Teoría de categorías aplicada a la programación funcional

Universidad de Murcia

Héctor Galbis Sanchis

October 2020

Declaración de originalidad

Héctor Galbis Sanchis, autor del Trabajo de Fin de Grado *Teoría de categorías aplicada a la programación funcional*, bajo la tutela de María Antonia Cárdenas Viedma y Leandro Marín Muñoz, declara que el trabajo que presenta es original, en el sentido de que ha puesto el mayor empeño en citar debidamente todas las fuentes utilizadas, y que la obra no infringe los derechos de autor de ninguna persona.

En Murcia, a 16 de junio de 2021

Fdo.: Héctor Galbis Sanchis

Resumen

A lo largo de la historia de la computación y la informática, grandes matemáticos como Alonzo Church o Alan Turing realizaron grandes contribuciones en este campo. Son considerados por muchos los precursores de la informática actual. Con la tesis de Church-Turing se demostró que el lambda cálculo era equivalente al modelo de las máquinas de Turing, por lo que fue usado posteriormente para crear los primeros lenguajes de programación funcionales. Cabe destacar LISP, el primer lenguaje funcional creado por John McCarthy. Por otro lado, Samuel Eilenberg y Saunders Mac Lane contribuyeron en el campo de las matemáticas con los conceptos de categorías y funtores, creando una rama del álgebra aplicable en casi cualquier campo de las matemáticas.

Hoy en día son varios los lenguajes basados en el lambda cálculo, pero sólo unos pocos son conscientes del poder que aporta la teoría de categorías, como OCaml, Haskell o F#. En este trabajo se muestran algunos de los conceptos que ofrece esta teoría y cómo aplicarlos en la programación funcional.

En primer lugar necesitamos una identificación de los lenguajes de programación con las denominadas categorías. Esto nos permitirá usar a nuestro favor muchos de los resultados matemáticos que conciernen a las categorías. Los tipos de datos y las funciones serán los principales protagonistas. Veremos que las estructuras de datos o el polimorfismo vienen respaldados por elementos matemáticos, así como la currificación de funciones o los *open multi-methods* para resolver el problema del *multiple dispatch*.

A continuación introduciremos los funtores y sus versiones homónimas en un lenguaje de programación. Un funtor es, después de la categoría, el concepto más importante de esta teoría. Mostraremos diversos ejemplos, estudiaremos el comportamiento de los funtores polinomiales y terminaremos con las transformaciones naturales.

Usando los funtores definiremos las F-álgebras y cómo nos ayudarán a

simular tipos de datos recursivos y a crear intérpretes usando los datos como representaciones de un lenguaje.

Finalmente estudiaremos las mónadas, un concepto capaz de generar numerosas herramientas que pueden ayudar al programador de muchas formas. Además, veremos cómo resolver el problema de usar las operaciones de entrada y salida en un lenguaje funcional puro como Haskell.

A lo largo del trabajo mostraremos cómo esta teoría es capaz de ayudarnos en lenguajes tan distintos como Haskell, C++ o Racket.

Extended abstract

The programming languages have evolved over the years. There are multiple type of langueges, but one of these types is the most known and used nowadays, the imperative languages. However, the functional paradigm can make the difference thanks to its closeness to mathematics. In fact, almost every programming language has the lambda calculus behind. In 1936, Alonzo Church introduced this theory and, meanwhile, Alan Turing was working in the halting problem. Later, they both proved the equivalence of lambda calculus and turing machines.

During 1942-1945, Samuel Eilenberg and Saunders Mac Lane introduced the concepts of categories and functors. They will be useful in functional languages as we will see in this project.

On the other side, John McCarthy created the first functional language, named LISP (LISt Processor). At the moment, there are multiple programming languages that take advantage of this paradigm, but only a few of them exploit the results of category theory.

In this project we will firstly identify the basic concepts of category theory with some programming elements. While we introduce these concepts and different results from the theory we will show how to apply them in our programming language. Finally, we study how to simulate recursive data types and how Haskell fixes the I/O operations problem.

The main concept of category theory is the category. A category is a triple $(O, A, \circ)$ where O is a collection of objects, A is a collection of morphisms or arrows that relate two objects and $\circ$ is a composition operation over two morphisms. The category also has two operations dom and cod, that return the domain and codomain of an arrow, for every object exists an identity morphism and the composition operation must be associative.

Our objective is to identify the elements of a programming language with the elements of a category. Initially we will use Haskell, as it is was built using mainly concepts from category theory. Haskell has a category structure, denoted by Hask, in which the objects are all the different types that we can create and the morphisms are the functions between types. The identity morphism is given by the function that given an element returns the element itself. Finally, the composition in Haskell can be easily obtained by the dot operator. This identification can be achieved analogously in the rest of programming languages.

Inside the collection of objects of a category we distinguish the initial and final objects. An object, denoted by 0, is initial if for every object b exists one morphism from 0 to b. The initial object usually corresponds to the empty set. In Haskell the empty type is called Void, and the morphism that goes from Void to another object is named absurd. On the other hand, a final object, denoted by 1, is final if for every object b exists one morphism from b to 1. Every type with just a single element is a final object in Hask. E.g, consider the Unit type denoted by '()' and the unit morphism.

