Grado en Matemática Computacional

Trabajo Final de Grado

Fundamentos de la Teoría de Categorías y sus aplicaciones en la programación funcional con Haskell

Autor:
Carme-Maria Arrufat Andreu

Tutor académico:
Antonio Beltrán Felip

Fecha de lectura: julio de 2022
Curso académico 2021/2022

Resumen

La programación funcional se basa en un hecho fundamental: la composición de funciones como forma de construir un programa. La Teoría de Categorías, que es en esencia el estudio de la composición de morfismos, nos sirve pues como base teórica para modelizar y describir patrones de los lenguajes de programación funcionales, entre ellos, la programación de efectos secundarios, entrada/salida de un programa, manejo de excepciones, etc., mediante mónadas, un concepto nacido puramente de la Teoría de Categorías.

En este trabajo de fin de grado damos una exposición general de aquellos fundamentos de la Teoría de Categorías necesarios para llegar al concepto de mónada (un monoide en la categoría de los endofunctores). Se introducirán para ello los conceptos de functor, transformación natural, adjunciones y, por supuesto, de mónada, no sin antes pasar por uno de los resultados más importantes de la Teoría de Categorías, el Lema de Yoneda y sus consecuencias. Desarrollaremos también estos conceptos no solo desde el punto de vista matemático, sino también desde la perspectiva del paradigma funcional mediante el lenguaje de programación Haskell.

Palabras clave

Haskell, Programación funcional, Teoría de Categorías.

Key words

Category Theory, functional programming, Haskell.

Índice

1 INTRODUCCIÓN

1.1. Motivación y objetivos

El paradigma de la programación funcional está cada día más presente en el diseño y desarrollo de los lenguajes de programación: desde la proliferación de lenguajes de programación funcionales como Haskell, Clojure, Erlang, etc., hasta la implementación de características funcionales en lenguajes imperativos, como Rust o JavaScript.

Fue por este motivo que empecé a estudiar y aprender a programar en dicho paradigma con el lenguaje de programación Haskell. Sin embargo, había un concepto que no paraba de repetirse y cuya importancia siempre se remarcaba: el concepto de mónada. Una simple búsqueda en Google hizo aparecer la famosa frase:

> *A monad is just a monoid in the category of endofunctors, what's the problem? [13]*

que posteriormente me percaté que se suele mencionar de manera irónica cuando algún programador o programadora pregunta inocentemente "¿Qué es una mónada?" debido a que este concepto no es, en absoluto, simple de comprender, y menos para una persona que no cuente con conocimientos matemáticos previos. Mi curiosidad por entender esta frase al completo fue lo que me motivó a comprender las matemáticas detrás de dicha frase, la Teoría de Categorías, y cómo esta se aplica en la programación funcional.

La Teoría de Categorías nos ofrece las herramientas conceptuales para definir estructuras matemáticas abstractas a lo largo de todas las disciplinas matemáticas, desde el álgebra o la topología, hasta la semántica de los lenguajes de programación funcionales. Gracias a esta abstracción podemos estudiar fenómenos que ocurren en cierto campo y trasladar las ideas obtenidas a otro. Es por este motivo que la Teoría de Categorías requiere de una nueva perspectiva a la hora de definir y estudiar los objetos que alcanza. A lo largo de este trabajo estudiaremos un teorema central en este campo, el Lema de Yoneda, y desarrollaremos los conceptos necesarios para comprender la noción con la que culmina este trabajo: la mónada. También realizaremos construcciones universales que nos permitirán generalizar estructuras más concretas desde un punto de vista categórico. Esto es, en cierto modo, poder llegar a generar los "planos" que contienen su estructura independientemente de la disciplina matemática con la que estemos tratando [21].

La capacidad de la Teoría de Categorías de abstraer y generalizar llega tan lejos como para que sus nociones nos sirvan como base teórica para modelizar y describir

aspectos de los lenguajes de programación funcionales. Como veremos más adelante, la esencia de una categoría es la composición de morfismos (o, particularmente, funciones), y precisamente los lenguajes de programación funcionales tienen como operación fundamental la aplicación de las mismas [11], al contrario de lo que sucede en la programación imperativa en la que la ejecución de funciones se realiza de forma secuencial.

Así pues, el estudio de este campo, concretamente mediante el lenguaje de programación funcional Haskell, va a constituir la parte fundamental de este trabajo, con una gran variedad de ejemplos y explicaciones de conceptos categóricos en este contexto, y culminará con la construcción de la estructura categórica de mónada.

1.2. Estructura del trabajo

Este trabajo consiste en el desarrollo de los conceptos más fundamentales de la Teoría de Categorías que se construyen conceptualmente uno tras otro para llegar al concepto de mónada.

Así pues, en los Capítulos 2 (Categorías), 3 (Construcciones) y 5 (Objeto exponencial) veremos las definiciones más fundamentales así como las construcciones categóricas más básicas a partir de las cuales generalizaremos hacia futuros conceptos.

Previamente al Capítulo 5, en el Capítulo 4 (Functores) se verá el concepto de functor así como sus distintos tipos, pasando por los endofunctores, los hom-functores (esenciales para comprender el Lema de Yoneda) y, por último, los límites y qué tipos de functores los preservan.

En el Capítulo 6 (Naturalidad) se introducirá el concepto de transformación natural, central en el desarrollo de la Teoría de Categorías, así como el Lema de Yoneda, sus consecuencias y sus aplicaciones.

Para acabar, en los Capítulos 7 (Adjunciones) y 8 (Mónadas), se desarrollará el concepto de adjunción, que es una relación particular entre functores, y a partir del cual se construirá la noción de mónada. Asimismo, se dará cuenta de las Categorías de Kleisli, que son una forma equivalente de representar una mónada, y cuya aplicación en la construcción de las mónadas en Haskell es fundamental.

Además, en las últimas secciones de cada capítulo se verán las implementaciones de los conceptos mostrados y ejemplos de los mismos en Haskell.

Y, finalmente, en el Apéndice A se expondrán detalladamente la sintaxis y los conceptos básicos de la programación en Haskell: la declaración de variables y funciones, el funcionamiento de sus tipos, polimorfismo y clases, las funciones de orden superior y un ejemplo de una de las mónadas más importante en Haskell usada para la programación de acciones.

1.3. Prefacio histórico

Por una parte, el desarrollo de la Teoría de Categorías fue impulsado en primera instancia por Eilenberg y Mac Lane en 1945 con la publicación del artículo titulado "Teoría general de las equivalencias naturales", cuyo concepto fundamental no era el de la categoría como estructura matemática sino el de la transformación natural. Su objetivo era formalizar el concepto de naturalidad, frecuente en muchos de los campos de las matemáticas [6]. Aunque inicialmente se vieran aplicaciones del lenguaje de la Teoría de Categorías en el álgebra abstracta, pronto se descubrirían más aplicaciones en topología algebraica y, en particular, en teoría de la homología.

Posteriormente, Grothendieck publicaría en 1957 el artículo titulado *Some aspects of homological algebra* en el que mostraría mediante el uso de las categorías cómo ciertos métodos en el álgebra homológica podrían ser trasladados a otras disciplinas como la geometría algebraica.

Con el desarrollo de la disciplina, la Teoría de Categorías se convirtió en un campo independiente de estudio y, en la década de los 60, Lawvere crearía un nuevo marco teórico con el que aproximarse a la lógica y a los fundamentos de las matemáticas mediante la Teoría de Categorías, pero concretamente con el concepto de *categoría de categorías*.

Más tarde, en la década de los 70, el concepto del *topos*, creado por Grothendieck e impulsado por Lawvere y Tierney, ganaría tracción hasta convertirse en un elemento fundamental de la disciplina. En pocas palabras, según Jean-Pierre Marquis, un *topos* es una categoría que posee la estructura lógica suficiente como para desarrollar lo que podríamos denominar "las matemáticas ordinarias" [16], es decir, el contenido que se suele impartir en los grados de matemáticas.

A partir de entonces la Teoría de Categorías ha ido desarrollándose y ha encontrado aplicaciones en múltiples disciplinas teóricas fuera de las matemáticas, como en la informática o en la física.

Por otra parte, los lenguajes de programación funcionales aún debían desarrollarse como para poderse llegar a producir la conexión entre la semántica de dichos lenguajes y la Teoría de Categorías. Los fundamentos del paradigma de la programación funcional se remontan a 1936 con la creación del *cálculo lambda* por parte de Church, que se trata de un sistema formal para la resolución del problema de si cierta función, representable mediante el cálculo lambda, era "computable" (o también llamado "decidible"). Además, paralelamente, Turing creó una clase de máquinas (las *máquinas de Turing*) donde la noción de función computable era equivalente a la del cálculo lambda [3]. Es bien conocido que el modelo actual de computadores se basa en el funcionamiento de la máquina de Turing, es decir, en ejecuciones secuenciales de órdenes, y que los lenguajes de programación imperativos se basan también en este principio. Ejemplos de lenguajes de dicho paradigma son los distintos ensambladores, C, Java, Python, etc. Por otro lado, los lenguajes de programación funcionales como Haskell, Miranda, ML o Adga se basan en el cálculo lambda, es decir, en la aplicación de funciones.

El desarrollo de dicho paradigma empieza a finales de la década de los 50, cuando

John McCarthy desarrolla en el MIT el primer lenguaje de programación funcional: LISP. Sin embargo, este no sería "puro" en el sentido de no contener expresiones imperativas (como `goto`), y solo admitir funciones puras. Además, también tendría problemas a la hora de asignar correctamente las variables libres a una función[1] tal y como se procede formalmente en el cálculo lambda [23]. A partir de entonces se darían los siguientes desarrollos en el campo de los lenguajes de programación funcionales:

- En 1966, Landin describe la familia de lenguajes de programación funcionales ISWIM como "Church sin lambda", donde funciones de orden superior se definen fácilmente. ISWIM también supone la primera aparición de la definición de tipos algebraicos.
- En 1968, en el MIT, Evans empieza a desarrollar PAL, influenciado por ISWIM. Este lenguaje contendría características como el uso de expresiones como `where` y `let` para la definición de funciones, o verificador de tipos en el *runtime*, pero con aspectos imperativos.
- En 1972, Turner empieza el desarrollo SASL, un lenguaje basado en LISP pero puramente funcional, con asignación correcta de variables libres acorde al cálculo lambda, coincidencia de patrones y evaluación perezosa.
- En última instancia, en 1983 empieza el desarrollo de Miranda, basado en SASL, que incluiría tipos polimórficos, definiciones de tipos algebraicos, y guardas además de las expresiones condicionales habituales. Así pues, podemos considerar Miranda como el primer lenguaje de programación funcional con todas las características principales que tienen los lenguajes funcionales modernos, como por ejemplo Haskell.

Paralelamente, a partir de los años 70, Christopher Strachey y Dana Scott empiezan el desarrollo de la semántica denotacional, que es un acercamiento concreto en el campo de la semántica (donde también existe, entre otras, la semántica operacional) que le atribuye significado a los lenguajes de programación mediante objetos matemáticos. Sin embargo, este acercamiento no sería capaz de abordar el problema de cómo modelar la programación efectiva (esto es, la que concierne a los efectos, como por ejemplo, obtener la entrada por teclado). No fue hasta 1991 que Eugenio Moggi [19] introduciría el concepto de mónada propio de la Teoría de Categorías para modelar los efectos. Todos estos avances permitieron finalmente a los lenguajes de programación funcionales ganar utilidad y obtener el status que tienen hoy en día.

1.4. Programación funcional. Haskell

Haskell comenzó su desarrollo en 1987 cuando se celebró la conferencia "Functional Programming Languages and Computer Architecture" en Portland, Oregon, donde se formó un comité para diseñar un lenguaje de programación funcional común con el

[1] Para introducirse en las nociones del cálculo lambda, véase [3].

objetivo de impulsar la investigación, las aplicaciones y la enseñanza de dicho campo. En 1990 se publicó el primer *Haskell Report* con el que se formó una comunidad de investigadores dispuestos a desarrollar dicho lenguaje. En 1992 se crearía el compilador GHC, el más compilador de Haskell usado hasta ahora, y en 1993 se lanzaría la versión 1.3 de Haskell, donde se implementaría por primera vez la mónada de entrada y salida, que se verá en el Apéndice A. Finalmente, tras una serie de pequeños desarrollos y revisiones graduales, se creó el estándar actual: *Haskell 2010* (y su correspondiente *Haskell Report*).

La principal característica de Haskell, además de todos los lenguajes de programación funcionales, es ser un lenguaje puramente funcional. Esto significa que las funciones siempre deben producir la misma salida dada la misma entrada. En lenguajes de programación imperativos, como en C o Java, una función puede retornar valores distintos dados los mismos argumentos debido a que es posible que dicha función dependa del estado general del programa (variables globales, contenido de listas con contenidos no inmutables, etc.). Esto implica que en Haskell una función no puede tener lo que se llaman "efectos secundarios", que consisten en, como se ha ejemplificado anteriormente, cambiar el *estado* del programa, asignando un valor a una variable global, modificando una dirección de memoria arbitraria, etc. Como explicaremos más adelante, el modelizado de los efectos secundarios en programación funcional se realiza mediante las mónadas.

Además, otras características que posee Haskell a día de hoy son:

- Admitir funciones de orden superior: la implementación de funciones de orden superior en Haskell, es decir, funciones que admiten otras funciones como parámetros, es trivial.
- Ser no estricto: o lo que es lo mismo, poseer evaluación perezosa. Esto significa que los argumentos que se le pasen a una función puedan ser evaluados posteriormente al momento en el que se le pasan a la misma.
- Tener un sistema de tipos estático y polimórfico: Haskell se puede basar o bien en el propio código fuente del programa, o bien en su sistema de inferencia de tipos para averiguar los tipos de los datos de entrada y salida de cierto programa. Tras eso, comprueba su seguridad de tipo (*type safety*), es decir, que la ejecución esté garantizada para una serie restringida de valores con un determinado tipo. Por otro lado, que tenga un sistema de tipos polimórfico permite definir tanto tipos como funciones genéricas que sean válidos para cualquier tipo o restringirlos para un conjunto de ellos. Esta última restricción puede hacerse mediante el mecanismo de clases de Haskell, con el que se definen conjuntos de datos que cumplan que sobre ellos se puedan implementar ciertas funciones.
- Soporte de tipos de datos algebraicos: es decir, la definición de tipos como combinación de otros. Esto implica también la posibilidad de definir tipos de forma recursiva, como árboles o listas. Veremos que estos se dividen principalmente en tipos producto y en tipos suma.

2 CATEGORÍAS

Una categoría es simplemente una colección más o menos estructurada de objetos con flechas, morfismos, que los unen. En ella se encuentra la esencia de la composición de dichos morfismos. Así pues, una categoría puede caracterizar a un tipo de estructura matemática de tal modo que posea transformaciones (morfismos) que preserven dicha estructura.

Vamos a ver ejemplos de categorías de varios tipos: desde las categorías que forman estructuras tan bien conocidas como los grupos, los espacios topológicos o los conjuntos, hasta la categoría que forman los tipos y las funciones de Haskell. Esta última será la motivación principal de este trabajo y aportará numerables ejemplos a los conceptos y construcciones categóricas que describiremos a continuación.

Definición 2.1. Una categoría $\mathbf{C}$ consiste en:

- Una colección de objetos, a la que llamamos Obj($\mathbf{C}$): $A, B, C \ldots$
- Una colección de morfismos o flechas, Mor($\mathbf{C}$): $f, g, h \ldots$ Para cada par de elementos $A, B \in \mathbf{C}$ no necesariamente distintos, se tiene el conjunto de morfismos entre A y B definido como

  $$Hom_{\mathbf{C}}(A, B) = \{f \in \text{Mor}(\mathbf{C}) \mid f : A \longrightarrow B\}.$$

  A este conjunto se le llama **hom-set** de A a B en $\mathbf{C}$.
- Para cada morfismo f, denotamos a los siguiente objetos como $A = \text{dom}(f)$ y $B = \text{cod}(f)$ de tal manera que tengamos que $f : A \longrightarrow B$.
- Dados dos morfismos $f : A \longrightarrow B$ y $g : B \longrightarrow C$ tales que $\text{cod}(f) = \text{dom}(g)$, entonces debe existir el morfismo composición de f y g definido como

  $$g \circ f : A \longrightarrow C$$

- Para cada objeto A, debe existir $id_A : A \longrightarrow A$.

De tal manera que cumplan las propiedades siguientes:

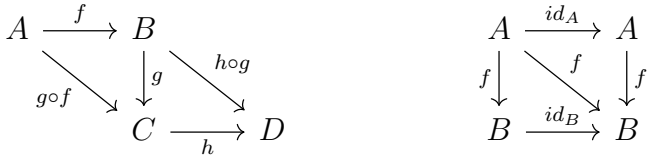
- Asociatividad: para toda terna de morfismos $f : A \longrightarrow B$, $g : B \longrightarrow C$ y $h : C \longrightarrow D$, entonces debe ocurrir:

  $$(h \circ g) \circ f = h \circ (g \circ f)$$

- Unidad: para todo morfismo $f : A \longrightarrow B$ debe ocurrir que:

$$f = f \circ id_A = id_B \circ f$$

Además, estas propiedades se pueden expresar gráficamente de tal forma que los siguiente diagramas conmuten.



Diagramas conmutativos del axioma de asociatividad y de la identidad.

Es habitual en Teoría de Categorías hacer uso de diagramas conmutativos para demostrar ciertas propiedades de algunas construcciones. Veamos ahora algunos ejemplos.

Ejemplo 2.1.

- La categoría **Sets** de la colección de todos los conjuntos y las funciones entre ellos.

- La categoría **Vec** de la colección de todos los espacios vectoriales y las transformaciones lineales entre ellos.

- La categoría **Top** de la colección de todos los espacios topológicos y las funciones continuas entre ellos.

- La categoría **Grp** de la colección de todos los grupos y las homomorfismos de grupos entre ellos. Además, un grupo cualquiera $G \in \mathbf{Grp}$, que es una abstracción de los automorfismos sobre cierto objeto, también cumple los axiomas de la Teoría de Categorías. En este sentido, G es una categoría de un solo objeto, cuyos morfismos son los propios elementos de G que actúan sobre dicho objeto formal, con la propiedad adicional de que todo morfismo tiene inversa en G.

- La categoría **Rng** de la colección de todos los anillos con unidad y los homomorfismos de anillos entre ellos.

- La categoría **Poset** de la colección de todos los conjuntos dotados de un orden parcial y las funciones monótonas (es decir, funciones que verifiquen que dados los posets $P, Q \in \mathbf{Poset}$, y siendo $p \in P$ y $q \in Q$, entonces $p \leq q \implies f(p) \leq f(q)$). Del mismo modo que en **Grp**, un poset en sí mismo también es una categoría. De hecho, es una categoría que tiene un solo morfismo entre cualquier par de objetos (debido a la propiedad antisimétrica del orden parcial). A estas categorías se las llama categorías poset, y son muy comunes.

Ejemplo 2.2. La categoría **Mon** de la colección de todos los monoides y los homomorfismos entre ellos. Un **monoide** es un conjunto M junto con una operación binaria $\cdot : M \times M \longrightarrow M$ tal que existe una unidad $u \in M$ y el producto de los elementos es asociativo. De la misma forma que hemos definido un grupo cualquiera como una categoría de un solo elemento, lo mismo ocurre con $(M, \cdot)$. Debido a las propiedades de los monoides, podemos ver a las categorías como una generalización del concepto de monoide, donde los homomorfismos entre ellos son los functores entre categorías. Es por este motivo que podremos, de nuevo, generalizar varios conceptos que se aplican concretamente a los monoides en la categorías.

Ejemplo 2.3. La categoría **Cat** de la colección de todas las categorías y los functores entre ellos. Veremos en secciones posteriores la definición más formal de **functor** como morfismo entre dos categorías cualesquiera.

Todos estos ejemplos forman parte de las **categorías concretas**, es decir, categorías sobre conjuntos con una estructura. Aunque no tengamos las herramientas conceptuales necesarias para entenderlo, esto significa que toda categoría pequeña (ver Sección 2.1) es concreta si se puede *sumergir* en **Sets**. Sin embargo, esto no es el caso en todas las categorías. Veamos algunos ejemplos de categorías no concretas:

Ejemplo 2.4. Las categorías finitas:

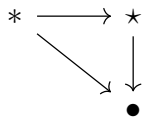
- Categoría **0**, sin objetos ni flechas; las categorías **1**, **2** ó **3**:

$$*$$

Categoría **1**.

$$* \longrightarrow \star$$

Categoría **2**.



Categoría **3**.

- Categoría $\mathbf{n}$, con $n \in \mathbb{N}$. Para formar categorías finitas solo es necesario especificar los n objetos y las flechas que los unen siguiendo los axiomas, pero todo esto sin formar bucles. Un bucle en el diagrama de dicha categoría haría que acabáramos con infinitas flechas: $f \circ g$, $f \circ g \circ f \circ g$, $f \circ g \circ f \circ g \circ f \circ g$, ..., lo cual produciría que no fuera una categoría finita.

Ejemplo 2.5. La categoría opuesta $\mathbf{C^{op}}$ de $\mathbf{C}$ (no necesariamente pequeña). En ella, los objetos siguen siendo $\mathrm{Obj}(\mathbf{C})$, pero por lo que respecta a $\mathrm{Mor}(\mathbf{C^{op}})$, para cualquier morfismo $f : C \longrightarrow D$, con $C, D \in \mathbf{C}$, un morfismo en $\mathbf{C^{op}}$ es $f^* : D^* \longrightarrow C^*$. A esta categoría también se la conoce como **categoría dual** de $\mathbf{C}$.

Ejemplo 2.6. La categoría de los tipos y funciones en lenguajes de programación funcionales. Dado un lenguaje de programación funcional L, la categoría asociada a dicho lenguaje $\mathbf{C}(L)$ tiene como objetos los tipos de L, y sus morfismos son las funciones computables en L. Con esto, la composición de dos funciones f y g se da al aplicar el output de f como input de g, y donde la función identidad es la función "no hacer nada" [2, p. 10]. Un ejemplo concreto de dicha categoría sería $\mathbf{C}(Haskell)$.

2.1. Categorías pequeñas y grandes

Al definir algunas de las categorías anteriores se ha podido notar cómo estas pueden entrar en conflicto con los axiomas de las teoría de conjuntos. Es decir, la categoría **Sets**, entre otras, se queda "pequeña" en dicho marco debido a que es un conjunto que contiene a todos los conjuntos posibles. Por ello hemos definido Obj y Mor como una colección, no como un conjunto.

Definición 2.2. Una categoría $\mathbf{C}$ es pequeña si tanto $\mathrm{Obj}(\mathbf{C})$ como $\mathrm{Mor}(\mathbf{C})$ son conjuntos. En caso contrario, dicha categoría es grande.

Es evidente que todas las categorías finitas vistas en el Ejemplo 2.4 son pequeñas. Por otra parte, la gran mayoría de las categorías con las que trataremos serán grandes.

Definición 2.3. Una categoría $\mathbf{C}$ es localmente pequeña si para todo par de objetos $X, Y \in \mathbf{C}$ la colección de morfismos entre X e Y, es decir, el conjunto $Hom_{\mathbf{C}} = \{f \in \mathrm{Mor}(\mathbf{C}) \mid f : X \longrightarrow Y\}$, es un conjunto.

2.2. Morfismos especiales

En ciertas categorías, como **Sets**, donde los morfismos son funciones, los conceptos de inyectividad y suprayectividad y los conceptos de cancelabilidad e invertibilidad por la izquierda y por la derecha son equivalentes. Sin embargo, este no es el caso en todas las categorías, especialmente porque la inyectividad y suprayectividad de una función depende de los elementos de la categoría. De nuevo, siguiendo una perspectiva categórica, veamos las definiciones formales de los conceptos de cancelabilidad e invertibilidad.

Definición 2.4. Sea $\mathbf{C}$ una categoría.

- Un morfismo $f : A \longrightarrow B$ es invertible por la derecha si existe un morfismo "inverso por la derecha" $f_d : B \longrightarrow A$ tal que

$$f \circ f_d = id_B$$

- Un morfismo $f : A \longrightarrow B$ es invertible por la izquierda si existe un morfismo "inverso por la izquierda" $f_i : B \longrightarrow A$ tal que

$$f_i \circ f = id_A$$

- Un morfismo $f : A \longrightarrow B$ es invertible o es un **isomorfismo** si existe un morfismo "inverso por ambos lados" $f^{-1} : B \longrightarrow A$ tal que

$$f^{-1} \circ f = id_A \text{ y } f \circ f^{-1} = id_B$$

Además, se dice que dos objetos $A, B \in \mathbf{C}$ son isomorfos si, dado $f : A \longrightarrow B$, existe el isomorfismo $f^{-1} : B \longrightarrow A$, y lo denotaremos mediante $A \cong B$.

Definición 2.5. Sea $\mathbf{C}$ una categoría.

- Un morfismo $f : A \longrightarrow B$ es cancelable por la derecha (**epimorfismo** o morfismo **épico**) si para cualquier par de morfismos $g, h : B \longrightarrow C$

$$g \circ f = h \circ f \implies g = h$$

- Un morfismo $f : A \longrightarrow B$ es cancelable por la izquierda (**monomorfismo** o morfismo **mónico**) si para cualquier par de morfismos $g, h : C \longrightarrow B$

$$f \circ g = f \circ h \implies g = h$$

Vistas las definiciones formales, veamos un ejemplo de cómo las definiciones categóricas de invertibilidad y las nociones de inyectividad y sobreyectividad no tienen por qué ser equivalentes en todas las categorías.

Ejemplo 2.7. Sea $P = \{a, b\} \in \mathbf{Poset}$ donde a y b son incomparables, y sea $Q = \{0, 1\} \in \mathbf{Poset}$ donde $0 < 1$. Definimos el morfismo $f : P \longrightarrow Q$ tal que $f(a) = 0$ y $f(b) = 1$. Claramente, f es un morfismo biyectivo, sin embargo, no es un isomorfismo de conjuntos parcialmente ordenados.

Sin embargo, en el caso de la cancelabilidad, el hecho de que un morfismo sea cancelable por ambos lados, es decir, tanto monomorfismo como epimorfismo, no implica que sea un isomorfismo. Pero el hecho de que sea cancelable por un lado e invertible por el otro sí implica que este sea un isomorfismo.

Proposición 2.1.

(a) Sea $f : A \longrightarrow B$, con $A, B \in \mathbf{C}$. Si f es invertible por la izquierda, entonces es mónico.

