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

1. Categorías y funtores **15**

1.1. Categorías . . . 15

1.1.1. Definición . . . 15

1.1.2. Categoría Producto . . . 17

1.2. Funtores . . . 17

1.2.1. Definición . . . 17

1.2.2. Funtores fieles y plenos . . . 19

1.2.3. Funtores Identidad . . . 19

1.2.4. Composición de funtores . . . 20

1.2.5. Bifuntores . . . 20

1.2.6. Producto de funtores . . . 20

1.3. En programación . . . 20

1.3.1. Categorías . . . 20

1.3.2. Funtores . . . 21

2. Construcciones elementales **29**

2.1. Elementos . . . 29

2.2. Monomorfismos . . . 30

2.3. Isomorfismos . . . 30

2.4. Productos . . . 31

2.4.1. Ejemplos . . . 31

2.5. Objetos finales . . . 32

2.5.1. Ejemplos . . . 32

2.6. Dualidad . . . 32

2.6.1. La categoría opuesta . . . 32

2.6.2. Epimorfismos . . . 33

2.6.3. Objetos iniciales . . . 34

2.6.4. Coproducto . . . 35

2.6.5. Funtores $\mathrm{Hom}(-, A)$ . . . 35

2.6.6. Funtor opuesto . . . 36

2.7. En programación . . . 36

2.7.1. Objetos iniciales y finales . . . 36

2.7.2. Productos . . . 37
2.7.3. Coproductos . . . 37

3. Transformaciones Naturales y el lema de Yoneda 39
3.1. Transformaciones Naturales . . . 39
3.1.1. Categorías de funtores . . . 40
3.1.2. Funtores a categorías de funtores . . . 41
3.1.3. Isomorfismos naturales . . . 41
3.1.4. Transformaciones naturales a través de funtores . . . 42
3.2. Lema de Yoneda . . . 42
3.3. Programación . . . 45
3.3.1. Transformaciones Naturales . . . 45

4. Adjunciones y Mónadas 49
4.1. Adjunciones . . . 49
4.1.1. Definición . . . 49
4.1.2. Unidad y counidad . . . 50
4.1.3. Ejemplos . . . 51
4.2. Mónadas . . . 53
4.2.1. Definición . . . 53
4.2.2. Las adjunciones dan lugar a mónadas . . . 55
4.2.3. Categoría de Kleisli de una mónada . . . 57
4.3. En Programación . . . 59
4.3.1. Adjunciones . . . 59
4.3.2. Mónadas . . . 60

5. Patrones de diseño 73
5.1. El patrón categórico . . . 73
5.1.1. Pipes . . . 74
5.2. Monad Transformers . . . 78
5.2.1. Ejemplo de aplicación . . . 78
5.3. Otros patrones de diseño . . . 81
5.3.1. `Applicative` . . . 81

A. Haskell Básico 89
A.1. Expresiones y tipos . . . 89
A.2. Funciones . . . 89
A.3. Expresiones lambda . . . 91
A.4. Definir tipos . . . 91
A.5. Funciones con tipos variables . . . 93
A.6. Tipos Parametrizados . . . 94
A.7. Listas . . . 94
A.8. Typeclasses . . . 95

Resumen y palabras clave

Abstract and keywords

To mum and dad

Declaration

I declare that..

Introducción

El lenguaje de programación haskell surge en el año 1990 con la idea de aunar los esfuerzos de múltiples instituciones académicas en el desarrollo de un lenguaje de programación funcional. Este lenguaje de programación estaba pensado para ser una base común sobre la que realizar docencia e investigación en la programación funcional.

Haskell cumplió su propósito de ser el vehículo de numerosa innovaciones en materia del diseño de lenguajes de programación. Entre otros desarrollos, haskell hizo de un concepto nacido en la teoría de categorías, las mónadas, una construcción fundamental para el lenguaje.

Conforme se refinaban las especificaciones del lenguaje, la idea de utilizar haskell en la industria era cada vez más factible. Hoy en día haskell es usado en empresas de sectores tan diversos como la banca, las redes sociales o la mensajería instantánea.

El objetivo del trabajo será comprender la forma en la que la teoría de categorías ha sido aplicada al diseño de software en el lenguaje de programación haskell. El objetivo del trabajo, naturalmente, no puede cumplirse sin estudiar la teoría de categorías de la que emanan estas aplicaciones.

Comenzamos el trabajo por las definiciones más básicas de la teoría de categorías: las categorías y los funtores. Ya en el primer capítulo podremos encontrar los primeros ejemplos de conceptos categóricos que se usan diariamente en el desarrollo de software en haskell.

En el segundo capítulo mostraremos la potencia expresiva de las categorías para la realización de distintas construcciones sobre estructuras matemáticas diferentes. Especializaremos algunas de estas construcciones a la programación.

El tercer capítulo tratará una de las nociones más importantes de la teoría de categorías, que nos servirá más adelante para enunciar la definición de mónada: las transformaciones naturales. Aprovecharemos la teoría construida hasta el momento para enunciar y demostrar el lema de Yoneda, uno de los resultados centrales de la teoría de categorías. En la parte de aplicaciones a la programación veremos cómo las transformaciones naturales permiten establecer relaciones entre constructores de tipos polimórficos.

En el cuarto capítulo conseguiremos definir qué son las monadas. Tendremos que pasar antes por la definición de adjunción, una abstracción que ha sido identificada por múltiples autores como uno de los conceptos clave de la teoría. Finalizaremos la parte del trabajo dedicada a las matemáticas analizando la relación entre las adjunciones y las mónadas. Al final del capítulo intentaremos justificar el rol de las mónadas en la programación funcional.

Finalizaremos el trabajo en el capítulo quinto. Dedicaremos este capítulo a la exposición de dos patrones de diseño que, junto con las mónadas, muestran la influencia de la teoría

de categorías en la forma en la que se programa en haskell. Aprovecharemos también para comentar brevemente otros patrones de diseño que indicarían líneas de futuro trabajo.

El libro *Topos, Triples and Theories* ha sido la referencia principal para el estudio de la teoría de categorías durante la elaboración de este trabajo. Ocasionalmente se ha utilizado el libro *Categories for the working mathematician* para la consulta de algún tema concreto.

El principal recurso de consulta sobre el lenguaje de programación haskell ha sido *Real World Haskell*. Esporádicamente se han consultado otras fuentes que serán referenciadas cuando corresponda a lo largo del texto.

Índice general

Capítulo 1

Categorías y funtores

1.1. Categorías

1.1.1. Definición

Tradicionalmente las matemáticas están fundamentadas en una teoría de conjuntos. Cuando partimos de una teoría de conjuntos no hace falta (*no se puede*) definir *qué* es un conjunto. Ocurre de forma similar con los conceptos de *elemento* y *pertenece*, que son básicos en la teoría. La teoría de categorías se puede utilizar también para fundamentar las matemáticas, y en este sentido no se podrían dar definiciones, en términos de otros conceptos, de nociones como *categoría, objeto, flecha o composición*. Siguiendo una línea de trabajo similar podríamos decir que tenemos una categoría 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 $\mathcal{C}$ y que notaremos por $f : A \to B$ 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 inicial a la teoría de categorías y con el suficiente esfuerzo se puede evitar hacer uso de la teoría de conjuntos. En vista de que este no es un trabajo que pretenda tratar sobre la fundamentación de las matemáticas no seguiremos un enfoque tan estricto y consideraremos que para cada par de objetos A y B de la categoría $\mathcal{C}$ las flechas entre A y B forman un conjunto, al que llamaremos $\mathrm{Hom}_{\mathcal{C}}(A, B)$ o simplemente $\mathrm{Hom}(A, B)$ cuando esté claro a que categoría $\mathcal{C}$ nos referimos. Por otro lado, y en vista de esta notación, la operación de composición induce una aplicación $\circ : \mathrm{Hom}(B, C) \times \mathrm{Hom}(A, B) \longrightarrow \mathrm{Hom}(A, C)$ para cada terna de objetos A, B y C. Las categorías en las que se puede hablar de los conjuntos $\mathrm{Hom}(A, B)$ son conocidas en la literatura como categorías *localmente pequeñas* y serán las únicas categorías consideradas a lo largo de este trabajo. Denotaremos por $\mathcal{O}b(\mathcal{C})$ y $\mathcal{A}r(\mathcal{C})$ a la *clase* de todos los objetos y a la clase de todas las flechas de la categoría $\mathcal{C}$ respectivamente, sin profundizar en la noción precisa de clase utilizada por estar fuera de nuestro objetivo.

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 y cada aplicación f entre el conjunto A y 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. No se puede asumir en general que los objetos de una categoría forman un conjunto por ejemplos como este: no tiene sentido hablar del conjunto de todos los conjuntos.

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 utilizar con 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 $g : B \longrightarrow C$, entonces $F(g \circ f) = Fg \circ Ff$.

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 éste 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:

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ & \searrow^{F(g\circ f)} & \downarrow{\scriptstyle Fg} \\ & & FC \end{array}$$

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 de conjunto subyacente 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 $\mathrm{Hom}(A, -) : \mathcal{C} \to \mathtt{Set}$. 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:

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

.

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

Viendo que $\mathrm{Hom}(A, -)$ 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 (respectivamente 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 (respectivamente 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 $f : C \longrightarrow C'$. Se tiene por tanto que $F \circ G$ 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 $- \times - : \texttt{Set} \times \texttt{Set} \longrightarrow \texttt{Set}$ que a cada par de conjuntos le asigna su producto cartesiano.

Dado un bifuntor podemos obtener otro funtor *fijando* uno de los parámetros. Por ejemplo, si C_1 es un objeto de $\mathcal{C}_1$ podemos considerar el funtor $F_{C_1} : \mathcal{C}_2 \longrightarrow \mathcal{D}$ tal que para cada objeto C_2 de $\mathcal{C}_2$ $F_{C_1}(C_2) = F(C_1, C_2)$ y para cada flecha de $\mathcal{C}_2$ de la forma $f : C_2 \longrightarrow C_2'$ definimos $F_{C_1}(f) = F(1_{C_1}, f)$. Comprobar que F_{C_1} es efectivamente un funtor es sencillo.