We need more complex types in any program we do. Specifically, every composite type is a combination of products and coproducts. The product of two objects a and b is an object, denoted by $a \times b$, and two morphisms, called projections, from $a \times b$ to a and b respectively. The product is unique up to isomorphisms. This type corresponds to the cartesian product. The product of Int and Char is $(\mathrm{Int}, \mathrm{Char})$. The projections are the functions fst and snd that return the left and right element from a product respectively. On the other hand, the coproduct of two objects a and b is an object, denoted by $a + b$, and two morphisms from a and b to $a + b$ respectively. The coproduct is unique up to isomorphisms. This type comes to be the direct sum of two types. In Haskell, we can make use of the Either type in order to create new coproduct types. It is worth mentioning that direct sum is not the union operation. The difference lies when the two types are the same. The union of the type Int with itself is the Int type. However, the direct sum of Int with itself allows us to distinguish each Int type by means of a label. The 'left 3' integer is different to the 'right 3' integer. Instead of using $(,)$ or the Either type, we can use the keyword data to create new and more complex data types. These new types will be a combination of coproducts and products.

The functions have their own types, and the category theory has a concept for them. If for every two types exists their product, we can define the

exponent. The exponent of a and b is an object, denoted by b^a and one morphism eval from $b^a \times a$ to b that indicates the exponent can be evaluated, specifically, a function can be evaluated.

If for every pair of objects there is an exponent, is verified $(c^b)^a \equiv c^{a \times b}$. This is the curryfication isomorphism. This result states that a function receiving a pair of objects is equivalent to a function receiving each object one by one.

Another statement can be obtained if our category contains finite product and coproduct and if every product satisfies the distributive property over the coproduct. In this situation, is verified $c^{a+b} \equiv c^a \times c^b$. That is, two different functions that receives different types and return the same type, are equivalent to a single function that can distinguish between two types and returns the same type as the other two. This isomorphism is justifying the overloaded functions and the polymorphic types. In the last case, arises the *multiple dispatch* problem. Given multiple polymorphic elements, the language must choose the correct function depending on the underlying type each polymorphic element has. The visitor pattern is a well known technique to solve the *double dispatch* problem. There are more sophisticated solutions like *open multi-methods* that solves the whole *multiple dispatch* problem. In fact, Haskell solves it using a combination of this technique and pattern matching. Other languages like C++ only implement a solution for the *single dispatch* problem.

Reusing code is very important in any programming language. Polymorphic functions lets us do it without much effort. In category theory they correspond to natural transformations, but first we need the concept of a functor. A functor F is a pair of functions that maps every object and morphism respectively to another category. In Haskell, a functor will be each data type that instantiates the Functor class. It allows us to modify the content of the structure using the function $fmap$ that corresponds with the function that maps each morphism to another.

The examples we will show will be important for the following results. The identity functor maps every object and morphism to itself. In Haskell, every type with only one contructor with one argument can be interpreted as a identity functor. Another example is the constant functor, where it maps every object to a given one, and maps every morphism to the identity morphism. Every type with only one constructor with no arguments can be considered a constant functor. Also, we can consider the product functor, which has

a constructor with more that one argument. The coproduct functor can be obtained by a type with more than one contructor, and the composition of two functors is a type with one constructor that receives another functor. With these examples we can define the polynomial functor. Recursively, a polynomial functor is an identity functor, a constant functor, or given two polynomial functors, the product, coproduct or composition of said functors.

While functors are used to modify the content of a structure, a natural transformation is used to change between functors. A natural transformation between two functors is the collection o morphisms from one functor to another keeping the underlying type intact. Also, the definition gives us an isomorphism each natural transformation must satisfy. Each function between two functors can be implemented in two different ways. We can select the best of both options depending on the problem we are facing.

Types like List or Trees are some examples of recursive data types. A language may not support this types, but in this case we can make use of F-algebras to simulate them. We need to create a category from a given functor. So, we will look for the objects and morphisms that make up this new category.

Given a functor F, a F-algebra is a pair (a, α) where a is an object α is a morphism from $F(a)$ to a. The F-algebras will be the objects of the category we are looking for. On the other hand, an homomorphism between two F-algebras (a, α) and (b, β) is a morphism between the objects belonging to each F-algebra that verifies $h \circ \alpha = \beta \circ Fh$. These homomorphisms will be the morphisms in the new category. Now we want to find a special object in this new category. It will be contained inside the set of fixed points. A fixed point is a F-algebra whose morphism is an isomorphism. Using the Lambek's lemma, we know that an initial object in this new category must be fixed point. Thus, the initial object in this category is named the least fixed point. This object verifies that it exists one morphism from it to every F-algebra. Given a F-algebra (a, α), the unique morphism from the least fixed point is named the catamorphism and is denoted by $(|\alpha|)$. Using the property of homomorphisms is verified $(|\alpha|) = \alpha \circ f((|\alpha|)) \circ unfix$ where unfix is the inverse of the morphism of the least fixed point. The least fixed point corresponds to the false recursive data type and the catamorphisms are the functions that manipulates the data of that type. However, the least fixed point does not necessarily exists. At this point, we can recover the polynomial functors, because given a polynomial functor, the new category associated to that functor

always contains a least fixed point. In Haskell, we need a restrictive subset of polynomial functors. Only those polyomial functors that contains in his definition a constant functor are practical. Otherwise, a catamorphism from that type is destined to fall into an infinite loop.

One of the most important problems in functional programming is deal with input and output functions. We need them if we want an useful program, but they are not purely functional. The solution Haskell offers us is using monads. This tool allows us to do auxiliary work while we focus on the main flow of computation.