Demostración. Como es invertible por la izquierda, entonces existe $f' : B \longrightarrow A$ tal que $f' \circ f = id_A$.

$$f \circ g = f \circ h \implies f' \circ (f \circ g) = f' \circ (f \circ h) \implies (f' \circ f) \circ g = (f' \circ f) \circ h \implies$$
$$id_A \circ g = id_A \circ h \implies g = h$$

□

(b) *Sea $f : A \longrightarrow B$, con $A, B \in \mathbf{C}$. Si f es invertible por la derecha, entonces es épico.*

Demostración. Análoga a la anterior. □

(c) *Sea $f : A \longrightarrow B$, con $A, B \in \mathbf{C}$. Si f es un isomorfismo, entonces es épico y mónico.*

Demostración. Trivial por las dos proposiciones anteriores. Además, probemos la unicidad de las inversas. Supongamos que existen $f_1^{-1}, f_2^{-1} : B \longrightarrow A$ morfismos en $\mathbf{C}$ inversos de f por ambos lados. Entonces:

$$f_1^{-1} \circ f = 1_A \quad \text{y} \quad f \circ f_2^{-1} = 1_B$$

Supongamos también que $f \circ f_1^{-1} = f \circ f_2^{-1}$ o que $f_1^{-1} \circ f = f_2^{-1} \circ f$. Debemos ver $f_1^{-1} = f_2^{-1}$.

$$f_1^{-1} = f_1^{-1} \circ 1_B = f_1^{-1} \circ (f \circ f_2^{-1}) = (f_1^{-1} \circ f) \circ f_2^{-1} = 1_A \circ f_2^{-1} = f_2^{-1}$$

Con esto también de demuestra que si existen dos inversas, entonces coinciden. □

Proposición 2.2. *Sea $f : A \longrightarrow B$.*

(a) *Si f es mónico y tiene inversa por la derecha (es decir, existe $g : B \longrightarrow A \mid f \circ g = id_B$), entonces f es un isomorfismo también.*

Demostración.

$$f \circ id_A = f = id_B \circ f = (f \circ g) \circ f = f \circ (g \circ f) \implies id_A = f \circ g$$

□

(b) *Si f es épico y tiene inversa por la izquierda (es decir, existe $g : B \longrightarrow A \mid g \circ f = id_A$), entonces f es un isomorfismo también.*

Demostración. Análoga a la anterior. □

2.3. Propiedades universales

Tradicionalmente, cuando se define un objeto matemático se hace describiendo lo que el objeto es, y tras esto se enumeran sus propiedades. Sin embargo, desde la perspectiva categórica, resulta más útil denotar un objeto por la propiedad universal (y única) que lo caracteriza. En las secciones posteriores veremos muchas de estas construcciones en las que se hace uso de una propiedad universal para definirlas. Al fin y al cabo, la noción subyacente a una propiedad universal es la de que dado un

objeto o una estructura que cumple ciertos requisitos, dicho objeto o dicha estructura es "la mejor" (la "más universal") porque solo desde cualquier objeto llega hasta ella un único morfismo en toda la categoría.

Así pues, vamos a ver varios ejemplos de cómo algunas estructuras que parecen a priori no necesariamente categorías lo acaban siendo. Esto es, las propiedades de dichos objetos no son de los objetos en sí, sino de los morfismos que los relacionan.

Ejemplo 2.8. En el Ejemplo 2.2 hemos visto la estructura algebraica de los monoides y cómo esta se puede generalizar mediante categorías. Veamos ahora una ejemplo aplicado a los monoides de una propiedad universal. Pero antes, veamos la definición de *monoide libre* según las propiedades de sus elementos.

Dado un conjunto A, un monoide libre $M(A)$ sobre A es el conjunto de secuencias finitas que se pueden crear a partir de los elementos de A, en el que también se ha definido una operación binaria (la concatenación), y en el que hay unidad. En otras palabras, A genera los elementos de $M(A)$. O sea, que dicho monoide cumple:

- Para todo $m \in M(A)$:

$$m = a_1 * a_2 * \ldots * a_n, \text{ con } a_i \in A$$

- No existen relaciones no triviales, es decir, si dos secuencias son iguales, $a_1 * a_2 * \ldots * a_n = a'_1 * a'_2 * \ldots * a'_n$, entonces $a_1 = a'_1, \ldots, a_n = a'_n$.

Estas dos condiciones, que son dependientes de los elementos del monoide en sí, se pueden redefinir de forma categórica mediante el uso de una propiedad universal que opere con morfismos en **Mon** (es decir, con homomorfismos de monoides).

Llamamos $|M|$ al conjunto que subyace al monoide M. Entonces para cada homomorfismo de monoides $f : M \longrightarrow N$, existe la función $|f| : |M| \longrightarrow |N|$. Definimos pues como monoide libre $M(A)$ al monoide que cumple la siguiente propiedad universal:

Existe una función $i : A \longrightarrow M(A)$, y dado cualquier monoide N y una función (en **Sets**) $\overline{f} : A \longrightarrow |N|$, entonces existe un **único** homomorfismo (en **Mon**) $\overline{f} : M(A) \longrightarrow N$ de tal forma que $f = |\overline{f}| \circ i$. Es decir, que el siguiente diagrama conmute:

$$M(A) \xrightarrow{\overline{f}} N$$

$$\begin{array}{ccc} |\, M(A)\, | & \xrightarrow{|\overline{f}|} & |\, N\, | \\ {\scriptstyle i}\uparrow & \nearrow_{f} & \\ A & & \end{array}$$

En la Proposición 1.9 de [2, p. 19] se demuestra cómo la definición de monoide libre mediante esta propiedad universal es equivalente a la definición mediante las dos propiedades descritas anteriormente que deben cumplir.

Ejemplo 2.9. Dado cualquier grafo dirigido G, que consiste en dos conjuntos V (vértices) y L (aristas) junto con dos funciones $origen : L \longrightarrow V$ y $destino : L \longrightarrow V$, podemos generar una categoría. La categoría generada por un grafo se llama *categoría libre*, y la representaremos como $\mathbf{C}(G)$. Así pues, los vértices de G, V, corresponden con los objetos de $\mathbf{C}(G)$; las aristas, L, con sus morfismos, y finalmente $origen = dom$ y $destino = cod$. Todo esto está representado mediante el functor (cuya definición formal se verá en la Definición 4.1) siguiente:

$$U : \mathbf{Cat} \longrightarrow \mathbf{Graphs} \tag{1}$$

De la misma forma que los monoides libres, una categoría libre está definida por ser una categoría con la siguiente propiedad universal:

Existe un homomorfismo de grafos $i : G \longrightarrow |\mathbf{C}(G)|$ (siendo $|\mathbf{C}(G)|$ el grafo subyacente de $\mathbf{C}(G)$), y para cualquier otro homomorfismo de grafos $h : G \longrightarrow |\mathbf{D}|$, siendo $\mathbf{D}$ una categoría, entonces hay un functor único $\overline{h} : \mathbf{C}(G) \longrightarrow D$:

$$\mathbf{C}(G) \xrightarrow{\overline{h}} \mathbf{D}$$

$$\begin{array}{ccc} |\,\mathbf{C}(G)\,| & \xrightarrow{|\overline{h}|} & |\,\mathbf{D}\,| \\ {\scriptstyle i}\uparrow & \nearrow_{h} & \\ G & & \end{array}$$

de tal forma que $|\overline{h}| \circ i = h$.

2.4. La categoría Hask

Siguiendo el Ejemplo 2.6, definimos la categoría **Hask** como el conjunto de tipos y de funciones posibles en Haskell. Como es esperable, los objetos de esta categoría son los tipos de Haskell. Por ejemplo, los número naturales, `Nat`:

```
data Nat = Zero | Succ Nat
```


```
data Nat = Zero | Succ Nat
```


O el tipo booleano, `Bool`:

```
data Bool = True | False
```

Asimismo, las funciones en *Haskell* son los morfismos de **Hask**. Esto es, para cada objeto en **Hask**, es decir, por cada tipo `a`, tenemos:

- La función identidad:

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

- El morfismo composición entre dos funciones de Haskell tales que `f :: a -> b` y `g :: b -> c`:

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

De esta definición es evidente que el requisito de asociatividad se cumple. Veamoslo:

```
(f . (g . h)) x
    = (por definición de .)
f ((g . h) x)
    = (por definición de .)
f (g (h x))
    = (por definición de .)
(f . g) (h x)
    = (por definición de .)
((f . g) . h) x
```

Y además, como `id` está definida para cualquier tipo `a`, se obtiene:

```
(id . f) x
    = (por definición de .)
(id . (f x))
    = (por definición de id)
f x
    = (por definición de id)
f (id x)
    = (por definición de .)
(f . id) x
```

Nota 2.1. *En realidad no todas las funciones de Haskell pueden llegar a retornar un valor en un número finito de pasos, y tampoco podemos llegar a saber cuáles son esas funciones debido a que este es un problema indecidible. Es por esto que todos los tipos de Haskell poseen un valor adicional que no viene dado explícitamente en la definición del tipo. Este valor es* `Bottom`*, que se refiere a una computación que nunca termina. Sin embargo, una función que retorna* `Bottom` *no es pura, sino parcial. Es por ello que consideramos* **Hask** *como la categoría de los tipos y funciones en Haskell que no tienen el valor de* `Bottom`*. Debido a estos problemas, usualmente se suele referir a este subconjunto de* **Hask** *qué sí tiene estructura categórica como* **Hask** *Platónico.*

2.5. Monoides en Haskell

A continuación vamos a ver un ejemplo de monoide en Haskell: el tipo lista (`[a]`).

Ejemplo 2.10. El tipo `[a]` es de hecho un monoide libre sobre `a`. En primer lugar, es un monoide porque `[a]` es una instancia de la clase `Monoid` de la siguiente forma:

```
instance Semigroup [a] where
  (<>) = (++)

instance Monoid [a] where
  mempty = []
```

En segundo lugar, `[a]` es libre porque:

- Para toda lista de tipo `[a]`, es decir, `l :: [a]`, se tiene que

  ```
  l = [x1] ++ [x2] ++ [x3] ++ ... ++ [xn]
  ```

  con `[xi] :: [a]`.

- Si dos listas son iguales, es porque sus elementos son iguales uno a uno.

No obstante, como hemos visto en el Ejemplo 2.8, la definición anterior es además equivalente a definir `[a]` mediante la siguiente propiedad universal:

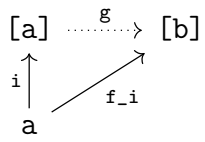
Sea la función

```
i :: a -> [a]
```

que toma cualquier valor `v :: a` y devuelve dicho valor en una lista, es decir, `[v] :: [a]`, y sea

```
f_i :: a -> [b]
```

una función cualquiera entre los tipos `a` y `[b]`. Entonces, si `[a]` es un monoide libre sobre `a`, debe existir una única función entre listas, `g :: [a] -> [b]`, que haga conmutar el siguiente diagrama:

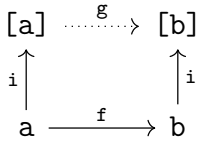


Para obtener `g`, recordemos en primer lugar que Haskell implementa la función `map` (véase el Apéndice A para ejemplos de su uso), definida de la siguiente manera:

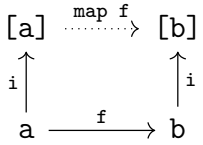
```
map :: (a -> b) -> ([a] -> [b])
```

tal que toma una función `f :: a -> b`, y devuelve una función `[a] :: [b]`.

En segundo lugar, podemos descomponer `f_i` como la composición de funciones `i` ∘ `f`, resultando en el siguiente diagrama conmutativo:



Así pues, podemos ver que, de hecho, la función `g` es determinada de forma única por `f` mediante `map`. Es decir, `map f`, además de ser un "homomorfismo de listas", hace conmutar el diagrama siguiente:



Con esto podemos concluir que `[a]` es un monoide libre sobre `a` porque la propiedad universal de los monoides libres se cumple.

3 CONSTRUCCIONES

Una vez definidos los conceptos fundamentales pertenecientes a las categorías, podemos empezar a pensar en construcciones universales que se pueden llegar a realizar a partir de ellas. Estas construcciones son importantes porque pueden llegar a caracterizar una categoría. Veremos un ejemplo de este fenómeno más adelante cuando se traten las categorías cartesianas cerradas. Además, identificar dichas construcciones nos sirve, al igual que lo hacíamos con la propiedad universal en la Sección 2.3, para definir objetos en una categoría en torno a sus relaciones con los demás.

3.1. Objetos iniciales y finales

Definición 3.1. Dada una categoría $\mathbf{C}$,

- un objeto $\mathbf{0}$ es **inicial** si para todo objeto $C \in \mathbf{C}$ existe un único morfismo

$$\mathbf{0} \longrightarrow C$$

- un objeto $\mathbf{1}$ es **terminal** si para todo objeto $C \in \mathbf{C}$ existe un único morfismo

$$C \longrightarrow \mathbf{1}$$

Cabe destacar que de la misma forma que todas las propiedades universales se definen salvo isomorfismo, lo mismo ocurre con las construcciones universales, y en concreto, con los objetos iniciales y terminales. Vamos a ver que a pesar de que pueda darse que en una categoría los objetos iniciales o terminales no sean únicos, estos entonces son al menos isomorfos.

Proposición 3.1. *Si $C, C' \in \mathbf{C}$ son objetos iniciales (o terminales) en una categoría $\mathbf{C}$, entonces $C \cong C'$.*

Demostración. Como C y C' son iniciales (o terminales) en la misma categoría $\mathbf{C}$, entonces deben existir los morfismos $m : C \longrightarrow C'$ y $m' : C \longrightarrow C'$. Además, deben existir $id_C : C \longrightarrow C$ y $id_{C'} : C' \longrightarrow C'$. Entonces, mediante el siguiente diagrama:

$$\begin{array}{ccc} C & \xrightarrow{m} & C' \\ & \searrow^{id_C} & \downarrow{\scriptstyle m'} \quad \searrow^{id_{C'}} \\ & & C \xrightarrow{m} C' \end{array}$$

Tenemos que $m \circ m' = id_C$, y $m' \circ m = id'_C$, luego m es un isomorfismo y $C \cong C'$. $\square$

Asimismo, una categoría puede no tener objetos iniciales o terminales. Veamos algunos ejemplos:

Ejemplo 3.1. En **Set**, los objetos iniciales son los conjuntos de un solo elemento $\{x\}$, y el objeto terminal es $\emptyset$.

Ejemplo 3.2. De forma análoga a **Set**, en **Cat** el objeto inicial es la categoría **0** y el objeto terminal es la categoría **1**.

Ejemplo 3.3. En **Grp**, los objetos iniciales y terminales son los grupos que contienen un elemento, $\{g\} = G \in \mathbf{Grp}$ (que por definición de grupo, como todo elemento debe tener inverso, entonces todo ellos deben ser la identidad en ese grupo), ya que para todo $G' \in \mathbf{Grp}$ hay un único homomorfismo entre G' y $\{g\}$ (que es el homomorfismo constante) y viceversa.

Ejemplo 3.4. Un *poset* entendido como una categoría puede no tener objeto inicial o terminal. De hecho, un objeto en un *poset* cualquiera P es inicial o terminal si y solo si es el menor o el mayor respectivamente en P. Entonces, el *poset* $(\mathbb{Z}, \leq)$ no tiene objeto inicial ni terminal.

3.2. Productos

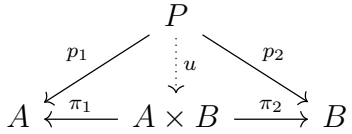
Definición 3.2. Un producto en una categoría $\mathbf{C}$ consiste en un objeto $A \times B$ junto con los objetos $A, B \in \mathbf{C}$ y los morfismos π_1, π_2 (llamados proyecciones), representados por el siguiente diagrama

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B,$$

donde se cumple que, si se tiene otro diagrama de la forma

$$A \xleftarrow{p_1} P \xrightarrow{p_2} B,$$

entonces existe un único morfismo $u : P \longrightarrow A \times B$ tal que el siguiente diagrama conmute:



Es decir, $p_1 = \pi_1 \circ u$ y $p_2 = \pi_2 \circ u$.

Aquí utilizamos, de nuevo, y como será habitual en las construcciones y conceptos que se expondrán más adelante, una propiedad universal de tal forma que nos garantiza la unicidad de la estructura salvo isomorfismo.

Veamos algunos ejemplos.

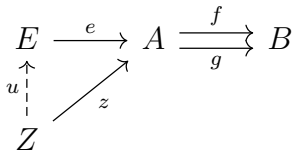
Ejemplo 3.5. El producto cartesiano en **Set**.

Ejemplo 3.6. En un *poset* entendido como categoría, la función ínfimo.

Ejemplo 3.7. En un anillo considerado como categoría, la función del máximo común divisor.

3.3. Igualador

Definición 3.3. Sea $\mathbf{C}$ una categoría y sean $f, g \in \mathrm{Mor}(\mathbf{C})$ dos morfismos paralelos (esto es, dos morfismos que comparten el mismo dominio y codominio) tales que $f, g : A \longrightarrow B$. Entonces, un igualador consiste en un objeto $E \in \mathbf{C}$ y un morfismo $e : E \longrightarrow A$ tal que dado cualquier objeto $Z \in \mathbf{C}$ y un morfismo $z : Z \longrightarrow A$, se tiene el morfismo único $u : Z \longrightarrow E$. Es decir, que e es universal, y que este diagrama conmuta:



Es decir, $e \circ u = z$.

El concepto de igualador generaliza los conceptos de kernel de un homomorfismo y de variedad definida mediante ecuaciones. Además, cabe destacar la naturaleza "monomórfica" del igualador, concretamente, del morfismo e anterior.

Proposición 3.2. *Dado un igualador $E \in \mathbf{C}$ de cualquier par de morfismos $f, g : A \longrightarrow B$, con $A, B \in \mathbf{C}$, junto con el morfismo $e : E \longrightarrow A$, entonces e es un monomorfismo.*

Demostración. Debemos demostrar que dados $h_1, h_2 : Z \longrightarrow E$, se cumple que

$$e \circ h_1 = e \circ h_2 \implies h_1 = h_2.$$

Mirando el diagrama de la Definición 3.3, vemos que, en este caso, $z = e \circ h_1 = e \circ h_2$, y que además, $f \circ z = f \circ e \circ h_1$. Como f y g son morfismos paralelos, entonces tenemos además que $g \circ z = f \circ e \circ h_1$. Finalmente, por la propiedad universal en la definición del igualador, debe existir un único morfismo $u : Z \longrightarrow E$ tal que $e \circ u = z$, entonces partiendo de que $z = e \circ h_1 = e \circ h_2$, se tiene que $u = h_1 = h_2$. □

3.4. Construcciones duales

Como hemos visto en el Ejemplo 2.5, podemos formar una nueva categoría, $\mathbf{C^{op}}$, a partir de una dada, $\mathbf{C}$, en la que $\mathrm{Obj}(\mathbf{C}) = \mathrm{Obj}(\mathbf{C^{op}})$ y en la que para todo morfismo

$f \in \mathrm{Mor}(\mathbf{C})$, se tiene un morfismo dual $f^* \in \mathrm{Mor}(\mathbf{C^{op}})$ tal que $dom(f) = cod(f^*)$ y $cod(f) = dom(f^*)$. A esta nueva categoría la hemos denotado como categoría dual de $\mathbf{C}$. Este tipo de categorías nos es útil porque cualquier proposición cierta para toda categoría, lo será también para cualquier categoría dual. Además, evidentemente, $(\mathbf{C^{op}})^{\mathbf{op}} = \mathbf{C}$.

Así pues, en el caso de los objetos iniciales y terminales, un objeto inicial en una categoría $\mathbf{C}$ cualquiera es terminal en $\mathbf{C^{op}}$ y viceversa.

En el caso de los productos, un producto en $\mathbf{C^{op}}$ se llama *coproducto*, y viene definido de la siguiente forma:

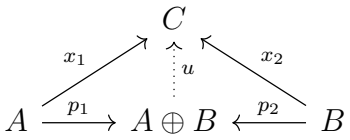
Definición 3.4. Un coproducto en una categoría $\mathbf{C}$ consiste en un objeto $A \oplus B$ junto con los objetos $A, B \in \mathbf{C}$ y los morfismos p_1, p_2 (llamados inyecciones canónicas), representados por el siguiente diagrama

$$A \xrightarrow{p_1} A \oplus B \xleftarrow{p_2} B,$$

donde se cumple que dado otro diagrama de la forma

$$A \xrightarrow{x_1} C \xleftarrow{x_2} B,$$

entonces existe un único morfismo $u : A \oplus B \longrightarrow C$ tal que el siguiente diagrama conmuta:



Es decir, $x_1 = u \circ p_1$ y $x_2 = u \circ p_2$.

El coproducto puede entenderse de forma análoga al producto como una "suma" de elementos de una categoría. Veamos algunos ejemplos.

Ejemplo 3.8. Unión disjunta en **Set**.

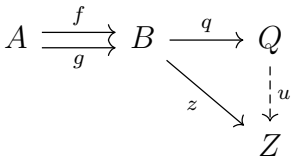
Ejemplo 3.9. En **Grp**, el producto libre de dos grupos.

Ejemplo 3.10. En un *poset* entendido como categoría, la función supremo.

Ejemplo 3.11. En un anillo considerado como categoría, el mínimo común múltiplo.

Lo mismo ocurre con los igualadores. Un igualador en $\mathbf{C^{op}}$ se llama *coigualador*, y viene definido de la siguiente manera.

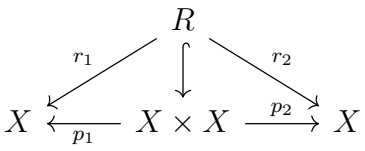
Definición 3.5. Sean $f, g \in \mathrm{Mor}(\mathbf{C})$ dos morfismos paralelos tales que $f, g : A \longrightarrow B$. Entonces, un coigualador consiste en un objeto $Q \in \mathbf{C}$ y un morfismo $q : B \longrightarrow Q$ tal que dado cualquier objeto $Z \in \mathbf{C}$ y un morfismo $z : B \longrightarrow Z$, se tiene el morfismo único $u : Q \longrightarrow Z$. Es decir, que q es universal y que el siguiente diagrama conmuta:



Es decir, $u \circ q = z$.

Podemos entender el coigualador como la construcción que describe, en general, los cocientes dada cierta relación de equivalencia.

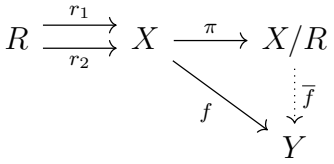
Ejemplo 3.12. Sea $R \subseteq X \times X$ una relación de equivalencia en $X \in \mathbf{Sets}$. Sean r_1, r_2 las proyecciones de la inclusión $R \subseteq X \times X$. Con esto tenemos el diagrama siguiente:



si consideramos la proyección canónica de la relación de equivalencia

$$\pi : x \longrightarrow X/R$$

definida de modo que lleva cada elemento $x \in X$ a su clase de equivalencia $[x]$, obtenemos el coigualador $(X/R, \pi)$. Es decir, dada una función $f : X \longrightarrow Y$ en **Sets** cualquiera, tenemos el siguiente diagrama del coigualador:



donde existe una función $\overline{f}$ tal que $\overline{f} \circ \pi = f$.

3.5. Construcciones en Haskell

En esta sección vamos a ver las diferentes construcciones que hemos visto generalizadas para toda categoría aplicadas particularmente a **Hask**. Veremos los objetos iniciales y terminales en dicha categoría, así como los tipos de datos algebraicos que representan en **Hask** el producto y el coproducto.

3.5.1. Objetos iniciales y terminales de Hask

El objeto inicial en **Hask** es el tipo `Void`, ya que no puede existir ninguna función desde ningún tipo en **Hask** hasta `Void` definida tal que:

```
f :: a -> Void
```

El tipo `Void` puede interpretarse pues como el conjunto vacío en **Set**, es decir, un tipo que no contiene ningún valor. De hecho, se define la función `absurd`, que nos muestra cómo existe un único morfismo entre `Void` y cualquier otro tipo en **Hask**:

```
absurd :: Void -> a
absurd = absurd
```

Por otro lado, el objeto terminal en **Hask** es el tipo `()`, que contiene únicamente un valor. Se define pues la función `unit` que nos muestra la unicidad entre cualquier tipo en **Hask** y `()`:

```
unit :: a -> ()
unit _ = ()
```

Nota 3.1. *Todo esto teniendo en cuenta que los tipos en* **Hask** *no contienen el valor* `Bottom`*. Si lo tuvieran, la unicidad de ambas funciones definidas anteriormente se perdería, ya que podríamos definirlas como* `undefined`*.*

3.5.2. Tipos producto

La implementación canónica de los productos en Haskell es la tupla [18].

Una tupla no es necesariamente conmutativa, aunque lo es salvo isomorfismo, siendo este la función `swap` (que es inversa de sí misma) definida de la siguiente forma:

```
swap :: (a, b) -> (b, a)
swap (a, b) = (b, a)
```

Las proyecciones del producto, de la tupla, vienen dadas por las funciones `fst` y `snd`, que se corresponden con la proyección de la primera y segunda componente respectivamente de la tupla.

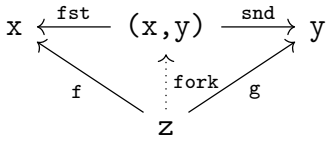
```
fst :: (a, b) -> a
fst (x, _) = x

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

Asimismo, para comprobar que de hecho una tupla es un producto en el sentido de teoría de categorías, tenemos la siguiente función:

```
fork :: (a -> b) -> (a -> c) -> a -> (b, c)
fork f g z = (f z, g z)
```

De tal forma que obtenemos el diagrama del producto siguiente:



3.5.3. Tipos suma

La implementación canónica de los tipos suma, que surgen a partir de los coproductos en **Hask**, viene dada por el tipo `Either`, que representa un valor con dos posibilidades, y se define de la siguiente forma:

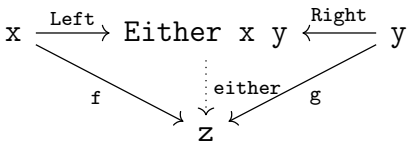
```
data Either a b = Left a | Right b
```

De la misma forma que antes, las instancias de tipo `Either`, es decir, los coproductos, son conmutativos salvo isomorfismo.

De nuevo, las función que nos permite saber que `Either` es un coproducto es `either`, definida de la siguiente forma:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left  x) = f x
either _ g (Right y) = g y
```

Con lo que obtenemos el diagrama del coproducto siguiente:



4 FUNCTORES

Como hemos visto en el Ejemplo 2.3, se puede formar la categoría de todas las categorías, **Cat**, cuyos morfismos se llaman functores. Estudiar este tipo especial de morfismos entre categorías es de especial utilidad para llegar a poder realizar construcciones más avanzadas, como los conos o, como veremos más adelante, las transformaciones naturales.

