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{D} \longrightarrow \mathcal{E}$ definimos $F\circ G : \mathcal{C} \longrightarrow \mathcal{E}$ como $(F\circ G)C = F(G(C))$ y $(F\circ G)(f) = F(Gf) : FGC \longrightarrow FGC'$ donde C, C' son objetos de $\mathcal{C}$ y $f : C \longrightarrow C'$. $F\circ G$ es un funtor. La composición de funtores cumple las siguientes propiedades:

- Es asociativa. Dado los funtores $\mathcal{C} \xrightarrow{F} \mathcal{C}' \xrightarrow{G} \mathcal{D} \xrightarrow{H} \mathcal{D}'$ tenemos que $(H\circ G)\circ F = H\circ(G\circ F)$.
- Para cualquier funtor $F : \mathcal{C} \longrightarrow \mathcal{D}$ se cumple que $F \circ 1_{\mathcal{C}} = F = 1_{\mathcal{D}} \circ F$.

Estas propiedades nos permiten considerar categorías en las que los objetos son categorías y las flechas son los funtores entre ellas.

1.2.5. 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.

1.2.6. Producto de funtores

Dados dos funtores $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{C}' \longrightarrow \mathcal{D}'$ podemos definir el funtor $F \times G : \mathcal{C} \times \mathcal{C}' \longrightarrow \mathcal{D} \times \mathcal{D}'$ de forma que:

- $(F \times G)(C, D) = (FC, GD)$ donde C es un objeto de $\mathcal{C}$ y D es un objeto de $\mathcal{D}$.
- $(F \times G)(f, g) = (Ff, Gg)$ donde f es una flecha de $\mathcal{C}$ y g es una flecha de $\mathcal{D}$.

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` $\in \mathrm{Hom}_{\texttt{Hask}}(\texttt{String}, \texttt{Int})$. 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 [4].

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 [5]).

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` [1] 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 algunas construcciones y definiciones elementales sobre categorías. Veremos algunos ejemplos en los que definiciones puramente categóricas nos permiten capturar nociones específicas de estructuras matemáticas como los grupos o los espacios topológicos. Además, introduciremos la categoría opuesta como herramienta para razonar sobre la dualidad.

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 conjuntos y las aplicaciones entre ellos a construcciones sobre categorías arbitrarias será útil saber cómo trasladar la noción de elemento.

Si consideramos el conjunto con un solo elemento, al que llamaremos $* = \{x\}$, y las aplicaciones que salen de él nos daremos cuenta de que podemos identificar las aplicaciones $* \longrightarrow A$ (estas aplicaciones son de la forma $x \mapsto a \in A$) con los elementos de A. Dicho de otra forma $\mathrm{Hom}(*, A)$ y A son *"lo mismo"* como conjuntos, pero $\mathrm{Hom}(*, A)$ está formado por flechas y eso es algo de lo que sí podemos hablar en el contexto de una categoría arbitraria. 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 $x \in^T A$. Diremos también que x es un elemento de A definido sobre T.*

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 o $f(x)$ en lugar de $f \circ x$) si nuestra intención es que algunas flechas (en este caso x) se interpreten como elementos categóricos.

Veremos 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 para cualquier objeto T de $\mathcal{C}$ y dados dos elementos $x, y \in^T A$ 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. *Dada la categoría $\mathcal{C}$ y y una flecha $f : A \longrightarrow B$ diremos que f es un isomorfismo si existe una flecha $g : B \longrightarrow A$ tal que $f \circ g = 1_B$ y $g \circ f = 1_A$.*

Nótese que esta definición la podemos caracterizar en términos de biyecciones sobre conjuntos de elementos.

Proposición 1. *Sean $\mathcal{C}$ una categoría y $f : A \longrightarrow B$ una flecha de esta. f es un isomorfismo si y solo si f es una biyección entre los elementos de A y los elementos de B definidos sobre T para cualquier objeto T de $\mathcal{C}$.*

Esta caracterización es similar a la definición habitual de biyección sobre conjuntos pero generalizando los elementos de los conjuntos a elementos de los objetos definidos sobre todos los objetos de la categoría.

Es inmediato ver 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

También es posible trasladar la definición de producto de conjuntos a una categoría arbitraria mediante la siguiente definición:

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 cualesquiera 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 de estos y además coincide (salvo el isomorfismo que comentamos antes) con el producto cartesiano de ambos junto a sus proyecciones canónicas. Desmotramos tal afirmación a continuación:

Sean $x : T \longrightarrow A, y : T \longrightarrow B$ dos aplicaciones. Podemos definir la aplicación $(x, y) : T \longrightarrow A \times B$ por $(x, y)(t) = (x(t), y(t))$. Esta aplicación cumple en efecto que $\pi_1 \circ (x, y) = x$ y $\pi_2 \circ y = y$. Pero además es la única que cumple esto pues si tenemos otra aplicación $f : T \longrightarrow A \times B$ cumpliendo $\pi_1 \circ f = x$ y $\pi_2 \circ f = y$ solo nos basta recordar que podemos escribir f como: $f(t) = (\pi_1 \circ f(t), \pi_2 \circ f(t)) = (x(t), y(t))$. y por tanto $f = (x, y)$.

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

- En `Grp` el producto 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.

2.5. Objetos finales

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 $*$ un objeto. Diremos que $*$ es un objeto final si dado cualquier objeto T de la categoría existe un único elemento de $*$ definido sobre T.*

Comprobar que de existir el objeto $*$ es único salvo isomorfismo es inmediato.

2.5.1. Ejemplos

En `Set` En la categoría de los conjuntos el objeto final es el $*$, el conjunto de un solo elemento.

En `Grp` El grupo trivial es el objeto final de la categoría `Grp`. Esto es sencillo de comprobar: dado cualquier grupo G tenemos que el homomorfismo trivial $G \longrightarrow *$ es el único morfismo entre G y $*$.

En `Ring` En la categoría de los anillos con unidad el objeto final es también el anillo trivial.

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

Definición 6. *Dada una categoría $\mathcal{C}$ definimos $\mathcal{C}^{op}$ (a la que llamaremos categoría opuesta a $\mathcal{C}$) como la categoría determinada por $\mathcal{O}b(\mathcal{C}^{op}) = \mathcal{O}b(\mathcal{C})$, $\mathrm{Hom}_{\mathcal{C}^{op}}(A, B) = \mathrm{Hom}_{\mathcal{C}}(B, A)$ y una operación de composición dada por $f \circ_{op} g = g \circ f$.*

Comprobar que esta construcción es efectivamente una categoría en sencillo:

- 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 en $\mathcal{C}^{op}$ son las mismas flechas que en la categoría original $\mathcal{C}$.

La categoría opuesta nos otorga una herramienta para reaprovechar definiciones realizadas anteriormente. Dada una propiedad P que se pueda o no cumplir en una categoría $\mathcal{C}$ llamamos propiedad dual de P a la propiedad de que se cumpla P en la categoría opuesta $\mathcal{C}^{op}$. Esto nos permite definir las siguientes propiedades:

- Ser epimorfismo es la propiedad dual de ser monomorfismo (es decir f es un epimorfismo en $\mathcal{C}$ si f es un monomorfismo en la categoría opuesta $\mathcal{C}^{op}$)

- La propiedad dual de ser isomorfismo es ella misma. La propiedad dual de ser único salvo isomorfismo es ella misma.

- El coproducto es el dual del producto, es decir llamaremos a $(A+B, i_1, i_2)$ el coproducto de A y B en $\mathcal{C}$ si $(A+B, i_1, i_2)$ es el producto de A y B en $\mathcal{C}^{op}$.

- La propiedad de ser objeto inicial es la dual a la de ser objeto final.

Merece la pena resaltar que $\mathcal{C}^{opop} = \mathcal{C}$ y por tanto la "propiedad dual a la propiedad dual a P" es la misma propiedad P. Esto nos permite decir también, por ejemplo, que la propiedad dual a ser epimorfismo es ser monomorfismo.

Comentamos estas propiedades en mayor profundidad en las siguientes secciones.

2.6.2. Epimorfismos

Como ya hemos dicho, la definición de epimorfismo es la siguiente.

Definición 7. *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 $\mathcal{C}^{op}$.*

Esta propiedad se puede caracterizar de forma sencilla desde dentro de $\mathcal{C}$.