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 a = f (g a)
```

En la declaración de tipo de la función `(.)` (la línea superior) utilizamos letras en minúsculas como variables de tipo. Esto quiere decir que la función es polimórfica en esos tipos, que luego se especializarán según los valores con los que llamemos al operador.

Además de la composición tenemos la función `id`, que nos servirá como la flecha identidad en `Hask` para cada tipo:

```
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 cómo especializando nociones categóricas a `Hask` obtendremos un marco desde el que comprender mejor algunas construcciones habituales que se dan en 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 que un endofuntor en `Hask` tiene que asignar tipos (los objetos de `Hask`) a otros tipos y funciones (las flechas de `Hask`) a otras funciones de manera que se cumplan 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`, el tipo de los enteros) le asigna otro tipo (`vector<int>`, el tipo de los vectores 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 éstas 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:

```
-- digamos que f :: a -> b
-- y que g :: b -> c
-- Entonces

-- el funtor se lleva bien con la composición
fmap (g . f) = (fmap g) . (fmap f) :: F a -> F c

-- el funtor lleva identidades a identidades
fmap id = id :: F a -> F a
```

Aclaramos que esto no es código en haskell sino la especificación de las propiedades que debe cumplir `fmap` expresadas con la sintaxis de haskell. Nótese además que solo escribimos `fmap` sin hacer referencia a la instancia de `Funtor` cuya implementación de `fmap` estamos usando. El compilador de haskell es capaz de inferir, en general, según el contexto del código, a que instancia de `Functor` nos referimos cuando usamos `fmap`.

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 posibilidad de que no exista resultado en el sistema de tipos.

`Maybe` es un constructor de tipos en el sentido de que a cada tipo `A` (una vez más, usaremos nombres de tipos que comienzan con mayúscula para referirnos a tipos concretos, mientras que usaremos nombres en minúscula para referirnos a tipos polimórficos) le asigna un tipo `Maybe A`, cuyos valores son o bien de la forma `Just v` para `v` algún valor de tipo `A` o bien `Nothing`. Resulta que se puede dotar a `Maybe` de 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 funciones de tipo `Maybe a ->Maybe b`. Esta función no hace nada si recibe un `Nothing` 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 según el caso. Un caso de uso muy habitual de `Either` es representar cómputos que, al igual que `Maybe`, podrían fallar, pero dando detalles sobre el error en caso de que lo hubiera.

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 PremiosTrabajadores
dar_premio age
  | age < 16 = Nothing
  | 16 <= age && age < 40 = Just PremiosJovenes
  | 40 <= age && 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 PremiosTrabajadores
dar_premio edad
  | edad < 16 = Left "Demasiado joven para estar en edad laboral"
```

```
  | 16 <= edad && edad < 40 = Right PremiosJovenes
  | 40 <= edad && edad < 65 = Right PremiosMayores
  | 65 <= edad = Left "Demasiado mayor para estar en edad laboral"
```

¿Es `Either` instancia de la clase `Functor`? 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 instancia de `Functor` necesitamos que sólo tenga un parámetro. Resulta, sin embargo, que `Either a` sí que lo es. 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 :: (b -> c) -> Either a b -> Either a c
  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)
```

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`. Básicamente el tipo `Reader a b` es *lo mismo* que el tipo `a ->b` pero en ocasiones es útil tener `Reader a b` como tipo aparte. 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.

Composición de Funtores Para mostrar la expresividad de haskell definimos el funtor composición. En primer lugar definimos el tipo:

```
data Composition f g a = Composition (f (g (a))
```

Un constructor de tipo parametrizado por tres tipos, donde los dos primeros son a su vez constructores de tipos. Se puede interpretar el constructor de tipo `Composition f g` como la composición de los constructores de tipo `f` y `g`. Por ejemplo podemos construir el tipo `Composition [] Maybe Int` que tiene como valores, por ejemplo:

```
Composition [Just 3, Just 4]
Composition [Nothing, Nothing, Nothing, Just 300]
Composition []
```

Pues bien, si `f` y `g` son funtores, resulta que también podemos definir una instancia válida de `Functor` para `Compose fg` de la siguiente manera:

```
instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Composition fga) = Composition (fmap (fmap f) fga)
```

Esta construcción se corresponde con la construcción que hicimos de composición de funtores.

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 $f(a) = f(a')$ implica $a = a'$.*

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 ésta. 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 $fx = fy$ implica $x = y$.*

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 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 ésta. Entonces 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.

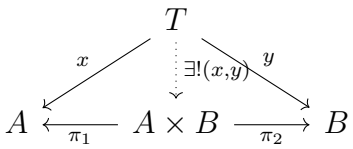
Es inmediato ver que los isomorfismos de la categoría `Set` se corresponden con las biyecciones y que los isomorfismos sobre `Grp`, `Ring`, `Top`, ... se corresponden con los isomorfismos de grupos, isomorfismos de anillos y homeomorfismos de espacios topológicos, respectivamente.

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 ésta. 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 $\pi_2((x, y)) = y$ (de ahora en adelante notaremos $\pi_i(x, y)$ por simplificar la notación).*

Expresaremos esto diciendo que el siguiente diagrama es conmutativo para cada T:



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, éste sólo 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 éstos 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 $\pi_2 \circ f = y$ sólo 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 tal y como vemos a continuación.

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 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 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 $g_1 \circ f = g_2 \circ f$ implica $g_1 = g_2$. 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 para todo $x \in \mathbb{Q}$ tenemos que existen a, b enteros tales que $x = \frac{a}{b}$ 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.

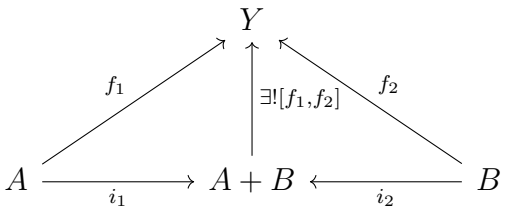
Otro ejemplo de categoría con objeto cero es la categoría de espacios vectoriales sobre un cuerpo K.

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 ésta. 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.*



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 alguna de 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`). El tipo `()` tiene como único valor el valor `()` (tipo y valor son homónimos en este caso) 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
snd (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
```

Comprobar que se lleva bien con las inyecciones es inmediato:

```
(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
```

En la biblioteca estándar tenemos la función `either :: (a ->c) ->(b ->c) ->Either a b ->c` que es precisamente la función que construye `f` a partir de `f1, 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:*

$$\begin{array}{ccc} FC & \xrightarrow{\lambda_C} & GC \\ {\scriptstyle Fg}\downarrow & & \downarrow{\scriptstyle Gg} \\ FC' & \xrightarrow{\lambda_{C'}} & GC' \end{array}$$

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

$$\begin{array}{ccc} G & \xrightarrow{\pi_G} & G^{ab} \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f^{ab}} \\ H & \xrightarrow{\pi_H} & H^{ab} \end{array}$$

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. Funtores a categorías de funtores

Sean $\mathcal{C}, \mathcal{C}'$ y $\mathcal{D}$ tres categorías y sea $F : \mathcal{C} \times \mathcal{C}' \longrightarrow \mathcal{D}$ un bifuntor. Veamos que F nos permite definir un funtor $H : \mathcal{C} \longrightarrow \mathcal{D}^{\mathcal{C}'}$. Definimos $HC = F_C = F(C, -) : \mathcal{C}' \longrightarrow \mathcal{D}$. Por otro lado, si $f : C_1 \longrightarrow C_2$ es una flecha de $\mathcal{C}$ definimos Hf como la transformación natural

$$Hf : F_{C_1} \Rightarrow F_{C_2},$$

$$Hf_{C'} = F(f, 1_{C'}) : F(C_1, C') \longrightarrow F(C_2, C').$$

Comprobar que $Hf : \mathcal{C} \longrightarrow \mathcal{D}^{\mathcal{C}'}$ es natural es rutinario.

3.1.3. 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:

$$\begin{array}{ccc} GC & \xrightarrow{\psi_C} & FC' \\ {\scriptstyle Gf}\downarrow & & \downarrow{\scriptstyle Ff} \\ GC' & \xrightarrow{\psi_{C'}} & FC' \end{array}$$

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. □

3.1.4. 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.

- 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 $H(A) = \mathrm{Hom}_{\mathcal{C}}(A, -)$ podemos notar que nuestro funtor L es precisamente

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

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:

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

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 $F : \mathcal{C} \longrightarrow \mathtt{Set}$ 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:

$$\begin{array}{ccc} \mathrm{Hom}(A, A) & \xrightarrow{\tau_A} & FA \\ {\scriptstyle \mathrm{Hom}(A,f)}\downarrow & & \downarrow{\scriptstyle Ff} \\ \mathrm{Hom}(A, C) & \xrightarrow{\tau_C} & FC \end{array}$$

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. □

El lema de Yoneda se puede interpretar en términos de elementos categóricos. Si tenemos una categoría $\mathcal{C}$, un objeto A de ésta 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:

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

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 cada 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 cada tipo `A` 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` (usamos `a` minúscula cuando nos referimos a una función polimórfica en la variable `a`) que, especializando a cada tipo del lenguaje, será la componente en cada objeto de `Hask` de nuestra transformación natural.

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 la variable de tipo `a` esté implementada con la misma expresión independientemente de futuras especializaciones de `a`. 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` (donde, insistimos, la `a` minúscula significa que se trata de una función polimórfica en el tipo `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 con el servidor"`. Los valores de la forma `Right v :: Either b a` representan cómputos exitosos con valor de retorno `v :: a`.

En este sentido podemos interpretar 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.

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 funtor es un funtor trivial porque lleva a todos los tipos esencialmente al mismo tipo y además lleva todas las funciones, también esencialmente, a la identidad. Supongamos que tenemos una función polimórfica de tipo `f :: F a ->B` donde `B` es un tipo concreto. Esta función se puede extender a otra función `fConst :: F a ->Const B a` de la siguiente forma:

```
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. Entonces cumple las condiciones de naturalidad. Especificando la condición de naturalidad con la sintaxis de Haskell llegamos a que para cualquier función `g :: C ->C'` (donde `C` y `C'` son dos tipos fijos pero arbitrarios):

```
fConst . fmap g = fmap g . fConst
```

donde el `fmap` que aparece en el miembro de la izquierda es el de la instancia de `Functor` de `F` y el de la derecha es el de `Const B`. Pero vimos anteriormente que la implementación de `fmap` de `Const b` es básicamente la identidad. Esto nos lleva a comprobar que: `f . fmap g = f`. Es decir, dada cualquier función de tipo `f :: F a ->B` los valores `f` son invariantes a través de aplicaciones de la forma `fmap g`. Ponemos dos ejemplos que aclararán esta proposición:

- Consideramos la función `length :: [a] ->Int`. En este caso lo que estamos diciendo es que dada una lista `l :: [A]` y cualquier función `h :: A ->B` tenemos que

  ```
  length (fmap h l) = length l
  ```

  Es decir, la longitud de una lista no cambia a través de funciones de la forma `fmap h`.