Definición 4.1. Sean **C** y **D** dos categorías cualesquiera. Un functor F es un morfismo en **Cat** de la forma $F : \mathbf{C} \Longrightarrow \mathbf{D}$ tal que asigna objetos de **C** a objetos **D**

$$F : \mathrm{Obj}(\mathbf{C}) \longrightarrow \mathrm{Obj}(\mathbf{D}),$$

y morfismos de **C** a morfismos **D**

$$F : \mathrm{Mor}(\mathbf{C}) \longrightarrow \mathrm{Mor}(\mathbf{D})$$

de manera que si $A, B \in \mathbf{C}$, entonces:

$$F : Hom_{\mathbf{C}}(A, B) \longrightarrow Hom_{\mathbf{D}}(FA, FB).$$

Además, debe preservar la estructura de categoría, es decir, la identidad y la composición:

- Dado $A \in \mathbf{C}$,
$$Fid_A = id_{FA} \tag{2}$$
- Dados dos morfismos $f, g \in \mathbf{C}$ tales que su composición exista,
$$F(g \circ f) = Fg \circ Ff \tag{3}$$

Usualmente los functores con los que tratamos son los **functores covariantes**, en los que la aplicación de este functor a una categoría preserva la dirección de los morfismos. Sin embargo, también se puede hablar de **functores contravariantes**, que invierten la dirección de los morfismos de la categoría a la que se aplica. Es decir, un functor contravariante de **C** a **D** no es más que un functor covariante tal que

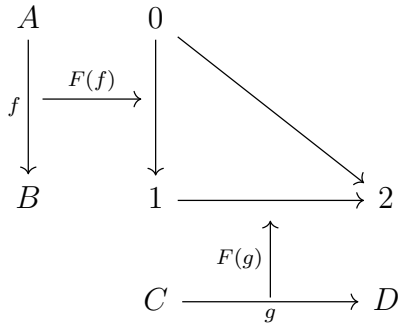
$$F : \mathbf{C}^{op} \Longrightarrow \mathbf{D},$$

donde la asignación de morfismos de **C** a morfismos **D** se produce de la siguiente manera:

$$F : Hom_{\mathbf{C}}(A, B) \longrightarrow Hom_{\mathbf{D}}(FB, FA).$$

Nota 4.1. *A pesar de que la imagen de un functor F sobre una categoría* $\mathbf{C}$ *preserve la estructura de categoría, esto no significa necesariamente que la imagen de F forme una categoría. Veamos un ejemplo:*

Ejemplo 4.1. Sea $\mathbf{C}$ la categoría cuyos elementos son A, B, C, D, con los morfismos identidad correspondientes y los morfismos $f : A \longrightarrow B$ y $g : C \longrightarrow D$ y sea la categoría $\mathbf{3}$. Consideremos el functor $F : \mathbf{C} \Longrightarrow \mathbf{3}$, donde $F(f) = (0 \to 1)$ y $F(g) = (1 \to 2)$.



Se puede ver que aunque el morfismo $0 \to 2$ existe en $\mathbf{3}$, este no pertenece a la imagen de F, es decir, no existe $F(g) \circ F(f)$, luego no forma una categoría.

Veamos una condición necesaria para que la imagen de F sí forme una categoría.

Proposición 4.1. *Sea $F : \mathbf{C} \Longrightarrow \mathbf{D}$ un functor inyectivo, es decir, un functor tal que la parte que asigna objetos en* $\mathbf{C}$ *a* $\mathbf{D}$ *y la parte que asigna morfismos en* $\mathbf{C}$ *a* $\mathbf{D}$ *es inyectiva. Entonces la imagen de F, $F\mathbf{C}$, forma una categoría.*

Demostración. Sean $A, B, C \in \mathbf{C}$. Para que la composición $Fg \circ Ff$ exista debe ocurrir que $Ff : FA \longrightarrow FB$ y que $Fg : FB \longrightarrow FC$. Por la inyectividad de F, particularmente de $F : \mathrm{Obj}(\mathbf{C}) \longrightarrow \mathrm{Obj}(\mathbf{D})$, sabemos que $f : A \longrightarrow B$ y $g : B \longrightarrow C$. Entonces, como $\mathbf{C}$ es una categoría, $g \circ f$ existe y se cumple que

$$Fg \circ Ff = F(g \circ f) \in F\mathbf{C}$$

□

4.1. Tipos de functores

Un tipo de functores que podemos encontrar normalmente a lo largo de la Teoría de Categorías son aquellos que envían elementos "estructurados" de una categoría a otra sin esta estructura. Estos functores, que carecen de una definción precisa, los llamamos **functores olvidadizos**.

Ejemplo 4.2. En **Grp** podemos crear un functor que envíe cada uno de sus elementos a su conjunto subyacente correspondiente en **Sets**.

$$F : \mathbf{Grp} \Longrightarrow \mathbf{Sets}$$

En este caso, el functor "olvida" la estructura de grupo, su operación asociada y propiedades, para enviarlo al conjunto de sus elementos.

Definición 4.2. Un **endofunctor** F es un functor caracterizado por tener dominio codominio idénticos. Esto es, un functor de una categoría $\mathbf{C}$ a sí misma.

$$F : \mathbf{C} \Longrightarrow \mathbf{C}$$

Ejemplo 4.3. Un endofunctor sobre un grupo G considerado como categoría es un automorfismo $F : G \Longrightarrow G$.

Definición 4.3. Un **bifunctor** $B : \mathbf{A} \times \mathbf{B} \Longrightarrow \mathbf{C}$ es un functor desde el producto de dos categorías a una tercera.

El producto de dos categorías se comporta de la forma esperada: por una parte, (A, B) es un par ordenado de elementos de $\mathrm{Obj}(\mathbf{A} \times \mathbf{B})$ (con $A \in \mathbf{A}$ y $B \in \mathbf{B}$), y por otra parte, (f, g) es un par ordenado de morfismos de $\mathrm{Mor}(\mathbf{A} \times \mathbf{B})$, donde la composición de dos morfismos se realiza componente a componente:

$$(f, g) \circ (f', g') = (f \circ f', g \circ g').$$

Ejemplo 4.4. Un hom-functor $Hom(=, -) : \mathbf{C}^{op} \times \mathbf{C} \Longrightarrow \mathbf{Sets}$ que veremos en la Sección 4.1.1 es un ejemplo de un bifunctor.

Definición 4.4. Sea $F : \mathbf{C} \Longrightarrow \mathbf{D}$ un functor, con $\mathbf{C}$ y $\mathbf{D}$ dos categorías cualesquiera. Entonces, para todo $A, B \in \mathbf{C}$,

- F es **completo** si $F : Hom_{\mathbf{C}}(A, B) \longrightarrow Hom_{\mathbf{D}}(FA, FB)$ es una función suprayectiva.
- F es **fiel** si $F : Hom_{\mathbf{C}}(A, B) \longrightarrow Hom_{\mathbf{D}}(FA, FB)$ es una función inyectiva.
- F es **completamente fiel** si $F : Hom_{\mathbf{C}}(A, B) \longrightarrow Hom_{\mathbf{D}}(FA, FB)$ es una función biyectiva.
- F es una **inmsersión** si $F : Hom_{\mathbf{C}}(A, B) \longrightarrow Hom_{\mathbf{D}}(FA, FB)$ es una función biyectiva y además $F : \mathrm{Obj}(\mathbf{C}) \longrightarrow \mathrm{Obj}(\mathbf{D})$ es una función inyectiva.

Ejemplo 4.5. Los functores olvidadizos $F : \mathbf{C} \Longrightarrow \mathbf{Sets}$ son generalmente (dependiendo de la definición de "olvidadizo" que tomemos) fieles.

Ejemplo 4.6. El functor $F : \mathbf{Sets} \Longrightarrow \mathbf{Sets}$ que envía $A \in \mathbf{Sets}$ a su conjunto potencia $\mathcal{P}(A) \in \mathbf{Sets}$, y cada función $f : A \longrightarrow B$ a $Ff : \mathcal{P}(A) \Longrightarrow \mathcal{P}(B)$, es inyectivo y por tanto fiel, pero no completo.

Ejemplo 4.7. El functor $F : \mathbf{Ab} \Longrightarrow \mathbf{Grp}$, que envía elementos y morfismos de la categoría de los grupos abelianos a la de los grupos, es tanto completo como fiel.

4.1.1. Hom-Functores

Este tipo particular de functores nos servirá como base más adelante en el estudio de dos conceptos muy importantes en Teoría de Categorías: el functor representable y sus aplicaciones en la demostración del Lema de Yoneda y sus consecuencias.

Definición 4.5. Sea $\mathbf{C}$ una categoría localmente pequeña.

Sean $A, B \in \mathbf{C}$. Llamamos **hom-functor covariante** al functor

$$Hom(A, -) : \mathbf{C} \Longrightarrow \mathbf{Sets},$$

que asigna a cada objeto $X \in \mathbf{C}$ el conjunto $Hom(A, X)$, y a cada morfismo $f : X \longrightarrow Y$, con $X, Y \in \mathbf{C}$, el morfismo siguiente:

$$Hom(A, f) : Hom(A, X) \longrightarrow Hom(A, Y),$$

definido por $(m : A \to X) \mapsto (f \circ m : A \to X \to Y)$.

Análogamente, llamamos **hom-functor contravariante** al functor

$$Hom(-, B) : \mathbf{C^{op}} \Longrightarrow \mathbf{Sets},$$

que asigna a cada objeto $X \in \mathbf{C}$ el conjunto $Hom(X, B)$, y a cada morfismo $g : X \longrightarrow Y$ en $\mathbf{C^{op}}$, con $X, Y \in \mathbf{C}$, el morfismo siguiente:

$$Hom(g, B) : Hom(Y, B) \longrightarrow Hom(X, B),$$

definido por $(m : Y \to B) \mapsto (m \circ g : X \to Y \to B)$.

Ahora vamos a ver que el hom-functor (covariante) es efectivamente un functor. Por una parte, debemos ver que $Hom(A, id_X) = id_{Hom(X,A)}$, siendo $x : A \longrightarrow X$:

$$\begin{aligned} Hom(A, id_X(x)) &= id_X \circ x \\ &= x \\ &= id_{Hom(A,X)(x)}. \end{aligned}$$

Por otra parte, veamos que dada la composición de morfismos $g \circ f$, su imagen a través del hom-functor existe:

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

4.2. Diagramas, conos y coconos

En secciones anteriores hemos visto construcciones que partían de un diagrama concreto. Por ejemplo, en el caso del producto, se partía de un diagrama como este:

$$A \longrightarrow A \times B \longleftarrow B$$

Vamos a ver que un diagrama no es más que un functor desde los objetos de una categoría índice $\mathbf{J}$ a los objetos de la categoría que nos interesa estudiar, $\mathbf{C}$.

Definición 4.6. Un diagrama de tipo **J** en **C** es un functor de la forma

$$D : \mathbf{J} \Longrightarrow \mathbf{C},$$

donde denotamos como $i, j, k, \ldots$ los objetos de la categoría índice **J**, y $D(i), D(j), D(k), \ldots$ como $D_i, D_j, D_k, \ldots$, que representan los valores en **C** del diagrama D.

Decimos que un diagrama es finito si Obj(**J**) tiene un número finito de elementos.

Al igual que ocurre con las categorías, existen diagramas pequeños y grandes.

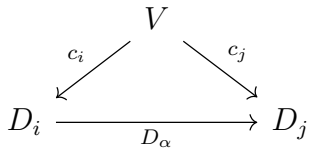
Definición 4.7. Un diagrama $D : \mathbf{J} \Longrightarrow \mathbf{C}$ es **pequeño** si la colección de objetos Obj(**J**) es un conjunto. Asimismo, si este conjunto es finito, entonces el diagrama D es **finito**.

Además, si partimos de un morfismo $\alpha : i \longrightarrow j$ en **J**, su imagen a través del functor será $D_\alpha : D_i \longrightarrow D_j$. Estos morfismos nos servirán para definir el concepto de *cono sobre un diagrama*.

Definición 4.8. Un **cono** sobre un diagrama $D : \mathbf{J} \Longrightarrow \mathbf{C}$ consiste en un vértice $V \in \mathbf{C}$ junto con una familia de morfismos también en **C**, a los que llamamos "aristas" del cono, donde para todo $i \in \mathbf{J}$ se tiene

$$c_i : V \longrightarrow D_i,$$

y donde para todo objeto $j \in \mathbf{J}$ y morfismo $\alpha : i \longrightarrow j$, el siguiente diagrama conmuta:



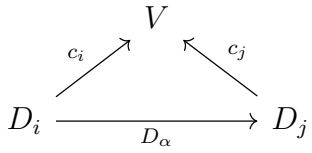
Es decir, $c_j = D_\alpha \circ c_i$. Representamos este cono mediante el par (V, c_i).

Veamos ahora el dual de un cono, el *cocono*:

Definición 4.9. Un **cocono** sobre un diagrama $D : \mathbf{J} \Longrightarrow \mathbf{C}$ consiste en un vértice $V \in \mathbf{C}$ junto con una familia de morfismos también en **C** donde para todo $i \in \mathbf{J}$ se tiene

$$c_i : D_i \longrightarrow V,$$

y donde para todo objeto $j \in \mathbf{J}$ y morfismo $\alpha : i \longrightarrow j$, el siguiente diagrama conmuta:



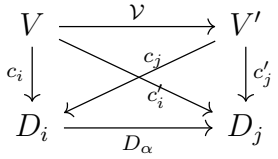
Es decir, $c_i = D_\alpha \circ c_j$.

Podemos imaginar los conos como pirámides de tantos lados como elementos tenga la categoría índice del diagrama $D : \mathbf{J} \Longrightarrow \mathbf{C}$, cuya base se compone de los elementos de $D(\mathbf{J})$ y de los morfismos entre ellos, y cuyo vértice y lados restantes son un elemento de $\mathbf{C}$ y morfismos en $\mathbf{C}$ respectivamente.

Podemos definir también morfismos entre conos. Dados dos conos (V, c_i) y (V', c'_i),

$$\mathcal{V} : (V, c_i) \longrightarrow (V', c'_i)$$

es un morfismo entre los vértices $V, V' \in \mathbf{C}$ tal que el siguiente diagrama (para todo $j \in \mathbf{J}$) conmuta:



En otros términos, $c_j = c'_j \circ \mathcal{V}$.

Una vez definidos estos conceptos, es intuitivo preguntarse si los conos sobre un diagrama como elementos y los morfismos entre ellos forman una categoría.

Proposición 4.2. *La colección de conos sobre un diagrama $D : \mathbf{J} \Longrightarrow \mathbf{C}$ forman una categoría, a la que llamamos* $\mathbf{Cono}(D)$*. Análogamente, a la categoría de coconos sobre D la llamamos* $\mathbf{Cocono}(D)$*.*

Demostración. En primer lugar, dado un vértice $V \in \mathbf{C}$ de un cono cualquiera sobre D, el morfismo $id_{\mathbf{Cono}(D)} : V \longrightarrow V$ existirá porque $\mathbf{C}$ es una categoría. En segundo lugar, dados dos morfismos entre tres conos cualquiera $\mathcal{V}_1 : V \longrightarrow V'$ y $\mathcal{V}_2 : V' \longrightarrow V''$, entonces, por ser V, V' y V'' elementos de $\mathbf{C}$, existirá $\mathcal{V}_2 \circ \mathcal{V}_1 : V \longrightarrow V''$. Por último, dados tres morfismos que admitan las composición entre ellos en $\mathbf{Cono}(D)$, su composición será asociativa porque dichos morfismos están, de nuevo, en $\mathbf{C}$, que es una categoría. □

4.3. Limites y colímites

Cuando un patrón, en este caso el de un cono, se repite en los morfismos de una categoría, entonces puede aparecer una construcción universal. Esta construcción es, valga la redundancia, la más universal de todas, ya que existe un único morfismo desde dicho patrón hasta esa construcción universal. Este es el concepto que motiva la idea de límite.

Definición 4.10. Un **límite** $(\varprojlim D, p)$ de un diagrama $D : \mathbf{J} \Longrightarrow \mathbf{C}$ es un objeto terminal de la categoría $\mathbf{Cono}(D)$. Denotamos los lados del límite como

$$p_i : \varprojlim_j D_j \longrightarrow D_i.$$

Definición 4.11. Un **colímite** $(\varinjlim D, p)$ es un objeto terminal en $\mathbf{Cocono}(D)$ (o un objeto inicial en $\mathbf{Cono}(D)$).

Nota 4.2. *De hecho, como el límite es un cono terminal en* $\mathbf{Cono}(D)$*, esto significa que para cualquier otro cono en dicha categoría existe solamente un morfismo hasta ese límite. Esto se puede formalizar como la siguiente propiedad universal: dado cualquier otro cono* $(C, c_i) \in \mathbf{Cono}(D)$*, entonces existe un único morfismo* $u : C \longrightarrow \varprojlim D$ *tal que para todo* $i \in \mathbf{J}$ *se cumple* $c_i = p_i \circ u$*.*

Definición 4.12. Una categoría $\mathbf{C}$ se dice que es **finitamente completa** si cada diagrama finito $D : \mathbf{J} \Longrightarrow \mathbf{C}$ tiene un límite, y **completa** si cada diagrama pequeño tiene límite. Al dual de esta defínción le llamamos categoría **cocompleta**.

Vamos a ver a continuación una condición necesaria y suficiente para que una categoría sea (finitamente) completa.

Proposición 4.3. *Una categoría es finitamente completa si y solo si tiene productos finitos e igualadores.*

Demostración. Debemos probar que todo límite se pueda construir a partir de un producto.

En primer lugar, sea el diagrama finito $D : \mathbf{J} \Longrightarrow \mathbf{C}$. Si tomamos el producto

$$\prod_{i \in \mathrm{Obj}(\mathbf{J})} D_i,$$

este tiene al menos las proyecciones $p_i : \prod_{i \in \mathrm{Obj}(\mathbf{J})} D_i \longrightarrow D_j$. Sin embargo, estos morfismos no tienen por qué necesariamente conmutar con los morfismos $D_\alpha : D_i \longrightarrow D_j$, con $(\alpha : i \longrightarrow j) \in \mathrm{Mor}(\mathbf{J})$, en el diagrama D.

Entonces, tomamos el producto sobre todos los morfismos de $\mathrm{Mor}(\mathbf{J})$

$$\prod_{\alpha \in \mathrm{Mor}(\mathbf{J})} D_j$$

y dos morfismos especiales

$$\prod_{i \in \mathrm{Obj}(\mathbf{J})} D_i \underset{\psi}{\overset{\phi}{\rightrightarrows}} \prod_{\alpha \in \mathrm{Mor}(\mathbf{J})} D_j$$

que muestran cómo varía el producto con respecto de los morfismos en el diagrama. Concretamente, definimos ϕ y ψ tomando la composición de estos dos morfismos con las proyecciones π_α del producto $\prod_{\alpha \in \mathrm{Mor}(\mathbf{J})} D_j$, obtenemos:

$$\pi_\alpha \circ \phi = \phi_\pi = \pi_{\mathrm{cod}(\alpha)}$$
$$\pi_\alpha \circ \psi = \psi_\pi = D_\alpha \circ \pi_{\mathrm{dom}(\alpha)}$$

donde $\phi_\alpha : \prod_{\alpha \in \mathrm{Mor}(\mathbf{J})} D_j \longrightarrow D_i$ y $\psi_\alpha : \prod_{\alpha \in \mathrm{Mor}(\mathbf{J})} D_j \longrightarrow D_j$. Entonces obtenemos que $\pi_{\mathrm{dom}(\alpha)}$ y $\pi_{\mathrm{cod}(\alpha)}$ son dos proyecciones del producto $\prod_{i \in \mathrm{Obj}(\mathbf{J})} D_i$ dado un morfismo $\alpha : i \longrightarrow j$ concreto.

En segundo lugar, necesitamos obtener el subconjunto de objetos de $\prod_{i \in \mathrm{Obj}(\mathbf{J})} D_i$ en el que los morfismos sí conmuten. Para ello, tomamos el igualador (E, e_i) siguiente:

$$E \xrightarrow{\ e\ } \prod_i D_i \underset{\psi}{\overset{\phi}{\rightrightarrows}} \prod_{\alpha : i \to j} D_j$$

donde podemos formar las aristas de un cono cuyo vértice es E tomando las proyecciones del producto $\prod_i D_i$ con el morfismo e del igualador. Es decir, que $e_i = \pi_i \circ e$.

Para ver que (E, e) es un cono, tomamos cualquier otro morfismo $c : V \longrightarrow \prod_i D_i$, donde $c_i = \pi_i \circ c$ y donde denotamos $c = \langle c_i \rangle$. Entonces (C, c) es un cono si y solo si $\phi\langle c_i \rangle = \psi\langle c_i \rangle$. Esto se cumple si y solo si para todo α

$$\pi_\alpha \phi \langle c_i \rangle = \pi_\alpha \psi \langle c_i \rangle$$

No obstante, por cómo hemos definido ϕ_α y ψ_α previamente, obtenemos que

$$\pi_\alpha \phi \langle c_i \rangle = \phi_\alpha \langle c_i \rangle = \pi_{\mathrm{cod}(\alpha)} \langle c_i \rangle = c_j$$
$$\pi_\alpha \psi \langle c_i \rangle = \psi_\alpha \langle c_i \rangle = D_\alpha \circ \pi_{\mathrm{dom}(\alpha)} \langle c_i \rangle = D_\alpha \circ c_i$$

Luego $c_j = D_\alpha \circ c_i$ si y solo si (C, c) es un cono, tal como habíamos afirmado al principio, y del mismo modo lo es (E, e). Como además de ser un cono es un igualador, por la propiedad universal del mismo, para cualquier otro cono (C, c) existirá un morfismo único $u : C \longrightarrow E$ tal que $c = u \circ e$. Finalmente, por este hecho y la Nota 4.2, se deduce que (E, e) es el límite del diagrama pequeño D. □

Por supuesto, el dual de este teorema también se cumple:

Proposición 4.4. *Una categoría es finitamente cocompleta si y solo si tiene coproductos pequeños finitos y coigualadores.*

Nota 4.3. *Estas dos últimas proposiciones se cumplen también para categorías con límites con cualquier cardinalidad, siempre y cuando tenga también productos y igualadores con dicha cardinalidad [2, p. 105].*

Con estos conceptos establecidos, podemos ver que las construcciones universales presentadas en la sección anterior no son más que ejemplos particulares de conos en cierta categoría.

Ejemplo 4.8. Un producto de dos elementos en una categoría $\mathbf{C}$ no es más que el límite del diagrama $D : \mathbf{2} \longrightarrow \mathbf{C}$, es decir, el cono límite $(D_0 \times D_1, \pi)$ siguiente:

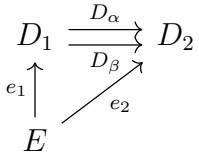
$$D_0 \xleftarrow{\ \pi_0\ } D_0 \times D_1 \xrightarrow{\ \pi_1\ } D_1$$

donde $\varprojlim_j D_j \cong D_1 \times D_2$.

Ejemplo 4.9. Un igualador en una categoría $\mathbf{C}$ no es más que el límite del diagrama $D : 1, 2 \longrightarrow \mathbf{C}$ siguiente:

$$D_1 \underset{D_\beta}{\overset{D_\alpha}{\rightrightarrows}} D_2$$

cuyo límite será el cono (E, e), con $e_1 : E \longrightarrow D_1$ y $e_2 : E \longrightarrow D_2$, siguiente:



en el que $D_\alpha \circ c_1 = D_\beta \circ c_1 = c_2$. Este límite (E, e) será pues el igualador para los morfismos D_α y D_β.

4.3.1. Functores continuos

Definir los límites da pie a estudiar un tipo concreto de functor, los functores continuos. Se puede ver fácilmente que este es un concepto reminiscente a la definición de función continua en análisis, pero además su estudio tiene aplicaciones en conceptos que veremos más adelante, como algunas propiedades los functores adjuntos.

Definición 4.13. Sea $F : \mathbf{C} \Longrightarrow \mathbf{D}$ un functor. Se dice que F es **continuo** o que preserva límites de tipo $\mathbf{J}$ si dado un diagrama $D : \mathbf{J} \Longrightarrow \mathbf{C}$, cuyo límite es $(\varprojlim D, p)$, con $p_i : \varprojlim D \longrightarrow D_i$, entonces el cono $(F \varprojlim D, Fp)$, con $Fp_i : F \varprojlim D \longrightarrow FD_i$ es un límite del diagrama $FD : \mathbf{J} \Longrightarrow \mathbf{D}$. Es decir,

$$F(\varprojlim D_i) \cong \varprojlim F(D_i)$$

Definición 4.14. Si F preserva los colímites de tipo $\mathbf{J}$, entonces se dice que F es **cocontinuo**.

De hecho, por lo visto en la Proposición 4.3, como todo (co)límite puede ser construido a partir de (co)productos (pequeños) y (co)igualadores en una categoría, un functor será (co)continuo si y solo si preserva los (co)productos y los (co)igualadores.

Paralelamente, podemos ver que los hom-functores son un tipo de functor que particularmente siempre preserva límites, es decir, es continuo.

Proposición 4.5. *Sea* $\mathbf{C}$ *una categoría pequeña cualquiera con productos, y sea* $X \in \mathbf{C}$. *Entonces el hom-functor*

$$Hom_{\mathbf{C}}(X, -) : \mathbf{C} \Longrightarrow \mathbf{Sets}$$

es continuo.

Demostración. Por la Proposición 4.3, decir que un functor preserva límites es lo mismo que decir que preserva productos e igualadores.

Así pues, en primer lugar, veamos que los productos se conservan. Es decir, que el siguiente isomorfismo existe:

$$Hom_{\mathbf{C}}(X, A \times B) \cong Hom_{\mathbf{C}}(X, A) \times Hom_{\mathbf{C}}(X, B).$$

Dado un diagrama de la forma

$$A \xleftarrow{\pi_1} P \xrightarrow{\pi_2} B$$

es un producto si y solo si para cualquier otro objeto $X \in \mathbf{C}$, existe el isomorfismo $\vartheta_X : Hom(X, P) \longrightarrow Hom(X, A) \times Hom(X, B)$. Este isomorfismo existe porque por la definición de producto, concretamente por la propiedad universal con el que lo definimos, dados $(x, y) \in Hom(X, A) \times Hom(X, B)$, entonces existe un único morfismo, ϑ_X, tal que si $z \in Hom(X, P)$, entonces $\vartheta_X(z) = (x, y)$. Luego, por definición, ϑ es biyectiva.

En segundo lugar, debemos ver que preserva igualadores. Dado el diagrama de un igualador en $\mathbf{C}$:

$$E \xrightarrow{e} A \underset{g}{\overset{f}{\rightrightarrows}} B,$$

consideramos el igualador a través de $Hom(X, -)$:

$$Hom(X, E) \xrightarrow{e^*} Hom(X, A) \underset{g^*}{\overset{f^*}{\rightrightarrows}} Hom(X, B).$$

Vamos a demostrar que este diagrama es un igualador en **Sets**. Sea $h : X \longrightarrow A \in Hom(X, A)$, tal que $f^* \circ h = g^* \circ h$. Entonces $f \circ h = g \circ h$. Siguiendo la propiedad universal del igualador en $\mathbf{C}$, existe un único morfismo $u : X \longrightarrow E$ tal que $e \circ u = h$. Esto significa que ese $u \in Hom(X, E)$, tal que $e^* \circ u = e \circ u = h$. Con todo esto, hemos probado que el morfismo $e^* : Hom(X, E) \longrightarrow Hom(X, A)$ es un igualador de los morfismos f^* y g^*. □

4.4. Functores en Haskell

Los functores en Haskell solo pueden operar dentro de la categoría **Hask**, luego en ese sentido, todo functor en Haskell es en realidad un endofunctor. Es por eso que los functores no son más que funciones que asignan tipos a tipos.