Proposición 2. *$f : A \longrightarrow B$ es un epimorfismo si y solo si dado cualquier objeto X y cualquier par de flechas $g_1, g_2 : B \longrightarrow X$ tenemos que $g_1 \circ f = g_2 \circ f \implies g_1 = g_2$. En este caso diremos también 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 Es sencillo probar que todo isomorfismo es a la vez 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 el hecho de que $f : R \longrightarrow S$ sea un monomorfismo es equivalente a que f sea una aplicación inyectiva.

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 $g_1 \circ i = g_2 \circ i$. Ahora $\forall x \in \mathbb{Q}$ tenemos que $\exists a, b \in \mathbb{Z} : x = \frac{a}{b}$ y entonces:

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

Con lo que $g_1 = g_2$. Esto prueba que $i : \mathbb{Z} \longrightarrow \mathbb{Q}$ es epimorfismo. Claramente es también un monomorfismo. La categoría de anillos es un ejemplo entonces de categoría en la que ser monomorfismo y epimorfismo no es equivalente a ser isomorfismo.

2.6.3. Objetos iniciales

Definición 8. *Dada una categoría $\mathcal{C}$ diremos que $\oslash$ es un objeto inicial si $\oslash$ es un objeto final en la categoría $\mathcal{C}^{op}$.*

Esta definición la podemos ver en exclusivamente en términos de la categoría $\mathcal{C}$ de la siguiente forma:

Proposición 3. *El objeto $\oslash$ es inicial si y solo si para cualquier otro objeto X existe una única flecha tiene a $\oslash$ como dominio y a X como codominio.*

2.6.3.1. Ejemplos

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

En `Grp` Vimos anteriormente que el objeto final de `Grp` era el grupo trivial. Resulta que el grupo trivial es también el objeto inicial de la categoría. Cuando en una categoría coinciden el objeto inicial $\oslash$ y el objeto final $*$ llamamos a tal objeto el objeto nulo y lo notamos por $\mathbf{0}$. La existencia de objetos nulos en una categoría nos garantiza la existencia de una flecha distinguida entre cada par de objetos A y B de la categoría, a la que llamaremos flecha cero y la definimos por $0_{A,B} = i \circ j : A \longrightarrow B$ donde $j : A \longrightarrow \mathbf{0}$ y $i : \mathbf{0} \longrightarrow B$ son respectivamente la única flecha que va de A al objeto nulo y la única flecha que va del objeto nulo a B.

Otro ejemplo de categoría con objeto cero es la categoría de espacios vectoriales sobre un cuerpo K.

En la categoría de cuerpos En la categoría de cuerpos no existen ni objetos finales ni objetos iniciales. Para justificar por qué supongamos que K es un objeto final de la categoría de cuerpos. La característica de K es o bien 0 o bien un primo. En cualquier caso si consideramos un cuerpo K' con característica distinta a la característica de K es bien conocido que no existen homomorfismos de cuerpos $K' \longrightarrow K$ y por tanto K no puede ser un objeto final. Para probar que no existen objetos iniciales se razona de forma análoga.

2.6.4. Coproducto

Coproducto es la propiedad dual de producto. Si vemos qué significa ser coproducto desde dentro de la misma categoría $\mathcal{C}$ obtenemos la siguiente definición:

Definición 9. *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 $A+B$ es un objeto de $\mathcal{C}$ y $i_1 : A \longrightarrow A+B, i_2 : B \longrightarrow A+B$ son dos flechas tales que para cualquier objeto Y y sendas flechas $f_1 : A \longrightarrow Y$, $f_2 : B \longrightarrow Y$ existe un único morfismo $f : A+B \longrightarrow Y$ tal que el siguiente diagrama es conmutativo:*

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

2.6.4.1. Ejemplos

En `Set` En `Set` el coproducto coincide con la unión disjunta.

En `Grp` En `Grp` el coproducto coincide con el producto libre.

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

Dado un objeto A de la categoría $\mathcal{C}$ definimos el funtor $\mathrm{Hom}_{\mathcal{C}}(-, A)$ como

$$\mathrm{Hom}_{\mathcal{C}}(-, A) = \mathrm{Hom}_{\mathcal{C}^{op}}(A, -) : \mathcal{C}^{op} \longrightarrow \texttt{Set}$$

Sabemos cómo actúa $\mathrm{Hom}_{\mathcal{C}}(-, A)$ sobre las flechas de $\mathcal{C}^{op}$ pero si tenemos una flecha $f \in \mathrm{Hom}_{\mathcal{C}}(C, C')$ tenemos que:

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

ya que $f \in \mathrm{Hom}_{\mathcal{C}^{op}}(C', C)$. Dicho de otra forma:

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

y su acción es:

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

además si tenemos dos flechas de $\mathcal{C}$ que se pueden componer f y g tenemos que:

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

con lo que la composición funciona *al revés*. Diremos que F es un funtor contravariante sobre sobre $\mathcal{C}$ cuando F sea un funtor definido sobre $\mathcal{C}^{op}$. En este sentido decimos que $\mathrm{Hom}_{\mathcal{C}}(-, A)$ es un funtor contravariante sobre $\mathcal{C}$.

2.6.6. Funtor opuesto

Dado un funtor $F : \mathcal{C} \longrightarrow \mathcal{D}$ podemos definir el funtor opuesto $F^{op} : \mathcal{C}^{op} \longrightarrow \mathcal{D}^{op}$ tal que $F^{op}C = FC$ y para cualquier flecha $f \in \mathrm{Hom}_{\mathcal{C}^{op}}(C, C') = \mathrm{Hom}_{\mathcal{C}}(C', C)$

$$F^{op}f = Ff \in \mathrm{Hom}_{\mathcal{C}}(FC', FC) = \mathrm{Hom}_{\mathcal{D}^{op}}(FC, FC')$$

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 la noción de funtor como morfismo entre categorías. Llegados a este punto podemos dar un paso más e introducir el concepto de transformación natural como el morfismo entre funtores.

Procedemos con la definición:

Definición 10. *Dados dos funtores $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{C} \longrightarrow \mathcal{D}$ decimos que $\lambda : F \Rightarrow G$ 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:*

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

3.1.0.1. Ejemplos

El doble dual Consideremos la categoría de espacios vectoriales de dimensión finita sobre un cuerpo K, a la que llamaremos $\texttt{Vect-}K$. Podemos definir el funtor $(-)^{**} : \texttt{Vect-}K \longrightarrow \texttt{Vect-}K$, al que llamaremos doble dual, que lleva un espacio vectorial V a su doble dual V^{**} y una aplicación lineal $f : V \longrightarrow W$ a la correspondiente aplicación lineal $f^{**} : V^{**} \longrightarrow W^{**}$ definida por

$$f^{**}(g)(\phi) = g(\phi \circ f)$$

Podemos probar que existe una transformación natural entre el funtor doble dual y el funtor identidad de la categoría $\texttt{Vect-}K$. La transformación natural es $\lambda : 1_{\texttt{Vect-}K} \Rightarrow (-)^{**}$ definido por:

$$\lambda_V : V \longrightarrow V^{**}$$

$$\lambda_V(v)(\phi) = \phi(v)$$

Nótese además que para cada espacio vectorial V se tiene que λ_V es un isomorfismo de espacios vectoriales.

Abelianización de grupos Definimos el funtor $(-)^{ab} : \mathtt{Grp} \longrightarrow \mathtt{Grp}$ como el funtor que lleva cada grupo G a su abelianización $G^{ab} = \frac{G}{[G,G]}$ y cada homomorfismo de grupos $f : G \longrightarrow H$ al homomorfismo de grupos dado por:

$$f^{ab} : G^{ab} \longrightarrow H^{ab}$$

$$f^{ab}(g[G,G]) = \pi_H(f(g))$$

Donde π_H es la proyección al cociente. La aplicación está bien definida porque los homomorfismos de grupos llevan conmutadores en conmutadores y como H^{ab} es un grupo abeliano y el único conmutador de un grupo abeliano es el 1 se tiene que $[G,G] \subseteq \ker(\pi_H \circ f)$.