A monad is triple (T, μ, η) where T is functor, and both μ and η are natural transformations. Intuitively, the μ transformation 'combines' two elements of type monad. This combination performs the auxiliary computation we were talking about. Also, this 'combination' has to be associative. However, the η transformation ensure that there is a monad that does nothing by combining it with another monad. That monad can be seen as a 'neutral' element. In Haskell, μ and η are given by the functions join and return respectively.

The 'combination' of two monads is used to compose the auxiliary calculations. Now we want a way to compose a main computation taking advantage of the auxiliary one. We need the concept of Kleisli categories.

The Kleisli category associated to a monad $C = (T, \mu, \eta)$ is the category where the objects are the same from the original category, the morphisms from X to Y are the morphisms from X to $T(Y)$, the composition is given by $g \circ_T f := \mu_Z \circ T(g) \circ f : X \to T(Z)$ and the identity morphism for every object X is $id_X := \eta_X : X \to T(X)$.

This category tells us how to compose the main computation, but can be cumbersome. Instead of this, we can create some syntactic sugar making use of the extension operator. This operator is defined as $f^* := \mu_Y \circ T(f)$. The corresponding function in Haskell is named bind and the syntactic sugar applied to monads is called the do notation. It transforms the code into nested bind functions and make easier to compose monads.

Haskell uses monads for the I/O operations. Instead of having a function that reads from or writes to a file, every I/O operation is represented by a IO monad. The main computation allows us to manipulate the data, while the auxiliary computation compose these I/O operations. However, there are many different monads: to concatenate strings, to obtain continuations, to propagate error messages, etc.

In this project we have shown how category theory helps us to create powerful programming tools. From currified functions to simulate recursive data types. There are many concepts that we have not studied like bi-functors, tri-funtors, the Yonneda lemma, or lenses and traversals. We can ensure the potential of category theory is immense.

Índice general

1. Introducción **12**

2. Estado del arte **14**

3. Análisis de objetivos y metodología **15**

4. El lenguaje como categoría **16**
- 4.1. Categorías 16
- 4.2. Tipos de datos, funciones y composición 18

5. Tipos de datos algebraicos **20**
- 5.1. El vacío y la unidad 20
- 5.2. Tipos de datos compuestos 21
- 5.3. Funciones de primera clase 24
- 5.4. Currificación y múltiple dispatch 26

6. Funciones polimórficas **32**
- 6.1. Funtores 33
- 6.2. Funtores polinomiales 34
- 6.3. Transformaciones naturales 37

7. Tipos de datos recursivos e intérpretes **39**
- 7.1. F-álgebras 40
- 7.2. Intérpretes 47

8. Computación auxiliar y E/S **50**
- 8.1. Mónadas 50
- 8.2. Categoría de Kleisli y el operador de extensión 54
- 8.3. Notación do 56

8.4. Entrada y salida en Haskell. La mónada IO 58

9. Conclusión y líneas futuras **60**

Apéndices **61**

A. El lenguaje de programación Racket **61**
A.1. S-expresiones 61
A.2. Formas especiales 62
A.3. Procedimientos 63
A.4. Quotation 64
A.5. Macros 65

B. Mónadas en C++ **67**

C. Mónadas en Racket **72**

D. La mónada IO en Racket **76**

Bibliografía **79**

Capítulo 1

Introducción

La programación funcional es un paradigma de programación basado principalmente en la composición de funciones. En contraste con el paradigma imperativo, que aún domina en la industria del software actual, el paradigma funcional no permite asignaciones, por lo que cada función debe devolver el mismo resultado a partir de los mismos datos de entrada.

Sus orígenes se remontan a 1936 donde Alonzo Church introdujo el lambda cálculo, un sistema formal en el que las funciones son la única herramienta disponible. Más tarde lo extendió al lambda cálculo tipado, que será el germen de todos los lenguajes funcionales con tipado estático.

Mientras tanto, Alan Turing publicó su trabajo sobre el problema de la parada, demostrando así, que existen problemas irresolubles. Tras esto, Church y Turing demostraron la equivalencia del lambda cálculo y las máquinas de Turing en la denominada Tesis de Church-Turing.

Durante los años 1942-1945, Samuel Eilenberg y Saunders Mac Lane introdujeron los conceptos de las categorías y los funtores en su estudio sobre topología algebraica con el objetivo de estudiar los procesos que preservan las estructuras matemáticas.

En el año 1958, John McCarthy desarrolló el primer lenguaje de programación funcional denominado LISP (LISt Processor) con el objetivo de crear programas de inteligencia artificial, lo que desafortunadamente marcó a este tipo de lenguajes con la etiqueta de 'lenguajes para la IA'. Durante los siguientes años hasta la actualidad han surgido numerosos lenguajes de programación funcional, donde caben destacar diferentes sucesores de Lisp como Scheme, Clojure o Racket, y otros como OCaml, Haskell y F#. Todos

ellos se basan en el lambda cálculo, pero lenguajes como OCaml y Haskell han demostrado que la teoría de categorías puede ser una herramienta muy potente en el campo de la programación.

En los diferentes capítulos que componen este trabajo veremos cómo la teoría de categorías nos ofrece diversos conceptos y herramientas para obtener un mayor poder en el ámbito de la programación funcional. Haremos especial atención al lenguaje Haskell al convertirse en la referencia actual en cuanto a teoría de categorías se refiere. Sin embargo, mostraremos ejemplos de uso en otros lenguajes como C++ y Racket.