- Consideramos ahora la función

  ```
  isNothing :: Maybe a -> Bool
  isNothing Nothing = True
  isNothing (Just x) = False
  ```

  En este caso dado un valor `m :: Maybe A` y cualquier función `h :: A ->B` tenemos que:

  ```
  isNothing (fmap f m) = isNothing m
  ```

  Es decir, que las funciones de la forma `fmap h` no pueden llevar un valor `Nothing :: Maybe C` a uno `Just x :: Maybe D` ni viceversa.

Transformaciones naturales a través de funtores Realizamos ahora la construcción de las transformaciones naturales a través de los funtores en Haskell. Utilizaremos el tipo `Composition f g a` que usamos en uno de los ejemplos de funtores.

```
natApplication :: (Functor f, Functor g, Functor h) =>
  (f a -> g a) -> Composition h f a -> Composition h g a
natApplication tau (Composition hfa) =
  Composition (fmap tau hfa)
```

Esta construcción es la que nos permite dada una transformación natural `tau :: F a ->G a` y un funtor `H` obtener una transformación natural `natApplication tau :: H F a ->H G a`. La transformación natural `F H a ->G H a` se obtiene simplemente restringiendo `tau` a tipos de la forma `H a`.

Veamos un ejemplo de uso de `natApplication`. Especializaremos `f` al funtor `Either String`, `g` al funtor `Maybe` y `h` al funtor `[]`. La transformación natural que usaremos será `discardLeft :: Either String a ->Maybe a`. A continuación ponemos algunos ejemplos de cómo funciona `natApplication discardLeft`.

```
-- natApplication discardLeft :: [Either String a] -> [Maybe a]
natApplication
    discardLeft
    (Composition [Left "error", Right 5, Left "test", Right 6])

-- resultado: [Nothing, Just 5, Nothing, Just 6]

natApplication
    discardLeft
    (Composition [Right "gg", Right "hh", Left "hh"])
-- resultado [Just "gg", Just "hh", Nothing ]
```

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*

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

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

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 $\mathcal{C}$:

$$\begin{array}{ccccc} C & \xrightarrow{\phi_{C,D}(f)} & GD & \xrightarrow{Gg} & GD' \\ & \searrow & & \nearrow & \\ & & \phi_{C,D'}(g\circ f) & & \end{array} \tag{4.1}$$

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 $\mathcal{D}$

$$\begin{array}{ccccc} FC' & \xrightarrow{Fh} & FC & \xrightarrow{\phi^{-1}_{C,D}(f)} & D \\ & \searrow & & \nearrow & \\ & & \phi^{-1}_{C',D}(f\circ h) & & \end{array} \tag{4.2}$$

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:

$$\begin{array}{ccc} C & \xrightarrow{\eta_C} & GFC \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle GFf} \\ C' & \xrightarrow{\eta_{C'}} & GFC' \end{array}$$

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):

$$C \xrightarrow{\eta_C} GFC \xrightarrow{GFf} GFC'$$
$$\phi_{C,FC'}(f \circ 1_C) = \phi_{C,FC'}(Ff)$$

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

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

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

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

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 $f(b,a) = f'(b)(a)$ para toda pareja $(b,a) \in B \times A$. Aplicando la definición de $\phi_{B,Z}$ tenemos $\phi_{B,Z}(f)(b)(a) = f(b,a) = f'(b)(a)$ también para toda pareja $(b,a) \in B \times A$, 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 $f_1(b,a) = \phi_{B,Z}(f_1)(b)(a) = \phi_{B,Z}(f_2)(b)(a) = f_2(b,a)$ para todo par $(b,a) \in B \times A$, con lo que $f_1 = f_2$.

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):

$$\begin{array}{ccccc} B \times A & \xrightarrow{f \times A} & B' \times A & \xrightarrow{\phi^{-1}_{B',Z}(g)} & Z \\ & \searrow^{\phi^{-1}_{B,Z}(g \circ f)} & & \nearrow & \end{array}$$

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:

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

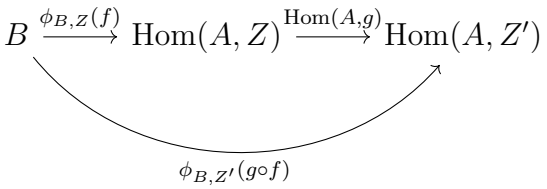
pero además

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

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):



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 $\eta : 1_{\mathcal{C}} \Rightarrow T$ y $\mu : T^2 \Rightarrow T$ son dos transformaciones naturales tales que los siguientes diagramas conmutan.*

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ {\scriptstyle \mu T}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow{\mu} & T \end{array} \tag{4.3}$$

$$\begin{array}{ccccc} T & \xrightarrow{T\eta} & T^2 & \xleftarrow{\eta T} & T \\ & \searrow & \downarrow{\scriptstyle \mu} & \swarrow & \\ & & T & & \end{array} \tag{4.4}$$

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 $T(X) = M \times X$;
- la transformación natural $\eta_X : X \longrightarrow T(X)$, dada por $\eta_X(x) = (e, x)$ (con $e \in M$ el elemento neutro);
- la transformación natural $\mu_X : T^2(X) \longrightarrow T(X)$ dada 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 licencia de identificar T^2X con $M \times M \times X$ en lugar de 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):

$$\begin{array}{ccc} M \times M \times M \times X & \xrightarrow{T\mu_X} & M \times M \times X \\ {\scriptstyle \mu_{TX}}\downarrow & & \downarrow{\scriptstyle \mu_X} \\ M \times M \times X & \xrightarrow{\mu_X} & M \times X \end{array}$$

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

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

y 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)

$$\begin{array}{ccccc} T(X) & \xrightarrow{\eta_{TX}} & T^2(X) & \xleftarrow{T\eta_X} & T(X) \\ & \searrow & \downarrow{\scriptstyle \mu_X} & \swarrow & \\ & & T(X) & & \end{array}$$

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

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

y además

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

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 $\mu_X : C + (C + X) \longrightarrow C + X$, $\mu_X = [[1_C, 1_C], 1_X]$ (donde estamos identificando $C + (C + X)$ con $(C + C) + X$ para simplificar la notación).

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

Aunque las nociones de adjunción y mónada surgieron de forma separada dentro del estudio de la teoría de categorías, no pasó mucho tiempo hasta que se descubrió que ambos conceptos estaban relacionados. En 1961, P. Huber demuestra que a partir de cada par de funtores adjuntos se puede obtener una mónada. Exponemos la construcción que realizó en el siguiente teorema.

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. El diagrama (4.3) de la definición de mónada es en este caso el diagrama:

$$\begin{array}{ccccc} GF & \xrightarrow{GF\eta} & GFGF & \xleftarrow{\eta GF} & GFGF \\ & \searrow^{=} & \downarrow{\scriptstyle G\epsilon F} & \swarrow^{=} & \\ & & GF & & \end{array}$$

es decir, hay que probar que:

$$G\epsilon F \circ GF\eta = 1_{GF} = G\epsilon F \circ \eta GF.$$

Probaremos la igualdad de la derecha. Sea C un objeto de $\mathcal{C}$. La componente en C de la transformación natural del miembro de la derecha de la ecuación es $G\epsilon_{FC} \circ \eta_{GFC}$. Por la condición (4.1) de la definición de adjunción, tenemos que el siguiente diagrama en $\mathcal{D}$:

$$FGFC \xrightarrow{1_{FGFC}} FGFC \xrightarrow{\epsilon_{FC}} FC$$

lo podemos trasladar al siguiente diagrama conmutativo en $\mathcal{C}$:

$$\begin{array}{ccccc} GFC & \xrightarrow{\eta_{GFC}} & GFGFC & \xrightarrow{G\epsilon_{FC}} & GFC \\ & \searrow & & \nearrow & \\ & & \phi_{GFC,FC}(\epsilon_{FC}) & & \end{array}$$

y entonces

$$\phi_{GFC,FC}(\epsilon_{FC}) = G\epsilon_{FC} \circ \eta_{GFC},$$

pero por la definición de ϵ_{FC}, tenemos que

$$1_{GFC} = \phi_{GFC,FC}(\epsilon_{FC}) = G\epsilon_{FC} \circ \eta_{GFC},$$

con lo que $G\epsilon F \circ \eta GF = 1_{GF}$. La prueba de que $G\epsilon F \circ GF\eta = 1_{GF}$ es análoga.

Queda probar la condición (4.4) de la definición de mónada. El diagrama (4.4) es en este caso:

$$\begin{array}{ccc} GFGFGF & \xrightarrow{G\epsilon FGF} & GFGF \\ \downarrow{\scriptstyle GFG\epsilon F} & & \downarrow{\scriptstyle G\epsilon F} \\ GFGF & \xrightarrow{G\epsilon F} & GFGF \end{array}$$

y la conmutatividad de este diagrama es equivalente a la del siguiente:

$$\begin{array}{ccc} FGFG & \xrightarrow{\epsilon FG} & FG \\ \downarrow{\scriptstyle FG\epsilon} & & \downarrow{\scriptstyle \epsilon} \\ FG & \xrightarrow{\epsilon} & 1_{\mathcal{D}} \end{array}$$

pero este diagrama en cualquier objeto D de la categoría $\mathcal{D}$ es

$$\begin{array}{ccc} FGFGD & \xrightarrow{\epsilon_{FGD}} & FG \\ \downarrow{\scriptstyle FG\epsilon_D} & & \downarrow{\scriptstyle \epsilon_D} \\ FG & \xrightarrow{\epsilon_D} & D \end{array}$$

que es precisamente la condición de naturalidad de $\epsilon : FG \Rightarrow 1_{\mathcal{D}}$, aplicada a la flecha $\epsilon_D : FGD \longrightarrow D$. □

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 por pasos.

1. Construimos $\mathcal{D}$. 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

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

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

     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.

   - 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

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

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

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

   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. Se cumple que

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

   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 lleva identidades a identidadese. Se cumple que $G\eta_A = \mu_A \circ T\eta_A = 1_{TA}$ 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.

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

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

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

```
biy :: forall a b z. (Product b a -> z) -> (a -> (Reader b z))
biy f a = Reader h
  where
    h :: b -> z
    h b = f (Product (b, a))

biy_inv :: forall a b z. (a -> (Reader b z)) -> (Product b a -> z)
biy_inv f (Product (b, a)) = h b
  where
    h :: b -> z
    (Reader h) = f a
```

Se puede comprobar que estas funciones son inversas fácilmente. La unidad de la adjunción la podemos implementar de la siguiente manera:

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

Y la counidad:

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

Y sabemos que son naturales porque es una función polimórfica de tipo `F a ->G a` con `F` y `G` funtores.