Comprobar que $\pi : 1_{\mathtt{Grp}} \Rightarrow (-)^{ab}$ es una transformación natural se reduce a comprobar que para cada homomorfismo de grupos $f : G \longrightarrow H$ el siguiente diagrama es conmutativo:

$$\begin{tikzcd} G \arrow[r, "\pi_G"] \arrow[d, "f"] & G^{ab} \arrow[d, "f^{ab}"] \\ H \arrow[r, "\pi_H"] & H^{ab} \end{tikzcd}$$

3.1.1. Categorías de funtores

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

Proposición 4. *1. Dado cualquier funtor $F : \mathcal{C} \longrightarrow \mathcal{D}$ podemos definir la transformación natural $1_F : F \Rightarrow F$ dada por $(1_F)_C = 1_{(FC)}$ para cada objeto C de $\mathcal{C}$.*

2. Podemos componer transformaciones naturales de la siguiente forma: dados funtores $F, G, H : \mathcal{C} \longrightarrow \mathcal{D}$ y transformaciones naturales $\lambda : F \Rightarrow G$, $\sigma : G \Rightarrow H$ definimos la transformación natural $\sigma \circ \lambda : F \Rightarrow H$ por $(\sigma \circ \lambda)_C = \sigma_C \circ \lambda_C$.

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. Notaremos esta categoría como $\mathcal{D}^{\mathcal{C}}$.

3.1.2. Isomorfismos naturales

Definición 11. *Dados dos funtores* $F, G : \mathcal{C} \longrightarrow \mathcal{D}$ *diremos que la transformación natural* $\lambda : F \Rightarrow G$ *es un isomorfismo natural si* $\lambda_C : FC \longrightarrow GC$ *es un isomorfismo para todo objeto* C *de* $\mathcal{C}$*.*

Podemos relacionar los isomorfismos naturales y las categorías de funtores de la siguiente manera:

Proposición 5. *Sean* $F, G : \mathcal{C} \longrightarrow \mathcal{D}$ *dos funtores. Entonces* $\phi : F \Rightarrow G$ *es un isomorfismo natural si y solo si es un isomorfismo en la categoría* $\mathcal{D}^{\mathcal{C}}$*.*

Demostración. Supongamos que $\phi : F \Rightarrow G$ es un isomorfismo natural. Definimos $\psi : G \Rightarrow F$ como $\psi_C = \phi_C^{-1}$ (podemos considerar la inversa de ϕ_C gracias a que ϕ_C es un isomorfismo de $\mathcal{C}$ por ser ϕ un isomorfismo natural). Comprobar la naturalidad de ψ consiste en ver que para cualquier flecha $f : C \longrightarrow C'$ se tiene que el siguiente diagrama es conmutativo:

$$\begin{tikzcd} GC \arrow[r, "\psi_C"] \arrow[d, "Gf"'] & FC' \arrow[d, "Ff"] \\ GC' \arrow[r, "\psi_{C'}"] & FC' \end{tikzcd}$$

Es decir, $\psi_{C'} \circ Gf = Ff\psi_C$. Esto es cierto sí y solo sí $Gf \circ \phi_C = \phi_{C'} \circ Ff$ (componiendo a la izquierda con el isomorfismo $\phi_{C'}$ y a la derecha con el isomorfismo ϕ_C), pero esta última igualdad se deduce de la naturalidad de $\phi : F \Rightarrow G$. Entonces ψ es la inversa de ϕ y por tanto ϕ es un isomorfismo en la categoría de funtores $\mathcal{D}^{\mathcal{C}}$.

Supongamos ahora que $\phi : F \Rightarrow G$ es un isomorfismo en la categoría de funtores $\mathcal{D}^{\mathcal{C}}$. Entonces ϕ tiene una inversa $\psi : G \Rightarrow F$ (es decir $\psi \circ \phi = 1_F$ y $\phi \circ \psi = 1_G$) por lo que para cada objeto C de $\mathcal{C}$ tenemos que $\phi_C \circ \psi_C = 1_{GC}$ y además $\phi_C \circ \psi_C = 1_{FC}$, es decir, ϕ_C es un isomorfismo en $\mathcal{C}$. Esto prueba que $\phi : F \Rightarrow G$ es un isomorfismo natural. □

3.1.3. Transformaciones naturales a través de funtores

Sean $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{C} \longrightarrow \mathcal{D}$ dos funtores y $\tau : F \Rightarrow G$ una transformación natural entre ellos. Entonces:

- Dado un funtor $H : \mathcal{D} \longrightarrow \mathcal{D}'$ definimos $(H\tau)_A = H(\tau_A)$ para cada objeto A de $\mathcal{C}$. $H\tau$ es una transformación natural $H\tau : HF \Rightarrow HG$.

- Dado un funtor $K : \mathcal{C}' \longrightarrow \mathcal{C}$ definimos $(\tau K)'_A = \tau_{KA'}$ para cualquier objeto A' de $\mathcal{C}'$. τK es una transformación natural $\tau K : FK \Rightarrow GK$.

3.2. Lema de Yoneda

El lema de Yoneda es uno de los primeros resultados inesperados que surgen a raíz del estudio de la teoría de categorías. Una de las consecuencias más importantes de este resultado es el encaje de Yoneda, que nos permite ver una categoría $\mathcal{C}$ arbitraria dentro de la categoría de funtores $\mathtt{Set}^{\mathcal{C}^{op}}$. Necesitaremos algunos resultados previos para poder enunciar el lema.

A lo largo de las siguientes secciones usaremos la notación $\mathrm{Nat}(F,G)$ en lugar de $\mathrm{Hom}_{\mathcal{D}^{\mathcal{C}}}(F,G)$ para referirnos al conjunto de transformaciones naturales entre dos funtores $F,G:\mathcal{C}\longrightarrow\mathcal{D}$. Probamos la funtorialidad de uno de los ingredientes principales del lema de Yoneda.

Proposición. *Dadas una categoría $\mathcal{C}$ y un objeto A de esta definimos:*

$$L:\mathcal{C}\times \mathit{Set}^{\mathcal{C}}\longrightarrow \mathit{Set}$$

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

L es un funtor.

Demostración. Llamando $H:\mathcal{C}^{op}\longrightarrow \mathtt{Set}^{\mathcal{C}}$ al funtor dado por $H(A)=\mathrm{Hom}_{\mathcal{C}}(A,-)$ (*probaré que esto es un funtor*) podemos notar que nuestro funtor L es precisamente

$$L:\mathcal{C}\times\mathtt{Set}^{\mathcal{C}}\xrightarrow{H^{op}\times 1_{\mathtt{Set}^{\mathcal{C}}}}(\mathtt{Set}^{C})^{op}\times\mathtt{Set}^{C}\xrightarrow{\mathrm{Nat}}\mathtt{Set}$$

□

Probamos la funtorialidad del otro ingrediente principal del lema de Yoneda.

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

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

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

Donde la igualdad $(*)$ se deduce de la naturalidad de $\sigma:F\Rightarrow G$. Que $Ev(1_C,1_F)=1_{FC}$ se prueba de forma sencilla, confirmando que $Ev:\mathcal{C}\times\mathtt{Set}^{\mathcal{C}}\longrightarrow\mathtt{Set}$ es un funtor □

Ya estamos preparados para enunciar el lema.

Teorema 1 (Lema de Yoneda)**.** *Sea $\mathcal{C}$ una categoría. Entonces los funtores L y Ev son naturalmente isomorfos. En particular, para cada objeto A de $\mathcal{C}$ y cada funtor $F:\mathcal{C}\longrightarrow\mathit{Set}$ se tiene una biyección natural*

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

Demostración. Definimos la aplicación

$$\phi_{A,F} : \mathrm{Nat}(\mathrm{Hom}(A,-),F) \longrightarrow FA$$

$$\phi_{A,F}(\tau) = \tau_A(1_A)$$

(recordamos que τ_A por ser τ transformación natural entre $\mathrm{Hom}(A,-)$ y F es induce una aplicación de conjuntos $\tau_A : \mathrm{Hom}(A,A) \longrightarrow FA$).

Veamos que $\phi_{A,F}$ es una biyección. En primer lugar veremos que es sobreyectiva. Sea $x \in FA$ Definimos la transformación natural $\lambda_x : \mathrm{Hom}(A,-) \Rightarrow F$ por

$$(\lambda_x)_C : \mathrm{Hom}(A,C) \longrightarrow FC$$

$$(\lambda_x)_C(f) = Ff(x)$$

Comprobar que λ_x es en efecto una transformación natural es sencillo. Pero $\phi_{A,F}(\lambda_x) = (\lambda_x)_A(1_A) = F1_A(x) = x$, luego $\phi_{A,F}$ es sobreyectiva.