Capítulo 2

Estado del arte

La teoría de categorías nació con los matemáticos Samuel Eilenberg y Saunders Mac Lane en los años 1942-1945, por lo que estamos ante una rama bastante reciente en comparación con el resto de disciplinas matemáticas. Con el tiempo han aparecido estudios donde aplican la teoría de categorías a la informática y, más concretamente, a la programación funcional.

El lambda cálculo de Alonzo Church está altamente relacionado con la teoría de categorías como muestran los autores Andrea Asperti y Giuseppe Longo en su libro *Types and Structures* [6]. Muy importantes también son los lenguajes de programación basados en esta teoría, donde el ejemplo canónico es Haskell. Sin embargo, podemos decir que el interés generado estos últimos años por la aplicación de la teoría de categorías a la programación funcional es debido a referentes como Bartosz Milewski y su libro *Category Theory For Programmers* [4].

Las principales herramientas que nos ofrecen las categorías son los funtores y las mónadas. Existen herramientas más complejas como las ópticas, para las cuales existen librerías como *lens* en Haskell o *lens-lib* en Racket. Otros conceptos como las F-álgebras están relacionadas con los intérpretes y los tipos de datos recursivos, así como las F-coálgebras lo están con las conocidas macros de los lenguajes derivados de Lisp. También cabe destacar su aplicación en arquitectura del software y los *Open Multi-Methods* [10], donde lenguajes como C# ya los implementan.

Capítulo 3

Análisis de objetivos y metodología

Numerosos conceptos de la teoría de categorías pueden ser aplicados en la programación. Comenzaremos este trabajo buscando una relación entre un lenguaje de programación y los conceptos básicos que hay detrás de esta rama matemática. A medida que introducimos nuevos resultados trataremos de explicar cómo trasladarlos a la programación, pasando por algunos isomorfismos para justificar herramientas como la sobrecarga de funciones o la currificación. Realizaremos un breve vistazo a las funciones polimórficas y cómo son explicadas usando los funtores, uno de los conceptos más importantes de esta teoría. Utilizando todo esto culminaremos con los últimos dos temas mostrando dos grandes resultados de la programación funcional. Por un lado, conseguiremos simular tipos de datos recursivos mediante F-álgebras. Esto puede recordarnos al combinador Y capaz de simular funciones recursivas en el lambda cálculo. Por el otro lado, estudiamos la solución que plantea Haskell para implementar la E/S, las mónadas.

Ha sido necesario para la realización de este trabajo una lectura exhaustiva de la bibliografía junto con un proceso de síntesis de la información. Además, todo el código presente en el trabajo es correcto y ha sido probado, exceptuando ciertas partes por motivos didácticos, donde se avisará de ello.

Capítulo 4

El lenguaje como categoría

La teoría de categorías es una de las ramas más abstractas de las matemáticas, por lo que muchos resultados de esta teoría pueden ser usados en el resto de campos. Tanto es así, que podemos usarla en el campo de la programación. Sin embargo, necesitamos un contexto con el que comenzar trabajar.

En este primer capítulo analizaremos los elementos de un lenguaje de programación para identificarlos con los diferentes conceptos existentes en la teoría de categorías.

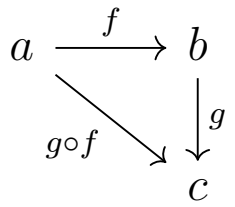
4.1. Categorías

El concepto principal de la teoría de categorías es, sorprendentemente, la categoría. Nuestro objetivo inicial es identificar una categoría adecuada al lenguaje de programación, por lo que es imprescindible conocer la definición de la misma.

Definición 4.1. *Una categoría es una terna* $(O, A, \circ)$ *donde:*

- O *es una colección de* **objetos** *denotados por* $a, b, ..., A, B, ... \in O$.
- A *es una colección de* **morfismos** *(flechas) denotados por* $f, g, ... \in A$ *que relacionan dos objetos, por ejemplo,* $f : a \to b \in A$.
- *Dos operaciones* **dom** *y* **cod**, *que devuelven los objetos dominio y codominio de una flecha, por ejemplo, si* $f : a \to b$, $dom(f) = a$ *y* $cod(f) = b$.

- *Existe una flecha identidad* $\mathbf{id_a} : a \to a$ *para cada objeto* $a \in O$.
- $\circ : A \times A \to A$ *es una operación (***composición***) que recibe dos flechas* f, g *tales que* $cod(f) = dom(g)$ *y retorna otra flecha tal que* $dom(f \circ g) = dom(g)$ *y* $cod(f \circ g) = cod(f)$.
- *Para cada* $f, g \in A$ *tales que* $cod(f) = b = dom(g)$ *se verifica* $id_b \circ f = f$ *y* $g \circ id_b = g$.
- *Para cada* $f, g, h \in A$ *tales que* $dom(f) = cod(g)$ *y* $dom(g) = cod(h)$ *se verifica* $(f \circ g) \circ h = f \circ (g \circ h)$.



Ejemplo 4.2. *Uno de los ejemplos más importantes es la categoría Set, cuyos objetos son los conjuntos y los morfismos son las funciones entre ellos. Hay que destacar que esta categoría es la más conocida matemáticamente hablando, por lo que los resultados más importantes suelen estar ligados a Set.*

Otro concepto importante es el de la **dualidad**. Dada una categoría, podemos invertir el sentido de todos los morfismos. Esta categoría nueva se le denomina la categoría dual.