Esta adjunción nos dice una cosa sobre haskell: es equivalente dar una función que recibe un parámetro de tipo `A` y otro de tipo `B` y devuelve un valor 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 funciones. Habitualmente cuando en un lenguaje se implementa una función `f` que recibe dos parámetros esta función ha de ser llamada como:

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

Esto no es así en haskell. Cuando en haskell escribimos `f a b` (lo que interpretamos como llamar a una función con dos parámetros) lo que ocurre en realidad es que se llama a la función `f` con el parámetro `a`. Esto devuelve una función que recibe un parámetro de tipo `b`, que se aplica finalmente sobre el valor `b`. En otras palabras:

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

La técnica de transformar funciones que reciben múltiples parámetros en funciones que reciben un solo parámetro pero devuelven otra función es conocida como currificación. La currificación automática en haskell permite ser muy expresivo a la hora de construir funciones. Un ejemplo sencillo que ilustra el tipo de operaciones que se pueden hacer gracias a la currificación automática es el siguiente:

```
add :: Int -> Int -> Int
add x y = x + y

mult :: Int -> Int -> Int
mult x y = x * y
```

Dadas estas dos funciones en haskell podemos considerar las funciones `add 3 :: Int ->Int` que a cada entero `x` le asigna el entero `x+3` o la función `mult 5 :: Int ->Int` que a cada entero `x` le asigna el entero `5 * x`. Las funciones `add 3` y `mult 5` se pueden utilizar como cualquier otra función del lenguaje. Por ejemplo:

```
fmap (add 3 . mult 5) [1,2,3] -- [8, 13, 18]
```

4.3.2. Mónadas

Haskell dispone en su biblioteca estándar de la typeclass `Monad`, así como de varias instancias de esta. La interfaz que presenta es la siguiente:

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

(la sintaxis utilizada en la función `>>=` significa que será utilizada como operador infijo) En la documentación se afirma que una instancia de `Monad` debe implementar las funciones `return` y `>>=` de forma que se satisfagan las siguientes leyes:

```
-- si v :: a
--    m :: m a
--    k :: a -> m b entonces
--    h :: b -> m c
(return v >>= k) = k v -- ley 1

(m >>= return) = m -- ley 2

m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h -- ley 3


-- en relación a fmap, f :: a  -> b
--                     m :: ma
fmap f m  =  m >>= return . f
```

No es inmediato ver que esta definición de mónada es equivalente a la que hemos trabajado durante el capítulo. De los tres ingredientes que necesitamos para definir una mónada tenemos dos:

- El funtor, que en este caso es `m`.
- La transformación natural $\eta : 1_{\texttt{Hask}} \Rightarrow \texttt{m}$, que en este caso se llama `return :: a ->m a`.

Sin embargo, la transformación natural $\mu : \texttt{m}^2 \Rightarrow \texttt{m}$ que nos falta la podemos obtener a partir de `bind` de la siguiente forma:

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

Las condiciones (4.3) y (4.4) de la definición de mónada se pueden expresar con sintaxis de haskell de la siguiente forma:

```
mu . mu = mu . (fmap mu) :: m (m (m a)) -> m a
-- y
mu . return = id = mu . (fmap return) :: m a -> m a
```

Veamos que ambas equaciones se cumplen si asumimos las leyes que debe cumplir cualquier instancia de `Monad`.

```
-- consideramos mmma :: m (m (m a))
mu . mu mmma = mu (mu mmma)
             = mu (mmma >>= id)
             = (mmma >>= id) >>= id
             -- por la ley 3
             = mmma >>= (\x -> (x >>= id))
             -- por la definición de mu
             = mmma >>= (\x -> mu x)
             = mmma >>= mu

-- por otro lado
mu . (fmap mu) mmma = mu (fmap mu mmma)
                    = mu (mmma >>= return . mu)
                    = (mmma >>= return . mu) >>= id
                    -- por la ley 3
                    = mmma >>= (\x -> return (mu x) >>= id)
                    -- por la ley 1
                    = mmma >>= (\x -> mu x)
                    = mmma >>= mu

-- y por tanto
mu . mu = mu . (fmap mu)


-- por otro lado
mu . return ma = mu (return ma)
               = (return ma) >>= id
               -- por la ley 1
               = ma

mu . (fmap return) ma = mu (fmap return ma)
                      = mu (ma >>= return . return)
                      = (ma >>= return . return) >>= id
                      -- por la ley 3
                      = ma >>= (\x -> return (return x) >>= id)
                      -- por la ley 1
                      = ma >>= (\x -> return x)
                      = ma >>= return
                      -- por la ley 2
                      = ma

-- lo que prueba definitivamente que
```

```
mu . return = id = mu . (fmap return)
```

Esto nos lleva a confirmar que una instancia de `Monad` en haskell es efectivamente una mónada en `Hask`. La implementación de `return` y `>>=` determina la implementación de otra función:

```
(>>) :: m a -> m b -> m b
(>>) ma mb = (ma >>= \x -> mb)
```

que secuencia los efectos de ambas instancias de la mónada descartando el valor del primero. Entenderemos mejor qué significa esto cuando veamos los ejemplos.

4.3.2.1. Ejemplos de Mónadas en Haskell

Maybe Resulta que `Maybe` no es solo un funtor sino que además es un `Monad`. La implementación de la instancia es la siguiente:

```
instance Monad Maybe where
  return x = Just x -- equivalentemente return = Just
  -- ma :: Maybe a, f :: a -> Maybe b
  ma >>= f = case ma of
    Nothing -> Nothing
    Just x -> f x
```

Esta instancia permite componer computaciones que pueden resultar en error sin tener que hacer gestión explícita de los casos de error. Proponemos un ejemplo a partir de la función `head_safe :: [a] ->Maybe a` que implementamos cuando introdujimos `Maybe`:

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

Supongamos que queremos implementar una función que recibe tres listas de enteros como parámetro y devuelve la suma de los primeros elementos de estas listas. Como las listas pueden ser vacías la computación puede fallar. Una forma de implementar esta lógica sería:

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

Donde hacemos distinción de casos de forma explícita para propagar el error. La implementación de la instancia de `Monad` de `Maybe` nos permite gestionar esta lógica de error de forma implícita y general:

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

Y la lógica de la propagación de errores la gestiona transparentemente la implementación de `>>=` para `Maybe`. Comprobar que ambas implementaciones de la función `sum_three_heads` son equivalentes es un ejercicio instructivo.

`Either a` También existe en la biblioteca estándar de haskell una instancia de `Monad` para el funtor `Either a`. La implementación es la siguiente:

```haskell
instance Monad (Either a) where
    return x = Right x

    -- (>>=) :: Either a b -> (b -> Either a c) -> Either a c
    mb >>= f = case mb of
        Left a -> Left a
        Right b -> f b
```

La implementación de `Monad` para `Either a` es similar a la de `Maybe` en el sentido de que los valores de error cortocircuitan la computación, en el sentido de que si a lo largo de la cadena de cómputos se encuentra un valor de la forma `Left a` ya no se ejecuta la función `f`.

Consideramos las siguientes funciones que modelan el funcionamiento de un compilador:

```haskell
data ErrorCompilacion = ErrorLexico | ErrorSintactico | ErrorSemantico

analisisLexico :: String -> Either ErrorCompilacion [Token]
analisisSintactico :: [Token] -> Either ErrorCompilacion ArbolSintactico
analisisSemantico :: ArbolSintactico -> Either ErrorCompilacion Programa
```

Podríamos definir ahora la función `compilar` que reúne las tres fases gestionando los posibles errores que surjan por el camino. Una forma de definir esta función sería:

```haskell
compilar :: String -> Either ErrorCompilacion Programa
compilar s = case analisisLexico s of
    Right tokens -> case analisisSintactico token of
        Right arbolSintactico -> case analisisSemantico arbolSintactico of
            Right programa -> Right programa
```

```
            Left errorSemantico -> Left errorSemantico
        Left errorSintactico -> Left errorSintactico
    Left errorLexico -> Left errorLexico
```

Pero la instancia de `Monad` nos permite gestionar los errores de forma transparente:

```
compilar :: String -> Either ErrorCompilacion Programa
compilar s = analisisLexico s >>= (\tokens ->
    analisisSintactico tokens >>= (\arbolSintactico ->
        analisisSemantico arbolSintactico
        )
    )
```

De hecho nótese que usando la ley 3 de las mónadas en haskell podemos simplificar la expresión anterior a la expresión equivalente:

```
compilar :: String -> Either ErrorCompilacion Programa
compilar src =
  (analisisLexico src) >>= analisisSintactico >>= analisisSemantico
```

que sigue propagando perfectamente los errores pero es más fácil de entender.

Ponemos un ejemplo de para qué podría servir el operador `»` en esta mónada. Recordemos que:

```
(>>) :: m a -> m b -> m b
ma >> mb = ma >>= \_ -> mb
```

(la barra baja se utiliza para explicitar que no se va a usar ese parámetro de la lambda). Dijimos anteriormente que `»` ignora el "contenido" de una mónada pero acumula sus efectos. Veamos qué significa eso en la mónada `Either a`. Supongamos que queremos implementar una función que dada una cadena de texto representando un programa devuelva el número de tokens del código siempre y cuando el programa sea válido. Para ello podríamos hacer lo siguiente:

```
numTokens :: String -> Either ErrorCompilacion Int
numTokens src =
    compilar src >> analisisLexico src >>= (\tokens ->
        return (length tokens)
        )
```

En este ejemplo decimos que `»` ignora el "contenido" de la mónada porque aunque la compilación fuera exitosa no utilizaríamos para nada el valor de tipo `Programa` que nos devolvería la función `compilar`. Sin embargo acumulamos los efectos en el sentido de que si a lo largo de la compilación se produce algún error, la computación de `numTokens` cortocircuitará y nos devolverá el error, aunque el análisis léxico fuera exitoso. Esto se puede comprobar a mano sustituyendo `»` por su implementación:

```
numTokens :: String -> Either ErrorCompilacion Int
numTokens src =
    compilar src >>= (\x -> analisisLexico src >>= (\tokens ->
        return (length tokens)
        ))
```

y como la implementacion de `>>=` dice que `(Left error >>= f) = Left error` (sin utilizar `f`) si tenemos que el resultado de compilar es de la forma `Left error` entonces no se ejecuta la lambda que recibe el parámetro `x`.