Veamos ahora que $\phi_{A,F}$ es inyectiva. Supongamos que tenemos dos transformaciones naturales $\tau, \tau' : \mathrm{Hom}(A,-) \Rightarrow F$ tales que $\phi_{A,F}(\tau) = \phi_{A,F}(\tau')$. Para todo $f \in \mathrm{Hom}(A,C)$ la naturalidad de τ nos garantiza la conmutatividad del siguiente diagrama:

$$\begin{tikzcd}
\mathrm{Hom}(A,A) \arrow[r, "\tau_A"] \arrow[d, "\mathrm{Hom}(A,f)"'] & FA \arrow[d, "Ff"] \\
\mathrm{Hom}(A,C) \arrow[r, "\tau_C"] & FC
\end{tikzcd}$$

Es decir $\tau_C \circ \mathrm{Hom}(A,f) = Ff \circ \tau_A$. Aplicando esta flecha sobre el valor 1_A tenemos que:

$$\tau_C(f) = \tau_C(f \circ 1_A) = \tau_C(\mathrm{Hom}(A,f)(1_A)) = Ff(\tau_A(1_A)) = Ff(\phi_{A,F}(\tau))$$

De la misma manera se puede obtener que $\tau'_C(f) = Ff(\phi_{A,F}(\tau'))$, pero entonces para cualquier objeto C y para cualquier flecha $f \in \mathrm{Hom}(A,C)$ se cumple:

$$\tau_C(f) = Ff(\phi_{A,F}(\tau)) = Ff(\phi_{A,F}(\tau')) = \tau'_C(f)$$

con lo que $\tau = \tau'$ y $\phi_{A,F}$ es biyectiva. Merece la pena apuntar cuál es la aplicación inversa de $\phi_{A,F}$. Unas cuentas sencillas nos permiten comprobar que:

$$\phi^{-1}_{A,F} : FA \longrightarrow \mathrm{Nat}(\mathrm{Hom}(A,-),F)$$

$$\phi^{-1}_{A,F}(a)_C(f) = (Ff)(a)$$

Probar que $\phi_{A,F}$ es natural en A y F es lo que queda para ver que $\phi : L \Rightarrow Ev$ es un isomorfismo natural. La demostración es rutinaria y no se incluye en el presente texto. $\square$

El lema de Yoneda se puede interpretar en términos de elementos categóricos. Si tenemos una categoría $\mathcal{C}$, un objeto A de esta y un funtor $F : \mathcal{C} \longrightarrow \mathtt{Set}$, el lema de Yoneda nos permite decir que los elementos categóricos de F definidos sobre el funtor $\mathrm{Hom}(A, -)$ son esencialmente los mismos que los elementos del conjunto FA.

Obtenemos un importante corolario del Lema de Yoneda cuando tomamos $F = \mathrm{Hom}(B, -)$ con B otro objeto de $\mathcal{C}$.

Teorema 2. *Dada una categoría $\mathcal{C}$ el funtor $T : C^{op} \longrightarrow \mathtt{Set}^{\mathcal{C}}$ dado por $T(A) = \mathrm{Hom}_{\mathcal{C}}(A, -)$ es fiel y pleno.*

Demostración. T actúa sobre flechas $f \in \mathrm{Hom}_{\mathcal{C}^{op}}(A, B)$ de la siguiente manera:

$$T(f) = \mathrm{Hom}_{\mathcal{C}}(f, -) : T(A) = \mathrm{Hom}(A, -) \Rightarrow \mathrm{Hom}(B, -) = T(B)$$

$$T(f)_C(g) = g \circ f$$

Repitiendo la demostración del lema de Yoneda con $F = \mathrm{Hom}(B, -)$ vemos que la aplicación inducida por T sobre los conjuntos $\mathrm{Hom}_{\mathcal{C}^{op}}(A, B)$ es precisamente la biyección:

$$\phi_{A,F}^{-1} : \mathrm{Hom}_{\mathcal{C}^{op}}(A, B) = FA \longrightarrow \mathrm{Nat}(\mathrm{Hom}(A, -), F) = \mathrm{Nat}(T(A), T(B))$$

que definimos anteriormente. Luego T es inyectiva y sobreyectiva sobre los conjuntos Hom y por tanto es un funtor fiel y pleno. □

Aplicando esta misma proposición sobre la categoría $\mathcal{C}^{op}$ llegamos al resultado dual:

Teorema 3. *Dada una categoría $\mathcal{C}$, el funtor que asigna a cada objeto A de la categoría el funtor $\mathrm{Hom}(-, A)$ es un funtor $\mathcal{C} \longrightarrow \mathtt{Set}^{\mathcal{C}^{op}}$ fiel y pleno.*

Los funtores descritos en los dos últimos teoremas son conocidos como **encajes de Yoneda**.

Los encajes de Yoneda son una herramienta fundamental para la teoría de categorías. Nos permiten ver la categoría $\mathcal{C}$ dentro de la categoría de funtores $\mathtt{Set}^{\mathcal{C}^{op}}$. En este sentido la categoría $\mathtt{Set}^{\mathcal{C}^{op}}$ se puede ver como una extensión de la categoría $\mathcal{C}$, identificando cada objeto A de $\mathcal{C}$ con el funtor $\mathrm{Hom}(-, A)$ y cada flecha $f : A \longrightarrow B$ con la transformación natural asociada $\mathrm{Hom}(-, f) : \mathrm{Hom}(-, A) \Rightarrow \mathrm{Hom}(-, B)$. El hecho de que el encaje de Yoneda sea fiel y pleno nos dice que esta identificación es buena en el sentido de que las flechas entre objetos A y B de $\mathcal{C}$ son *las mismas* que las flechas (transformaciones naturales) entre los funtores $\mathrm{Hom}(-, A)$ y $\mathrm{Hom}(-, B)$

Cuando en el capítulo 2 introdujimos la noción de elemento categórico subrayamos que dados $x \in^T A$ y $f : A \longrightarrow B$ entonces $f \circ x \in^T B$. Que f lleve elementos de A en elementos de B motivó que utilizáramos la notación $f(x)$ en lugar de $f \circ x$. Esta notación podría llevarnos a interpretar f como una familia de aplicaciones $f_T : \mathrm{Hom}(T, A) \longrightarrow \mathrm{Hom}(T, B)$ para objeto T de $\mathcal{C}$, pero esta familia es precisamente la imagen de f a través del encaje de Yoneda: $f_T = \mathrm{Hom}(-, f)_T$. Recíprocamente, dada cualquier familia de aplicaciones $g_T : \mathrm{Hom}(T, A) \longrightarrow \mathrm{Hom}(T, B)$ natural en T, en el sentido de que para cualquier $h : S \longrightarrow T$ se dé $g_T(x) \circ h = g_S(x \circ h)$, tendríamos que g es una transformación natural $g : \mathrm{Hom}(-, A) \Rightarrow \mathrm{Hom}(-, B)$ y el encaje de Yoneda nos permitiría realizar la identificación $g : A \longrightarrow B$.

Todo esto nos permite decir que dar una flecha $f : A \longrightarrow B$ en una categoría arbitraria $\mathcal{C}$ es esencialmente lo mismo que dar una aplicación que lleva elementos de A a elementos de B de una forma coherente (la condición de naturalidad).

3.3. Programación

3.3.1. Transformaciones Naturales

Para construir una transformación natural entre dos endofuntores de `Hask` F y G necesitamos, en primer lugar, una función de tipo `FA ->GA` para `A` cualquier tipo del lenguaje.

Usamos el sistema de polimorfismo de Haskell para suplir nuestra primera necesidad. El lenguaje nos permite definir una función de tipo `F a ->G a` (una función polimórfica en la variable de tipo `a`) que, especializando a cada tipo del lenguaje, será la componente en cada objeto de `Hask` de nuestra transformación natural. usas A y a para tipo casi en el mismo párrafo ?¿?¿?