Definición 4.3. *Sea C una categoría.* $\mathbf{C^{op}}$ *es la categoría que contiene los mismos objetos que C y existe una correspondencia entre los morfismos de C y los de* C^{op} *de forma que:*

$$f \in C \equiv f^{op} \in C^{op}$$

$$dom(f^{op}) = cod(f), cod(f^{op}) = dom(f)$$

$$f^{op} \circ g^{op} \equiv g \circ f$$

4.2. Tipos de datos, funciones y composición

Usaremos inicialmente el lenguaje Haskell, ya que se ha ido construyendo usando principalmente la teoría de categorías. La categoría que podemos formar a partir del lenguaje se denomina **Hask**. El conjunto de objetos de esta categoría la forman todos los tipos de datos diferentes que se pueden crear. Es decir, tipos como Int, Bool o String serán objetos en la categoría Hask. Los morfismos serán las funciones del lenguaje, los elementos que reciben unos datos de cierto tipo y devuelven un resultado con otro tipo concreto.

```
positive :: Int -> Bool
positive x = x > 0
```

El morfismo identidad viene dada por la función con su mismo nombre que devuelve la propia entrada.

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

Observamos que para la función identidad estamos usando un tipo genérico a, por lo que estamos definiendo la identidad para cada tipo en Haskell, es decir, para cada objeto de Hask.

Por último, la composición en Haskell se obtiene fácilmente utilizando el operador '.'.

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

suma2 :: Int -> Int
suma2 = suma1 . suma1

suma2 5 -- Resultado: 7
```

Las funciones que podemos definir en cualquier lenguaje de programación se comportan de igual manera que las funciones matemáticas, por lo que las propiedades que se establecen en la definición de categoría se van a verificar siempre.

En otros lenguajes podemos hacer esta identificación de forma análoga. En C++, que también es un lenguaje de tipado estático, basta tener como objetos los diferentes tipos de datos que podemos tener y como morfismos las funciones que manipularán estos datos.

```cpp
// Funcion identidad
template<typename T>
T id(T x){
    return x;
}

// Funcion composicion
template<typename F, typename G>
auto compose(F&& f, G&& g){
    return [=](auto x){return f(g(x));};
}
```

Otros lenguajes como Racket tienen tipado dinámico, pero esto no es un problema para poder usar la teoría de categorías. Aunque las variables no tengan un tipo establecido, sí lo tienen los datos que manejamos.

```racket
; Funcion identidad
(define (id x) x)

; Funcion composicion
(define (compose f g)
    (lambda (x) (f (g x))))
```

Capítulo 5

Tipos de datos algebraicos

Cuando hablamos de tipos de datos realmente estamos hablando de un conjunto de valores. Mediante herramientas como las estructuras de datos o la herencia podemos crear conjuntos de valores más complejos. Vamos a ver que conceptualmente estamos usando operaciones algebraicas sobre conjuntos como lo son el producto o la suma directa. Veremos también cómo representar de forma teórica las funciones de primera clase y obtendremos resultados importantes que derivarán en potentes herramientas de programación.

5.1. El vacío y la unidad

De entre el conjunto de objetos que forman parte de una categoría podemos destacar (si existen) los objetos iniciales y finales.

Definición 5.1. *Sea C una categoría. Un objeto, que denotaremos por 0, es inicial si para cada objeto $b \in Ob_C$ existe un único morfismo $f \in C[0, b]$, donde $C[0, b]$ es el conjunto de morfismos que van de 0 hacia b.*

En *Set* (la categoría de conjuntos) el objeto inicial viene dado por el conjunto vacío, aunque para entender bien esto hace falta usar el concepto de función desde el punto de vista de conjuntos. Es decir, una función f es una terna $f = (M, D, C)$ donde D es el dominio de la función, C es el codominio, y M es el conjunto de pares de elementos $(x, f(x))$. De esta forma para cada conjunto C, la única función del conjunto vacío a C es

$$h = (\emptyset, \emptyset, C)$$

De la misma forma podemos definir el objeto inicial en cualquier lenguaje de programación. En Haskell tenemos el tipo de dato Void que no contiene ningún valor. Así, la única función existente entre Void y cualquier otro tipo es la función absurd.

```haskell
-- Definicion de Void sin constructores
data Void

-- Definicion de la funcion absurd
absurd :: Void -> c
```

Hay que destacar que al ser Void un tipo sin valores es imposible ejecutar la función absurd.

Definición 5.2. *Sea C una categoría. Un objeto, que denotaremos por 1, es terminal si para cada objeto $b \in Ob_C$ existe un único morfismo $f \in C[b, 1]$, donde $C[b, 1]$ es el conjunto de morfismos que van de b hacia 1.*

Apoyándonos de nuevo en la categoría *Set*, sabemos que cualquier conjunto que sólo contenga un único elemento es un objeto final. En Haskell este objeto suele identificarse con el tipo de dato Unit, que se representa mediante '`()`' y cuyo único elemento también es '`()`'. El único morfismo entre cualquier tipo de dato y Unit es la función unit.