`Reader a` `Reader a` también dispone de una instancia de `Monad` que podemos implementar de la siguiente forma:

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

instance Monad (Reader a) where
  return x = Reader (\a -> x) -- la función constantemente x
  -- (>>=) :: Reader a b -> (b -> Reader a c) -> Reader a c
  (Reader aToB) >>= f = Reader (\a ->
      let b = aToB b
          (Reader aToC) = f b
      in aToC a
      )
```

Esta implementación puede resultar confusa a quien no esté versado en haskell pero se puede interpretar como que si tenemos una función que va de un tipo fijo `A` a un tipo fijo `B` y tenemos otra función que va de un tipo fijo `B` a una función que va del tipo fijo `A` a un tipo fijo `C` podemos obtener una función que va de `A` a `C` (sin la necesidad de `B`). Podemos realizar una implementación de la misma lógica sin `Reader` de por medio de la siguiente forma:

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

La instancia de `Monad` de `Reader a` es útil para cuando tenemos varias funciones que reciben un parámetro del mismo tipo y queremos combinarlas entre ellas. Proponemos un ejemplo análogo al anterior en el que la compilación se realiza de acuerdo a un valor de configuración que todas las fases tienen que tener en cuenta. Una posible implementación sería (asumiendo que ninguna de las fases puede fallar para simplificar el ejemplo):

```
analisisLexico :: Config -> String -> [Token]
analisisSintactico :: Config -> [Token] -> ArbolSintactico
analisisSemantico :: Config -> ArbolSintactico -> Programa

compilar :: Config -> String -> Programa
```

```
compilar config s =
    let tokens = analisisLexico config s
        arbolSintactico = analisisSemantico config tokens
    in
        analisisSemantico config arbolSintactico
```

En este ejemplo tenemos que pasar el parámetro de configuración a todas las etapas. El `Reader a` monad nos permite simplificar este patrón. Para ello, definimos nuestras etapas como:

```
rAnalisisLexico :: String -> Reader Config ([Token])
rAnalisisLexico s = Reader (\conf -> analisisLexico conf s)

rAnalisisSintactico :: [Token] -> Reader Config (ArbolSintactico)
rAnalisisSintactico tokens = Reader (\conf -> analisisSintactico conf tokens)

rAnalisisSemantico :: ArbolSintactico -> Reader Config Programa
rAnalisisSintactico aSintactico = Reader (\conf -> analisisSemantico conf aSintactico)
```

Y finalmente nuestra función `compilar` se puede escribir cómo:

```
compilar :: Config -> String -> Programa
compilar config s =
  runReader (rAnalisisLexico s >>= rAnalisisSintactico >>= rAnalisisSemantico) config
```

donde `runReader` está definida en la biblioteca estándar y está definida como:

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

En este caso hemos visto como la instancia de `Monad` de `Reader a` se encarga transparentemente de pasar el parámetro de tipo `a` a las funciones que lo necesiten.

4.3.2.2. Interpretación en programación

Hemos visto que las expresiones monádicas (expresiones formadas por valores de un `Monad` combinadas con los operadores `return` y `>>=`) pueden volverse farragosas rápidamente. Podemos poner como ejemplo la implementación de `sum_three_heads` realizada anteriormente:

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

Es común que las expresiones monádicas en haskell puedan volverse complejas. Para ello se introduce la notación `do`. Esta notación permite transformar expresiones monádicas en bloques de código que resultarán familiares a los programadores que hayan utilizado lenguajes de programación imperativos.

Describimos informalmente la transformación sintáctica. Expresiones de la forma:

```
m1 >> m2 >> m3
```

las llevamos a:

```
do { m1;
     m2;
     m3;
}
```

Y expresiones de la forma

```
m >>= \x -> f x
```

las llevamos a

```
do {
     x <- m;
     f x
}
```

este código es similar al que podría encontrarse uno en un lenguaje de programación imperativo pero utiliza por debajo el operador `>>=` de la instancia de `Monad` que estemos utilizando. Las llaves y el `;` (punto y coma) se pueden omitir si se utiliza una identación adecuada.

Para justificar que esta notación se puede entender como una extensión de la programación imperativa introducimos la mónada identidad.

```
data Id a = Id a

instance Functor Id where
 fmap f (Id a) = Id (f a)

instance Monad Id where
  return a = Id a
  (Id a) (>>=) f = f a
```

Utilizando la notación `do` con esta mónada tenemos un código que funciona de forma muy parecida a cómo lo hace el código imperativo:

```
suma :: Int -> Int -> Id Int
suma x y = Id (x + y)

x :: Id Int
x = do
  y <- suma 3 4
  z <- suma y 7
  return z -- x = Id 14
```

La notación `do` desarrollaría esta expresión en la siguiente:

```
x :: Id Int
x = suma 3 4 >>= (\y ->
        suma y 7 >>= (\z ->
            return z
            )
        )
```

lo que nos permite establecer que el código se ejecuta de arriba abajo línea por línea (porque ese es el orden de la composición), y el operador `<-` se comporta como un operador de asignación.

Decimos que la notación `do` para mónadas *generaliza* ese aspecto de la programación imperativa porque la notación `do` puede usarse también con el resto de mónadas del lenguaje. Podemos escribir los ejemplos que hicimos anteriormente con la notación `do`.

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

compilar :: String -> Either ErrorCompilacion Programa
compilar s = do
   tokens <- analisisLexico s
   arbolSintactico <- analisisSintactico tokens
   programa <- analisisSemantico arbolSintactico
   return programa
```

Y por estar usándose por debajo las instancias de mónada de `Maybe` y de `Either a` respectivamente podemos estar seguro que este código se está encargando de la propagación de errores de forma correcta. Esta vez de forma totalmente transparente.

Las leyes de las mónadas se pueden expresar usando la notación `do`. La ley 1 dice que las siguientes dos expresiones son equivalentes:

```haskell
-- ley 1
-- la expresión
do
  x <- return v
  k x
-- es igual a
k v
```

Esto quiere decir que *inyectar* un valor (por la izquierda) en el contexto monádico no produce ningún efecto. La interpretación de esta afirmación varía en función de la instancia de `Monad` que estemos utilizando pero en la mónada `Either a` significaría que inyectar un valor usando `return` no va a cortocircuitar la computación.

```haskell
-- ley 2
-- la expresión
do
  x <- m
  return x
-- es igual a
m
```

Esta ley es igual a la anterior salvo que inyectamos el valor por la derecha. La tercera ley es más interesante desde el punto de vista de la notación `do`:

```haskell
-- ley 3
-- la expresión
do
  x <- m
  y <- k x
  h y
-- es igual a
do
  y <- (do
    x <- m
    k x)
  h y
```

Esta ley nos permite refactorizar código para organizarlo como creamos más conveniente sin afectar al funcionamiento de este. Por ejemplo, aplicado a nuestro ejemplo `sum_three_heads` la tercera ley nos permite implementar la función de la siguiente forma:

```haskell
sum_two_heads :: [Int] -> [Int] -> Maybe Int
sum_two_heads xs ys = do
    x <- head_safe xs
    y <- head_safe ys
```

```
    return (x + y)

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

Concluimos que las leyes de las mónadas son las que hacen que sea fácil razonar con código escrito con la notación `do`.

Ahora que disponemos de la notación `do` introducimos brevemente otras dos instancias de `Monad` que se encuentran en la biblioteca estándar de haskell y que serán útiles para entender ejemplos futuros.

L `IO` Haskell utiliza la mónada `IO` para modelar computaciones que requieren de entrada/salida (conectar con una base de datos remota, imprimir texto por pantalla). Un ejemplo de código que se ejecuta dentro de `IO` es el siguiente:

```
main :: IO ()
main = do
    putStrLn "Cual es tu nombre"
    nombre <- readLn
    putStrLn ("Hola " ++ nombre)
```

De hecho, la única forma de ejecutar operaciones de entrada salida es dentro de una función que devuelva una acción de tipo `IO A` que acabe evaluándose durante la ejecución de la función `main` del programa. Esto significa que una función con tipo:

```
sum :: Int -> Int -> Int
```

nunca podrá ejecutar operaciones de entrada salida. Esto contrasta con otros lenguajes de programación en los que independientemente del tipo que tenga una función se pueden realizar operaciones de entrada/salida arbitrarias.

En haskell las funciones son puras en el sentido de que se siempre que se llamen con los mismos parámetros devolverán los mismos resultados. Siguiendo con el ejemplo de la función `sum`, podemos asegurar que sea cual sea el momento de ejecución de nuestro programa en el que la llamemos, si la llamamos con los mismos parámetros obtendremos el mismo resultado. Esto no sería cierto si las funciones en haskell pudieran efectuar acciones de entrada/salida. Si pudieran, una función podría, por ejemplo, leer un archivo y devolver el contenido de este. Sin embargo si llamáramos a esta función, modificáramos el contenido del archivo y la volviéramos a llamar obtendríamos un resultado diferente. La mónada de `IO` permite que nuestro programa se comunique con *el mundo exterior*. Al final del día, nuestro programa tiene que hacer algo útil.

`State a` Una consecuencia de que haskell sea un lenguaje puro es que las funciones que no se ejecuten dentro de `IO` no disponen de un estado que puedan mutar. La mónada `State a` permite modelar funciones que trabajan con estado mutable de forma totalmente pura.

Capítulo 5

Patrones de diseño

En esta sección estudiaremos tres ejemplos de patrones de diseño y conjuntos de buenas prácticas que se encuentran habitualmente en código escrito en haskell y que son inicialmente motivados por la teoría de categorías.

5.1. El patrón categórico

La filosofía Unix, originada principios de los setenta durante el desarrollo del sistema operativo homónimo, aboga por el diseño de componentes de software con una superficie muy reducida y altamente combinable. Peter Salus resumió la filosofía Unix en tres puntos:

- Escribe programas que hagan una sola cosa y que la hagan bien.
- Escribe programas para que puedan funciona conjuntamente.
- Escribe programas que trabajen con texto, porque el texto es un formato que entienden todos los programas.

La vigencia de la filosofía Unix aproximadamente 50 años más tarde de su concepción es muestra de su potencia. La modularidad es también una cualidad que se desea del software y para la que se desarrollan patrones de diseño.

La programación funcional es otro ejemplo de ambiente en el que se sigue esta filosofía, ya que se busca crear software a base de componer de forma arbitrariamente elaborada funciones sencillas.

El patrón categórico es un refinamiento de estas ideas, buscando que las primitivas básicas que comprende tu software sean las flechas de una categoría. Esto tiene las siguientes ventajas:

- Exista una forma de componer las primitivas. Tal como en la filosofía unix.
- La composición es asociativa. Es mucho más sencillo razonar sobre una combinación de elementos asociativa que sobre una que no lo es. La asociatividad hace más sencillo comprender qué hace el código.

- La existencia de identidades supone la existencia de primitivas que son *triviales* respecto a la composición. Este es otro factor que favorece la comprensión de la operación de composición.

Estudiaremos un ejemplo de librería en haskell que fue concebida con el patrón categórico en mente.

5.1.1. Pipes

`pipes` es una librería de software libre hecha en haskell para la gestión de `streams`. Esta librería provee abstracciones para el tratamiento de secuencias de datos y transformaciones sobre estas. Un ejemplo de secuencias que podrían gestionarse con la librería `pipes` sería una sucesión de mensajes que llegan de un servidor remoto o las líneas de un fichero en disco.

La librería ofrece tres abstracciones principales para el manejo de `streams`:

- `Producers`: a los que en el presente texto nos referiremos como productores. Modelan fuentes de datos. Un `Producer`, por ejemplo, podría ser el que emite las líneas de un fichero de datos para su consumo por otra parte del software.

- `Consumers`: modelan destinos de datos. *Piden* datos para realizar tareas con ellos. Un ejemplo de `Consumer` recibiría mensajes de tipo `String` y los almacenaría en una base de datos.

- `Pipes`: modelan transformaciones de los datos. Pueden recibir datos de un tipo y emitir datos de otro tipo. Un ejemplo sería un `pipe` que recibe enteros y emite esos mismos enteros multiplicados por dos.

En este texto nos centraremos en los productores y pondremos de manifiesto la naturaleza categórica en la que se centra su diseño. Explicaciones análogas se pueden ofrecer para los consumidores y para los `pipes`.

5.1.1.1. Productores

El tipo que describe a los productores es:

```
Producer tipoEmitido monadaBase tipoResultado
```

En lo que concierne a los ejemplos que mostraremos a lo largo de esta sección consideraremos que la mónada base es `IO`. Esto permite al productor realizar acciones de entrada/salida como escribir en ficheros, mandar información por la red o mostrar información por pantalla. El `tipoEmitido` es el tipo de los mensajes que emite el productor. Cuando el productor acaba de emitir los mensajes que haya considerado necesarios puede optar por devolver un valor. El tipo de ese valor es `tipoResultado`. El siguiente es un ejemplo de productor con tipos concretos.

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

La función `lift` nos permite ejecutar acciones de la mónada base en el código del productor. En este caso esa acción es mostrar un mensaje por pantalla. Este productor emitirá tres valores enteros y entre el segundo y el tercer valor ejecutará una acción de entrada/salida.

Existe un tipo especial de productor que no emite ningún valor, y se llama `Effect`. En este texto nos referiremos a los valores de tipo `Effect` como efectos. `pipes` no nos permite ejecutar un productor si no lo *conectamos* a algún sitio, es decir, si no hay alguna parte de nuestro código que utilice los valores emitidos. No existe este problema con los efectos, ya que no emiten ningún valor. Para ejecutar un efecto podemos usar la función:

```
runEffect :: Effect IO () -> IO ()
```

Consideremos el efecto:

```
print :: Int -> Effect IO ()
print x = do
  lift (putStrLn $ "el numero es " ++ show x)
```

Esta función recibe como parámetro un entero y devuelve un efecto que consiste en imprimir por pantalla tal entero. Podemos *conectar* esta función que recibe un entero con nuestro productor anterior, que emitía enteros. La forma de combinarlos es usar la función `for` de `pipes` que podemos considerar que tiene la siguiente cabecera:

```
for :: Monad m => Producer a m r -> (a -> Effect m ()) -> Effect m r
```

Entonces podemos construir la expresión:

```
(for threeNumbers print) :: Effect IO ()
```

La forma en la que `for` combinará el productor con la función `print` es sustituyendo cada llamada a `yield` en el productor por una llamada a la función `print`. Es decir, la expresión anterior generaría un efecto equivalente a:

```
do
  lift (putStrLn $ "el numero es " ++ show 3)
  lift (putStrLn $ "el numero es " ++ show 4)
  lift $ putStrLn "Accion de IO"
  lift (putStrLn $ "el numeroes " ++ show 5)
  return ()
```

En general, `for` no solo permite combinar productores con efectos sino que también se pueden combinar productores con otros productores. Definimos la siguiente función:

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

es decir, recibe un parámetro y devuelve un productor que emite ese valor dos veces. Este productor también se puede combinar con `for` siguiendo la misma lógica que se seguía anteriormente: cada llamada a `yield` se sustituye por una llamada a la función `dup`. Entonces el productor

```
for threeNumbers dup
```

es equivalente al productor

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

¿Podemos combinar también `dup` y `print`? La respuesta es afirmativa. En `pipes` se define un combinador con el siguiente tipo:

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

Es decir si tenemos una función que recibe un parámetro de tipo `a` y devuelve un productor que emite valores de tipo `b` y tenemos otra función que acepta un parámetro de tipo `b` y emite valores de tipo `c` podemos combinarlos para obtener una función que recibe un parámetro de tipo `a` y devuelve un productor que emite valores de tipo `c`. Esta cabecera puede resultar confusa para alguien que no haya programado nunca en haskell por lo que proponemos el siguiente ejemplo:

```
dup2 :: String -> Producer String IO ()
dup2 s = do
```

```
  yield s
  yield (s ++ s)

leng :: String -> Producer Int IO ()
leng s = yield (length s)

numberAndSquare :: Int -> Producer Int IO ()
numberAndSquare x = do
  yield x
  lift $ putStrLn "Ahora emitimos el cuadrado"
  yield (x * x)
```

El operador ~> nos permite combinar tanto `dup2` y `leng` como `leng` y `numberAndSquare`. Detallamos qué significa cada una de las composiciones:

```
dup2 ~> leng :: String -> Producer Int IO ()
```

es una función equivalente a:

```
dup2Leng :: String -> Producer Int IO ()
dup2Leng  = do
  yield (length s)
  yield (length (s ++ s))
```

Por otro lado:

```
leng ~> numberAndSquare :: String -> Producer Int IO ()
```

es equivalente a:

```
lengNumberAndSquare :: String -> Producer Int IO ()
lengNumberAndSquare s = do
  yield (length s)
  lift $ putStrLn "Ahora emitimos el cuadrado"
  yield ((length s) * (length s))
```

Pero aún hay más, también son posibles las siguientes combinaciones:

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

y no solo eso sino que además a partir de la implementación de `pipes` se puede demostrar que las dos expresiones anteriores producen productores equivalentes. En ese sentido podemos decir que el operador ~> es asociativo.

De hecho podemos decir que los productores forman una categoría. En esta categoría los objetos son los tipos de haskell y las flechas $a \longrightarrow b$ son las funciones de haskell de tipo `a`

`->Producer b IO ()`. La composición es el operador `~>` que hemos descrito anteriormente. Ya hemos visto que el operador es asociativo, quedaría ver que existen las identidades para cada tipo y existen. La identidad es precisamente

```
yield :: a -> Producer a IO ()
```

es decir, `yield ~>f = f` y `g ~>yield = g`.

5.2. Monad Transformers

En el capítulo anterior comentamos el uso de mónadas para realizar labores como:

- Ejecutar operaciones de entrada/salida
- Gestionar computaciones puras con estado
- Gestión de fallos de forma transparente
- Acceso global a un repositorio de información inmutable

Una aplicación normalmente necesita varias de estas funcionalidades al mismo tiempo. Los *monad transformers* permiten combinar las funcionalidades de varias mónadas. Un ejemplo de clase que regula el funcionamiento de esta abstracción es:

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a
```

Es decir, un `t` t

5.2.1. Ejemplo de aplicación

Podemos suponer que estamos escribiendo una aplicación que necesita acceso global a unos datos de configuración, tiene que realizar transformaciones que modifiquen estado y además tiene que mostrar información por pantalla. Si tuviéramos que realizar esas tareas por separado usaríamos respectivamente el `Reader` monad, el `State` monad y el `IO` monad. Para hacer todo a la vez tendremos que usar monad transformers.

Nuestra aplicación hará lo siguiente: recibirá por entrada estándar enteros que escriba el usuario y los irá almacenando en una lista. Cuando se acabe la entrada mostrará la lista completa de los enteros que el usuario ha introducido. Es una aplicación sencilla pero que mostrará la utilidad de usar monad transformers.

En primer lugar definiremos el tipo `Config` de la aplicación que representa la configuración inicial de la aplicación. En este caso el tipo `Config` solo almacenará el título de la aplicación y el carácter prompt que usará a la hora de mostrar salida. El tipo es el siguiente:

```
data Config = Config {
  title :: String,
  prompt :: String
  }
```

El estado a modificar de nuestra aplicación será precisamente la lista de enteros. Usaremos un alias de tipo para representar este tipo.

```
type Stack = [Int]
```

Definimos la pila de mónadas sobre la que diseñaremos nuestra aplicación. Usaremos una mónada `Reader Config` para que las acciones que se ejecuten en el dominio de nuestra aplicación tengan acceso a la configuración global. Utilizaremos también una mónada `State Stack` para que las acciones puedan modificar el estado de la lista de enteros. Finalmente, utilizaremos la mónada `IO` para poder efectuar acciones de entrada/salida. El tipo que utilizaremos para definir las acciones de nuestra aplicación será entonces:

```
type AppT = ReaderT Config (StateT Stack IO)

newtype AppM a = AppM { unApp :: AppT a }
  deriving (Functor, Applicative, Monad,
            MonadReader Config, MonadState Stack, MonadIO)
```

El tipo `AppT` es un alias para la pila de monad transformers que usaremos. Encapsulamos ese tipo en otro llamado `AppM` (esto es algo habitual en haskell y representa ventajas a la hora de aumentar la seguridad en cuanto a la comprobación de tipos que el compilador puede hacer). La librería `mtl` nos proporciona typeclasses como `MonadReader Config` que nos permiten utilizar la funcionalidad del monad `Reader` dentro de una pila de monad transformers. Análogamente sucede con `MonadState Stack`.

Si tenemos un valor de `AppM a` necesitamos una forma de transformarlo en un valor de `IO a` para poder ejecutarlo. Para ello usamos la función:

```
runApp :: Config -> AppM a -> IO a
runApp config (AppM m) = evalStateT (runReaderT m config) []
```

Esta aplicación recibe una configuración y un valor de `AppM a` y devuelve la acción de `IO a` asociada. Ejecuta la acción inicializando el valor del `Stack` a la lista vacía.