Un ejemplo de este tipo de función polimórfica sería:

```
discardLeft :: Either b a -> Maybe a
discardLeft (Left stringQueNoQuiero) = Nothing
discardLeft (Right v) = Just v
```

(recordemos que es `Either b`, y no `Either` a secas, la instancia de la clase `Functor`)

En segundo lugar, necesitamos que se cumplan las condiciones de naturalidad. Podríamos emplear nuestro tiempo en demostrar que la función `discardLeft` que hemos definido cumple tales condiciones pero eso no será necesario: el sistema de polimorfismo de Haskell (*parametric polymorphism*) nos lo garantiza. Haskell nos impone la restricción de que una función polimórfica en el la ??? variable de tipo `a` esté implementada con la misma expresión. Por ejemplo, en Haskell no es posible escribir una función polimórfica de esta forma:

```
polimorfica :: a -> a
polimorfica v = v

polimorfica :: Int -> Int
polimorfica n = n + 1
```

Esta restricción, junto con otras características del polimorfismo paramétrico, permite comprobar que cualquier función de tipo `F a->G a` con `F` y `G` funtores es una transformación natural [6].

Mostramos a continuación algunos ejemplos de transformaciones naturales que podemos escribir entre los funtores de Haskell que hemos visto anteriormente.

3.3.1.1. Ejemplos

Entre `Maybe` y `Either` La función `discardLeft :: Either b a ->Maybe a` que hemos definido antes es un ejemplo de transformación natural. La interpretación que dimos de estos tipos en el primer capítulo fue la siguiente:

- Los valores de la forma `Nothing :: Maybe a` representan situaciones de error durante un cómputo. Los valores de la forma `Just v :: Maybe a` representan un cómputo exitoso que tiene al valor `v :: a` como valor de retorno.

- Los valores de la forma `Left error :: Either b a` representan situaciones de error durante un cómputo y además aportan más información sobre el error (por ejemplo si la variable `b` estuviera especializada a `String` podríamos tener un valor de error que fuera `Left "No fue posible conectar al servidor"`. Los valores de la forma `Right v :: Either b a` representan cómputos exitosos con valor de retorno `v :: a` .

En este sentido podemos implementar la función `discardLeft` como una aplicación que recibe un cómputo posiblemente fallido y descarta la información asociada al fallo en caso de que la haya. En caso de computación exitosa deja invariante su resultado.

Podemos escribir una transformación natural en el sentido inverso fijando un valor de tipo `b`. Por ejemplo

```
maybeToRight :: b -> Maybe a -> Either b a
maybeToRight defaultError Nothing = Left defaultError
maybeToRight defaultError (Just v) = Right v
```

Y `maybeToRight` no es una transformación natural pero para cualquier valor `x :: B` tenemos que `maybeToRight x :: Maybe a ->Either B a` sí que lo es. Bajo la interpretación de cómputos propensos a error, esta transformación natural añade una justificación del error a un cómputo fallido sin contexto del fallo.

Transformaciones naturales al funtor constante En Haskell podemos definir el funtor constante de la siguiente manera:

```
data Const b a = Const b

instance Functor (Const b) where
  fmap f (Const b) = Const b
```

Este es un funtor distinto a los anteriores que hemos definido en Haskell demás ???? en el sentido de que no utiliza la variable de tipo de ninguna forma. *Escribir esto mejor* pues si.

Supongamos que tenemos una función polimórfica de tipo `f :: F a ->B` donde `B` es un tipo fijo. Esta función se puede extender a otra función `fConst :: F a ->Const B a` de la siguiente forma:

```
-- recordamos f :: F a -> B
fConst :: F a -> Const B a
fConst x = Const (f x)
```

El tipo de `fConst` (es de la forma `F a ->G a` donde tanto `F` como `G` son funtores) nos garantiza que es una transformación natural. En particular cumple las condiciones de naturalidad. Si escribimos estas condiciones de naturalidad en haskell llegamos a:

```
-- dado g :: c -> c'
fConst . fmap g = fmap g . fConst
```

donde el `fmap` de la izquierda es el de `F` y el de la izquierda es el de `Const B`.izquierda derecha?¿?¿? Pero el de `Const b` deja invariante el resultado. Esto nos lleva a comprobar que: `f . fmap g = f`. Es decir, si modificamos un valor de nuestro funtor con alguna función de la forma `fmap g` no estaremos variando la métrica. Un ejemplo de esto lo podemos ver con la función `length :: [a] ->Int`.

Capítulo 4

Adjunciones y Mónadas

4.1. Adjunciones

De camino a nuestro objetivo de definir el concepto de mónada tendremos que examinar una de las construcciones más importantes de la teoría de las categorías: las adjunciones. Como veremos en las próximas secciones las mónadas y las adjunciones están estrechamente relacionadas, pero esto no debe hacernos pensar que las mónadas son el único motivo por el que las adjunciones son importante en matemáticas. Las adjunciones han sido identificadas por algunos autores como uno de los conceptos clave de la teoría de categorías. En palabras de McLane los funtores adjuntos "se encuentran por todas partes". [3].

Empezaremos por precisar qué es una adjunción.

4.1.1. Definición

Definición 12. *Una adjunción entre categorías $\mathcal{C}$ y $\mathcal{D}$ es un par de funtores $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{D} \longrightarrow \mathcal{C}$ tales que los funtores*

$$\mathrm{Hom}_{\mathcal{D}}(F-,-) : \mathcal{C}^{op} \times \mathcal{D} \longrightarrow \mathbf{\mathit{Set}}$$

$$\mathrm{Hom}_{\mathcal{C}}(-,G-) : \mathcal{C}^{op} \times \mathcal{D} \longrightarrow \mathbf{\mathit{Set}}$$

son naturalmente isomorfos. Diremos que F es el adjunto por la izquierda de G y que G es el adjunto por la derecha de F. Notaremos esta relación entre ambos funtores como $F \dashv G$.

Nótese que para cualquier objeto C de $\mathcal{C}$ y cualquier objeto D de $\mathcal{D}$ un par de funtores adjuntos $F \dashv G$ nos permite establecer biyecciones entre los conjuntos:

$$\mathrm{Hom}_{\mathcal{D}}(FC, D) \cong \mathrm{Hom}_{\mathcal{C}}(C, GD)$$

Sea $\phi_{C,D} : \mathrm{Hom}_{\mathcal{D}}(FC, D) \longrightarrow \mathrm{Hom}_{\mathcal{C}}(C, GD)$ una tal familia de biyecciones. Comprobar que ϕ es natural en C y en D se reduce a, por un lado, comprobar que el siguiente diagrama en $\mathcal{D}$:

$$FC \xrightarrow{f} D \xrightarrow{g} D'$$

Se traslada al siguiente diagrama conmutativo en $\mathcal{C}$:

$$\begin{tikzcd} C \arrow[r, "\phi_{C,D}(f)"] \arrow[rr, bend right, "\phi_{C,D'}(g\circ f)"'] & GD \arrow[r, "Gg"] & GD' \end{tikzcd} \tag{4.1}$$

Y que por otro lado, el siguiente diagrama en $\mathcal{C}$:

$$C' \xrightarrow{h} C \xrightarrow{f} GD$$

se traslada al siguiente diagrama conmutativo en $\mathcal{D}$

$$\begin{tikzcd} FC' \arrow[r, "Fh"] \arrow[rr, bend right, "\phi^{-1}_{C',D}(f\circ h)"'] & FC \arrow[r, "\phi^{-1}_{C,D}(f)"] & D \end{tikzcd} \tag{4.2}$$

4.1.2. Unidad y counidad

Sean $\mathcal{C}$ y $\mathcal{D}$ dos categorías y $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{D} \longrightarrow \mathcal{C}$ dos funtores adjuntos $F \dashv G$. Esta adjunción induce un par de transformaciones naturales $\eta : 1_{\mathcal{C}} \Rightarrow GF$ y $\epsilon : FG \Rightarrow 1_{\mathcal{D}}$ que conoceremos con el nombre de **la unidad** y **la counidad** de la adjunción. Mostramos a continuación la construcción de la unidad.