```haskell
-- Definicion de la funcion unit
unit :: c -> ()
unit x = ()
```

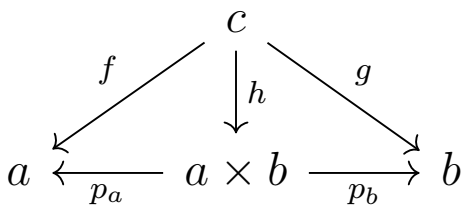
5.2. Tipos de datos compuestos

En cualquier lenguaje de programación necesitamos herramientas que nos permitan crear tipos de datos compuestos a partir de otros más primitivos. En C++ disponemos de las clases y la herencia, mientras que en Racket podemos usar una combinación de listas y símbolos. Sin embargo, todas estas soluciones tienen implícitas las nociones de productos y coproductos como veremos más adelante.

Definición 5.3. *Sea C una categoría y $a, b \in Ob_C$. El producto de a y b es un objeto, denotado por $a \times b$, y dos morfismos $p_a : a \times b \to a$, $p_b : a \times b \to b$, llamados proyecciones, tales que para cada objeto $c \in Ob_C$ y cada $f \in C[c, a]$ y cada $g \in C[c, b]$, existe un único morfismo $h \in C[c, a \times b]$ tal que se verifica:*

$$p_b \circ h = g$$

$$p_a \circ h = f$$



Aunque la definición parezca un poco engorrosa estamos ante el producto cartesiano que a menudo se usa en la categoría *Set*. En Haskell la mejor representación del producto viene dada por el tipo (a, b) donde a y b son dos tipos cualquiera, y las proyecciones se corresponden con las funciones `fst` y `snd`.

```haskell
-- Producto de un entero y un caracter
prod :: (Int,Char)
prod = (3,'x')

fst prod -- Resultado: 3
snd prod -- Resultado: 'x'
```

En C++ podemos crear el producto de 2 tipos de la siguiente manera:

```cpp
// Producto de dos tipos
template<typename A, typename B>
struct product{
    A v1;
    B v2;
};
```

```
7
8  // Proyecciones
9  template<typename A, typename B>
10 A fst(const product<A,B>& p){
11     return p.v1;
12 }
13
14 template<typename A, typename B>
15 B snd(const product<A,B>& p){
16     return p.v2;
17 }
```

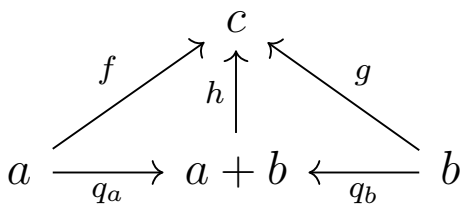
En Racket el producto, más conocido como par, es la principal estructura del lenguaje. Podemos crearla con la función `cons` y sus proyecciones son las funciones `car` y `cdr`.

```
1 (define p (cons 3 4))
2 (car p) ; Resultado: 3
3 (cdr p) ; Resultado: 4
```

Definición 5.4. *Sea C una categoría, y $a, \in Ob_C$. El coproducto de a y b es un objeto, denotado por $a+b$, y dos morfismos $q_a : a \to a+b$, $q_b : b \to a+b$ tales que, para cada $f \in C[a,c]$ y $g \in C[b,c]$, existe un único morfismo $h \in C[a+b,c]$ tal que se verifica:*

$$h \circ q_a = f$$

$$h \circ q_b = g$$



El coproducto viene a ser la suma directa de subconjuntos en *Set*. En casi todos los lenguajes de programación con tipado estático se obtienen

nuevos tipos de datos más complejos usando una combinación de productos y coproductos:

```
1 data MiTipo a b c = Nada | Prod a b | SuperProd a b c
```

Si interpretamos un constructor sin argumentos como una función con dominio el tipo `Unit`, en este ejemplo se observa que el tipo `MiTipo` se puede interpretar como un coproducto de `Unit`, $a \times b$ y $a \times b \times c$. Es decir, podemos escribir, con la notación que hemos definido hasta ahora, que:

$$MiTipo = 1 + a \times b + a \times b \times c$$

Fijándonos en el lenguaje C++, mientras que el producto se correspondía con estructuras de datos, los coproductos vendrán dados por la herencia. La relación 'ser hijo de' se puede interpretar como un morfismo que va del tipo 'hijo' hacia el tipo 'padre'. En efecto, cuando construimos un objeto de un tipo 'hijo', el propio constructor es una función que dados unos datos de entrada nos devuelve un objeto de tipo 'padre'.

Para finalizar esta sección cabe mencionar la siguiente definición que usaremos más adelante.

Definición 5.5. *Una categoría C es* **Cartesiana** *(C es una CC) si:*

- *Contiene un objeto terminal.*
- *Para cada par de objetos $a, b \in Ob_C$ existe su producto categórico ($a \times b$, $p_a : a \times b \to a$, $p_b : a \times b \to b$)*

Como ya hemos visto en los ejemplos anteriores los lenguajes que tratamos son categorías cartesianas. Pero aún podemos pedir más, y lo veremos en la siguiente sección.

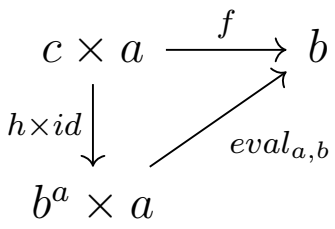
5.3. Funciones de primera clase

Hemos visto que combinando tipos primitivos mediante productos y coproductos somos capaces de obtener tipos más complejos. Sin embargo, uno de los elementos más importantes de la programación funcional es la función. Vamos a ver ahora cómo formalizar en la teoría de categorías lo que es una función e introduciremos algunos resultados importantes sobre éstas.

La función en la programación funcional es un dato más, como el resto que usamos normalmente. En particular, deben ser de primera clase, es decir, podemos pasarlos como argumentos y poder devolverlos como resultado de otra función. Por tanto tendrá su propio objeto en cualquier categoría. Estos objetos serán los exponenciales.

Definición 5.6. *Sea C una categoría cartesiana y $a, b \in Ob_C$. El exponente de a y b es un objeto, denotado por b^a y un morfismo $eval_{a,b} : b^a \times a \to b$, tal que para cualquier morfismo $f : c \times a \to b$, existe un único $h : c \to b^a$ tal que se verifica:*

$$eval_{a,b} \circ (h \times id) = f$$



En Haskell los exponenciales son fácilmente reconocibles por tener una flecha ($\to$) en la descripción del tipo.