La clase `Functor` en Haskell se define de la siguiente manera:

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

Un tipo que sea instancia de la clase `Functor` es todo aquel que pueda implementar una función de manera que dada otra función `f :: a -> b`, devuelva `Ff :: f a -> f b`. En otras palabras, la función que devuelve `fmap`, `Ff`, se corresponde con el morfismo $F : \text{Obj}(\mathbf{C}) \longrightarrow \text{Obj}(\mathbf{C})$ de la Definición 4.1, siendo $\mathbf{C} = \mathbf{Hask}$.

Por otra parte, aunque las propiedades de los functores vistas en la Definición 4.1 no se establezcan explícitamente en la definición de la clase en Haskell, estas se corresponden con:

- $Fid_A = id_{FA}$:

  ```
  fmap id = id,
  ```

  siendo el primer `id` la identidad `a -> a`, y el segundo `id` la identidad `f a -> f a`.

- $F(g \circ f) = Fg \circ Ff$:

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

En la Sección A.3 se examina la función `map` definida de la siguiente forma:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Así pues, podemos ver cómo de hecho `[]` es un functor, cuya implementación de `fmap` es la función `map` anterior:

```
instance Functor [] where
  -- fmap :: (a -> b) -> [a] -> [b]
  fmap _ []    = []
  fmap (x:xs) = f x : fmap xs
```

Además, cumple las propiedades de los functores que hemos visto anteriormente:

- Identidad:

  ```
  fmap id []
      = (por definición de fmap)
      []
      = (por definición de id)
      id []

  fmap id (x:xs)
      = (por definición de fmap)
      x:(id xs)
  ```

```
    = (por definición de id)
    x:xs
    = (por definición de id)
    id x:xs
```

- Composición de funciones:

```
fmap (g . f) []
    = (por definición de fmap)
    []
    = (por definición de fmap)
    fmap g []
    = (por definición de fmap)
    fmap g (fmap f [])
    = (por definición de .)
    (fmap g . fmap f) []

fmap (g . f) (x:xs)
    = (por definición de fmap)
    (g . f) x : fmap (g . f) xs
    = (por definición de .)
    g (f x) : fmap g (fmap f xs)
    = (por definición de fmap)
    fmap g (f x) : fmap f xs
    = (por definición de fmap)
    fmap g (fmap f x:xs)
    = (por definición de .)
    (fmap g . fmap f) (x:xs)
```

Otros ejemplos básicos de functores incluyen el tipo `Maybe` o el tipo `(t,)` (siendo este último el tipo de las tuplas cuya primera componente es un valor de tipo `t`, y la segunda componente un valor de cualquier otro tipo que le pasemos). Vamos a ver detalladamente cómo este primero es un functor.

El tipo `Maybe` encapsula la noción de un valor opcional. Por ejemplo, un valor de tipo `Maybe Int` es o bien ese `Int`, o bien nada, o sea, `Nothing`. Se define de la siguiente manera:

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

Además, este tipo implementa `fmap` de la siguiente forma:

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

Comprobemos de nuevo que las propiedades de los functores se cumplen para `Maybe`:

- Identidad:

```
fmap id Nothing
    = (por definición de fmap)
    Nothing
    = (por definición de id)
    id Nothing

fmap id (Just a)
    = (por definición de fmap)
    Just (id a)
    = (por definición de id)
    Just a
    = (por definición de id)
    id (Just a)
```

- Composición de funciones:

```
fmap (g . f) Nothing
    = (por definición de fmap)
    Nothing
    = (por definición de fmap)
    fmap g Nothing
    = (por definición de fmap)
    fmap g (fmap f Nothing)
    = (por definición de .)
    (fmap g . fmap f) Nothing

fmap (g . f) (Just a)
    = (por definición de fmap)
    Just ((g . f) a)
    = (por definición de .)
    Just (g (f a))
    = (por definición de fmap)
    fmap g (Just (f a))
    = (por definición de fmap)
    fmap g (fmap f (Just a))
    = (por definición de .)
    (fmap g . fmap f) (Just a)
```

Observamos que los tipos que pueden ser instancia de la clase `Functor` son tipos parametrizados, como `Maybe`, `[]`, `(t,)`, etc., que toman un tipo `a`, y devuelven `Maybe a`,`[a]`, `(t, a)`, etc. Es por esto que podemos entender los functores en Haskell como "contenedores" de otros tipos.

4.5. Hom-functores en Haskell

Vamos a introducir el tipo `Reader`

```
type Reader a x = a -> x,
```

que es de hecho un functor, ya que podemos definir la función `fmap` acordemente:

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

Este functor representa un hom-functor covariante (es decir, functorial solo en `a`) en **Hask**.

Nota 4.4. *Si hubiéramos definido el hom-functor* `Reader` *de forma contravariante, sería de la siguiente manera:*

```
type Reader a x = x -> a
```

donde `contramap` *(análogo contravairnate de* `fmap`*) se definiría, tal y como se esperaba, como la precomposición con* `f`*:*

```
instance Contravariant (Reader a) where
    -- contramap :: (b -> a) -> Reader a -> Reader b
    contramap f h = h . f
```

Como mencionamos en la Sección 4.1.1, este tipo de functores serán muy importantes a la hora de entender el Lema de Yoneda y, en este caso, de las consecuencias del mismo en la programación funcional.

5 OBJETO EXPONENCIAL

La motivación detrás de la construcción universal del objeto exponencial es la de la generalización del conjunto de funciones de un conjunto a otro, normalmente expresado como $B^A = \{f : A \longrightarrow B\} = Hom_{\mathbf{Sets}}(A, B)$, con $A, B \in \mathbf{Sets}$. Vamos a estudiar en qué casos el objeto exponencial existe en una categoría dada, o dicho de otra forma, bajo qué condiciones en una categoría $\mathbf{C}$, dados dos elementos $A, B \in \mathbf{C}$, B^A es un objeto en ella.

5.1. Construcción

En el caso de **Sets**, dados los conjuntos A, B y C en **Sets**, consideramos la función

$$eval : A \times B^A \longrightarrow B$$

tal que asigna el par ordenado (a, g) a la imagen de a a través de g, es decir, $(a, g) \mapsto g(a) \in B$. Así pues,

$$eval(a, g) = g(a).$$

Por otra parte, dada una función $f : A \times B \longrightarrow C$, podemos fijar un $a \in A$ de tal forma que

$$f(a, y) : B \longrightarrow C$$

y por tanto $f(a, y) \in C^B$. Si variamos a en A, obtenemos la función

$$\tilde{f} : A \longrightarrow C^B$$

tal que $\tilde{f}(a)(b) = f(a, b)$. Al operador $\tilde{}$: $(A \times B \to C) \longrightarrow (A \to C^B)$ definido como $f \mapsto \tilde{f}$ se le llama traspuesta o "*currificación*" de f.

Con esto podemos ver que la función *eval* posee la siguiente propiedad universal: dada una función $f : A \times B \longrightarrow C$, existe una única función $\tilde{f} : A \longrightarrow C^B$ tal que.

$$eval \circ (\tilde{f} \times id_B) = f$$

Concretamente $eval(\tilde{f}(a), b) = f(a, b)$. A esta composición se la llama "*descurrificación*" de $\tilde{f}$.

$$\begin{array}{ccc} C^B & & C^B \times B \xrightarrow{eval} C \\ {\scriptstyle \tilde{f}}\uparrow & & {\scriptstyle \tilde{f}\times id_B}\uparrow \quad \nearrow_{f} \\ A & & A \times B \end{array}$$

Gracias a que la operación de trasposición es reversible mediante *eval*, podemos establecer el siguiente isomorfismo en **Sets**:

$$Hom_{\mathbf{Sets}}(A \times B, C) \cong Hom_{\mathbf{Sets}}(A, C^B).$$

Sin embargo, la función evaluación solo podrá existir en primer lugar si la categoría en la que estemos trabajando tiene productos. Así pues, vamos a establecer la definición formal del objeto exponencial:

Definición 5.1. Sea $\mathbf{C}$ una categoría con productos binarios (es decir, que para cada para de objetos en la categoría, existe la construcción expuesta en la Sección 3.2). Dados los objetos $B, C \in \mathbf{C}$, un exponencial de B a C es un objeto C^B junto con un morfismo *evaluación*

$$\epsilon : C^B \times B \longrightarrow C$$

tal que para cada $A \in \mathbf{C}$ y para cada morfismo

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

hay un único morfismo *traspuesto*

$$\tilde{f} : A \longrightarrow C^B$$

de forma que $\epsilon \circ (\tilde{f} \times id_B) = f$.

Además, dado un morfismo $g : A \longrightarrow C^B$ en $\mathbf{C}$, llamamos también *trasposición* a $\overline{g} := \epsilon \circ (g \times id_B)$. De esta forma, $\tilde{\overline{g}} = g$, y para cualquier morfismo $f : A \times B \longrightarrow C$, $\overline{\tilde{f}} = f$.

Como se ha mostrado antes en un caso concreto, la trasposición es una operación invertible, luego obtenemos el isomorfismo siguiente, esta vez en $\mathbf{C}$:

$$Hom_{\mathbf{C}}(A \times B, C) \cong Hom_{\mathbf{C}}(A, C^B).$$

5.2. Categorías cartesianas cerradas

Definición 5.2. Una categoría se dice que es cartesiana cerrada si tiene productos finitos y exponenciales.

Nota 5.1. *Se pueden encontrar otras definiciones en las que el requisito de tener productos finitos sea reemplezado por los dos siguientes:*

- *Que la categoría tenga productos binarios para todo $A, B \in \mathbf{C}$.*
- *Que la categoría tenga un objeto terminal.*

Estas dos definiciones son equivalentes ya que si queremos un producto arbitrario de elementos, necesitamos al menos un producto binario, que dará pie a un producto ternario, cuaternario, etc., junto con un producto vacío. El único elemento que cumple esto último, es decir, que $a \times b \cong a$, es el objeto terminal de la categoría en la que estemos: $b = 1$.

El propósito de los objetos exponenciales en las categorías cartesianas cerradas es el de establecer un objeto formado por la colección de morfismos entre dos objetos A y B de esa categoría dentro de la categoría en sí. Este objeto normalmente lo denotamos como $Hom_{\mathbf{C}}(A, B)$, que cuando existe en una categoría cartesiana cerrada, significa que es un objeto dentro de la categoría misma, y no necesariamente un elemento de una categoría distinta.

Además, las categorías cartesianas cerradas nos ofrecen una forma de modelizar el cálculo lambda [4, p. 55]. Esto significa que todo resultado de la Teoría de Categorías se puede trasladar a este, y así poder llegar a obtener una reducción todavía mayor de sus expresiones [4, p. 60].

A continuación vamos a ver que las exponenciación por un elemento fijo de una categoría cartesiana cerrada es de hecho un functor. Veámoslo.

Proposición 5.1. *Sea* $\mathbf{C}$ *una categoría cartesiana cerrada. Dado un elemento* $A \in \mathbf{C}$*, la exponenciación*

$$(-)^A : \mathbf{C} \Longrightarrow \mathbf{C}$$

es un functor.

Demostración. Dado un morfismo $\beta : B \longrightarrow C$, vamos a definir su exponenciación por A. En primer lugar, tomamos la traspuesta de la composición del morfismo β con la evaluación ϵ, es decir:

$$(\beta \circ \epsilon) : B^A \times A \to B \to C$$

donde le aplicamos la trasposición:

$$\widetilde{(\beta \circ \epsilon)} : B^A \longrightarrow C^A.$$

Con esto hemos obtenido

$$\beta^A : B^A \longrightarrow C^A,$$

que cumple el siguiente diagrama conmutativo:

$$\begin{array}{ccc} C^A & C^A \times A & \xrightarrow{\epsilon} C \\ {\scriptstyle \beta^A}\uparrow & {\scriptstyle \beta^A \times id_A}\uparrow & \uparrow{\scriptstyle \beta} \\ B^A & B^A \times A & \xrightarrow[\epsilon]{} B \end{array}$$

Una vez establecido β^A, veamos que cumple las propiedades de los functores. Por una parte, veamos si conserva la identidad. Es evidente que

$$(id_B)^A = 1_{B^A} : B^A \longrightarrow B^A.$$

Por otra parte, veamos que la exponenciación de la composición de morfismos es la composición de exponenciaciones.

Dado otro morfismo $\gamma : C \longrightarrow D$, tenemos que:

$$\begin{array}{ccc} D^A \times A & \xrightarrow{\epsilon} & D \\ {\scriptstyle \gamma^A \times id_A}\uparrow & & \uparrow{\scriptstyle \gamma} \\ C^A \times A & \xrightarrow{\epsilon} & C \\ {\scriptstyle \beta^A \times id_A}\uparrow & & \uparrow{\scriptstyle \beta} \\ B^A \times A & \xrightarrow[\epsilon]{} & B \end{array}$$

Tomando $(\gamma \circ \beta)$, el diagrama anterior queda simplificado de la siguiente manera:

$$\begin{array}{ccc} D^A \times A & \xrightarrow{\epsilon} & D \\ {\scriptstyle (\gamma\circ\beta)^A \times id_A}\uparrow & & \uparrow{\scriptstyle \gamma\circ\beta} \\ B^A \times A & \xrightarrow[\epsilon]{} & B \end{array}$$

Y por otra parte, usando el hecho de que $(\gamma^A \times id_A) \circ (\beta^A \times id_A) = (\gamma^A \circ \beta^A) \times id_A$, tenemos que:

$$\begin{array}{ccc} D^A \times A & \xrightarrow{\epsilon} & D \\ {\scriptstyle (\gamma^A\circ\beta^A)\times id_A}\uparrow & & \uparrow{\scriptstyle \gamma\circ\beta} \\ B^A \times A & \xrightarrow[\epsilon]{} & B \end{array}$$

Luego tenemos que $(\gamma \circ \beta)^A = \gamma^A \circ \beta^A$. □

Veamos ahora algunos ejemplos de categorías cartesianas cerradas.

Ejemplo 5.1. Uno de los ejemplos más comunes, a parte del de la categoría **Sets** visto al principio de esta sección, es el de la categoría **Poset**. En ella, dados dos posets P y Q, los morfismos $f : P \longrightarrow Q$ son funciones monótonas, y los elementos de su producto $P \times Q$ son los pares parcialmente ordenados tales que $(p, q) \leq (p', q')$ si y solo si $p \leq p'$ y $q \leq q'$.

Veamos ahora su objeto exponencial. Tomamos la colección de funciones monótonas de P a Q como:

$$Q^P = \{f : P \longrightarrow Q \mid f \text{ es monótona}\}$$

donde $f \leq g$ si y solo si $f(p) \leq g(p), \forall p \in P$.

Tomamos las funciones evaluación y trasposición como hemos definido anteriormente, es decir:

- Función evaluación:
$$\epsilon : Q^P \times P \longrightarrow Q.$$
- Función trasposición de una función $f : X \times P \longrightarrow Q$:
$$\tilde{f} : X \longrightarrow Q^p.$$

Si mostramos que todas ellas con monótonas, entonces $Q^P \in \mathbf{Posets}$.

Tomamos $(f,p) \leq (f',p') \in Q^P \times P$. En primer lugar:

$$\begin{aligned}\epsilon(f,p) &= f(p)\\ &\leq f(p')\\ &\leq f'(p')\\ &= \epsilon(f',p'),\end{aligned}$$

luego ϵ es monótona. En segundo lugar, tomamos la función monótona $f : X \times P \longrightarrow Q$ con $x \leq x'$. Vamos a ver que $\tilde{f}(x) \leq \tilde{f}(x')$, y para ello mostramos que $\tilde{f}(x)(p) \leq \tilde{f}(x')(p)$ para todo $p \in P$.

Como $\tilde{f}(x)(p) = f(x,p) \leq f(x',p) = \tilde{f}(x')(p)$, entonces

$$\tilde{f}(x)(p) \leq \tilde{f}(x')(p),$$

luego $\tilde{f}(x)$ es monótona en Q^P.

5.3. Objeto exponencial en Haskell

Recordemos la definición de objeto exponencial, esta vez limitada a la categoría **Hask**. Dados dos tipos `b, c` $\in$ **Hask**, un exponencial de `b` a `c` es un morfismo `b -> c` junto con un morfismo evaluación

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

tal que para otro tipo `a` y para cada función

```
f :: (a, b) -> c,
```

tengamos una función única llamada *traspuesta*

```
traspuesta_f :: a -> (b -> c),
```

de forma que `eval (traspuesta_f, id) = f`.

La única función en **Hask** que traspone a `f` es la función `curry`:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)
```

Luego `curry f = traspuesta_f`.

De la misma forma, dada otra función

```
g :: a -> (b -> c),
```

podemos obtener el morfismo único

```
traspuesta_g :: (a, b) -> c
```

a través de la función `uncurry`, definida de la siguiente manera:

```
uncurry ::  (a -> b -> c) -> ((a, b) -> c)
uncurry f (x, y) = f x y
```

Del mismo modo que antes, obtenemos que `uncurry g = traspuesta_g`, o que `traspuesta_g = eval (g, id)`.

Aunque podríamos haber afirmado de primeras que **Hask** (platónico) es una categoría cartesiana cerrada por la existencia de productos binarios y objeto terminal en ella, el haber visto la construcción del objeto exponencial en ella nos ha permitido mostrar dos funciones muy importantes en la programación funcional: `curry` y `uncurry`. Para ver los usos de estas funciones con más detalle, véase el Apéndice A.

6 NATURALIDAD

El concepto de naturalidad trata de formalizar el hecho de que existen ciertos morfismos entre estructuras que aparecen de forma intrínseca, "natural", a ellas, sin la elección arbitraria de ningún otro elemento. Esta idea, consecuentemente, se corresponde con el hecho de que cierto diagrama entre dichas estructuras conmute. Veamos un ejemplo previo a la definición formal de transformación natural.

Ejemplo 6.1. (De no naturalidad) Tomemos un espacio vectorial de dimensión finita V sobre el cuerpo $\mathbb{R}$ y su correspondiente espacio dual V^* de funciones lineales $f : V \longrightarrow \mathbb{R}$.

Para demostrar que hay un isomorfismo entre V y V^*, es necesario en primer lugar tomar una base $B = \{v_1, \ldots, v_n\}$ (arbitrariamente elegida) de V. Para crear el isomorfismo usamos las funciones $v_i^* : V \longrightarrow \mathbb{R}$ tal que $v_i^*(v_j) = 1$ si $i = j$ y $v_i^*(v_j) = 0$ si $i \neq j$. Es sencillo comprobar pues que una base de V^* es $B^* = \{v_1^*, \ldots, v_n^*\}$, y la función lineal $\varphi : V \longrightarrow V^*$ definida como $\varphi_B(v_i) = v_i^*$ es un isomorfismo.

La elección de base que hemos hecho en el ejemplo anterior ha sido arbitraria. Ninguna base para generar los isomorfismos es más o menos "natural" que otra. Sin embargo, considerando ahora además el espacio doble dual V^{**} de V, podemos ver que se puede formar un isomorfismo entre estas dos estructuras de forma en que no se haga ninguna elección arbitraria.

Ejemplo 6.2. Tomemos el espacio vectorial doble dual V^{**} de V, esto es, el espacio vectorial de aplicaciones lineales $g : V^* \longrightarrow \mathbb{R}$. Sin necesidad de elegir ninguna base, para construir el isomorfismo entre V y su doble dual, definimos $\psi_V : V \longrightarrow V^{**}$ tal que $v \mapsto \psi_V(v) : V^* \to \mathbb{R}$, que asigna una función lineal $f : V \longrightarrow \mathbb{R}$ a su evaluación mediante $v \in V$, es decir, a $f(v)$. En otras palabras $\psi_V(v)(f) = f(v) \in R$, y se demuestra que es un isomorfismo utilizando el hecho de que V es de dimensión finita.

Es evidente que la construcción de este isomorfismo ψ en concreto es independiente de cualquier base. Podremos decir, consecuentemente, que el isomorfismo entre V y V^* no es natural. Sin embargo, sí lo es en el caso del isomorfismo entre V y V^{**}.

Habiendo visto este ejemplo a modo de motivación del concepto de naturalidad, veamos ahora la definición formal de transformación natural.

6.1. Transformaciones naturales

Definición 6.1. Sea **C** y **D** dos categorías, y sean F y G dos functores paralelos, $F, G : \mathbf{C} \Longrightarrow \mathbf{D}$.

Una transformación natural $\lambda : F \Longrightarrow G$ es una familia de morfismos en **D**, tal que para todo elemento $C \in \mathbf{C}$

$$\lambda_C : FC \longrightarrow GC,$$

(a los que llamamos componentes de la transformacional natural) y dado cualquier morfismo $f : C \longrightarrow C'$ en **C**, se tiene que el siguiente diagrama conmuta:

$$\begin{array}{ccc} C & & FC \xrightarrow{\lambda_C} GC \\ {\scriptstyle f}\downarrow & & {\scriptstyle Ff}\downarrow \qquad\quad \downarrow{\scriptstyle Gf} \\ C' & & FC' \xrightarrow[\lambda_{C'}]{} GC' \end{array}$$

Es decir, que $Gf \circ \lambda_C = \lambda_{C'} \circ Ff$. A este hecho le llamamos **condición de naturalidad**. Cuando se tiene un elemento $C \in \mathbf{C}$, un morfismo de dicho elemento a cualquier otro arbitrario, y se cumple además la condición de naturalidad, entonces se dice que λ *es natural en* C.

Asimismo, una transformación natural también puede ser considerada como un **morfismo entre functores** en una categoría **D**.

Habiendo establecido cómo conmuta el diagrama de una transformación natural, vamos a volver al Ejemplo 6.2 de nuevo. Podemos interpretar la naturalidad del isomorfismo ψ como que dado dos espacios vectoriales de dimensión finita V y W cualesquiera y sus correspondientes dobles duales V^{**} y W^{**}, da el mismo resultado evaluar mediante ψ_V el doble dual de una función lineal cualquiera que preceder la evaluación ψ_W con dicha función lineal.

Veamos más ejemplos.

Ejemplo 6.3. Sea **CRng** la categoría de los anillos conmutativos y sea **Grp** la categoría de los grupos. Sean también los functores paralelos $G, U : \mathbf{CRng} \Longrightarrow \mathbf{Grp}$ definidos de la siguiente manera:

- G: Toma un elemento $R \in \mathbf{CRng}$ y lo envía al grupo general lineal $GL_n(R)$, y toma un homomorfismo de anillos $f : R \longrightarrow S$ y lo envía a la aplicación f_e, que está definida de tal modo que dada una matriz $M \in GL_n(R)$ actúa elemento a elemento sobre todos los elementos de dicha matriz.

- U: Toma un anillo $R \in \mathbf{CRng}$ y lo envía al grupo R^* de unidades (elementos invertibles respecto al operador $*$ del anillo), y toma un homomorfismo de anillos $f : R \longrightarrow S$ y lo envía al homomorfismo de grupos $f_u : R^* \longrightarrow S^*$.

Con esto disponemos de un morfismo en **CRng** que mediante G y U se aplica a los elementos de una matriz y a la unidades de un anillo respectivamente. Así pues, una operación que cumpla la condición de naturalidad respecto a los morfismos f_e y f_u introducidos anteriormente, es decir

$$\lambda_s \circ f_e = f_u \circ \lambda_R,$$

es la del determinante. Vamos a verlo particularmente para $n = 2$, a pesar de que se cumpla para todo $n \in \mathbb{N}$.

Sea el anillo R y el homomorfismo de anillos $f : R \longrightarrow S$. El determinante en un anillo cualquiera A se define como $det_A : GL_2(A) \longrightarrow A^*$ tal que dada una matriz $M = [[a, b], [c, d]] \in GL_2(A)$, $det_A(M) = ad - bc \in A^*$.

Por una parte, $(det_S \circ f_e)(M) = det_s[[f(a), f(b)], [f(c), f(d)]] = f(a)f(d) - f(b)f(c) = f(ad - bc)$ (por ser f un homomorfismo de anillos).

Por otra parte, $(f_u \circ det_R)(M) = f_u(ad - bc) = f(ad - bc)$.

Por tanto, podemos concluir que

$$det_S \circ f_e = f_u \circ det_R,$$

es decir, que el conjunto $\{det_R \mid R \in \mathbf{CRng}\}$ es natural en R y que el siguiente diagrama conmuta:

$$\begin{array}{ccccc} R & & GL_n(R) & \xrightarrow{det_R} & R^* \\ \downarrow{\scriptstyle f} & & {\scriptstyle Gf}\downarrow & & \downarrow{\scriptstyle Uf} \\ S & & GL_n(S) & \xrightarrow{det_S} & S^* \end{array}$$

6.2. Isomorfismos naturales

El primer ejemplo con el que se ha presentado esta sección ha sido uno que presentaba una transformación natural, más concretamente, un isomorfismo natural. Una vez vista la definición más general de transformación natural, veamos el caso concreto en el que dicha transformación es de hecho un isomorfismo.

Definición 6.2. Sean **C** y **D** dos categorías y F y G dos functores entre ellas. Dada una transformación natural $\lambda : F \longrightarrow D$ donde cada componente $\lambda_C : FC \longrightarrow GC$ es un isomorfismo, tenemos que la condición de naturalidad es:

$$Gf \circ \lambda_C = \lambda_{C'} \circ Ff.$$

Además, usando el hecho de que cada componente es un isomorfismo, tenemos que

$$\lambda_{C'}^{-1} \circ (Gf \circ \lambda_C) \circ \lambda_C^{-1} = \lambda_{C'}^{-1} \circ (\lambda_{C'} \circ Ff) \circ \lambda_C^{-1},$$

lo cual implica que

$$\lambda_{C'}^{-1} \circ Gf = Ff \circ \lambda_C^{-1},$$

luego la familia de morfismos $\lambda^{-1} = \{\lambda_C^{-1} \mid C \in \mathbf{C}\}$ cumple también la condición de naturalidad.

Proposición 6.1. *Sean* $\mathbf{C}$ *y* $\mathbf{D}$ *dos categorías y* F *y* G *dos functores entre ellas. Sea* $\lambda : F \longrightarrow G$ *una transformación natural. Los siguientes hechos son equivalentes:*

- *Cada componente de* λ *es un isomorfismo.*
- *Hay una transformación natural* $\mu : G \longrightarrow F$ *tal que*

$$\lambda \circ \mu = 1 \ y \ \mu \circ \lambda = 1$$

donde 1 *es la familia de morfismos identidad.*

Demostración. Inmediata por lo mostrado en la Definición 6.2. □

Ejemplo 6.4. El isomorfismo natural de $Hom_{\mathbf{Grp}}(F(1), G) \cong U(G)$. Recordemos que un elemento terminal 1 en la categoría **Sets** era un conjunto de un solo elemento. Dicho elemento terminal puede ser el generador de un grupo libre, es decir, el grupo $F(1)$. El hom-functor $Hom_{\mathbf{Grp}}(F(1), -) : \mathbf{Grp} \Longrightarrow \mathbf{Sets}$ es isomorfo naturalmente isomorfo al functor olvidadizo $U : \mathbf{Grp} \Longrightarrow \mathbf{Sets}$.

Esto significa que para cualquier $G \in \mathbf{Grp}$, se tiene que

$$Hom_{\mathbf{Grp}}(F(1), G) \cong U(G),$$

y que para cualquier homomorfismo de grupos $f : G \longrightarrow H$, el siguiente diagrama conmuta:

$$\begin{array}{ccc} G & Hom_{\mathbf{Grp}}(F(1), G) & \xrightarrow{\cong} U(G) \\ \downarrow f & \downarrow Hom_{\mathbf{Grp}}(F(1), f) & \downarrow Uf \\ H & Hom_{\mathbf{Grp}}(F(1), H) & \xrightarrow{\cong} U(h) \end{array}$$

En otras palabras, que este isomorfismo es natural en $G \in \mathbf{Grp}$.

Otro ejemplo importante de isomorfismo natural es el de la asociatividad del producto en un categoría. Para ello veremos que existe un isomorfismo natural entre las correspondientes construcciones.

Ejemplo 6.5. Debemos probar que el siguiente isomorfismo

$$\lambda_A : (A \times B) \times C \cong A \times (B \times C)$$

cumple la condición de naturalidad y es de hecho un isomorfismo natural.

Sea $f : A \longrightarrow A'$ un morfismo en $\mathbf{C}$. Entonces tenemos el siguiente diagrama conmutativo:

$$\begin{array}{ccc} A & (A \times B) \times C & \xrightarrow{\lambda_A} A \times (B \times C) \\ f \downarrow & \downarrow & \downarrow \\ A' & (A' \times B) \times C & \xrightarrow[\lambda'_A]{} A' \times (B \times C) \end{array}$$

Es evidente que se cumple la condición de naturalidad, donde además

$$(- \times B) \times C \cong - \times (B \times C)$$

para functores $\mathbf{C} \Longrightarrow \mathbf{C}$. Luego este isomorfismo es natural en el elemento $A \in \mathbf{C}$.

Siguiendo un procedimiento similar, se puede demostrar también que

$$(- \times \star) \times * \cong - \times (\star \times *).$$

6.3. Categorías de functores

Una vez definidas las transformaciones naturales como morfismos entre functores, vamos a ver que es posible crear una categoría donde sus objetos son functores y sus morfismos son transformaciones naturales entre ellos.

Definición 6.3. Sea $\mathbf{C}$ una categoría pequeña. Una categoría de functores $\mathbf{Fun}(\mathbf{C}, \mathbf{D})$ (también denotada como $\mathbf{D}^{\mathbf{C}}$) tiene como objetos los functores (covariantes) $F : \mathbf{C} \Longrightarrow \mathbf{D}$ y $G : \mathbf{C} \Longrightarrow \mathbf{D}$ y como morfismos las transformaciones naturales $\lambda : F \longrightarrow D$. Cumple los axiomas de las categorías:

- Identidad: para cada objeto F, la transformación natural id_F tiene como componentes
$$(id_F)_C = id_{FC} : FC \longrightarrow FC, \ \forall C \in \mathbf{C}$$
- Composición: para cada par de morfismos en $\mathbf{Fun}(\mathbf{C}, \mathbf{D})$, es decir, transformaciones naturales, λ y μ que se puedan componer tal que $F \xrightarrow{\lambda} G \xrightarrow{\mu} H$, las componentes de dicha composición de transformaciones naturales son
$$(\mu \circ \lambda)_C = \mu_C \circ \lambda_C$$

Nota 6.1. *Es necesario que* $\mathbf{C}$ *sea una categoría pequeña. Esto se debe a que la colección de transformaciones naturales entre dos functores no es necesariamente un conjunto. Para ello, hacemos que la colección de objetos y de morfismos de* $\mathbf{C}$ *formen un conjunto.*

Con esta definición podemos ver que, de hecho, los isomorfismos naturales no son más que isomorfismos en la categoría de functores.

Asimismo, es importante notar que el exponencial $\mathbf{D}^{\mathbf{C}} = \mathbf{Fun}(\mathbf{C}, \mathbf{D})$, que como hemos visto tiene estructura de categoría y que por tanto pertenece a $\mathbf{Cat}$, solo existirá si la categoría $\mathbf{Cat}$ de todas las categorías pequeñas es cartesiana cerrada. Vamos a verlo.

Proposición 6.2. $\mathbf{Cat}$ *es una categoría cartesiana cerrada con exponencial* $\mathbf{D}^{\mathbf{C}}$ *dadas* $\mathbf{C}, \mathbf{D} \in \mathbf{Cat}$.

Demostración. Véase [2, p. 161]. □

6.4. Equivalencia

Una condición todavía más fuerte que el isomorfismo natural que nos puede llevar al concepto de que dos categorías son "la misma" es el de la *equivalencia* natural entre categorías.

Definición 6.4. Sean $\mathbf{C}$ y $\mathbf{D}$ dos categorías cualesquiera. Se dice que $\mathbf{C}$ y $\mathbf{D}$ son equivalentes (o naturalmente equivalentes) si existen un par de functores $F : \mathbf{C} \Longrightarrow \mathbf{D}$ y $G : \mathbf{D} \Longrightarrow \mathbf{C}$ tales que

$$F \circ G \cong id_{\mathbf{D}}$$
$$G \circ F \cong id_{\mathbf{C}}$$

Esto significa que las composiciones de ambos functores sean naturalmente isomorfas a los correspondientes functores identidad, o que F y G sean objetos isomorfos en la categoría $\mathbf{Fun}(\mathbf{C}, \mathbf{D})$.

Es por este motivo que los functores que cumplan estos requisitos serán *completamente fieles*.

6.5. Functores representables

El concepto que vamos a ver a continuación será fundamental para entender el lema que demostraremos en la sección siguiente. Encapsula la noción de que un objeto es "el representante" de cierto functor. O dicho de otra manera, que para crear un functor con ciertas características particulares que veremos a continuación, solo es necesario dar cierto objeto en la categoría.

Definición 6.5. Sea $\mathbf{C}$ una categoría. Un functor $F : \mathbf{C} \Longrightarrow \mathbf{Sets}$ se dice que es representable si es naturalmente isomorfo al hom-functor covariante $Hom_{\mathbf{C}}(A, -)$ para algún objeto $A \in \mathbf{C}$.

Asimismo, también existe la versión dual de esta definición: se dice que functor $F : \mathbf{C}^{op} \Longrightarrow \mathbf{Sets}$ es un functor contravariante representable si es naturalmente isomorfo al hom-functor contravariante $Hom_{\mathbf{C}}(-, A)$, con $A \in \mathbf{C}$.

Entonces, entendemos que F es un functor representable por el objeto A de la categoría $\mathbf{C}$ porque se puede establecer un isomorfismo natural entre F y el hom-functor (covariante) producido por A.

Un ejemplo de functor representable se puede ver en el Ejemplo 6.4.

6.6. Lema de Yoneda

El lema de Yoneda es uno de los resultados más importantes de la Teoría de Categorías y se puede entender como una generalización del Teorema de Cayley, en el que un grupo cualquiera se puede representar como un grupo de permutaciones

de un conjunto. El lema de Yoneda, por otra parte, nos muestra cómo una categoría cualquiera es equivalente a una formada por functores representables y los morfismos que los unen (transformaciones naturales).

Asimismo, también veremos que una consecuencia muy importante de este lema es el hecho de que podemos construir una inmersión (es decir, un functor fiel y completo) de una categoría $\mathbf{C}$ a la categoría de functores $\mathbf{Fun}(\mathbf{C}, \mathbf{Set})$. En otras palabras, "inyectar" mediante un functor una copia uno-a-uno de los objetos de $\mathbf{C}$ en los objetos de la categoría $\mathbf{Fun}(\mathbf{C}, \mathbf{Set})$ y, posteriormente, usando el hecho de que dicho functor sea una inmersión, "inyectar" los morfismos entre los objetos de $\mathbf{C}$ para que coincidan uno-a-uno con los morfismos de $\mathbf{Fun}(\mathbf{C}, \mathbf{Set})$. Esto significará que de cualquier categoría siempre habrá un isomorfismo canónico, natural, hacia la categoría de conjuntos y funciones entre ellos.

Antes de empezar, vamos a ver una serie de definiciones y notas que nos serán útiles a lo largo del enunciado y demostración del Lema de Yoneda.

Nota 6.2. *A lo largo de toda esta sección se asumirá que todas las categorías con las que trabajemos son localmente pequeñas. De este modo, los hom-sets y hom-functores producidos con elementos y morfismos de dichas categorías serán conjuntos y no habrá problema a la hora de trabajar con ellos.*

Definición 6.6. Sea $\mathbf{C}$ una categoría y sean $A, B \in \mathbf{C}$. Se define

$$Nat(Hom_{\mathbf{C}}(A, -), Hom_{\mathbf{C}}(B, -))$$

como el conjunto de transformaciones naturales entre los hom-functores (covariantes) $Hom_{\mathbf{C}}(A, -)$ y $Hom_{\mathbf{C}}(B, -)$. Es decir, un hom-set en $\mathbf{Fun}(\mathbf{C}, \mathbf{Sets})$.

6.6.1. Lema de Yoneda

Previamente a la demostración completa del Lema de Yoneda, vamos a ver una serie de proposiciones que nos ayudarán a entender las nociones básicas a partir de las cuales construiremos la demostración completa y generalizada de dicho lema.

En primer lugar, debemos ver que cada morfismo $f : B \longrightarrow A$ en una categoría $\mathbf{C}$ genera una transformación natural del functor $Hom_{\mathbf{C}}(A, -)$ al functor $Hom_{\mathbf{C}}(B, -)$.

Proposición 6.3. *Sean $A, B \in \mathbf{C}$ dos objetos cualquiera, y $Hom_{\mathbf{C}}(A, -)$ y $Hom_{\mathbf{C}}(B, -)$ los hom-functores representables de dichos dos objetos. Entonces, dado un morfismo $f : B \longrightarrow A$, existe la transformación natural*

$$Hom_{\mathbf{C}}(f, -) : Hom_{\mathbf{C}}(A, -) \longrightarrow Hom_{\mathbf{C}}(B, -)$$

correspondiente de tal modo que para cada $Z \in \mathbf{C}$, la componente $Hom_{\mathbf{C}}(f, -)_Z$ envía un morfismo $k : A \longrightarrow Z$ al morfismo $k \circ f : B \longrightarrow Z$.

Demostración. Se demuestra directamente a partir de la Definición 6.2 de componente de una transformación natural y la Definición 4.5 de hom-functor. □

En segundo lugar, debemos ver que lo contrario se cumple: cada transformación natural de $Hom_{\mathbf{C}}(A,-)$ a $Hom_{\mathbf{C}}(B,-)$ se corresponde con un único morfismo $f : B \longrightarrow A$ en $\mathbf{C}$.

Proposición 6.4. *Sean los hom-functores $Hom_{\mathbf{C}}(A,-)$ y $Hom_{\mathbf{C}}(B,-)$ para los objetos $A, B \in \mathbf{C}$. Entonces si existe la transformación natural $\alpha : Hom_{\mathbf{C}}(A,-) \longrightarrow Hom_{\mathbf{C}}(B,-)$ es porque existe un único morfismo $f : B \longrightarrow A$ tal que $\alpha = Hom_{\mathbf{C}}(f,-)$.*

Demostración. Sea $g : A \longrightarrow Z$. Como α es una transformación natural, el siguiente diagrama conmuta:

$$\begin{array}{ccc} Hom_{\mathbf{C}}(A,A) & \xrightarrow{Hom_{\mathbf{C}}(A,g)} & Hom_{\mathbf{C}}(A,Z) \\ {\scriptstyle \alpha_A}\downarrow & & \downarrow{\scriptstyle \alpha_Z} \\ Hom_{\mathbf{C}}(B,A) & \xrightarrow[Hom_{\mathbf{C}}(B,g)]{} & Hom_{\mathbf{C}}(B,Z) \end{array}$$

Sabemos que $Hom_{\mathbf{C}}(A,A)$ contiene al menos a id_A, luego por la conmutatividad del diagrama, tendremos que

$$\alpha_z \circ g = g \circ \alpha_A(id_A),$$

y mediante la definición utilizada en el teorema anterior,

$$g \circ \alpha_A(id_A) = Hom_{\mathbf{C}}(\alpha_A(id_A),-)_Z \circ g.$$

Asimismo, como el diagrama conmuta para todo $Z \in \mathbf{C}$ y para todo $g : A \longrightarrow Z$, tenemos que

$$\alpha = Hom_{\mathbf{C}}(\alpha_A(id_A),-).$$

Luego si tenemos $f : B \longrightarrow A$, entonces f existe y se define como $f := \alpha_A(id_A)$.

Finalmente, si tenemos $f, f' : B \longrightarrow A$ tales que $\alpha = Hom_{\mathbf{C}}(f,-) = Hom_{\mathbf{C}}(f',-)$, se cumplirá que

$$f = Hom_{\mathbf{C}}(f,-)_A(id_A) = Hom_{\mathbf{C}}(f',-)_A(id_A) = f',$$

porque para cualquier $h : B \longrightarrow A$, el hom-functor $Hom_{\mathbf{C}}(h,-)_A$ mandará cualquier $j : A \longrightarrow A$ a $j \circ h : B \longrightarrow A$, luego, en particular, $id_A \circ h = h$. Por tanto, f es única. □

Con las Proposiciones 6.3 y 6.4 hemos obtenido dos conjuntos de funciones $\mathcal{X}_{A,B}$ y $\mathcal{E}_{A,B}$ (indexadas por los objetos A y B de $\mathbf{C}$) definidas de modo que

- $\mathcal{X}_{A,B} : Hom_{\mathbf{C}}(B,A) \longrightarrow Nat(Hom_{\mathbf{C}}(A,-), Hom_{\mathbf{C}}(B,-))$ por la Proposición 6.3.
- $\mathcal{E}_{A,B} : Nat(Hom_{\mathbf{C}}(A,-), Hom_{\mathbf{C}}(B,-)) \longrightarrow Hom_{\mathbf{C}}(B,A)$ por la Proposición 6.4.

Comprobamos ahora que de hecho una es inversa de la otra.

Por un lado, sea $f : B \longrightarrow A \in Hom_{\mathbf{C}}(B, A)$. Entonces

$$(\mathcal{E}_{A,B} \circ \mathcal{X}_{A,B})(f) = \mathcal{E}_{A,B}(Hom_{\mathbf{C}}(f, -)) = (Hom_{\mathbf{C}}(f, -)_A(id_A) = f.$$

Como f es un morfismo arbitrario, se tiene que $\mathcal{E}_{A,B} \circ \mathcal{X}_{A,B} = id$.

Por otro lado, sea la transformación natural $\alpha : Hom_{\mathbf{C}}(A, -) \longrightarrow Hom_{\mathbf{C}}(B, -)$. Entonces

$$(\mathcal{X}_{A,B} \circ \mathcal{E}_{A,B})(\alpha) = \mathcal{X}_{A,B}(\alpha_A(id_A)) = Hom_{\mathbf{C}}(\alpha_A(id_A), -) = \alpha,$$

por la Proposición 6.4. De nuevo, como la elección de α ha sido arbitraria, tenemos que $\mathcal{X}_{A,B} \circ \mathcal{E}_{A,B} = id$.

Luego $\mathcal{X}_{A,B}$ y $\mathcal{E}_{A,B}$ son funciones isomorfas. Este hecho nos permite concluir finalmente que

$$Nat(Hom_{\mathbf{C}}(A, -), Hom_{\mathbf{C}}(B, -)) \cong Hom_{\mathbf{C}}(B, A).$$

Los pasos para generalizar el resultado anterior son los siguientes:

1. En ningún momento hemos utilizado el hecho de que $Hom_{\mathbf{C}}(B, -) : \mathbf{C} \Longrightarrow \mathbf{Sets}$ sea un hom-functor, luego podemos sustituirlo por cualquier otro functor (no necesariamente representable por B) con el mismo dominio y codominio, es decir, $F : \mathbf{C} \Longrightarrow \mathbf{Sets}$. Esto se verá en la Proposición 6.5.

2. Una vez hayamos probado lo anterior, es decir, el isomorfismo $Nat(Hom_{\mathbf{C}}(A, -), F) \cong FA$, deberemos probar rigurosamente que este es también natural, tanto en el functor $F \in \mathbf{Fun}(\mathbf{C}, \mathbf{Sets})$ como en el objeto $A \in \mathbf{C}$. Esto se verá en el Lema 6.1.

Proposición 6.5. *Sea* $\mathbf{C}$ *una categoría localmente pequeña. Sea* $A \in \mathbf{C}$ *y sea el functor* $F : \mathbf{C} \Longrightarrow \mathbf{Sets}$. *Entonces*

$$Nat(Hom_{\mathbf{C}}(A, -), F) \cong FA.$$

Demostración. En primer lugar, vamos a generalizar la función $\mathcal{X}_{A,B}$ definida anteriormente de la siguiente forma: Sea $\mathcal{X}_{A,F}$ la transformación natural que envía $f \in FA$ a la transformación natural

$$\chi = \mathcal{X}_{A,F} f : Hom_{\mathbf{C}}(A, -) \longrightarrow F.$$

Esta transformación natural la definimos requiriendo que cada uno de sus componentes mediante un $Z \in \mathbf{C}$ cualquiera sea la aplicación que toma un morfismo $g : A \longrightarrow Z$ cualquiera y lo envíe a $Fg(f)$.

Vamos a ver que χ es en efecto una transformación natural comprobando si el siguiente diagrama conmuta:

Sea $u : Z \longrightarrow Z'$. Entonces el diagrama

$$\begin{array}{ccc} Hom_{\mathbf{C}}(A,Z) & \xrightarrow{Hom_{\mathbf{C}}(A,u)} & Hom_{\mathbf{C}}(A,Z') \\ {\scriptstyle \chi_z}\downarrow & & \downarrow{\scriptstyle \chi_{z'}} \\ FZ & \xrightarrow[Fu]{} & FZ' \end{array}$$

conmuta porque si $j : A \longrightarrow Z \in Hom_{\mathbf{C}}(A, Z)$, entonces $F(u \circ j)(f) = Fu \circ Fj(f)$. Como F es de hecho un functor, entonces la igualdad anterior se cumple siempre.

En segundo lugar, vamos a generalizar la función $\mathcal{E}_{A,B}$ de la siguiente forma: $\mathcal{E}_{A,F}$ envía una transformación natural $\alpha : Hom_{\mathbf{C}}(A, -) \longrightarrow F$ al elemento $\alpha_A(id_A)$.

Finalmente, vamos a ver que estas dos funciones, $\mathcal{X}_{A,F}$ y $\mathcal{E}_{A,F}$ son inversas la una de la otra.

- Sea $f \in FA$. Entonces

$$(\mathcal{E}_{A,F} \circ \mathcal{X}_{A,F})(f) = \mathcal{E}_{A,F} \circ \chi = \chi_A(id_A) = Fid_A(f) = id_{FA}(f),$$

  luego $\mathcal{E}_{A,F} \circ \mathcal{X}_{A,F} = id$.

- Sea $\alpha : Hom_{\mathbf{C}}(A, -) \longrightarrow F$. Entonces

$$(\mathcal{X}_{A,F} \circ \mathcal{E}_{A,F})(\alpha) = \mathcal{X}_{A,F}(\alpha_A(id_A)).$$

  Por cómo se ha definido $\mathcal{X}_{A,F}$, cada componente de $\mathcal{X}_{A,F}(\alpha_A(id_A))$ dado un $Z \in \mathbf{C}$ enviará un morfismo $g : A \longrightarrow Z$ a $Fg(\alpha_A(id_A))$. Además, como α es una transformación natural, entonces el siguiente diagrama conmutará:

$$\begin{array}{ccc} Hom_{\mathbf{C}}(A,A) & \xrightarrow{Hom_{\mathbf{C}}(A,g)} & Hom_{\mathbf{C}}(A,Z) \\ {\scriptstyle \alpha_A}\downarrow & & \downarrow{\scriptstyle \alpha_Z} \\ FA & \xrightarrow[Fg]{} & FZ \end{array}$$

  Entonces, dado el morfismo $id_A \in Hom_{\mathbf{C}}(A, A)$, tendremos que

$$\alpha_Z(Hom_{\mathbf{C}}(A, g)(id_A)) = \alpha_Z(g) = Fg(\alpha_A(id_A)).$$

  Con esto podemos concluir que cada componente de

$$(\mathcal{X}_{A,F} \circ \mathcal{E}_{A,F})(\alpha) = \mathcal{X}_{A,F}(\alpha_A(id_A))$$

  actúa de la misma manera que cualquier componente de α, luego

$$(\mathcal{X}_{A,F} \circ \mathcal{E}_{A,F})(\alpha) = \alpha.$$

Como la elección de α ha sido arbitraria, entonces

$$\mathcal{X}_{A,F} \circ \mathcal{E}_{A,F} = id.$$

□

Finalmente, solo nos queda probar la naturalidad de A y de F en el isomorfismo que constituye el lema de Yoneda.

Lema 6.1 (**Lema de Yoneda**). *Sea* $\mathbf{C}$ *una categoría localmente pequeña. Sea* $A \in \mathbf{C}$ *y sea el functor covariante* $F : \mathbf{C} \Longrightarrow \mathbf{Sets}$. *Entonces*

$$Nat(Hom_{\mathbf{C}}(A, -), F) \cong FA,$$

naturalmente tanto en A *como en* F.

Demostración. En la proposición anterior hemos demostrado que dicho isomorfismo existe. Veamos primero la naturalidad en A:

Debemos probar que los functores $N = nat(-, F) \circ \chi$ y F son isomorfos.

Por cómo hemos definido N, vemos que este envía objetos de $\mathbf{C}$ al conjunto $Nat(Hom_{\mathbf{C}}(A, -), F)$, y morfismos $f : A \longrightarrow B$ de $\mathbf{C}$ a morfismos

$$Nat(Hom_{\mathbf{C}}(A, -), F) \longrightarrow Nat(Hom_{\mathbf{C}}(B, -), F)$$

Es decir, el morfismo que envía cualquier transformación natural

$$\alpha : Hom_{\mathbf{C}}(A, -) \longrightarrow F$$

a su correspondiente composición

$$\alpha \circ Hom_{\mathbf{C}}(f, -) : Hom_{\mathbf{C}}(B, -) \longrightarrow F.$$

Entonces, dado cualquier morfismo $f : A \longrightarrow B$ en $\mathbf{C}$, debemos ver si se cumple la condición de naturalidad en el siguiente diagrama:

$$\begin{array}{ccc} Nat(Hom_{\mathbf{C}}(A, -), F) & \xrightarrow{Nf} & Nat(Hom_{\mathbf{C}}(B, -), F) \\ \Big\downarrow{\scriptstyle \mathcal{E}_{A,F}} & & \Big\downarrow{\scriptstyle \mathcal{E}_{B,F}} \\ FA & \xrightarrow[Ff]{} & FB \end{array}$$

De nuevo, como hemos estado haciendo hasta ahora, tomamos un elemento de $Nat(Hom_{\mathbf{C}}(A, -), F)$, es decir, $\alpha : Hom_{\mathbf{C}}(A, -) \longrightarrow F$, y vemos si diagrama conmuta.

Por una parte,

$$\begin{aligned} (\mathcal{E}_{B,F} \circ Nf)(\alpha) = \mathcal{E}_{B,F}(\alpha \circ Hom_{\mathbf{C}}(f, -)) = (\alpha \circ Hom_{\mathbf{C}}(f, -))_B(id_B) = \\ = \alpha_B \circ Hom_{\mathbf{C}}(f, -)_B(id_B) = \alpha_B(f). \end{aligned}$$

Por otra parte,

$$(F(f) \circ \mathcal{E}_{A,F})(\alpha) = Ff(\alpha_A(id_A)) = \alpha_B \circ Hom_{\mathbf{C}}(A, f)(id_A) = \alpha_B(f).$$

Por consiguiente, el diagrama conmuta, luego como la elección de A ha sido arbitraria, podemos concluir que hay un isomorfismo natural $\mathcal{E}_F : N \longrightarrow F$ con componentes $(\mathcal{E}_F)_A = \mathcal{E}_{A,F}$ para cada $A \in \mathbf{C}$.

Probemos ahora la naturalidad en F:

Previamente, consideramos el functor evaluación en $A \in \mathbf{C}$, definido como $ev_A : F \longrightarrow FA$. Con esto, y dada cualquier transformación natural $\gamma : F \longrightarrow G$ debemos ver si el diagrama siguiente conmuta:

$$\begin{array}{ccc} Nat(Hom_{\mathbf{C}}(A,-),F) & \xrightarrow{Nat(Hom_{\mathbf{C}}(A,-),\gamma)} & Nat(Hom_{\mathbf{C}}(A,-),G) \\ \downarrow{\scriptstyle \mathcal{E}_{A,F}} & & \downarrow{\scriptstyle \mathcal{E}_{A,G}} \\ ev_A(F) = FA & \xrightarrow[ev_A(\gamma)]{} & ev_A(G) = GB \end{array}$$

Seleccionamos un elemento de $Nat(Hom_{\mathbf{C}}(A,-),F)$, a saber, $\alpha : Hom_{\mathbf{C}}(A,-) \longrightarrow F$.

Por una parte, $(\mathcal{E}_{A,G} \circ Nat(Hom_{\mathbf{C}}(A,-),\gamma))(\alpha) = \mathcal{E}_{A,G}(\gamma \circ \alpha) = (\gamma \circ \alpha)_A(id_A) = \gamma_A(\alpha_A(id_A))$.

Por otra parte, $(ev_A(\gamma) \circ \mathcal{E}_{A,F})(\alpha) = \gamma_A(\mathcal{E}_{A,F})(\alpha)) = \gamma_A(\alpha_A(id_A))$.

Por tanto, hay un isomorfismo natural $\mathcal{E}_A : Nat(Hom_{\mathbf{C}}(A,-),-) \longrightarrow ev_A$ con componentes $(\mathcal{E}_A)_F = \mathcal{E}_{A,F}$ para cada functor $F \in \mathbf{Fun}(\mathbf{C}, \mathbf{Sets}) = \mathbf{Sets}^{\mathbf{C}}$. □

También se puede obtener la versión dual del Lema de Yoneda:

Lema 6.2 (**Lema de Co-Yoneda**). *Sea* $\mathbf{C}$ *una categoría localmente pequeña. Sea* $A \in \mathbf{C}$ *y sea el functor contravariante* $F : \mathbf{C^{op}} \Longrightarrow \mathbf{Sets}$. *Entonces*

$$Nat(Hom_{\mathbf{C}}(-, A), F) \cong FA,$$

naturalmente tanto en A *como en* F.

6.6.2. Inmersión de Yoneda

Definición 6.7. La inmersión de Yoneda es el functor $\mathcal{X} : \mathbf{C^{op}} \Longrightarrow \mathbf{Sets}^{\mathbf{C}}$, que toma un elemento $A \in \mathbf{C^{op}}$ y lo lleva al functor covariante

$$\mathcal{X}A = Hom_{\mathbf{C}}(A,-) : \mathbf{C} \Longrightarrow \mathbf{Sets},$$

y un morfismo $f : A \longrightarrow B$ en $\mathbf{C^{op}}$ a la transformación natural

$$\mathcal{X}f = Hom_{\mathbf{C}}(f,-) : Hom_{\mathbf{C}}(B,-) \longrightarrow Hom_{\mathbf{C}}(A,-).$$

Asimismo, la versión dual de la inmersión de Yoneda es el functor $\mathcal{Y} : \mathbf{C} \Longrightarrow \mathbf{Sets}^{\mathbf{C^{op}}}, \mathbf{Sets})$ que toma una elemento $A \in \mathbf{C}$ y lo lleva al functor contravariante

$$\mathcal{Y}A = Hom_{\mathbf{C}}(-, A) : \mathbf{C^{op}} \Longrightarrow \mathbf{Sets},$$

y un morfismo $f : A \longrightarrow B$ en $\mathbf{C}$a la transformación natural

$$\mathcal{Y}f = Hom_{\mathbf{C}}(-, f) : Hom_{\mathbf{C}}(-, A) \longrightarrow Hom_{\mathbf{C}}(-, B).$$

Vamos a ver que este functor es de hecho una inmersión (es decir, un functor que es completo, fiel e inyectivo sobre los objetos) como corolario del lema de Yoneda.

Proposición 6.6. *La inmersión de Yoneda es completa, fiel e inyectiva sobre los objetos.*

Demostración. El hecho de que sea completa se ha probado mediante la parte de existencia de la Proposición 6.4, y el hecho de que sea fiel por la parte de unicidad de dicha proposición.

Solo nos queda demostrar que sea inyectiva sobre los objetos. Esto significa que si $A \neq B$, entonces $\mathcal{X}A \neq \mathcal{X}B$. Supongamos que $\mathcal{X}A = \mathcal{X}B$, es decir, que $Hom_{\mathbf{C}}(A, -) = Hom_{\mathbf{C}}(B, -)$. Entonces, dado $C \in \mathbf{C}$, tendremos que

$$Hom_{\mathbf{C}}(A, -)(C) = Hom_{\mathbf{C}}(B, -)(C),$$

luego

$$Hom_{\mathbf{C}}(A, C) = Hom_{\mathbf{C}}(B, C).$$

Sin embargo, esto último solo es posible si $A = B$, luego hemos probado que $\mathcal{X}$ es inyectiva. El procedimiento es análogo para demostrar que $\mathcal{Y}$ es una inmersión. □

Asimismo, como corolario de la inmersión de Yoneda, obtenemos el principio de Yoneda:

Corolario 6.1 (**Principio de Yoneda**)**.** *Para cualquier par de objetos $A, B \in \mathbf{C}$, $A \cong B$ si y solo si $\mathcal{X}A \cong \mathcal{X}B$, siendo $\mathcal{X}$ la inmersión de Yoneda.*

Demostración. Por una parte, supongamos que $\mathcal{X}A \cong \mathcal{X}B$. Esto implica que hay un isomorfismo natural de $Hom_{\mathbf{C}}(A, -)$ a $Hom_{\mathbf{C}}(B, -)$, que por la Proposición 6.4 viene dado por $Hom_{\mathbf{C}}(f, -)$, con $f : B \longrightarrow A$. Como hemos demostrado que $\mathcal{X}$ es completamente fiel, entonces $\mathcal{X}f$ es un isomorfismo, luego f también lo es. Esto implica que $A \cong B$.

Por otra parte, supongamos que $A \cong B$. esto implica que hay un isomorfismo $f : B \longrightarrow A$. La Proposición 6.3 nos garantiza que con dicho isomorfismo obtenemos el isomorfismo natural $Hom_{\mathbf{C}}(f, -) : Hom_{\mathbf{C}}(A, -) \xrightarrow{\sim} Hom_{\mathbf{C}}(B, -)$. Por definición de $\mathcal{X}$, obtenemos

$$\mathcal{X}f : \mathcal{X}A \xrightarrow{\sim} \mathcal{X}B,$$

luego $\mathcal{X}A \cong \mathcal{X}B$. □

6.6.3. Aplicaciones

El principio de Yoneda es muy útil para de demostrar que si dos objetos son isomorfos en la categoría **Sets**, usando por ejemplo isomorfismos de hom-sets (en la categoría en la que estemos trabajando) que se han demostrado con anterioridad, entonces, mediante el principio de Yoneda, podemos concluir que esos dos objetos serán isomorfos en dicha categoría.

En primer lugar, podemos comprobar la afirmación hecha en la última frase de la Nota 5.1.

Proposición 6.7. *Sea* $\mathbf{C}$ *una categoría localmente pequeña, y sean* $A \in \mathbf{C}$ *un objeto cualquiera y* $1 \in \mathbf{C}$ *el objeto terminal de la misma. Entonces* $A \times 1 \cong A$.

Demostración.

$$\begin{aligned} Hom_{\mathbf{C}}(-, A \times 1) &\cong Hom_{\mathbf{C}}(-, A) \times Hom_{\mathbf{C}}(-, 1) \\ &\quad \text{(hom-functores preservan límites)} \\ &\cong Hom_{\mathbf{C}}(-, A) \times 1 \\ &\quad \text{(por definición de objeto terminal)} \\ &\cong Hom_{\mathbf{C}}(-, A) \end{aligned}$$

Luego por el principio de Yoneda, $A \times 1 \cong A$. □

Asimismo, también podemos demostrar que una categoría cartesiana cerrada con coproductos (binarios) tiene la propiedad distributiva:

Proposición 6.8. *Sea* $\mathbf{C}$ *una categoría localmente pequeña y cartesiana cerrada con coproductos binarios. Sean también* $A, B, C \in \mathbf{C}$. *Entonces*

$$A \times (B \oplus C) \cong (A \times B) \oplus (A \times C),$$

donde $\oplus$ *es el operador del coproducto de* $\mathbf{C}$.

Demostración.

$$\begin{aligned} Hom_{\mathbf{C}}(A \times (B \oplus C), -) &\cong Hom_{\mathbf{C}}(B \oplus C, -^A) \\ &\quad (\mathbf{C} \text{ es cartesiana cerrada}) \\ &\cong Hom_{\mathbf{C}}(B, -^A) \oplus Hom_{\mathbf{C}}(C, -^A) \\ &\quad \text{(hom-functores preservan límites)} \\ &\cong Hom_{\mathbf{C}}(A \times B, -) \oplus Hom_{\mathbf{C}}(A \times C, -) \\ &\quad (\mathbf{C} \text{ es cartesiana cerrada}) \end{aligned}$$

De nuevo, por el principio de Yoneda, se tiene que $A \times (B \oplus C) \cong (A \times B) \oplus (A \times C)$. □

6.7. Transformaciones naturales en Haskell

Para construir una transformación natural en Haskell necesitamos dos functores, `F` y `G`, que envíen el tipo `a` a `F a` y `G a` respectivamente. Dicha transformación en Haskell será la función del functor `F` al functor `G` siguiente:

```
lambda :: F a -> G a
```

Tengamos en cuenta que `lambda` no es más que una función parametrizada mediante el tipo `a`. Es decir, `lambda` es una función polimórfica (véase la Sección A.2.3). En Haskell, todas la funciones de este tipo (las definidas como `lambda`) cumplen de forma automática la condición de naturalidad [24]. Lo que significa que si tenemos una función cualquiera `f :: a -> b`, los dos functores y transformación natural anteriores, obtenemos que

$$Gf \circ \lambda_a = \lambda_b \circ Ff,$$

o, lo que es lo mismo en Haskell,

```
fmap f . lambda = lambda . fmap f
```

Nota 6.3. *No es necesario especificar de ninguna forma qué `fmap` estamos usando ya que la inferencia de tipos de Haskell se encarga de aplicar el `fmap` adecuado. Lo veremos al comprobar la condición de naturalidad de la transformación natural del ejemplo siguiente.*

Vamos a ver un ejemplo de transformación natural entre los functores `[]` y `Maybe` estudiados en la sección anterior.

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

Vamos a verificar que la condición de naturalidad se cumple:

- Caso `[]`:

```
(fmap f . headSeguro) []
    = (por definición de .)
    fmap f (headSeguro [])
    = (por definición de headSeguro)
    fmap f Nothing
    = (por definición de fmap)
    Nothing
    = (por definición de headSeguro)
    headSeguro []
    = (por definición de fmap)
    headSeguro (fmap f [])
    = (por definición de .)
    (headSeguro . fmap f) []
```

- Caso `(x:xs)`:

```
(fmap f . headSeguro) (x:xs)
    = (por definición de .)
    fmap f (headSeguro (x:xs))
    = (por definición de headSeguro)
    fmap f (Just x)
    = (por definición de fmap)
    Just (f x)
    = (por definición de headSeguro)
    headSeguro (f x : fmap f xs)
    = (por definición de fmap)
    headSeguro (fmap f (x:xs))
    = (por definición de .)
    (headSeguro . fmap f) (x:xs)
```

6.8. Functores representables en Haskell

Por definición, un functor representable `F` es todo aquel functor isomorfo a un hom-functor. Es decir, en Haskell, cuando definamos un (endo)functor representable, este será isomorfo al functor `Reader a` que hemos estado utilizando hasta ahora. Tendremos pues las dos funciones siguientes

```
alpha :: Reader a -> F a

beta :: F a -> Reader a
```

tales que `alpha . beta = id` y `beta . alpha = id`.

En Haskell definiremos un functor representable de la siguiente forma:

```
class Representable F where
    type Rep F :: *
    tabulate :: (Rep F -> x) -> F x
    index    :: F x -> (Rep F -> x)
```

Aquí `tabulate` coincide con la función `alpha`, e `index` con `beta`. El tipo `Rep F` se refiere al tipo representante; en las definiciones previas se correspondería con `a`.

6.9. Yoneda en Haskell

El conjunto de transformaciones naturales de un hom-functor (que como vimos, era el functor `Reader a x = a -> x`) y cualquier otro functor `F` viene dada por la función polimórfica siguiente:

```
alpha :: (a -> x) -> F x
```

El Lema de Yoneda nos dice que hay una relación de isomorfía entre los elementos de `alpha` y los elementos de `F a`:

$$\texttt{alpha :: (a -> x) -> F x} \cong \texttt{F a}.$$

Lo que significa que podemos representar el "contenedor" de `a`, `F a`, a través de una función polimórfica `alpha`[2]. Siguiendo la notación y procedimientos de la Proposición 6.5, vamos a ver que hay un isomorfismo natural entre:

```
epsilon :: ((a -> x) -> F x) -> F a
epsilon f = f id

chi :: F a -> ((a -> x) -> F x)
chi x f = fmap f x
```

Definimos `epsilon` de esa forma ya que `id` $\in$ `F x`, luego `f id` $\in$ `F a`, y definimos `chi` de tal modo que obtengamos un elemento de `F x` a partir de usar `fmap` con una función cualquier `f` a un elemento de `F a`.

Vamos a tener que probar que, en **Hask**, la composición de ambas funciones sea la identidad:

```
epsilon . chi = id
chi . epsilon = id
```

Veámoslo.

```
(epsilon . chi) f
    = (por definición de .)
    epsilon (chi f)
    = (por definición de epsilon)
    (chi f) id
    = (por definición de chi)
    fmap id f
    = (por definición de fmap)
    id f

(chi . epsilon) f
    = (por definición de .)
    chi (epsilon f)
    = (por definición de epsilon)
    chi f id
    = (por definición de chi)
    fmap id f
    = (por definición de fmap)
    id f
```

[2]O lo que es lo mismo, hay un isomorfismo entre cualquier función polimórfica y un tipo.

El Lema de Yoneda en Haskell nos muestra que un tipo puede tener varias representaciones: una función polimórfica o un tipo, y ambas son igual de válidas. Por ejemplo, en la construcción de compiladores, el Lema de Yoneda se puede usar para tener el siguiente isomorfismo natural (usando `id` como el functor `F` en la explicación anterior):

$$\texttt{(a -> r) -> r} \cong \texttt{a},$$

que nos dice que un tipo cualquiera `a` puede ser sustituido por un *handler* para `a`, es decir, por una computación que tome dicho tipo `a` y ejecute el resto de ella, donde el tipo `r` es normalmente algún código de estado). Este patrón se suele utilizar en la programación de interfaces de usuario o en la programación concurrente [18, p. 240].

Por otra parte, la Inmersión de Yoneda nos muestra que en Haskell hay un isomorfismo entre functores `Reader` y funciones. Es decir

$$\texttt{(a -> x) -> (b -> x)} \cong \texttt{b -> a},$$

o lo que es lo mismo,

$$\texttt{Reader a -> Reader b} \cong \texttt{b -> a}.$$

7 ADJUNCIONES

En la noción de functor adjunto intervienen los conceptos más importantes que hemos mostrado hasta ahora: los functores, las transformaciones naturales, etc., y encapsula un patrón que aparece a lo largo de muchos campos de las matemáticas. Citando el famoso eslogan de MacLane, *adjoint functors arise everywhere* [14, p. vii] y, de hecho, los teoremas de los functores adjuntos (el Teorema General del Functor Adjunto y el Teorema Especial del Functor Adjunto) tienen un gran variedad de aplicaciones [1, p. 335]. Sin la perspectiva de la Teoría de Categorías, este patrón no sería posible reconocerlo. Sin embargo, su importancia no se limita solamente a las matemáticas. La noción de functor adjunto es fundamental para la programación funcional ya que será el concepto que nos lleve finalmente a la noción de mónada. Además, los functores adjuntos representan otro tipo, esta vez más débil, de equivalencia (entre objetos, categorías, etc.). En la Definición 6.4, hemos visto que dos categorías son equivalentes si existen dos functores cuya composición es la identidad apropiada. En el caso de los functores adjuntos, nos dan una idea de la relación entre functores entre dos categorías, no necesariamente de equivalencia, en la que lo importante no es el par de categorías en sí, sino los functores que forman la adjunción. En otras palabras, la adjunción es una relación particular entre un par de functores. No es de extrañar, ya que como hemos visto hasta ahora, la perspectiva de la Teoría de Categorías es la que se centra no en los objetos de las mismas, sino en las relaciones entre ellos.

Veremos, en primer lugar, la definición de functores adjuntos y, tras ello, ejemplos y propiedades de los mismos.

Definición 7.1. Sean $\mathbf{C}$ y $\mathbf{D}$ dos categorías y $F : \mathbf{C} \Longrightarrow \mathbf{D}$ y $G : \mathbf{D} \longrightarrow \mathbf{C}$ dos functores entre ellas. Se dice que F es un adjunto a la izquierda de G y G es un adjunto a la derecha de F si y solo si

$$Hom_{\mathbf{D}}(FC, D) \cong Hom_{\mathbf{C}}(C, GD),$$

naturalmente en $C \in \mathbf{C}$ y $D \in \mathbf{D}$. Esto se denota como $F \dashv G$ (o $F \dashv G : \mathbf{C} \longrightarrow \mathbf{D}$), y en forma de diagrama, como

$$\mathbf{C} \underset{G}{\overset{F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathbf{D}$$

Cuando esto sucede, se dice que F y G forman una adjunción.

Ejemplo 7.1. (de adjunción libre-olvidadiza) Recordemos el Ejemplo 2.8, en el que hemos visto la definición y la propiedad universal que define los monoides libres. De allí podemos formular las afirmaciones siguientes:

- Un monoide puede ser generado a partir de un conjunto cualquiera $X \in \mathbf{Sets}$. Es decir, tenemos el functor "libre"

$$F : \mathbf{Sets} \Longrightarrow \mathbf{Mon}$$

tal que FX es un monoide libre.

- De un monoide $M \in \mathbf{Mon}$ puede obtenerse el conjunto subyacente de sus elementos a través del functor olvidadizo

$$U : \mathbf{Mon} \Longrightarrow \mathbf{Sets}$$

tal que $U(M)$ es el conjunto subyacente de M.

La composición de estos functores en un conjunto cualquiera X resulta

$$i_X : X \longrightarrow (U \circ F)(X).$$

Por la propiedad universal de los monoides, tenemos que para cada monoide M y cada función $f : X \longrightarrow UM$, hay un único isomorfismo $g : FX \longrightarrow M$ tal que $f = Ug \circ id_X$. Es decir, el siguiente diagrama conmuta:

$$FX \overset{g}{\dashrightarrow} M$$

$$\begin{array}{ccc} (U \circ F)(X) & \xrightarrow{Ug} & UM \\ {\scriptstyle id_X}\uparrow & \nearrow_{f} & \\ X & & \end{array}$$

Consideremos la aplicación

$$\phi : Hom_{\mathbf{Mon}}(FX, M) \longrightarrow Hom_{\mathbf{Sets}}(X, UM)$$

tal que $\phi(g) = Ug \circ id_X$, donde un morfismo $g \in \mathbf{Mon}$ (parte de arriba del diagrama) lo envía a un morfismo $\phi(g) \in \mathbf{Sets}$ (parte de abajo del diagrama).

Entonces por la propiedad universal definida anteriormente, podemos concluir que ϕ es un isomorfismo:

$$Hom_{\mathbf{Mon}}(FX, M) \cong Hom_{\mathbf{Sets}}(X, UM).$$

Esto significa que dado un functor olvidadizo como U, este tiene un functor adjunto a la izquierda, F, y viceversa; partiendo de un functor libre como F, este tiene un functor adjunto a la derecha, U.

Veamos ahora una definición equivalente:

Definición 7.2. Sean $\mathbf{C}$ y $\mathbf{D}$ dos categorías y $F : \mathbf{C} \Longrightarrow \mathbf{D}$ y $G : \mathbf{D} \longrightarrow \mathbf{C}$ dos functores entre ellas. Entonces $F \vdash G$ si y solo si existen las transformaciones naturales $\eta : id_{\mathbf{C}} \longrightarrow G \circ F$ y $\epsilon : F \circ G \longrightarrow id_{\mathbf{D}}$ tales que

$$(\epsilon \circ F)(C) \circ (F \circ \eta)(C) = id_F(C)$$
$$(G \circ \epsilon)(D) \circ (\eta \circ G)(D) = id_G(D)$$

para todo $C \in \mathbf{C}$ y $D \in \mathbf{D}$.

O, equivalentemente, que las identidades triangulares siguientes (en las categorías $\mathbf{Fun}(\mathbf{C}, \mathbf{D})$ y $\mathbf{Fun}(\mathbf{D}, \mathbf{C})$ respectivamente) se cumplan:

Las transformaciones naturales η y ϵ se denominan **unidad** y **counidad** de la adjunción.

Nota 7.1. *Por lo que respecta al diagrama anterior, en los morfismos de la parte superior del diagrama, $F \circ \eta$ y $\eta \circ G$, nos referimos a F o G en realidad como la identidad de dichos elementos, es decir:*

$$(F \circ \eta)(F) = (F \circ \eta)(F \circ id_{\mathbf{C}}) \overset{notación}{=} (id_F \circ \eta)(F \circ id_{\mathbf{C}}) = F \circ (G \circ F)$$
$$(\eta \circ G)(G) = (\eta \circ G)(id_{\mathbf{C}} \circ G) \overset{notación}{=} (\eta \circ id_G)(id_{\mathbf{C}} \circ G) = (G \circ F) \circ G$$

A continuación vamos a ver un ejemplo de adjunción en construcciones categóricas.

Ejemplo 7.2. Las acciones de *currificar* y *descurrificar* una función guardan una relación de adjunción. En otras palabras, que tomar el producto por $A \in \mathbf{C}$ es un ajunto por la izquierda de exponenciar por A.

Sea $\mathbf{C}$ una categoría con productos binarios. Consideremos el objeto $A \in \mathbf{C}$ y el (endo)functor producto

$$- \times A : \mathbf{C} \Longrightarrow \mathbf{C}$$

definido de modo que

- Envía una objeto X a $X \times A$, y
- envía un morfismo $f : X \longrightarrow Y$ a $f \times id_A : X \times A \longrightarrow Y \times A$.

Para que este functor tenga un adjunto a la derecha, deberíamos tener un functor $U : \mathbf{C} \Longrightarrow \mathbf{C}$ tal que

$$Hom_{\mathbf{C}}((- \times A)(X), Y) \cong Hom_{\mathbf{C}}(X, UY)$$

Si definimos U como el functor $U = (-)^A : \mathbf{C} \Longrightarrow \mathbf{C}$ (que hemos comprobado en la Proposición 5.1 que lo era), y tal que

- U envía un objeto Y a Y^A, y
- U envía un morfismo $g : Y \longrightarrow Z$ a $g^A : Y^A \longrightarrow Z^A$,

se cumple que

$$Hom_{\mathbf{C}}((- \times A)(X), Y) \cong Hom_{\mathbf{C}}(X, (-^A)(Y)),$$

debido a la construcción del objeto exponencial en la Definición 5.1.

Entonces podemos concluir que $(- \times A) \vdash (-)^A$.

7.1. Propiedades de los functores adjuntos

Como hemos probado para todas las construcciones anteriores, vamos a ver que los functores adjuntos son únicos salvo isomorfismo.

Proposición 7.1. *Sea* $\mathbf{C}$ *y* $\mathbf{D}$ *dos categorías. Dado un functor* $F : \mathbf{C} \Longrightarrow \mathbf{D}$ *y los functores adjuntos a la derecha* $U, V : \mathbf{D} \Longrightarrow \mathbf{C}$ *que cumplen que*

$$F \vdash U \text{ y } F \vdash V,$$

entonces $U \cong V$.

Demostración. Sean $C \in \mathbf{C}$ y $D \in \mathbf{D}$ cualesquiera. Entonces, por la Definición 7.1

$$\begin{aligned} Hom_{\mathbf{C}}(C, UD) &\cong Hom_{\mathbf{C}}(FC, D) \text{ (ya que } F \vdash U) \\ &\cong Hom_{\mathbf{C}}(C, VD) \text{ (ya que } F \vdash V) \end{aligned}$$

Por el principio de Yoneda, $UD \cong VD$, y por ser functores adjuntos, este isomorfismo es natural en D, entonces $U \cong V$. □

Por último, veamos una de las propiedades más importantes de los functores adjuntos.

Proposición 7.2. *Las functores adjuntos a la derecha preservan límites y, dualmente, los functores adjuntos a la izquierda preservan colímites.*

Demostración. Sean $\mathbf{C}$ y $\mathbf{D}$ do categorías. Supongamos que tenemos la adjunción siguiente

$$\mathbf{C} \underset{G}{\overset{F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathbf{D},$$

con el diagrama $D : \mathbf{J} \Longrightarrow \mathbf{D}$ (siendo $\mathbf{J}$ la categoría índice) donde el límite $\varprojlim_j D_j$ existe.

Entonces para todo $X \in \mathbf{C}$ se tiene que

$$\begin{aligned} Hom_{\mathbf{C}}(X, G(\varprojlim_j D_j)) &\cong Hom_{\mathbf{D}}(FX, \varprojlim_j D_j) \text{ (ya que } F \vdash G) \\ &\cong \varprojlim_j Hom_{\mathbf{D}}(FX, D_j) \text{ (hom-functores preservan límites)} \\ &\cong \varprojlim_j Hom_{\mathbf{C}}(X, GD_j) \text{ (ya que } F \vdash G) \\ &\cong Hom_{\mathbf{C}}(X, \varprojlim_j GD_j) \text{ (hom-functores preservan límites)} \end{aligned}$$

Luego por el principio de Yoneda, $G(\varprojlim_j D_j) = \varprojlim_j GD_j$. □

Si bien esta proposición no nos dice exactamente qué caracteriza un functor para que forme parte de una adjunción, sí nos dice una condición necesaria para ello. No obstante, el Teorema General del Functor Adjunto [7] nos dice exactamente qué características debe cumplir un functor para tener un adjunto por la izquierda. El enunciado y demostración de dicho teorema se puede encontrar en [14, p. 117].

7.2. Adjunciones en Haskell

Esta sección nos servirá como preliminar a la sección de las mónadas en Haskell. Como hemos visto en la Definición 7.2, una adjunción entre dos functores está constituida por dos transformaciones naturales: la unidad y la counidad. En Haskell, estas dos se representan mediante las funciones `return` y `extract`:

```
return :: d -> m d
extract :: w c -> c
```

Siguiendo la Definición 7.2 de nuevo, aquí `m` se corresponde con el endofunctor $G \circ F$, y `w` con el endofunctor $F \circ G$.

Siguiendo con la analogía de los functores (endofunctores en particular, ya que estamos en **Hask**) como "contenedores" de tipos, la función `return` crea un "contenedor" por defecto sobre un valor de cualquier tipo, y la función `extract`, "extrae" el valor de su correspondiente "contenedor".

Asimismo, también podemos definir la adjunción de endofunctores en Haskell de la siguiente forma:

```
class (Functor f, Representable g)
  => Adjunction f g | f -> g, g -> f where
    unit   :: a ->  (g . f) a
    counit :: (f . g) a -> a
```

La línea con `Adjunction f g | f -> g, g -> f` significa que entre los dos functores `f` y `g` hay una dependencia funcional. Es decir, `f -> g` significa que `g` está determinado por `f`, y viceversa.

O bien, usando la Definición 7.1, podemos definirlo de la siguiente manera:

```
class (Functor f, Representable g)
  => Adjunction f g | f -> g, g -> f where
    leftAdjunct :: (f a -> b) -> (a -> g b)
    rightAdjunct :: (a -> g b) -> (f a -> b)
```

La relación entre estas dos definiciones, y por lo que acaban siendo equivalentes en Haskell, es la siguiente:

```
unit           = leftAdjunct id
counit         = rightAdjunct id
leftAdjunct f  = fmap f . unit
rightAdjunct f = counit . fmap f
```

8 MÓNADAS

Recordemos el concepto de monoide dado en la Ejemplo 2.2. Un monoide puede ser visto como una categoría de un solo objeto donde los elementos del monoide son los morfismos de dicho objeto a sí mismo, y la operación binaria de dicho monoide como la composición de sus morfismos. Sin embargo, un monoide también puede ser visto como un elemento de una categoría monoidal. Vamos a verlo.

8.1. Categorías monoidales

Definición 8.1. Una categoría monoidal estricta es una categoría $\mathbf{C}$ junto con un operador binario

$$\otimes : \mathbf{C} \times \mathbf{C} \Longrightarrow \mathbf{C}$$

que es un bifunctor asociativo,

$$A \otimes (B \otimes C) \cong (A \otimes B) \otimes C,$$

y junto a un elemento $I \in \mathbf{C}$ que actúa como unidad,

$$I \otimes C \cong C \cong C \otimes I.$$

En otras palabras, una categoría monoidal es un monoide en **Cat**.

Esta definición no es lo suficientemente general debido a que la asociatividad y la unidad se han definido mediante isomorfismos, no mediante la igualdad. En la mayoría de categorías, el producto (o el coproducto) no son asociativos, es decir, que para tres elementos A, B, C cualquiera, no se cumple que

$$A \times (B \times C) = (A \times B) \times C,$$

pero sí que ocurre para cualquier categoría con productos o coproductos si se define las asociatividad mediante isomorfismo, como hemos visto en el Ejemplo 6.5. Del mismo modo, no se cumple en general que para un objeto terminal 1,

$$1 \times A = A = A \times 1,$$

lo cual sí que ocurre si se define mediante isomorfismo en cualquier categoría con objetos terminales como se ha visto en la Proposición 6.7.

Vamos a ver entonces la definición más general de categoría monoidal.

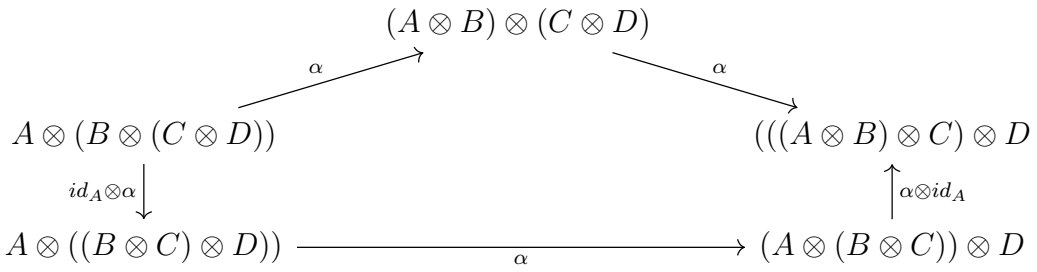
Definición 8.2. Una categoría monoidal consiste en una categoría $\mathbf{C}$ junto con un functor

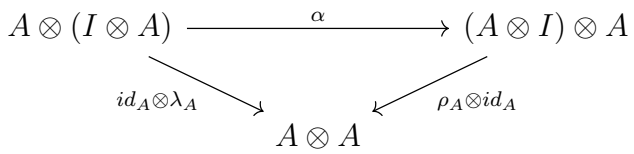
$$\otimes : \mathbf{C} \times \mathbf{C} \Longrightarrow \mathbf{C},$$

un objeto $I \in \mathbf{C}$, y los isomorfismos naturales

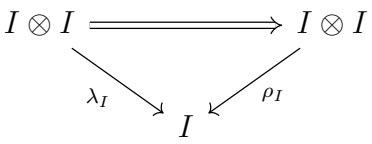
$$\alpha_{ABC} : A \otimes (B \otimes C) \xrightarrow{\sim} (A \otimes B) \otimes C$$
$$\lambda_A : I \otimes A \xrightarrow{\sim} A,\ \rho_A : A \otimes I \xrightarrow{\sim} A,$$

de forma que los diagramas siguientes siempre conmutan:





Y por último,



que es lo mismo que afirmar que $\lambda_I = \rho_I$.

Ahora ya podemos denominar como categoría monoidal a cualquier categoría con productos finitos y con objeto terminal, siendo el operador $\otimes$ el producto $\times$ de la categoría, y la unidad I un objeto terminal 1 también de dicha categoría. Dualmente, también serán categorías monoidales aquellas categorías con coproductos finitos y objetos iniciales. Un ejemplo básico de este concepto es el siguiente.

Ejemplo 8.1. La categoría **Sets** con el producto cartesiano y cualquier conjunto de un elemento como unidad (los objetos terminales de **Sets**).

Con el concepto de categoría monoidal ya definido, podemos generalizar el concepto de monoide como un elemento de una categoría monoidal.

Definición 8.3. Un monoide M es un elemento de una categoría monoidal $(\mathbf{C}, \otimes, I)$ tal que dispone de los siguientes dos morfismos:

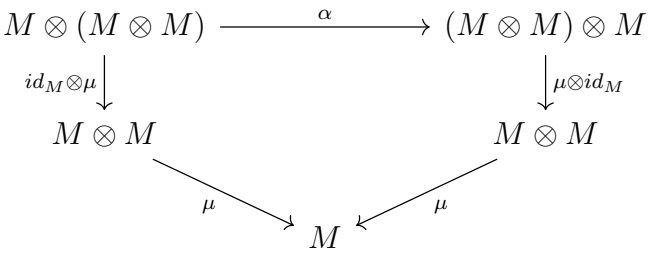
- El morfismo de multiplicación,

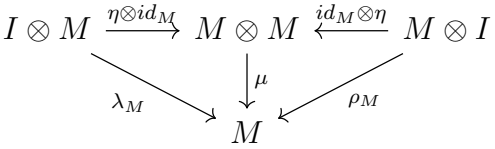
$$\mu : M \otimes M \longrightarrow M.$$

- El morfismo de la unidad,

$$\eta : I \longrightarrow M.$$

Además, debe cumplir la propiedad asociativa y la de la unidad, es decir, que los siguientes diagramas conmuten:





Esta generalización del monoide desde la perspectiva de Teoría de Categorías va a servirnos para definir uno de los conceptos fundamentales de la programación funcional: la mónada.

8.2. Mónadas

Consideremos la adjunción $F \vdash G$, cuya composición es

$$F \circ G : \mathbf{C} \Longrightarrow \mathbf{C}.$$

Sea el endofunctor $T = F \circ G : \mathbf{C} \Longrightarrow \mathbf{C}$. Entonces tenemos las siguientes transformaciones naturales:

$$\eta : id_T \longrightarrow T$$
$$\epsilon : T \longrightarrow id_T$$

Esta última, dado $C \in \mathbf{C}$, la consideramos aplicada a FC, es decir,

$$\epsilon_{FC} : T \circ FC = F \circ G \circ FC \longrightarrow FC.$$

Si componemos G con esta transformación, obtenemos

$$G \circ \epsilon_{FC} : G \circ T \circ FC = G \circ F \circ G \circ FC = (T \circ T)(C) \longrightarrow G \circ FC = TC.$$

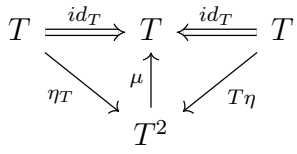
Nota 8.1 (Notación)**.** *Llamaremos* T^2 *a* $(T \circ T)$*,* T^3 *a* $(T \circ T \circ T)$*, etc.*

A la composición $G \circ \epsilon_{FC}$ la llamamos μ:

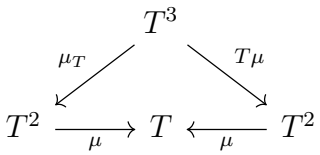
$$\mu : T^2 \longrightarrow T.$$

Mediante las identidades triangulares de las adjunciones (Definición 7.2) vamos a ver se cumplen los diagramas conmutativos siguientes para la estructura (T, η, μ) que hemos creado.

En primer lugar,



con lo que obtenemos que $\mu \circ \eta_T = id_T = \mu \circ T\eta$. Y, en segundo lugar,



con lo que obtenemos, finalmente, que $\mu \circ \mu_T = \mu \circ T\mu$.

Lo que nos enseña este "sketch" de demostración es el hecho de que para cualquier endofunctor que surja de una adjunción se obtiene una estructura muy similar a la de un monoide [2, p. 258]. De hecho, estos dos diagramas conmutativos coinciden con las propiedades de la unidad y de la asociatividad respectivamente del monoide, siendo μ el morfismo de multiplicación y η el de la unidad. La siguiente definición encapsula este concepto:

Definición 8.4. Una mónada (T, η, μ) sobre una categoría $\mathbf{C}$ consiste en un endofunctor $T : \mathbf{C} \Longrightarrow \mathbf{C}$ junto con dos transformaciones naturales

$$\eta : id_T \longrightarrow T$$

y

$$\mu : T \circ T = T^2 \longrightarrow T$$

de modo que satisfacen las igualdades siguientes:

$$\mu \circ \mu_T = \mu \circ T\mu$$
$$\mu \circ \eta_T = id_T = \mu \circ T\eta$$

De hecho, debido a que la categoría $\mathbf{Fun}(\mathbf{C},\mathbf{C}) := \mathbf{End}(\mathbf{C})$ de endofunctores en $\mathbf{C}$ donde se encuentra T es monoidal, el operador composición en ella es el operador producto en el monoide, y el functor identidad en $\mathbf{End}(\mathbf{C})$ es la unidad en dicho monoide. Es decir,

Una mónada es un monoide en la categoría de los endofunctores.

8.3. Categoría de Kleisli

Una descripción alternativa de las mónadas es mediante una categoría de Kleisli. Esta descripción es más fácil de justificar a nivel computacional y se corresponde con la representación que se encuentra implementada en las librerías de lenguajes de programación funcionales como Haskell.

Definición 8.5. Sea (T,η,μ) una mónada sobre una categoría $\mathbf{C}$. La categoría de Kleisli de $\mathbf{C}$ es la categoría $\mathbf{C}_T$ donde:

- Objetos: $\mathrm{Obj}(\mathbf{C}_T) = \mathrm{Obj}(\mathbf{C})$.
- Morfismos: dados $X, Y \in \mathbf{C}$, entonces $Hom_{\mathbf{C}_T}(X,Y) = Hom_{\mathbf{C}}(X,TY)$.
- Composición: dados los morfismos $f : X \longrightarrow TY$ y $g : TY \longrightarrow Z$ en $\mathbf{C}$, la composición en $\mathbf{C}_T$ viene dada por:

  $$g \circ f = \mu_Z \circ Tg \circ f : X \longrightarrow TY \longrightarrow T^2Z \longrightarrow TZ,$$

  o, puesto de otra manera, el siguiente diagrama conmutativo:

  $$\begin{array}{ccc} X & \xrightarrow{f} & TY \\ {\scriptstyle g\circ_T f}\downarrow & & \downarrow{\scriptstyle Tg} \\ TZ & \xleftarrow[\mu_Z]{} & T^2Z \end{array}$$

- Identidad: el morfismo identidad en $\mathbf{C}_T$ se corresponde con η en $\mathbf{C}$.

La equivalencia entre las mónadas y las categorías de Kleisli fue demostrada por Ernest Manes en 1976 [15].

A continuación vamos a ver el concepto de mónada en Haskell desarrollado desde cero y, tras ello, ejemplos que muestran su funcionamiento de manera práctica.

8.4. Functores aplicativos

En la Sección 4.4 hemos visto la construcción de un functor en Haskell, que no es más que un "contenedor" de un tipo que puede implementar una función, `fmap`, que envía valores y funciones de un tipo a valores de dicho "contenedor". Sin embargo, un functor solamente puede asignar una función de un solo argumento, no de varios. Es decir, no disponemos de una generalización de `fmap` que tome una función de, por ejemplo, 2 argumentos, como esta:

```
fmap :: (a -> b -> c) -> f a -> f b -> f c
```

Pero sabemos por la Sección 5.3 que tenemos esta equivalencia:

$$\texttt{a -> b -> c} \cong \texttt{(a, b) -> c}.$$

Con esto, se puede llegar a definir un nuevo tipo de functor que preserve productos (que como hemos visto en la Sección 3.5.2, en Haskell estos son las tuplas) y, de esa forma, poder asignar funciones de más de dos elementos.

Por otra parte, a lo largo de esta sección hemos visto que las estructuras monoidales (ya sean categorías monoidales u otras estructuras algebraicas) tienen la operación producto (junto con un elemento unidad) y sus propiedades correspondientes (asociatividad y la de la unidad).

Entonces, este nuevo tipo de functor podría ser aquel que preserve la estructura monoidal, es decir, el producto, la unidad y sus respectivas propiedades, definido de la siguiente manera:

```
class Functor f => Monoidal f where
    unit :: f ()
    (**) :: f a -> f b -> f (a, b)
```

Este nuevo tipo de functor `Monoidal` es equivalente al ya existente **functor aplicativo** en Haskell, definido canónicamente de la siguiente forma:

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

que es equivalente a `Monoidal` ya que:

```
pure x = fmap (const x) unit
unit = pure ()
```

(donde `const :: a -> b -> a` es la función definida tal que `const x _ = x`), y:

```
fa ** fb = pure (,) <*> fa <*> fb
fg <*> fa = fmap (\f -> (\x -> f x)) (fg ** fa)
```

Con esto obtenemos las equivalencias entre las dos clases de functores [17].

Ejemplo 8.2. El functor `Maybe` es una instancia de `Applicative`.

Definimos `Maybe` como instancia de `Applicative` de la siguiente manera:

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure = Just

    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    Nothing  <*> _  = Nothing
    (Just g) <*> mx = fmap g mx
```

Algunos ejemplos de uso de estos operadores son:

```
> pure (+) <*> Just 1 <*> Just 2
Just 3

> pure (*) <*> Just 2 <*> Nothing
Nothing
```

Ejemplo 8.3. El functor `[]` es una instancia de `Applicative`.

Definimos `[]` como instancia de `Applicative` de la siguiente manera:

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure x = [x]

    -- (<*>) :: [(a -> b)] -> [a] -> [b]
    []  <*> _  = []
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Algunos ejemplos de uso de estos operadores son:

```
> pure (+) <*> [1, 2] <*> [3, 4]
[(+1), (+2)] <*> [3, 4] =
    = [4, 5, 5, 6]

> pure (*) <*> [1] <*> []
[]
```

Los functores aplicativos están estrechamente relacionados con la programación de efectos secundarios. Podemos aplicar funciones puras a un número ilimitado de argumentos sin preocuparnos por la propagación de, por ejemplo, errores (véase el ejemplo anterior con el functor aplicativo `Maybe` y su valor `Nothing`), ya que esto estará implementado en los propios mecanismos de los functores aplicativos. Es por esto que los functores aplicativos pueden verse como el concepto que abstrae la idea de aplicar funciones puras a argumentos con efectos [12, p. 162].

8.5. Mónadas en Haskell

Recapitulando hasta ahora, hemos definido la clase de functores en Haskell, que no son más que *endofunctores* en **Hask**. A continuación hemos definido los functores aplicativos, que son un tipo de endofunctores que, en particular, preservan la *estructura monoidal*. Y por último vamos a ver las mónadas en su representación equivalente mediante categorías de Kleisli.

Como hemos visto en la Definición 8.5, un morfismo en una categoría de Kleisli es una función

```
a -> m b
```

en **Hask**, siendo `a` y `b` tipos y `m` un functor.

Veamos ahora cómo definimos la composición de este tipo de funciones. Sean `f :: a -> m b` y `g :: b -> m c` dos funciones que queremos componer. Definimos la *composición de Kleisli* como el operador

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c),
```

también conocido como el "operador pez".

Por último, la identidad será el morfismo de Kleisli `a -> m a`, que se corresponde con la unidad de una adjunción, η, vista en la Sección 7.2, que en Haskell se denomina `return`.

Entonces, una mónada estará definida de la siguiente manera:

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

Podemos comprobar fácilmente que las propiedades asociativa y de la unidad de esta definición de mónada se cumplen:

- Asociatividad:

```
(f >=> g) >=> h  ≅  f >=> (g >=> h)
```

- Unidad:

```
return >=> f  ≅  f
f >=> return  ≅  f
```

Podemos, sin embargo, definir el operador pez de manera diferente. La primera función que toma `>=>`, es decir, `f :: a -> m b`, tendrá como resultado un valor de tipo `m b` en todos los casos. Entonces, partiendo de un valor de dicho tipo, `m b`, y una

función `g :: b -> m c`, debemos acabar con un valor de tipo `m c`. Cambiando `b` por `a` y `c` por `b` en el procedimiento anterior, obtenemos un operador de composición nuevo

$$\texttt{(>>=) :: m a -> (a -> m b) -> m b}.$$

Pero, de nuevo, este operador de composición se puede simplificar. Sabemos desde el principio que `m` es un functor, luego podemos aplicar `fmap` a los argumentos de `>>=`. Esta aplicación tomará un valor de `m a` y mediante la función `(a -> m b)` nos devolverá `m (m b)`. Sin embargo, si introducimos una función nueva, llamada `join`, definida de la siguiente manera

$$\texttt{join :: m (m a) -> m a},$$

podemos obtener `x >>= f = join (fmap f x)`, donde `x :: m a` y `f :: a -> m b`. Es fácil ver que `join` se corresponde con μ en la Definición 8.5.

Hemos obtenido finalmente una estructura que implementa los morfismos η y μ como hemos visto a lo largo de la explicación, conserva las propiedades de las mónadas (asociatividad y unidad) y tiene el operador de composición bien definido. Mediante las categorías de Kleisli hemos obtenido una representación de la mónada en Haskell, cuya definición es:

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

Y que, de nuevo, conserva las propiedades de las mónadas:

- Asociatividad:

$$\texttt{(m >>= g) >>= h} \cong \texttt{m >>= (x -> g x >>= h)} \cong \texttt{m >>= (g >>= h)}$$

- Unidad:

$$\texttt{return x >>= f} \cong \texttt{f x}$$
$$\texttt{m >>= return} \cong \texttt{m}$$

Nota 8.2. *El hecho de que* ***`Applicative`*** *sea una superclase de* ***`Monad`*** *se detalla en el* Functor-Applicative-Monad Proposal *de 2014 [9].*

Ejemplo 8.4. Vamos a ver que el functor `Maybe` es una mónada.

Definimos `Maybe` como instancia de `Monad` de la siguiente forma:

```
instance Monad Maybe where
    return = Just
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
```

y cumple las propiedades de las mónadas:

- Asociatividad:

```
Nothing >>= (\x -> g x >>= h)
    = (por definición de >>=)
    Nothing
    = (por definición de >>=)
    (Nothing >>= g)
    = (por definición de >>=)
    (Nothing >> = g) >>= h

(Just y) >>= (\x -> g x >>= h)
    = (por definición de >>=)
    (\x -> g x >>= h) y
    = (reducción de la expresión)
    g y >>= h
    = (por definición de >>=)
    ((Just y) >>= g) >>= h
```

- Unidad:

```
return Nothing >>= f
    = (por definición de return)
    Just Nothing >> = f
    = (por definición de >>=)
    f Nothing

return x >>= f
    = (por definición de return)
    Just x >>= f
    = (por definición de >>=)
    f x

Nothing >>= return
    = (por definición de >>=)
    return Nothing
    = (por definición de return)
    Nothing

(Just x) >>= return
    = (por definición de >>=)
    return x
    = (por definición de return)
    Just x
```

Podremos encontrar más ejemplos de usos de las mónadas en el Apéndice A para la programación de acciones mediante la mónada `IO`.

9 CONCLUSIONES

Este TFG ha puesto de manifiesto, en nuestra opinión, la importancia de realizar un estudio formal y en profundidad, desde el punto de vista matemático, de la Teoría de Categorías, para así poder comprender en toda su extensión los conceptos y patrones que intervienen en los lenguajes de programación funcionales, particularmente en Haskell, como por ejemplo las mónadas y los functores. Es conveniente, para ello, haber adquirido en primer lugar una buena formación y dominio en el conocimiento de las estructuras algebraicas (monoides, posets, grupos, anillos, cuerpos, espacios vectoriales, álgebras, etc.) que permita alcanzar un grado de abstracción suficiente para poder comprender, sin dificultad, los conceptos abstractos de la Teoría de Categorías y sus aplicaciones.

La realización de este trabajo ha resultado enriquecedora por varios motivos. En primer lugar, porque ha dado pie a aprender un lenguaje de programación nuevo, que opera fundamentalmente de forma distinta a los lenguajes imperativos a los que estamos acostumbrados. En segundo lugar, todo ello ha motivado el estudio de la Teoría de Categorías, un campo matemático genuinamente interesante en mi opinión y que me ha hecho ganar una nueva perspectiva a la hora de estudiar matemáticas (i.e., el punto de vista de las categorías). En tercer y último lugar, el haber adquirido estos conocimientos (Haskell y sus fundamentos matemáticos) amplía y mejora mi perfil profesional, aportándome más capacidad de realizar tareas de índole matemática y no solo de desarrollo de software. Se podría afirmar que Haskell ocupa una posición única entre la academia y la industria y, además, que este lenguaje destaca por una combinación de investigación de vanguardia y tecnología probada y comprobada en el tiempo.

En este trabajo se han abarcado los conceptos necesarios para llegar a entender las mónadas: la categoría, el functor, las transformaciones naturales y las adjunciones. Sin embargo, esto no constituye en absoluto la totalidad de la Teoría de Categorías como campo de estudio matemático. A continuación proponemos una serie de temas y nociones adyacentes a los estudiados en este trabajo que podrían extender algunas de las secciones presentes y constituir un posteriores estudios.

9.1. Estudios futuros

- Teoría de Categorías de orden superior.

  A lo largo del trabajo hemos expuesto lo que se llaman 1-categorías. Es decir, categorías que consisten en objetos y morfismos entre ellos (1-morfismos). Sin

embargo, podemos generalizar esta noción para manejar hasta *n-morfismos*. En ese caso se obtiene una *n-categoría*. Por ejemplo, una 2-categoría es una categoría que consiste en objetos, 1-morfismos entre objetos, y 2-morfismos entre 1-morfismos. Un ejemplo particular de una 2-categoría es una categoría que hemos visto ya: **Cat**. En ella, los objetos son categorías, los 1-morfismos los functores entre ellas, y los 2-morfismos son las transformaciones naturales entre dichos functores.

Para exponer dicha generalización necesitaríamos, en primer lugar, las noción de conjunto simplicial y, en segundo lugar, el de Teoría de Categorías Modelo. Por supuesto, ambos conceptos quedan con creces fuera del alcance de este trabajo. Sin embargo, [8] y [20] parecen un buen punto de partida.

- Álgebras para mónadas. *Lentes* en Haskel.

  Tras exponer las mónadas en el Capítulo 8, el siguiente paso sería estudiar las álgebras sobre las mismas. En pocas palabras, un álgebra para una mónada T en $\mathbf{C}$ es un objeto $A \in \mathbf{C}$ junto con un morfismo

  $$a : TA \longrightarrow A$$

  de tal forma que los siguientes diagramas conmutan:

  $$\begin{array}{ccc} A & \xrightarrow{\eta} & TA \\ & \searrow^{id} & \downarrow{\scriptstyle \alpha} \\ & & A \end{array} \qquad \begin{array}{ccc} T^2A & \xrightarrow{Ta} & TA \\ {\scriptstyle \mu}\downarrow & & \downarrow{\scriptstyle a} \\ TA & \xrightarrow{a} & A \end{array}$$

  Por otra parte, también podemos considerar una coálgebra para una (co)mónada. Esta consiste en una (co)mónada T en $\mathbf{C}$ junto con un objeto $A \in \mathbf{C}$ y un morfismo:

  $$c : A \longrightarrow TA,$$

  donde los diagramas anteriores conmutan dualmente.

  Esta noción de coálgebra sobre una comónada nos puede servir para modelizar [18, p. 383] las *lentes* (implementadas en la librería `lens` de Haskell), un tipo particular de referencia funcional. Las referencias funcionales son muy útiles a la hora de acceder y modificar partes de un valor que de otra manera resultaría incómoda o inconveniente.

- Correspondencia de Curry-Howard-Lambek y la teoría de tipos.

  En el Capítulo 5 hemos visto las categorías cartesianas cerradas, y se ha comentado de forma poco detallada cómo este tipo de categorías sirven para modelizar el cálculo lambda. La correspondencia de Curry-Howard, por otra parte, establece una equivalencia entre el cálculo lambda con tipos simples (Haskell implementa dicho sistema) y la lógica intuicionista, que posteriormente se extendería a la lógica de primer orden mediante la introducción de tipos dependientes. Así pues, se establece una correspondencia entre un modelo de computación (el cálculo

lambda con tipos simples) y un sistema de demostraciones (la lógica de primer orden).

Una posible extensión de este trabajo podría haber sido, por una parte, el desarrollo de esta correspondencia entre las categorías cartesianas cerradas, la teoría de tipos y la lógica (correspondencia de Curry-Howard-Lambek), y por otra parte, el desarrollo de la Teoría de Categorías en una lenguaje de programación funcional que, al contrario que Haskell, sí implemente tipos dependientes. Un ejemplo de dicho lenguaje es Agda, que es además un asistente de demostraciones que se basa en el paradigma de *proposiciones-como-tipos*. Es decir, un programa correcto en Agda es una demostración matemática correcta.

Por lo que respecta al estudio y desarrollo de la correspondencia de de Curry-Howard-Lambek, el libro *Type theory and functional programming* [22] parece ser un buen comienzo. Por otra parte, todavía no se ha llegado a una formalización estándar de la Teoría de Categorías en Agda [10]. A pesar de ello, la librería [5], si bien no es la librería estándar (todavía), podría servirnos para mostrar la implementación formal de la Teoría de Categorías en un lenguaje de programación funcional con tipos dependientes.

A INTRODUCCIÓN A HASKELL

A.1. Declaración de variables y funciones

En Haskell, la declaración de variables (inmutables, en el sentido matemático) se hace convencionalmente declarando la firma de tipos seguida de la expresión que la define:

```
valor :: Int
valor = 10
```

Las funciones se declaran también de la misma forma:

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

No es necesario declarar la expresión con la firma de tipos incluida debido a que Haskell puede inferir los tipos a partir de la definición de la misma. Sin embargo, es recomendable hacerlo para evitar que el compilador llegue a inferir tipos que no son los esperados debido a una mala implementación de la definición de la función.

Asimismo, podemos introducir definiciones de expresiones de forma local dentro de una función mediante las siguiente palabras reservadas: `where` y `let ... in`. El uso de una u otra solo presenta diferencias a nivel superficial, entre un estilo declarativo o un estilo expresivo respectivamente. Así pues, podemos definir una función que incluye `where` como:

```
productoArbitrario :: Int -> Int -> Int
productoArbitrario x y = x' * y'
  where x' = x * 2
        y' = y * 2
```

y `let ... in` como:

```
productoArbitrario :: Int -> Int -> Int
productoArbitrario x y = let x' = x * 2
                             y' = y * 2
                         in x' * y'
```

A.1.1. Funciones *currificadas*

Nota A.1. *"Currificada" del inglés* curried.

En el ejemplo anterior de la implementación de la función `producto`, a la función se le pasan dos parámetros de tipo `Int`, y devuelve un valor de tipo `Int`. No obstante, la función anterior podría reescribirse como una a la que se le pasa un solo parámetro, una tupla `(Int, Int)`, y devuelve un valor de tipo `Int`:

```
producto' :: (Int, Int) -> Int
producto' (x, y) = x * y
```

La diferencia entre estas dos implementaciones de la función `producto` pueden parecer a simple vista no demasiado significativa.

No obstante, las funciones de orden superior, como sería la función `producto`, se pueden beneficiar de la evaluación perezosa por parte del compilador. Como hemos explicado en la Sección 5.1, existe una correspondencia de tipos entre funciones como:

```
f :: (a, b) -> c
```

y funciones como:

```
g :: a -> (b -> c)
```

(Los paréntesis que simbolizan la aplicación de funciones son asociativos por la derecha. Esto significa que los paréntesis empleados en el ejemplo anterior no son necesarios realmente, pero son añadidos para hacer más clara la siguiente explicación).

En el segundo caso, la función `g` que toma como parámetro un valor de tipo `a` y devuelve una función de tipo `b -> c` se dice que está *currificada*. Toda función *currificada* es de orden superior, y devuelve funciones hasta que todos los parámetros se hayan pasado.

Como se comentaba más arriba, gracias a la evaluación perezosa, Haskell puede llegar a no evaluar la función retornada de tipo `b -> c` si no es necesario.

A.1.2. Expresiones lambda

Una expresión lambda se puede describir como una función anónima, en el sentido en que toma parámetros y retorna un valor calculado mediante dichos parámetros y nada más. Así pues, la función `producto` de antes se puede escribir como:

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

De nuevo, aquí se puede apreciar cómo las funciones *currificadas* son funciones de orden superior, es decir, funciones que devuelven funciones.

A.1.3. Composición de funciones

Es habitual escribir funciones que toman uno o varios parámetros como una composición de funciones sin hacer referencia a los parámetros tomados. Por ejemplo, la función

```
esImpar :: Int -> Bool
esImpar x = (not (even x))
```

la podemos escribir como:

```
esImpar' :: Int -> Bool
esImpar' = not . even
```

Las funciones de este estilo se llaman de "punto libre" (*point free*). Asimismo, en este ejemplo podemos ver en funcionamiento del operador '.' de composición, definido mediante una expresión lambda de la siguiente forma:

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

Gracias al operador de composición podemos transformar funciones al estilo de punto libre. Y esto no solo se reduce a simples funciones booleanas, sino que también podemos crear una función que eleve al cuadrado, sume 10, y finalmente nos diga si es impar, todo eso sin hacer uso del parámetro pasado en ningún momento:

```
operacionesEncadenadas :: Int -> Bool
operacionesEncadenadas = not . even . (10+) . (^2)
```

Además, Haskell proporciona otro operador de composición, `$`, que nos da la posibilidad de evitar escribir paréntesis. Este, sin embargo, no difiere en funcionalidad del operador de composición mostrado anteriormente, pero ofrece una mayor riqueza sintáctica. Se define de la siguiente forma:

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

Veamos un ejemplo. El resultado de hacer:

```
(not . even) (1+1)
```

será el mismo que:

```
not . even $ 1+1
```

Es decir, `$` nos permite omitir los paréntesis de la expresión `1+1` y de todas las funciones (o composiciones de ellas) que le precedan.

A.1.4. Condicionales y Guardas

Podemos escribir expresiones condicionales en Haskell siguiendo la sintaxis del siguiente ejemplo:

```
signo :: Int -> Int
signo n = if n < 0 then -1
          else if n == 0 then 0 else 1
```

donde la rama del `else` debe estar siempre presente.

Asimismo, se pueden usar *guardas* para definir una función como una serie de expresiones lógicas con resultado del mismo tipo. Siguiendo el ejemplo anterior, se podría definir `signo` mediante guardas de la siguiente manera:

```
signo' :: Int -> Int
signo' n | n < 0      = -1
         | n == 0     = 0
         | otherwise  = 1
```

Aquí el símbolo '`|`' significa "tal que", al igual que en matemáticas, y la palabra reservada `otherwise`, aunque no siempre necesaria, es útil para definir la expresión resultante para cualquier caso no cubierto anteriormente.

A.1.5. Coincidencia de patrones

A la hora de construir una función es posible definir cada uno de sus casos a través de patrones: si el primer patrón coincide, la función con la definición de ese caso se ejecuta; si no, pasa al siguiente, y así sucesivamente. Definimos con '`_`' la expresión con la que Haskell puede hacer coincidir cualquier valor. Un ejemplo de esto podría ser la función `&&`:

```
() :: Bool -> Bool -> Bool
True  True = True
_  _ = False
```

Un aspecto a destacar de este ejemplo es que nos muestra que podemos definir una función infija colocando su nombre en la firma de tipos entre paréntesis.

Veamos ahora otros dos ejemplo de coincidencia de patrones en la definición de funciones:

```
primero :: (a, b) -> a
primero (a, _) = a

segundo :: (a, b) -> b
segundo (_, b) = b
```

Por último, veamos un ejemplo de esta utilidad en funciones que toman cadenas de caracteres como parámetro:

```
empiezaPorA :: [Char] -> Bool
empiezaPorA ('a': _) = True
empiezaPorA _        = False
```

(Veremos a continuación el significado del operador ':'.)

A.2. Tipos

Hasta ahora hemos estado utilizando la notación "`e :: T`", siendo `T` un tipo y `e` un valor, para referirnos a que el valor `e` tiene como tipo a `T`. Hemos visto además que tanto, por ejemplo, `Int` como `Int -> Int` son tipos válidos; uno hace referencia al tipo de todos los enteros y el otro al de las funciones que toman un entero y devuelven otro entero.

Los tipos primitivos implementados en Haskell son:

- `Bool`, valores lógicos.
- `Char`, caracteres.
- `String`, cadenas de caracteres.
- `Int` e `Integer`, enteros de precisión fija y arbitraria respectivamente.
- `Float` y `Doubles`, números flotantes de precisión simple y doble respectivamente.

1. Tipos tupla.

   Las tuplas son secuencias de componentes, y cada uno de ellos puede ser de un tipo diferente. Por ejemplo, la tupla

   ```
   (2, "char", (3.567, True)) :: (Int, String, (Float, Bool))
   ```

   es una tupla de aridad 3, donde `(Float, Bool)` comprende un solo tipo (es decir, el tipo de un par cuya primera componente es un `Float` y la segunda un `Bool`).

2. Tipos lista.

   Al contrario que las tuplas, las listas deben ser secuencias de elementos del mismo tipo. El tipo de todas las listas de un tipo T se escribe `[T]`. Por ejemplo,

   ```
   ["Uno", "Dos", "Tres"] :: [String]
   ```

   es una lista de cadenas de caracteres cuya longitud es 3.

   También podemos tener listas de listas, de tipo `[[T]]`:

```
[[1, 2], [1, 2, 3]] :: [[Int]]
```

donde las "sublistas" que la componen pueden ser de cualquier longitud.

Dos operadores importantes de listas son el de anteposición (':') y el de concatenación. El primero inserta en la cabeza de una lista un elemento (del mismo tipo de la lista). Por ejemplo:

```
insertar :: Int -> [Int] -> [Int]
insertar n ns = n:ns

> insertar 1 [2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Nota A.2. *Es importante remarcar el uso de notación en este ejemplo, que es el habitual en Haskell: usamos 'n', 'x', 'c', etc. para denotar un elemento de tipo Int, String, Char, etc., y usamos 'ns', 'xs', 'cs', etc. para denotar las listas de dichos tipos.*

Este operador también se puede usar de formar "invertida". Es decir, dada una lista, podemos extraer su primer elemento de la siguiente forma:

```
primerElemento :: [Int] -> Int
primerElemento n:ns = n

> primerElemento [1, 2, 3, 4, 5]
1
```

y también podemos extraer su cola de la siguiente forma:

```
cola :: [Int] -> [Int]
cola n:ns = ns

> cola [1, 2, 3, 4, 5]
[2, 3, 4, 5]
```

Con este operador ya explicado, podemos definir el operador de concatenación:

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

que obtiene como parámetros dos listas y devuelve su concatenación. Por ejemplo:

```
> [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]
```

A.2.1. Comprensión de listas

De la misma forma que utilizamos una expresión matemática como

$$\{x^2 \mid x \in \{1, 2, 3, 4, 5\}\}$$

para definir el conjunto discreto $\{1, 4, 9, 16, 25\}$, podemos hacerlo de una forma muy similar en Haskell, usando guardas y el operador `<-`, que significa "extraído de". Con ello podemos definir el siguiente *generador*:

```
> [x^2 | x <- [1..5]]
[1, 4, 9, 16, 25]
```

Algo a destacar de este ejemplo es el uso de la función de librería `[i..f]`, que nos permite generar una lista de números no necesariamente enteros del `i` al `f`. Veamos algunos ejemplos:

```
> [1..5]
[1, 2, 3, 4, 5]

> [1,1.5..3]
[1.0,1.5,2.0,2.5,3.0]
```

Es importante mencionar que los generadores son perezosamente evaluados por Haskell, luego se evaluarán los elementos de las listas que sean necesarios.

Veamos ahora unos ejemplos. En primer lugar, podemos colocar varias expresiones tras las guardas y que dichas expresiones dependan las unas de las otras.

```
> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

También podemos usar los generadores para redefinir ciertas funciones, como por ejemplo:

```
concatenar :: [[a]] -> [a]
concatenar xss = [x | xs <- xss, x <- xs]

> concatenar [[1, 2], [3, 4]]
[1, 2, 3, 4]
```

```
tamañoLista :: [a] -> Int
tamañoLista xs = sum [1 | _ <- xs]
```

```
factores :: Int -> [Int]
factores n = [x | x <- [1..n], n `mod` x == 0]
```

Nota A.3. *Rodear una función con ‘ permite que sea usada de forma infija a pesar de estar esta definida como prefija. Es decir,* `n ‘mod‘ x` *equivale a* `mod n x`*.*

Nota A.4. *Haskell ya proporciona funciones como* `primerElemento`*,* `cola`*,* `tamañoLista`*, etc. Estas son* `head`*,* `tail`*, y* `length`*. Las contiene el módulo* Prelude*, que es el único módulo que carga Haskell por defecto. Contiene declaraciones de clases, definiciones de funciones, tipos, . . .*

A.2.2. Declaración de tipos

Con el mecanismo `type` de Haskell podemos crear nuevos *sinónimos* de tipos. El ejemplo más obvio es el caso de las `Strings`:

```
type String = [Char]
```

Esta declaración es similar a la directiva `#define` en C. Podemos, además, parametrizar estos sinónimos con una o más variables de tipo. Por ejemplo, en el caso del tipo de una tabla de asociaciones, formada por pares `(clave, valor)`, podemos declarar el tipo:

```
type Tabla c v = [(c, v)]
```

donde podemos definir:

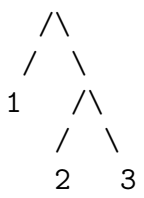
```
buscarTabla :: c -> Assoc c v -> v
buscarTabla c tabla = head [ v | (c', v) <- tabla, c' == c]
```

Ahora bien, si queremos definir tipos más complejos, necesitamos hacer uso de la declaración `data`. Esta nos permite crear tipos recursivos, tipos algebraicos (que no son más que tipos que resultan de la combinación de varios otros) y tipos propios que no son sinónimos. Veamos unos ejemplos:

```
data Arbol a = Hoja a | Nodo (Arbol a) (Arbol a)
```

El símbolo `|` se lee como un "o", y este tipo se puede interpretar como que un árbol es o bien un valor de tipo `Hoja a` (siendo `Hoja` el constructor), o bien un valor de tipo `Nodo (Arbol a) (Arbol a)` (y siendo `Nodo` de nuevo el constructor). De esta forma podemos definir un valor de tipo `Arbol` como:

```
miArbol :: Arbol Int
miArbol = Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
```



Un constructor de un tipo sin ningún argumento se llama `constante`. El tipo `Maybe`, visto extensamente a lo largo del trabajo, contiene una constante, `Nothing`, y un constructor, `Just`.

Por último, existe una última declaración de tipos: `newtype`. Funciona de manera similar a la declaración `data`, pero tiene las siguiente características:

- Solo se puede declarar un tipo mediante `newtype` si este tiene un solo contructor con un solo parámetro.
- Esta restricción permite considerar el tipo en sí y el tipo de su parámetro como el mismo. Lo que significa que, por ejemplo, dado

  ```
  newtype Nat = N Int
  ```

  se tiene que `N Int = Int`.
- Esta consideración solo se da durante la ejecución. Sin embargo, el *type checker* los considerará distintos.

A.2.3. Tipos polimórficos

Hemos visto varios ejemplos donde las funciones o valores toman parámetros de tipos no concretos como `(Int, Int), Bool, Int -> Char`, etc., sino que toman una *variable de tipo*, como `a` (o cualquier nombre que empiece por una letra minúscula). A esta clase de tipos, que contienen una o más varibles de tipo, se llaman *tipos polimórficos*.

Tomando como ejemplo la función `tamañoLista` anterior, vemos que puede tomar como parámetro una lista de cualquier tipo. Así pues, la firma de la función nos indica que el tipo de esta función es `[a] -> Int`, que es polimórfico.

Vamos a ver a continuación cómo podemos limitar el polimorifsmo de tipos a solamente ciertas *clases* de ellos.

A.2.4. Clases

Las clases son conjuntos de tipos de datos que deben implementar una o varias funciones concretas. Declarar un tipo como instancia de una clase se hace mediante

el mecanismo `class`, y solo es posible hacerlo en tipos declarados mediante `data` o `newtype`. El ejemplo más básico es el de la clase `Eq`:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

En este caso, para instanciar un tipo `a` en la clase `Eq`, solo es necesario definir el significado de `==` para dicho tipo.

También existe la clase `Ord`, definida como:

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max :: a -> a -> a

  min x y | x <= y    = x
          | otherwise = y
  max x y | x <= y    = y
          | otherwise = x
```

donde, por ejemplo, podemos instanciar el tipo `Bool`:

```
instance Ord Bool where
  False < True = True
  _ < _ = False

  b <= c = (b < c) || (b == c)
  b > c  = c < b
  b >= c = c <= b
```

Podemos, además, instanciar automáticamente tipos en clases con el mecanismo `deriving`. Esto significa que dado

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord, Show, Read)
```

el tipo `Maybe` formará parte de esas clases si y solo si el tipo del parámetro, `a`, ya lo está. Si no es así, lo haremos manualmente como se ha explicado en el ejemplo del tipo `Bool`.

Con las clases ya vistas, podemos definir los tipo *sobrecargados* (*overload*), que consisten en tipos cuya definición viene dada por una restricción de clase. Por ejemplo, podemos definir la función `valorAbsouto` como:

```
abs :: Num a => a -> a
abs a | a < 0     = -a
      | otherwise =  a
```

donde `a` es una variable de tipo que debe ser una instancia de la clase `Num`, es decir, la clase de los valores numéricos, y esto debe ser indicado en la firma de tipos de la función mediante la siguiente sintaxis: `Clase a => Tipo(a)`.

A.3. Funciones de orden superior

Haskell implementa en su librería una serie de funciones de orden superior que nos pueden resultar útiles para encapsular en forma de funciones patrones de programación o para tratar listas de forma eficiente.

Nota A.5. *Existe un buscador de funciones, ya sea por nombre o por firma de tipo, en las librerías de Haskell situado en la siguiente dirección:* ***https://hoogle.haskell.org/*** *.*

1. `take`, `drop`, y derivados. Podemos obtener sublistas gracias a las 4 funciones (entre otras) que veremos a continuación:

   En primer lugar, `take` es una función que toma los `n` primeros elementos de una lista:

   ```
   take :: Int -> [a] -> [a]

   > take 4 [1..10]
   [1, 2, 3, 4]

   > take 4 [1, 2]
   [1]

   > take (-1) [1..10]
   []

   > take 4 "Hola mundo"
   "Hola"
   ```

   También podemos, mediante `takeWhile`, tomar la sublista prefija que cumpla una condición, es decir, que se le pase una función `a -> Bool` y que devuelva los elementos tales que dicha función devuelva `True` para cada elemento de la lista.

   ```
   takeWhile :: (a -> Bool) -> [a] -> [a]

   > takeWhile (>=0) [-2, -1, 0, 1, 2]
   []

   > takeWhile (>=0) [0, 1, 2, -5, -10]
   [0, 1, 2]
   ```

En segundo lugar, podemos desechar los primer `n` elementos de una lista mediante `drop`:

```
drop :: Int -> [a] -> [a]

> drop 4 [1..10]
[5, 6, 7, 8, 9, 10]

> drop 4 [1, 2]
[]

> drop (-1) [1..10]
[1..10]

> drop 4 "Hola mundo"
" mundo"
```

Del mismo modo que antes, se puede obtener la sublista sufija en la que todos sus elementos cumplan una condición mediante `dropWhile`:

```
dropWhile :: (a -> Bool) -> [a] -> [a]

> dropWhile (>=0) [-2, -1, 0, 1, 2]
[-2, -1, 0, 1, 2]

> dropWhile (<0) [-2, -1, 0, 1, 2]
[0, 1, 2]
```

2. `zip` y `zipWith`. La función `zip` toma dos listas y devuelve una nueva lista de tuplas con pares formados a partir de los elementos de estas dos primeras listas.

```
zip :: [a] -> [b] -> [(a, b)]

> zip [1, 2] ['a', 'b']
[(1,'a'),(2,'b')]

> zip [] [1..5]
[]
```

Esta función se puede generalizar de forma en que tome un nuevo argumento, una función de tipo `a -> b`, que sea la función que se encargue de "empaquetarlas".

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith f [x1,x2,x3, ...] [y1,y2,y3, ...] =
        = [f x1 y1, f x2 y2, f x3 y3, ...]
```

Así pues, `zip` se puede definir mediante `zipWith` como:

```
zip xs ys = zipWith (,) xs ys
```

Veamos algunos ejemplos:

```
> zipWith (+) [1, 2, 3] [4, 5, 6]
[5, 7, 9]

> zipWith (++) [[1, 2], [5, 6]] [[3, 4], [7, 8]]
[[1, 2, 3, 4], [5, 6, 7, 8]]

> zipWith (:) [1, 4] [[2, 3], [5, 6]]
[[1, 2, 3], [4, 5, 6]]
```

3. `map` y `filter`. `map` es una aplicación particular de la función más general `fmap` vista en el Capítulo 4. En concreto, sobre el functor `[]`. Se puede definir de la siguiente forma:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Veamos algunos ejemplos:

```
> map (+1) [1, 2, 3, 4]
[2, 3, 4, 5]

> map esImpar [1,2,3,4]
[True, False, True, False]

> map reverse ["123", "abc"]
["321", "cba"]
```

Podemos limitar la función que toma `map` a que sea una condición. Así obtenemos `filter`, que devuelve los elementos de una lista que cumplan la condición dada.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

Algunos ejemplos de uso son:

```
> filter esImpar [1..10]
[1, 3, 5, 7, 9]

> filter (> 5) [1..10]
[6,7,8,9,10]

> filter (/= ' ') "Hola mundo"
"Holamundo"
```

4. `foldr` y `foldl`. Las siguientes funciones se encargan de encapsular un patrón de recursión muy común en las definiciones de funciones recursivas en Haskell.

   Veamos primeramente la función `foldr`. El patrón sobre listas que encapsula es este:

```
f [] = v
f (x:xs) = x # f xs
```

   Siendo `#` un operador cualquiera. En ejemplos como

```
sumaLista [] = 0
sumaLista (x:xs) = x + sumaLista xs

productoLista [] = 1
productoLista (x:xs) = x * productoLista xs

tamañoLista [] = 0
tamañoLista (x:xs) = 1 + tamañoLista xs
```

   podemos ver un patrón sobre el valor que tiene la función cuando se le pasa la lista vacía, `[]`, y el operador entre la el primer elemento de la lista y el resto. Con esto podemos definir `foldr` como:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

   Luego `foldr`, como hemos visto antes, asigna el valor `v` a la lista vacía `[]`, y a lo que no sea una lista vacía le asigna `f` tanto al primer elemento como al resto de la lista de forma recursiva.

   Ahora ya podemos reescribir los ejemplos anteriores usando está función de orden superior.

```
sumaLista = foldr (+) 0

productoLista = foldr (*) 1

tamañoLista = foldr (\_ n -> 1 + n) 0
```

   En resumen, podemos considerar `foldr` como una función que aplica el operador que se le pasa como argumento asociativamente por la derecha (`foldr`, *fold right*):

```
foldr (#) v [x0, x1, ..., xn] = x0 # (x1 # (... (xn # v) ...))
```

   Si el operador no es asociativo por la derecha, entonces el patrón de recursividad no se puede encapsular mediante `foldr`.

Esta no es, sin embargo, la única forma de encapsular un patrón de recursividad. La función `foldl` (*fold left*) tiene el mismo comportamiento que `foldr` pero asume que la asociatividad del operador es por la izquierda.

Partimos del siguiente patrón:

```
f v [] = v
f v (x:xs) = f (v # x) xs
```

Que se puede ver en ejemplos de definiciones de funciones como:

```
sumaLista = sumaLista' 0
  where
    sumaLista' v [] = v
    sumaLista' v (x:xs) = sumaLista' (v+x) xs

tamañoLista = tamañoLista' 0
  where
    tamañoLista' v [] = v
    tamañoLista' v (x:xs) = tamañoLista (v + 1) xs

invertirLista = invertirLista' []
  where
    invertirLista' ys [] = ys
    invertirLista' ys x:xs = invertirLista (x:ys) xs
```

Definimos pues `foldl` como:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Y ahora ya podemos ver los tres ejemplos anteriores reescritos mediante `foldl`:

```
sumaLista = foldl (+) 0

tamañoLista = foldl (\n _ -> n + 1) 0

invertirLista = foldl (\xs x -> x:xs) []
```

De nuevo, podemos considerar `foldl` como una función que aplica el operador que se le pasa como argumento asociativamente por la izquierda, y del mismo modo que antes, si dicho operador no es asociativo por la izquierda, `foldl` no podrá usarse.

A.4. Mónadas

En el Capítulo 8 hemos visto la definición canónica de mónada en Haskell mediante el operador >>=. Sin embargo, Haskell cuenta con una notación específica para simplificar

expresiones con dicho operador. Esta notación (llamada "notación `do`") nos permite pasar de expresiones como

```
m1 >>= \x1 ->
m2 >>= \x2 ->
...
mn >>= \xn ->
f x1 x2 ... xn
```

a expresiones "secuenciales" más similares a las que encontramos en programación imperativa como

```
do x1 <- m1
   x2 <- m2
   ...
   xn <- mn
   f x1 x2 ... xn
```

A continuación vamos a ver ejemplos de mónadas usando esta notación.

A.4.1. Mónada `IO`

Está mónada nos permite programar acciones interactivas: leer la entrada por teclado, escribir texto por pantalla, etc.

Cada expresión de tipo `IO a` es una acción que devuelve un valor de tipo `a`. Por ejemplo

```
getLine :: IO String
```

toma la entrada del teclado y devuelve una cadena de texto, o

```
putStrLn :: String -> IO ()
```

toma una cadena de texto y la imprime por pantalla.

Nota A.6. *No entraremos en detalles sobre la implementación real de `IO` y la consideraremos como un tipo primitivo más que ofrece Haskell.*

A la hora de construir un programa completo debemos tener en cuenta que el compilador buscará un valor (que llamamos normalmente `main`) de tipo `main :: IO ()` y lo ejecutará. Es entonces en `main` donde colocaremos nuestra secuencia de acciones `IO` que se querrán ejecutar. Un programa básico a modo de ejemplo es:

```
main :: IO ()
main = do putStrLn "¿Cómo te llamas?"
          nombre <- getLine
          putStrLn ("Hola, " ++ nombre)
```

donde obtenemos por pantalla una cadena de texto `nombre` e imprimimos posteriormente por pantalla "Hola, `nombre`".

Nota A.7. *La primera linea de la expresión* `do` *anterior podría reemplazarse por*

```
main = do _ <- putStrLn "¿Cómo te llamas?"
          ...
```

Sin embargo, como el tipo que retorna `putStrLn` *es* `IO ()` *(es decir, hace una acción y no devuelve nada) podemos omitir asignar el resultado a un valor cualquiera.*

Bibliografía

[1] Jirí Adámek, Horst Herrlich y George E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. 2005. URL: http://katmat.math.uni-bremen.de/acc/acc.pdf (visitado 23-05-2022).

[2] Steve Awodey. *Category theory*. Oxford university press, 2010.

[3] Henk Barendregt y Erik Barendsen. "Introduction to Lambda Calculus". En: (2000).

[4] Jan-Willem Buurlage. "Categories and Haskell". En: (2018). URL: https://raw.githubusercontent.com/jwbuurlage/category-theory-programmers/master/doc/categories_for_programmers.pdf.

[5] P.W.D. Charles. *agda-categories library*. https://github.com/agda/agda-categories. 2022.

[6] Samuel Eilenberg y Saunders MacLane. "General Theory of Natural Equivalences". En: *Transactions of the American Mathematical Society* 58.2 (1945), págs. 231-294.

[7] Peter J Freyd y Andre Scedrov. *Categories, allegories*. Elsevier, 1990.

[8] Greg Friedman. "An elementary illustrated introduction to simplicial sets". En: (2008). DOI: 10.48550/ARXIV.0809.4221. URL: https://arxiv.org/abs/0809.4221.

[9] HaskellWiki. *Functor-Applicative-Monad Proposal — HaskellWiki*. [Online; accessed 27-June-2022]. 2015. URL: https://wiki.haskell.org/index.php?title=Functor-Applicative-Monad_Proposal&oldid=60290.

[10] Jason Z. S. Hu y Jacques Carette. "Formalizing category theory in Agda". En: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, ene. de 2021. DOI: 10.1145/3437992.3439922. URL: https://doi.org/10.1145%2F3437992.3439922.

[11] John Hughes y Chalmers Hogskola. "Why Functional Programming Matters". En: (1999).

[12] Graham Hutton. *Programming in Haskell*. 2.ª ed. Cambridge Univeristy Press, 2016.

[13] James Iry. *A brief, incomplete, and mostly wrong history of programming languages*. [Online; accessed 03-July-2022]. 2009. URL: http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html.

[14] Saunders Mac Lane. *Categories for the working mathematician*. Vol. 5. Springer Science & Business Media, 2013.

[15] Ernest G Manes. *Algebraic theories*. Vol. 26. Springer Science & Business Media, 2012.

[16] Jean-Pierre Marquis. "Category Theory". En: *The Stanford Encyclopedia of Philosophy*. Ed. por Edward N. Zalta. Fall 2021. Metaphysics Research Lab, Stanford University, 2021.

[17] Conor McBride y Ross Paterson. "Applicative programming with effects". En: *Journal of functional programming* 18.1 (2008), págs. 1-13.

[18] Bartosz Milewski. *Category theory for programmers*. Bartosz Milewski, 2019.

[19] Eugenio Moggi. "Notions of computation and monads". En: *Information and computation* 93.1 (1991), págs. 55-92.

[20] Emily Riehl. *Categorical homotopy theory*. Vol. 24. Cambridge University Press, 2014.

[21] Emily Riehl. *Category Theory in Context*. Dover Publications, Inc, 2016.

[22] Simon Thompson. *Type theory and functional programming*. Addison Wesley, 1991.

[23] David A. Turner. "Some history of functional programming languages". En: *International Symposium on Trends in Functional Programming*. Springer. 2012.

[24] Philip Wadler. "Theorems for free!" En: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 1989, págs. 347-359.