Sea $\phi : \mathrm{Hom}(F-,-) \Rightarrow \mathrm{Hom}(-,G-)$ el isomorfismo natural que nos viene dado por la adjunción. Definimos $\eta : 1_{\mathcal{C}} \Rightarrow GF$ como:

$$\eta_C = \phi_{C,FC}(1_{FC}) : C \longrightarrow GFC$$

Veamos que en efecto es natural. Sea $f : C \longrightarrow C'$ una flecha de $\mathcal{C}$. El diagrama cuya conmutatividad tenemos que verificar es el siguiente:

$$\begin{tikzcd} C \arrow[r, "\eta_C"] \arrow[d, "f"'] & GFC \arrow[d, "GFf"] \\ C' \arrow[r, "\eta_{C'}"] & GFC' \end{tikzcd}$$

Es decir tenemos que comprobar que

$$\phi_{C',FC'}(1_{FC'}) \circ f = \eta_{C'} \circ f = GFf \circ \eta_C = GFf \circ \phi_{C,FC}(1_{FC})$$

Pero teniendo en cuenta cuenta que ϕ es natural tenemos que el diagrama

$$FC \xrightarrow{1_{FC}} FC \xrightarrow{Ff} FC'$$

Lo podemos trasladar al siguiente diagrama conmutativo (por el diagrama (4.1) de la definición de adjunción):

$$\begin{tikzcd} C \arrow[r, "\eta_C"] \arrow[rr, bend right, "\phi_{C,FC'}(f\circ 1_C)=\phi_{C,FC'}(Ff)"'] & GFC \arrow[r, "GFf"] & GFC' \end{tikzcd}$$

Por lo que $GFf \circ \eta_C = \phi_{C,FC'}(Ff)$ pero a su vez:

$$C \xrightarrow{f} C' \xrightarrow{\eta_{C'}} GFC'$$

Lo podemos trasladar a (por el diagrama (4.2) de la definición de adjunción):

$$\begin{tikzcd} FC \arrow[r, "Ff"] \arrow[rr, bend right, "\phi^{-1}_{C,FC'}(\eta_{C'}\circ f)"'] & FC' \arrow[r, "1_{FC'}"] & FC' \end{tikzcd}$$

Este último diagrama muestra que

$$Ff = \phi^{-1}_{C,FC'}(\eta_{C'} \circ f)$$

Y por tanto

$$GFf \circ \eta_C = \phi_{C,FC'}(Ff) = \phi_{C,FC'}(\phi^{-1}_{C,FC'}(\eta_{C'} \circ f)) = \eta_{C'} \circ f$$

lo que prueba la naturalidad de $\eta : 1_{\mathcal{C}} \Rightarrow GF$.

La counidad $\epsilon : FG \Rightarrow 1_{\mathcal{D}}$ se define como:

$$\epsilon_D = \phi^{-1}_{GD,D}(1_D) : FGD \longrightarrow D$$

La demostración de la naturalidad no presenta complicaciones.

4.1.3. Ejemplos

Producto Sea A un conjunto. Veamos que los funtores $- \times A : \texttt{Set} \longrightarrow \texttt{Set}$ y $\mathrm{Hom}(A, -) : \texttt{Set} \longrightarrow \texttt{Set}$ conforman una adjunción. Necesitamos encontrar un isomorfismo natural

$$\phi : \mathrm{Hom}(- \times A, -) \Rightarrow \mathrm{Hom}(-, \mathrm{Hom}(A, -))$$

Para ello definimos

$$\phi_{B,Z} : \mathrm{Hom}(B \times A, Z) \longrightarrow \mathrm{Hom}(B, \mathrm{Hom}(A, Z))$$

$$\phi_{B,Z}(f)(b)(a) = f(b,a)$$

Veamos que $\phi_{B,Z}$ es una biyección. En primer lugar vemos que es sobreyectiva. Sea $f' \in \mathrm{Hom}(B, \mathrm{Hom}(A,Z))$. Definimos la aplicación $f : B \times A \longrightarrow Z$ dada por $f(b,a) = f'(b)(a)$, $\forall (b,a) \in B \times A$. Aplicando la definición de $\phi_{B,Z}$ tenemos $\phi_{B,Z}(f)(b)(a) = f(b,a) = f'(b)(a)$ también $\forall (b,a) \in B \times A$, y por tanto $\phi_{B,Z}(f) = f'$. Veamos ahora que ϕ es inyectiva. Supongamos que tenemos $f_1, f_2 : B \times A \longrightarrow Z$ tales que $\phi_{B,Z}(f_1) = \phi_{B,Z}(f_2)$ pero entonces tenemos que $f_1(b,a) = \phi_{B,Z}(f_1)(b)(a) = \phi_{B,Z}(f_2)(b)(a) = f_2(b,a)$ $\forall (b,a) \in B \times A$ y por tanto $f_1 = f_2$.

Comprobamos ahora las condiciones de naturalidad. En primer lugar dado el diagrama:

$$B \xrightarrow{f} B' \xrightarrow{g} \mathrm{Hom}(A,Z)$$

Hemos de ver que el siguiente diagrama es conmutativo (diagrama (4.2) de la definición de adjunción):

$$\begin{tikzcd} B \times A \arrow[r, "f \times A"] \arrow[rr, bend right, "\phi^{-1}_{B,Z}(g \circ f)"'] & B' \times A \arrow[r, "\phi^{-1}_{B',Z}(g)"] & Z \end{tikzcd}$$

Y esto equivale a probar

$$\phi_{B,Z}(g \circ f) = \phi_{B',Z}(g) \circ (f \times A)$$

donde $f \times A$ está definida por

$$f \times A : B \times A \longrightarrow B' \times A$$

$$(f \times A)(b,a) = (f(b),a)$$

Aplicando sobre un valor cualquiera $(b,a) \in B \times A$ tenemos que:

$$\phi^{-1}_{B,Z}(g \circ f)(b,a) = (g \circ f)(b)(a) = g(f(b))(a)$$

y

$$(\phi^{-1}_{B',Z}(g) \circ (f \times A))(b,a) = \phi^{-1}_{B',Z}(g)(f(b),a) = g(f(b))(a)$$

En segundo lugar dado el diagrama:

$$B \times A \xrightarrow{f} Z \xrightarrow{g} Z'$$

Hemos de ver que el siguiene diagrama es conmutativo (diagrama (4.1) de la definición de adjunción):

$$\begin{tikzcd} B \arrow[r, "\phi_{B,Z}(f)"] \arrow[rr, bend right, "\phi_{B,Z'}(g \circ f)"'] & \mathrm{Hom}(A,Z) \arrow[r, "\mathrm{Hom}(A,g)"] & \mathrm{Hom}(A,Z') \end{tikzcd}$$

Es decir, $\phi_{B,Z'}(g \circ f) = \mathrm{Hom}(A, g) \circ \phi_{B,Z}(f)$. Para todo $(b, a) \in B \times A$ tenemos que:

$$\phi_{B,Z'}(g \circ f)(b)(a) = (g \circ f)(b, a) = g(f(b, a))$$

y

$$\begin{aligned}(\mathrm{Hom}(A, g) \circ \phi_{B,Z}(f))(b)(a) &= (\mathrm{Hom}(A, g)(\phi_{B,Z}(f)(b))(a) \\ &= (g \circ \phi_{B,Z}(f)(b))(a) = g(f(b, a))\end{aligned}$$

Y por tanto ϕ es una transformación natural.

Esta adjunción nos dice que dar una aplicación que va de $B \times A$ a Z es esencialmente lo mismo que dar una aplicación que vaya de B al conjunto de aplicaciones entre A y Z.

Grupos libres Los funtores $U : \mathtt{Grp} \longrightarrow \mathtt{Set}$, conjunto subyacente, y $F : \mathtt{Set} \longrightarrow \mathtt{Grp}$, grupo libre, son adjuntos $F \dashv U$.

La unidad $\eta : 1_{\mathtt{Set}} \Rightarrow UF$ es la inclusión canónica $\eta_X : X \subseteq UFX$ de un conjunto en el grupo libre generado por él. La counidad $\mu : FU \Rightarrow 1_{\mathtt{Grp}}$ determina para cada grupo G el homomorfismo de grupos $\mu_G : FUG \longrightarrow G$ tal que a cada palabra $g_1 g_2 \dots g_n \in FUG$ le asigna el valor $\prod_{i=1}^{n} g_i \in G$.

4.2. Mónadas

Comenzamos con el estudio de una de las nociones nacidas en teoría de categorías que son más importantes para nuestro trabajo: las mónadas.