```haskell
-- Definicion de un exponencial que recibe otro exponencial
litmap :: (Int -> Bool) -> [Bool]
litmap f = map f [-4,5,7,-3,0]

litmap positivo -- Resultado: [False,True,True,False,False]
```

La función *eval* que menciona la definición del objeto exponencial es el hecho de poder ejecutar las funciones. Se puede observar de forma más clara en el lenguaje Racket, donde dicha función *eval* se corresponde con encerrar entre paréntesis nuestro objeto exponencial.

```racket
(define (positivo x)
    (> x 0))
```

```
3
4 ; Usamos lambda para crear un exponencial y lo evaluamos
5 ((lambda (f) (map f '(-4,5,7,-3,0))) positivo)
```

Nos va a interesar poder crear un exponencial siempre que queramos. Es decir, nos interesa que para cada par de objetos siempre exista el exponente de dichos objetos. Esta idea es lo que nos lleva a la siguiente definición.

Definición 5.7. *C será una categoría cartesiana cerrada (CCC) si se verifica:*

- *C es una categoría cartesiana*
- *Para cada par de objetos $a, b \in Ob_C$, existe un exponencial $b^a \in Ob_C$.*

5.4. Currificación y múltiple dispatch

A continuación veremos algunas propiedades sobre los objetos exponenciales. En particular se establecerán algunos isomorfismos que podemos aprovechar para obtener potentes herramientas.

Proposición 5.8. *Sea C una categoría cartesiana cerrada (CCC). Se verifica:*

$$(c^b)^a \equiv c^{b \times a}$$

Este resultado se denomina isomorfismo de currificación. Lo que nos dice es que es equivalente tener una función que recibe dos argumentos (derecha), que tener una función que al recibir un argumento devuelva otra función que pueda recibir un segundo argumento (izquierda).

Si una función puede asimilar sus argumentos de uno en uno, de forma que retorne una nueva función que pueda recibir el resto, estaremos hablando de una función currificada. En Haskell todas las funciones están currificadas, lo que nos da bastante poder a la hora de crear nuevas funciones.

```
1 -- Definicion tradicional
2 suma1 :: Int -> Int
3 suma1 x = x + 1
4
5 map suma1 [1,1,1,1,1] -- Resultado: [2,2,2,2,2]
```

```haskell

-- Usando la currificacion
map ((+) 1) [1,1,1,1,1] -- Resultado: [2,2,2,2,2]
```

Otros lenguajes como C++ o Racket no tienen funciones currificadas por defecto, pero podemos crear métodos para currificar funciones. Un ejemplo sencillo se puede crear en Racket.

```racket
(define (curry f)
  ; curry-aux: Funcion que, dados los viejos arguentos
  ; retorna una funcion que recibe nuevos argumentos
  ; para llamar, si es posible, a la funcion f uniendo
  ; viejos y nuevos argumentos
  (define (curry-aux old-args)
    (lambda new-args
     (define all-args (foldl cons old-args new-args))
     ; Cantidad valida de argumentos?
     (if (procedure-arity-includes? f (length all-args))
       ; True -> Se ejecuta la funcion
       (apply f (reverse all-args))
       ; False -> Retornamos otra funcion
       ; para seguir recolectando argumentos
       (curry-aux all-args))))
  (curry-aux '()))
```

De manera similar lo conseguimos en C++ (usando el estándar C++20).

```cpp
// Comprueba si una funcion sin argumentos se puede
   ejecutar
template<typename T>
concept Valid_function = requires (T f){
    {f()};
};

// Si se puede ejecutar, se ejecuta
template<Valid_function F>
auto curry(F&& f){
    return f();
```

```
}

// En otro caso retornamos una funcion para recolectar
// mas argumentos
template<typename F>
auto curry(F&& f){
   return [=](auto&&... os){
      return curry(
         [=](auto&&... ns) -> decltype(f(os...,ns...)) {
            return f(os...,ns...);
         }
      );
   };
}
```

Para finalizar el capítulo hablaremos del múltiple dispatch como resultado de otro isomorfismo que verifican los objetos exponenciales. Sin embargo, necesitaremos de una última definición.

Definición 5.9. *Una categoría C con productos y coproductos finitos, se dice que es distributiva si para todos $X, Y, Z \in Ob_C$ se verifica que el morfismo canónico de distributividad es un isomorfismo:*

$$X \times Y + X \times Z \equiv X \times (Y + Z)$$

El morfismo canónico es el único morfismo que viene dado por $X \times i_Y : X \times Y \to X \times (Y + Z)$ y $X \times i_Z : X \times Z \to X \times (Y + Z)$, donde $i_Y : Y \to Y + Z$ y $i_Z : Z \to Y + Z$ son las funciones inyección.

Se puede comprobar fácilmente que los lenguajes que estamos usando contienen categorías distributivas. Los productos y coproductos serán siempre finitos y dicho isomorfismo existe.