Mostramos algunas de las acciones que se ejecutarán en el contexto de nuestra aplicación:

```
addNumber ::
  (MonadState Stack m) => Int -> m ()
addNumber x = modify $ \state -> (x:state)
```

Esta acción añade al estado de la aplicación un entero `x`. Nótese que esta acción se podría implementar con este tipo y sería equivalente:

```
addNumber ::
  Int -> AppM ()
addNumber x = modify $ \state -> (x:state)
```

La ventaja de implementarla con un tipo polimórfico que sea `MonadState Stack m` es que el compilador entiende que esa función puede ejecutarse sobre cualquier pila de mónadas que gestione un estado de tipo `Stack`. Esto no es útil porque existan muchas pilas de mónadas sobre la que sea útil ejecutar esa acción sino porque con ese tipo se hace imposible ejecutar ninguna acción `IO ()` en la función. Para que se pueda ejecutar una acción de entrada/salida se debe estar dentro de un `MonadIO` pero la *constraint* `MonadState Stack m` no implica que se pueda ejecutar ninguna operación `IO`. La ventaja de este hecho es que viendo simplemente el tipo de la función podemos saber que la única funcionalidad que puede ejecutar esta función es una transformación sobre el estado `Stack`. En este caso es fácil leer directamente qué hace la función y saber que solo modifica el estado, pero delimitar estrictamente sobre qué partes de un sistema actúa una función es útil a la hora de trabajar sobre sistemas más complejos.

Otro ejemplo de esto lo encontramos en la acción:

```
appPrint ::
  (MonadReader Config m, MonadIO m) => String -> m ()
appPrint msg = do
  promptText <- fmap prompt ask
  liftIO $ putStrLn (promptText ++ " " ++ msg)
```

Viendo este código sabemos que la función `appPrint` no puede modificar el estado. En las constraints asumimos `MonadIO` porque necesitamos imprimir por la salida estándar y asumimos `MonadReader` porque necesitamos leer la configuración para ver el carácter de `prompt`.

```
showTitle ::
  (MonadReader Config m, MonadIO m) => m ()
showTitle = do
  titleStr <- fmap title ask
  appPrint titleStr
```

Es análoga a `appPrint`. Implementamos también una función que muestra por pantalla el estado de la aplicación.

```
showStack :: AppM ()
showStack = do
  stack <- get
  appPrint (show stack)
```

En esta función no usamos un tipo polimórfico porque necesitamos utilizar las tres mónadas para cumplir esta funcionalidad. Necesitamos el `State` monad para leer el estado de la

aplicación, necesitamos `Reader` para poder usar `appPrint`, que necesita conocer el `prompt` y por supuesto necesitamos `MonadIO` para imprimir mensajes por pantalla.

Finalmente mostramos el código que falta para que la aplicación funcione:

```
appLoop :: AppM ()
appLoop = do
  eof <- liftIO isEOF
  if eof
    then showStack
    else (do n <- liftIO readLn
             addNumber n
             appLoop
         )

main :: IO ()
main = runApp (Config "AppTitle" ">") $ do
  showTitle
  appLoop
```

La función `main` inicializa la aplicación con un valor de configuración, imrpime el título por pantalla y ejecuta el bucle central de la aplicación. Este bucle comprueba si se ha llegado al final de la entrada estándar. Si no, se lee un entero por la entrada, se añade a la pila y se vuelve a ejecutar el bucle central. Si se ha llegado al final de la entrada estándar se muestra por pantalla el contenido de la pila y se sale.

5.3. Otros patrones de diseño

Finalizaremos el trabajo con la exposición breve de un par de patrones de diseño conocidos y bien documentados en la comunidad de haskell que muestran que la influencia de la teoría de categorías no se limita al patrón categórico y a los monads transformers. Analizar en más profundidad las siguientes construcciones podría ser una buena forma de continuar con la línea de este trabajo.

5.3.1. `Applicative`

`Applicative` es otra typeclass de la biblioteca estándar de haskell. Es una superclase de `Monad`, en el sentido de que toda instancia de `Monad` es una instancia de `Applicative`. La interfaz de `Applicative` es la siguiente:

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

La función `pure` nos recuerda a la función `return` de la instancia de `Monad`, y de hecho si un applicative dado es un `Monad` las funciones `pure` y `return` deben coincidir. La función `(<*>)` se parece tanto a `fmap` de la clase `Functor` como a `(>>=)` de la clase `Monad`, pero no es ninguna de las dos. Veamos las leyes que una instancia de `Applicative` debe cumplir según la documentación de haskell:

```
pure id <*> v = v -- identidad
pure (.) <*> u <*> v <*> w = v <*> (v <*> w) -- composición
pure f <*> pure x = pure (f x) -- homomorfismo
u <*> pure y = pure (\f -> f y) <*> u -- intercambio
```

Se puede probar que si tenemos una instancia de mónada y definimos:

```
pure = return

-- (<*>) :: f (a -> b) -> f a -> f b
ff <*> fa = do
  f <- ff
  a <- fa
  return (f a)
```

(hemos podido usar notación `do` porque estamos asumiendo que `f` es un `Monad`) Entonces se cumplen las leyes de applicative. A partir de una instancia de applicative también podemos definir una instancia de funtor que cumple las leyes de la siguiente forma:

```
fmap :: (a -> b) -> f a -> f b
fmap f fa = (pure f) <*> fa
```

Para entender la diferencia cualitativa entre `Applicative` y `Monad` enunciaremos el siguiente resultado: toda expresión creada usando los combinadores aplicativos (`pure` y `(<*>)`) se puede transformar en una expresión de la siguiente forma:

```
pure f <*> u1 <*> u2 <*> ... <*> un
```

donde `f :: u1 ->u2 ->... ->z` y `ui :: f ui` son valores aplicativos. Esto no es posible en general con expresiones monádicas, donde la *estructura* de la computación puede depender de resultados intermedios.

Esta diferencia ha permitido que `Applicative` se identifique como la abstracción adecuada para la resolución de múltiples problemas como el parseo de gramáticas libres de contexto o la validación de formularios. Facebook liberó recientemente una librería de haskell llamada Haxl para la gestión de la concurrencia de forma totalmente implícita para el programador, permitiendo que funciones como:

```
numCommonFriends :: User -> User -> Haxl Int
numCommonFriends x y =
    length <$> (intersect <$> friendsOf x <*> friendsOf y)
```

ejecuten las llamadas `friendsOf` a la base de datos de forma concurrente (e incluso integrando soluciones de cacheo de forma transparente para el programador). Usando la instancia de applicative de Haxl el programador permite que la librería extraiga el paralelismo implícito de su código sin tener que programar de forma distinta a como lo haría con código secuencial.

Categóricamente, `applicative` ha sido identificado como una clase especial de funtores de categorías monoidales.

5.3.1.1. Free Monads

Los free monads son una abstracción que proviene también de la teoría de categorías y que ha sido utilizada exitosamente para el diseño de software. La aplicación de las free monads a haskell permite el encaje natural de lenguajes de dominio específico así como la creación de intérpretes muy generales para estos lenguajes.

Una de las librerías más utilizada que implementan la construcción es `free`, y es utilizada por numerosas aplicaciones en haskell entre las que podemos destacar:

- `engine-io`: una implementación de un protocolo de websockets.
- `reddit`: una librería para interactuar con el popular servicio homónimo.

La idea que sugiere la utilización de las free monads es la creación de una DSL que permita expresar toda la lógica de negocio de tu aplicación de forma agnóstica al resto de tecnologías utilizadas (es decir, no mezclar código del framework que estés utilizando con tu lógica de negocio) y que luego sean los intérpretes de esa DSL los que tengan que interactuar con el resto de componentes del sistema.

Conclusiones y vías futuras

En la asignatura de Geometría I, durante mi primer año de carrera pude leer una afirmación que despertó mi curiosidad: " *un espacio vectorial es naturalmente isomorfo a su doble dual* ". Lo que entendí en su momento es que entre esos dos espacios existía un isomorfismo que era más evidente que otros, aunque no creí que esa noción se pudiera formalizar.

Este trabajo, cinco años más tarde, me permite por fin darle un sentido preciso a esa afirmación. Digo más, la maquinaria que me permite entender qué significa que un espacio vectorial sea naturalmente isomorfo a su doble dual me sugiere una forma nueva de entender gran parte de los contenidos de la carrera, especialmente (pero no exclusivamente) las asignaturas de álgebra.

La complejidad del software construido a lo larga de la escasa historia de la disciplina no ha hecho más que crecer, y no hay motivos para pensar que esta tendencia parará. El desarrollo de nuevos paradigmas como la programación orientada a objetos permitió a los ingenieros domar la complejidad y llevar a cabo proyectos que hubieran sido imposibles con técnicas más rudimentarias. La utilización de la teoría de categorías para formalizar abstracciones en lenguajes de programación funcional podría ser un paso adelante. Muchas empresas se benefician ya del uso de haskell en sus sistemas y de las numerosas innovaciones que este lenguaje de programación introdujo.

Desde mi punto de vista he hecho una buena exposición de los contenidos de teoría de categorías que permiten entender los patrones de diseño más usados en haskell. No estoy convencido de que este texto, por sí mismo, pueda convencer a nadie de que estas técnicas pueden ser beneficiosas para el desarrollo de software. Haskell es un lenguaje de programación muy diferente, y cómo tal, aceptar sus propuestas puede requerir un trabajo continuado. En este sentido podemos afirmar que los objetivos han sido alcanzados.

La línea exacta de este trabajo puede continuarse con el estudio de mónadas libres y categorías de álgebras sobre mónadas. Estos conceptos, que se estudian en los textos de categorías por su relevancia en las matemáticas, tienen aplicaciones a la programación. Otra línea de trabajo sería la de la teoría de categorías para el estudio de sistemas de tipos.

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

Haskell básico para entender los ejemplos

Comentamos brevemente algunas funcionalidades de haskell que facilitarán la comprensión de los ejemplos utilizados.

A.1. Expresiones y tipos

Toda expresión en Haskell tiene un tipo. Los siguientes son ejemplos de expresiones junto con sus tipos:

```
True :: Bool
False :: Bool
'c' :: Char
"hola" :: String
3 :: Int
```

En general la sintaxis `a :: A` se utiliza en haskell para afirmar que la expresión `a` tiene tipo `A`. Que toda expresión en haskell esté tipada no significa que tengamos que ser explícito acerca de estos tipos, el compilador es capaz en la mayoría de las ocasiones de inferir por nosotros cual es el tipo de todas las expresiones de un programa. Será habitual, sin embargo, que especifiquemos un tipo aunque no sea necesario, para ayudar al lector.

A.2. Funciones

En haskell las funciones son ciudadanos de primera clase (*first class citizens*) en el sentido de que son valores como cualquier otro. Esto significa, entre otras cosas, que las funciones tienen un tipo.