4.2.1. Definición

Definición 13. *Una mónada sobre una categoría $\mathcal{C}$ es una terna (T, η, μ) donde $T : \mathcal{C} \longrightarrow \mathcal{C}$ es un endofuntor y $\eta : 1_{\mathcal{C}} \Rightarrow T$, $\mu : T^2 \Rightarrow T$ son dos transformaciones naturales tales que los siguientes diagramas conmutan:*

$$\begin{tikzcd} T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"] & T \end{tikzcd} \tag{4.3}$$

$$\begin{tikzcd} T \arrow[r, "T\eta"] \arrow[dr, equal] & T^2 \arrow[d, "\mu"] & T \arrow[l, "\eta T"'] \arrow[dl, equal] \\ & T & \end{tikzcd} \tag{4.4}$$

Estudiar algunos ejemplos será útil para comprender mejor los distintos elementos que forman parte de la definición.

4.2.1.1. Ejemplos

Monoides Sea M un monoide. Definimos:

- El funtor $T : \texttt{Set} \longrightarrow \texttt{Set}$ como $T(X) = M \times X$
- $\eta_X : X \longrightarrow T(X)$ definida por $\eta_X(x) = (e, x)$ (con $e \in M$ el elemento neutro)
- $\mu_X : T^2(X) \longrightarrow T(X)$ definida por $\mu_X(m, n, x) = (m \cdot n, x)$

Veamos que (T, η, μ) es una mónada. *A lo largo de este ejemplo nos tomaremos la libertad de identificar T^2X con $M \times M \times X$ en lugar con $M \times (M \times X)$ con la intención de simplificar la notación.* En primer lugar tenemos que ver que para cualquier conjunto X el siguiente diagrama es conmutativo (diagrama (4.3) de la definición de mónada):

$$\begin{tikzcd} M \times M \times M \times X \arrow[r, "T\mu_X"] \arrow[d, "\mu_{TX}"] & M \times M \times X \arrow[d, "\mu_X"] \\ M \times M \times X \arrow[r, "\mu_X"] & M \times X \end{tikzcd}$$

Es decir $\mu_X \circ \mu_{TX} = \mu_X \circ T\mu_X$. Para comprobar esta igualdad tomamos $(m, n, k, x) \in T^3(X)$. Por un lado:

$$(\mu_X \circ \mu_{TX})(m, n, k, x) = \mu_X(\mu_{TX}(m, n, k, x) = \mu_X(m \cdot n, k, x) = ((m \cdot n) \cdot k, x)$$

Por otro lado:

$$(\mu_X \circ T\mu_X)(m, n, k, x) = \mu_X(T\mu_X(m, n, k, x)) = \mu_X(m, n \cdot k, x) = (m \cdot (n \cdot k), x)$$

y por la asociatividad del producto en M tenemos que el diagrama es conmutativo. Queda ver que para cualquier conjunto X el diagrama (4.4 de la definición de mónada)

$$\begin{tikzcd} T(X) \arrow[r, "\eta_{TX}"] \arrow[dr, equal] & T^2(X) \arrow[d, "\mu_X"] & T(X) \arrow[l, "T\eta_X"'] \arrow[dl, equal] \\ & T(X) & \end{tikzcd}$$

es conmutativo, es decir, $\mu_X \circ T\eta_X = 1_{TX} = \mu_X \circ \eta_{TX}$. Podemos comprobar esta igualdad basta ver que para cualquier $(m, x) \in TX$:

$$\mu_X(T\eta_X(m, x)) = \mu_X(m, e, x) = (m \cdot e, x) = (m, x)$$

$$\mu_X(\eta_{TX}(m, x)) = \mu_X(e, m, x) = (e \cdot m, x) = (m, x)$$

Sumas Sea $\mathcal{C}$ una categoría con coproductos finitos. Dado un objeto C de $\mathcal{C}$ de definimos:

- El endofuntor $T : \mathcal{C} \longrightarrow \mathcal{C}$ dado por $T(X) = C + X$
- La transformación natural $\eta : 1_{\mathcal{C}} \Rightarrow T$ dada por $\eta_X : X \longrightarrow C + X$ la inyección canónica en el coproducto.
- La transformación natural $\mu : T^2 \Rightarrow T$ dada por $\mu_X : C + (C + X) \longrightarrow C + X$ *Definir*

La terna (T, η, μ) es una mónada.

Mónada $\mathrm{Hom}(A, -)$ 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.

4.2.2. Las adjunciones dan lugar a mónadas

Teorema 4. *Sean $\mathcal{C}$ y $\mathcal{D}$ dos categorías y $F : \mathcal{C} \longrightarrow \mathcal{D}$ y $G : \mathcal{D} \longrightarrow \mathcal{C}$ un par de funtores adjuntos $F \dashv G$ con unidad $\eta : 1_{\mathcal{C}} \Rightarrow GF$ y counidad $\epsilon : FG \Rightarrow 1_{\mathcal{D}}$. La terna $(GF, \eta, G\epsilon F)$ es una mónada.*

Demostración. Hacer demostración □

Pero este resultado lo sabemos. El siguiente resultado.

4.2.3. Categoría de Kleisli de una mónada

Hemos visto que todo par de funtores adjuntos inducen una mónada. Es natural plantearse si el recíproco es cierto: ¿es toda mónada la composición de un par de funtores adjuntos? La respuesta es afirmativa.

Teorema 5. *Sea $\mathcal{C}$ una categoría y (T, η, μ) una mónada sobre $\mathcal{C}$. Entonces existe una categoría $\mathcal{D}$ y un par de funtores $F : \mathcal{C} \longrightarrow \mathcal{D}$, $G : \mathcal{D} \longrightarrow \mathcal{C}$ tales que $F \dashv G$ y además:*

1. *$T = GF$*
2. *η es la unidad de la adjunción*

3. $\mu = F\epsilon G$ *donde* $\epsilon : FG \Rightarrow 1_{\mathcal{D}}$ *es la counidad de la adjunción.*

Demostración. *Categoría de Kleisli.* Probamos el resultado paso por paso.

1. Construcción de $\mathcal{D}$. Definimos $\mathcal{O}b(\mathcal{D}) = \mathcal{O}b(\mathcal{C})$ y $\mathrm{Hom}_{\mathcal{D}}(A,B) = \mathrm{Hom}_{\mathcal{C}}(A,TB)$. Dadas dos flechas $f \in \mathrm{Hom}_{\mathcal{D}}(A,B)$ y $g \in \mathrm{Hom}_{\mathcal{D}}(B,C)$ definimos la composición $g \circ_{\mathcal{D}} f \in \mathrm{Hom}_{\mathcal{D}}(A,C)$ como la flecha

$$g \circ_{\mathcal{D}} f : A \xrightarrow{f} TB \xrightarrow{Tg} T^2C \xrightarrow{\mu_C} TC$$

Veamos que esta operación de composición cumple los axiomas:

- Es asociativa. Sean $f : \mathrm{Hom}_{\mathcal{D}}(A,B)$, $g : \mathrm{Hom}_{\mathcal{D}}(B,C)$ y $h : \mathrm{Hom}_{\mathcal{D}}(C,D)$ entonces

$$(h \circ_{\mathcal{D}} g) \circ_{\mathcal{D}} f = \mu_D \circ T(h \circ_{\mathcal{D}} g) \circ f = \mu_D \circ T(\mu_D \circ Th \circ g) \circ f = \mu_D \circ T\mu_D \circ T^2h \circ Tg \circ f$$

$$=^* \mu_D \circ \mu_{TD} \circ T^2h \circ Tg \circ f =^{**} \mu_D \circ Th \circ \mu_C \circ Tg \circ f = \mu_D \circ Th \circ (g \circ_{\mathcal{D}} f) = h \circ_{\mathcal{D}} (g \circ_{\mathcal{D}} f)$$

Donde $(*)$ es consecuencia del diagrama (4.3) de la definición de mónada aplicado sobre D y $(**)$ es consecuencia de la naturalidad de $\mu : T^2 \Rightarrow T$.