Proponemos ejemplos de funciones a continuación:

```
suma1 :: Int -> Int
suma1 x = x + 1

tres :: Int
tres = suma 2

esA :: Char -> Bool
esA c = if c == 'A'
        then True
        else False
```

El tipo de `suma1` indica que la función recibe un parámetro de tipo entero y devuelve un valor de tipo entero. Se pueden definir también funciones de múltiples parámetros:

```
suma :: Int -> Int -> Int
suma x y = x + y

cinco :: Int
cinco = suma 3 2
```

El tipo de `suma` puede resulta sorprendente a quien esté acostumbrado a trabajar con lenguajes imperativos. Haskell solo considera funciones de un parámetro. La trampa es que `suma` es una función que recibe un parámetro de tipo `Int` y devuelve *otra función* que recibe un parámetro `Int` y devuelve un valor de tipo `Int`. El siguiente ejemplo es equivalente al anterior:

```
suma :: Int -> Int -> Int
suma x y = suma x + y

sumaTres :: Int -> Int
sumaTres = suma 3

cinco :: Int
cinco = sumaTres 2
```

Entonces cuando haskell evalúa la expresión

```
suma 3 2
```

lo está haciendo asociado a la izquierda:

```
(suma 3) 2
```

La técnica de transformar una función que recibe varios parámetros en una función que recibe un parámetro y devuelve otra función se llama currificación. Haskell hace currificación automática de todas las funciones.

De hecho no existe ninguna diferencia entre el tipo de la función `suma` tal como estaba definido anteriormente y el tipo:

```
suma :: Int -> (Int -> Int)
suma x y = x + y
```

A.3. Expresiones lambda

Existe un tipo especial de expresión en haskell llamado expresión lambda y que sirve para definir funciones. Ponemos un ejemplo a continuación:

```
suma1 :: Int -> Int
suma1 = (\x -> x + 1)
```

La expresión construye la función que recibe un valor `x` y devuelve el valor `x+1`. Las expresiones lambda sirven también para definir funciones que reciben múltiples parámetros:

```
multiplica :: Int -> Int -> Int
multiplica = (\x y -> x * y)
```

Es importante notar que la expresión
`x ->f x` es equivalente a la expresión `f`.

A.4. Definir tipos

Haskell permite al usuario definir sus propios tipos. Veamos un ejemplo sencillo:

```
data Dni = Dni Int Char
```

Esta línea define el tipo `Dni` (izquierda) y el constructor homónimo (derecha) que acepta como parámetros un entero y un carácter. Mostramos a continuación ejemplos de valores del tipo `Dni`.

```
(Dni 44669279 'J') :: Dni
(Dni 12345678 'C') :: Dni
```

Todos los valores del tipo `Dni` son de la forma `(Dni i l)` donde `i :: Int` y `l :: Char`. Además, podemos considerar al constructor `Dni` una función con tipo `Dni :: Int ->Char ->Dni`. Nótese que `Dni` antes de los dos puntos representa al constructor pero que el `Dni` de la derecha representa al tipo.

Como todos los valores del tipo `Dni` son de la forma especificada arriba, haskell nos permite definir funciones sobre ese tipo de la siguiente manera:

```
suma3AlNumero :: Dni -> Int
suma3AlNumero (Dni numero caracter) = numero + 3

-- devuelve el dni completo como una cadena
devuelveCadena :: Dni -> String
devuelveCadena (Dni n c) = (show n) ++ (show c)
```

Podemos definir tipos con más de un constructor. Por ejemplo:

```
data PersonaConAltura = Hombre Int | Mujer Int | Otro String Int
```

donde definimos el tipo `PersonaConAltura` y los constructores `Hombre :: Int ->PersonaConAltura`, `Mujer :: Int ->PersonaConAltura`, `Otro :: String ->Int ->PersonaConAltura`. Todo valor de `PersonaConAltura` será o bien de la forma `Hombre n`, o bien `Mujer n`, o bien `Otro o n`, donde `n :: Int` y `o :: String`. Esto hace que haskell nos permita definir funciones sobre este tipo como:

```
comentarioSobreAltura :: PersonaConAltura -> String
comentarioSobreAltura (Hombre altura) = if altura < 174
                                        then "Tienes altura inferior a la media"
                                        else "Tienes altura superior a la media"
comentarioSobreAltura (Mujer altura) = if altura < 163
                                       then "Tienes altura inferior a la media"
                                       else "Tienes altura superior a la media"
comentarioSobreAltura (Otro genero x) =
            "No tengo informacion sobre la altura media en el género " ++ genero
```

Esta funcionalidad de hacer distinción de casos sobre los constructores de un tipo es conocida como *pattern matching*. También se puede hacer *pattern matching* dentro de una expresión con la siguiente sintaxis:

```
genero :: PersonaConAltura -> String
genero persona = case persona of
    Hombre _ -> "Hombre"
    Mujer _ -> "Mujer"
    Otro g _ -> g

-- equivalentemente podríamos haber implementado la función como
genero (Hombre _) = "Hombre"
genero (Mujer _) = "Mujer"
genero (Otro g _) = g
```

donde usamos el carácter `_` para indicar (tanto a lectores como al compilador) que no usaremos esa variable.

Nótese que podemos definir tipos *recursivos* en el siguiente sentido:

```
data Peano = Cero | Succ Peano
```

Aquí estamos definiendo el tipo `Peano` que tiene dos constructores: `Cero :: Peano` (no admite parámetros) y `Succ :: Peano ->Peano`. Podemos hacer *pattern matching* sobre este tipo:

```
aNumero :: Peano -> Int
aNumero Cero = 0
aNumero (Succ peano) = 1 + (aNumero peano)

esCero :: Peano -> Bool
esCero Cero = True
esCero (Succ _) = False
```

Existe otra sintaxis para definir tipos con un solo constructor y que reciben un solo parámetro:

```
newtype EnvuelveEntero = EnvuelveEntero Int
```

que para los propósitos de este trabajo consideraremos igual a la sintaxis que usa `data` en lugar de `newtype`.

Cuando usamos la sintaxis:

```
data Dni = Dni {
  numero :: Int
  letra :: Char
  }
```

No solo estamos definiendo el tipo `Dni` junto con el constructor de tipo `Dni :: Int ->Char ->Dni` sino que además estamos definiendo las proyecciones evidente `numero :: Dni ->Int` y `letra :: Dni ->Char`.

A.5. Funciones con tipos variables

Podemos definir funciones polimórficas como la siguiente:

```
aplicar :: (a -> b) -> a -> b
aplicar f x = f x

seis :: Int
seis = aplicar suma1 5

hombre :: String
hombre = aplicar genero (Hombre 180)
```

La función `aplicar` se puede especializar a tipos concretos como `a = Int, b = Int` en el caso del valor `seis` o a los tipos `a = PersonaConAltura, b = String` en el caso del valor `hombre`.

A.6. Tipos Parametrizados

Podemos definir tipos parametrizados por otros tipos. Por ejemplo:

```
data ParConNombre a = ParConNombre String a a
```

esto define todos los tipos `ParConNombre A` donde `A` es un tipo fijo pero arbitrario. Nótese que `ParConNombre` no es un tipo, pero `ParConNombre Int` sí que lo es. Un ejemplo de valor de este tipo sería `ParConNombre "tresdos"3 2 :: ParConNombre Int`.

El constructor `ParConNombre :: String ->a ->a ->ParConNombre a` es ahora una función con tipos variables. Se puede estar parametrizado por más de un tipo:

```
data Par a b = Par a b

-- ejemplos de valores
Par 3 'c' :: Par Int Char
Par 'c' 3 :: Par Char Int
Par 4 5 :: Par Int Int
```

A.7. Listas

Podemos definir listas en haskell de forma similar a como definimos el tipo `Peano`:

```
data Lista a = ListaVacia | NoVacia a Lista

-- ejemplos de valores
ListaVacia :: Lista Int
NoVacia 3 ListaVacia :: Lista Int
NoVacia 3 (NoVacia 4 ListaVacia) :: Lista Int
```

y podemos definir funciones sobre este tipo:

```
sumarLista :: Lista Int -> Int
sumarLista ListaVacia = 0
sumarLista (NoVacia x otraLista) = x + (sumarLista otraLista)
```

No necesitamos definir este tipo por nuestra cuenta porque ya está definido en la biblioteca estándar de haskell y cuenta con una sintaxis más intuitiva:

```
sumarLista :: [Int] -> Int
sumarLista [] = 0
sumarLista (x:xs) = x + (sumarLista xs)
```

Cuando queramos hacer referencia a que `[]` es un tipo parametrizado a veces escribiremos `[] Int` en lugar de `[Int]`.

A.8. Typeclasses

Haskell soporta una funcionalidad similar a las interfaces de java conocida como typeclasses. Una typeclass define una interfaz que cualquier tipo que quiera ser instancia de esa clase debe implementar. Por ejemplo:

```
data Tamaño = Grande | Pequeño

class TieneTamaño t where
  tamaño :: t -> Tamaño

instance TieneTamaño Int where
  tamaño x = if x > 179 then Grande else Pequeño

instance TieneTamaño String where
  tamaño s = if (length s) > 10 then Grande else Pequeño

instance TieneTamaño PersonaConAltura
  tamaño (Hombre altura) = tamaño altura
  tamaño (Mujer altura) = tamaño altura
  tamaño (Otro _ altura) = tamaño altura
```

Nótese como hemos implementado la instancia de `Tamaño` de `PersonaConAltura` aprovechando la instancia de `Int`. Podemos escribir funciones que solo acepten instancias de esta typeclass:

```
nombreTamaño :: Tamaño -> String
nombreTamaño Grande = "grande"
nombreTamaño Pequeño = "pequeño"

decirTamaño :: TieneTamaño a => String -> a -> String
decirTamaño nombre a =
  nombre ++ " tiene tamaño " ++ nombreTamaño (tamaño a)

-- ejemplos
decirTamaño "tres" 3 -- tres tiene tamaño pequeño
decirTamaño "Braulio" (Hombre 180) -- Braulio tiene tamaño grande
decirTamaño "cadena de texto" "eeeeeeeeeeee" -- ...
```

De hecho, también se pueden definir clases que solo puedan ser instanciadas por instancias de otras clases:

```
class TieneTamaño t => TieneTamañoNumerico t where
  tamañoNumerico :: t -> Int
```

```haskell
instance TieneTamañoNumerico Int where
  tamañoNumerico x = x

instance TieneTamañoNumerico Char where
  tamañoNumerico _ = 100 -- Error: Char no es instancia de TieneTamaño
```