- Existen identidades. Sean $f \in \mathrm{Hom}_{\mathcal{D}}(A,X)$ y $g \in \mathrm{Hom}_{\mathcal{D}}(X,A)$. Observamos que

$$f \circ_{\mathcal{D}} \eta_A = \mu_X \circ Tf \circ \eta_A =^* \mu_X \circ \eta_{TB} \circ f =^{**} f$$

Donde $(*)$ es consecuencia de la naturalidad de $\eta : 1_{\mathcal{C}} \Rightarrow T$ y $(**)$ es consecuencia del diagrama (4.4) de la definición de mónada. Por otro lado

$$\eta_A \circ_{\mathcal{D}} g = \mu_A \circ T\eta_A \circ g =^* g$$

con $(*)$ consecuencia otra vez del diagrama (4.4).

Esto muestra que en la categoría $\mathcal{D}$ la flecha η_A es la identidad del objeto A para cualquier objeto A de la categoría.

2. Ahora que conocemos la estructura de categoría de $\mathcal{D}$ definimos $F : \mathcal{C} \longrightarrow \mathcal{D}$ como $FA = A$ para cada objeto A de $\mathcal{C}$ y $Ff = Tf \circ \eta_A$ para toda flecha $f : A \longrightarrow B$ de $\mathcal{C}$. Veamos que F es un funtor. Por partes:

- F se lleva bien con la composición. Dado el diagrama $A \xrightarrow{f} B \xrightarrow{C}$ en la categoría $\mathcal{C}$ tenemos que:

$$F(g \circ f) = T(g \circ f) \circ \eta_A = Tg \circ Tf \circ \eta_A =^* Tg \circ \eta_B \circ f$$

Donde $(*)$ se deduce de la naturalidad de $\eta : 1_{\mathcal{C}} \Rightarrow T$. Igualmente:

$$Fg \circ_{\mathcal{D}} Ff = (Tg \circ \eta_B) \circ_{\mathcal{D}} (Tf \circ \eta_A) = \mu_C \circ T(Tg \circ \eta_B) \circ (Tf \circ \eta_A)$$

$$= \mu_C \circ T^2g \circ T\eta_B \circ Tf \circ \eta_A =^* \mu_C \circ T^2g \circ T\eta_B \circ \eta_B \circ f$$

$$=^{**} Tg \circ \mu_B \circ T\eta_B \circ \eta_B \circ f =^{***} Tg \circ \eta_B \circ f$$

Con $(*)$ consecuencia de la naturalidad de $\eta : 1_{\mathcal{C}} \Rightarrow T$, $(**)$ consecuencia de la naturalidad de $\mu : T^2 \Rightarrow T$, y $(***)$ por el diagrama (4.4) de la definición de mónadas.

- F lleva identidades a identidades:

$$F1_A = T1_A \circ \eta_A = 1_{TA} \circ \eta_A = \eta_A$$

que es la flecha identidad de A en $\mathcal{D}$.

3. Definimos $GA = TA$ para cada objeto A de $\mathcal{D}$ y $Gf = \mu_B \circ Tf$ para cada flecha $f \in \mathrm{Hom}_{\mathcal{D}}(A, B)$. Mostramos que $G : \mathcal{D} \longrightarrow \mathcal{C}$ es un funtor:

- G se lleva bien con la composición. Dadas $f \in \mathrm{Hom}_{\mathcal{D}}(A, B)$ y $g \in \mathrm{Hom}_{\mathcal{D}}(B, C)$ tenemos que:

$$G(g \circ_{\mathcal{D}} f) = \mu_C \circ T(g \circ_{\mathcal{D}} f) = \mu_C \circ T(\mu_C \circ Tg \circ f) = \mu_C \circ T\mu_C \circ T^2 g \circ Tf$$

$$=^* \mu_C \circ \mu_{TC} \circ T^2 g \circ Tf =^{**} \mu_C \circ Tg \circ \mu_B \circ Tf = Gg \circ Gf$$

Con $(*)$ consecuencia del diagrama (4.3) de la definición de mónada y $(**)$ consecuencia de la naturalidad de $\mu : T^2 \Rightarrow T$.

- G se lleva bien con las identidades. $G\eta_A = \mu_A \circ T\eta_A = 1_{TA}$ por el diagrama (4.4) de la definición de mónada.

4. La prueba de que $F \dashv G$ es sencilla. El isomorfismo natural que necesitamos para establecer la adjunción es precisamente:

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

$$\phi_{A,B}(f) = f$$

La comprobación de la naturalidad de ϕ es rutinaria.

□

4.3. En Programación

4.3.1. Adjunciones

Currificación Definimos el siguiente funtor producto en Haskell.

```
type Product b a = (b, a)

instance Functor (Product fijo) where
  -- fmap :: a -> c -> (fijo, a) -> (fijo, c)
  fmap f (x, a) = (x, f a)
```

Veamos que los funtores `Product fijo` y y `Reader fijo` son funtores adjuntos. Para comprobarlo necesitamos definir una biyección natural entre los conjuntos $\mathrm{Hom}_{\texttt{Hask}}(\texttt{Product fijo a}, z) \cong \mathrm{Hom}_{\texttt{Hask}}(a, \texttt{Reader fijo z})$. La biyección se puede implementar de la siguiente forma:

```
biy :: ((Product fijo a) -> z) -> (a -> (Reader fijo z))
biy f a = Reader (\fijo -> f (a, fijo))

-- y la inversa
biy_inv :: (a -> Reader fijo z) -> (Product fijo a -> z)
biy_inv f (fijo, a) = (f a) fijo
```

Se puede comprobar que son inversas fácilmente. La unidad la podemos implementar tambień:

```
unit :: a -> Reader fijo (Product fijo a)
unit a = Reader (\fijo -> (fijo, a))
```

Y la counidad:

```
counit :: Product fijo (Reader fijo a) -> a
counit (fijo, Reader f) = f fijo
```

Y sabemos que son naturales porque son polimórficos en la variable `a`.

Esta adjunción nos dice una cosa sobre Haskell: es equivalente dar una función que recibe dos parámetros de tipo `A` y de tipo `B` y devuelve uno de tipo `Z` que dar una función que recibe un parámetro de tipo `A` y devuelve una función de tipo `B ->Z`. Esto motiva la sintaxis de Haskell para llamar a función. Habitualmente cuando en un lenguaje se implementa una función `f` que recibe dos parámetros ha de ser llamada como:

```
f(a, b)
```

En Haskell esto no es así. Debido a que *es lo mismo* definir una función que recibe dos parámetros que una función que recibe un parámetro y devuelve otra función tenemos que Haskell opta por la segunda. Entonces la sintaxis de Haskell

```
f a b = (f a) b
```

Es decir, se aplica la función `f` al valor `a`, lo que devuelve una función que recibe un parámetro `b` y esta se aplica a continuación al valor `b`. Esto es útil en Haskell por varios motivos que explicitaremos.

4.3.2. Mónadas

Al igual que existe una typeclass `Functor` también existe una typeclass `Monad` en la bibiloteca estándar de Haskell. La definición es la siguiente:

```
class Monad M where
  return :: a -> M a
  bind :: M a -> (a -> M b) -> M b
```

Esta definición de Mónada es similar a la que dimos originalmente. Podemos ver que `return` es precisamente la transformación natural η. ¿Pero cómo se relaciona `bind` con la terminología que usamos anteriormente? Implementamos la siguiente función:

```
mu :: Monad m => m (m a) -> m a
mu mma = bind mma id
```

4.3.2.1. Notación `do`

Capítulo 5

Conclusion

Bibliografía

[1] The University Of Glasgow. *Data.Functor*. URL: `http://hackage.haskell.org/package/base-4.11.1.0/docs/Data-Functor.html` (visitado 09-06-2018).

[2] Saunders McLane. "Categories for the Working Mathematician". En: second. University of Chicago: Springer, 1997, pág. 8.

[3] Saunders McLane. "Categories for the Working Mathematician". En: second. University of Chicago: Springer, 1997, pág. 7.

[4] Patrik Jansson Nils Anders Danielsson John Hughes y Jeremy Gibbons. "Fast and Loose Reasoning is Morally Correct". En: (POPL '06 Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages).

[5] Kevin D. Smith y col. *PEP 318 – Decorators for Functions and Methods*. https://www.python.org/dev/ 0318/, 2013.

[6] Philip Wadler. "Theorems for Free!" En: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: ACM, 1989, págs. 347-359. ISBN: 0-89791-328-0. DOI: `10.1145/99370.99404`. URL: `http://doi.acm.org/10.1145/99370.99404`.

Apéndice A

Appendix Title