```
canonical :: Either (x,y) (x,z) -> (x,Either y z)
canonical (Left (a,b)) = (a,Left b)
canonical (Right (a,c)) = (a,Right c)

canonical_inv :: (x,Either y z) -> Either (x,y) (x,z)
canonical_inv (a,Left b) = Left (a,b)
canonical_inv (a,Right c) = Right (a,c)
```

Proposición 5.10. *Sea C una categoría cartesiana cerrada (CCC). Si C es distributiva, se verifica el siguiente isomorfismo:*

$$X^{Y+Z} \equiv X^Y \times X^Z$$

Hay que observar que, desde el punto de vista de la programación, este resultado nos indica que es equivalente tener dos morfismos diferentes con distintos tipos de entrada (derecha), que tener un sólo morfismo que pueda distinguir el tipo de entrada (izquierda).

```
type Vector2 = (Fractional,Fractional)
data Point = Point Vector2
data Circle = Circle Vector2 Fractional
data Square = Square Vector2 Fractional

-- Opcion 1
point_in_circle :: Point -> Circle -> Bool
point_in_circle p (Circle c r) = dist_2(p,c) < r

point_in_rect :: Point -> Rect -> Bool
point_in_rect p (Square c l) = dist_inf(p,c) < l/2

-- Opcion 2
data Figure = Point Vector2 | Circle Vector2 Fractional
   | Square Vector2 Fractional

point_in_figure :: Point -> Figure -> Bool
point_in_figure p (Circle c r) = dist_2(p,c) < r
point_in_figure p (Square c l) = dist_inf(p,c) < l/2
```

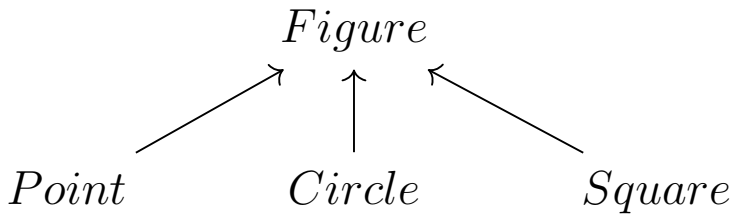
Claramente nuestro objetivo será buscar siempre un único morfismo que pueda distinguir entre los diferentes tipos de entrada. Un caso más interesnte es el de las funciones que deben realizar este tipo de decisiones para más de un argumento. Un ejemplo típico es el intersección de figuras.

```
type Vector2 = (Fractional,Fractional)
data Figure = Point Vector2 | Circle Vector2 Fractional
   | Square Vector2 Fractional

```

```haskell
intersection :: Figure -> Figure -> Bool
intersection (Point p) (Point q) = p == q
intersection (Circle c r) (Square d l) = -- ...
intersection (Point p) (Square d l) = -- ...
inter...
```

Como se puede observar en el ejemplo Haskell ofrece mucha facilidad para resolver este tipo de problemas. Hay que destacar que Haskell está siendo capaz de inferir el tipo concreto de un coproducto. Se está resolviendo el problema del múltiple dispatch.



En C++ este problema es mucho más complicado. Recordemos que el coproducto se corresponde con la herencia, por lo que tratamos de averiguar cómo implementar la siguiente función.

```cpp
struct Figure{ /* ... */ }

struct Point : Figure { /* ... */ }
struct Circle : Figure { /* ... */ }
struct Square : Figure { /* ... */ }

bool intersect(const Figure& f1, const Figure& f2){
   // ????????????????????
}
```

Lo que andamos buscando se denomina open multi-methods. Desgraciadamente C++ no los ha implementado aún, por lo que hay que recurrir a librerías externas, como lo son la librería yomm2 (de John Louis Leroy) o la librería omm. Un ejemplo ideal de open multi-methods en C++ sería el siguiente.

```
// Notese la palabra 'virtual'
bool intersect(virtual const Figure& f1, virtual const
   Figure& f2);

// Implementaciones de la funcion intersect
bool intersect(const Point& p, const Circle& c){
    // ...
}

bool intersect(const Square& s1, const Square& s2){
    // ...
}
```

Capítulo 6

Funciones polimórficas

Uno de los objetivos principales de la programación es reutilizar código, tener la máxima flexibilidad sin tener que escribir más de lo necesario. El polimorfismo nos ayuda a conseguirlo permitiendo que una función se comporte de diferente manera dependiendo de los datos de entrada que le proporcionemos.

Funciones como la identidad o las proyecciones de un producto que aceptan diferentes tipos de variables son un ejemplo claro de funciones polimórficas.

```
-- Acepta cualquier tipo
id :: a -> a
id x = x

-- Acepta productos de cualquier tipo
fst :: (a,a) -> a
fst (x,y) = x
```

También lo hemos visto en C++ mediante la sobrecarga de funciones.

```
bool intersect(const Point& p, const Circle& c);
bool intersect(const Square& s, const Point& p);
bool intersect(const Circle& c1, const Circle& c2);
```

La teoría de categorías nos permitirá aprovechar al máximo el polimor-

fismo usando uno de los conceptos más importantes de esta rama de las matemáticas, el funtor junto con las transformaciones naturales.

6.1. Funtores

Definición 6.1. *Sean C y D dos categorías. Un funtor $F : C \to D$ es un par de funciones $F_{ob} : Ob_C \to Ob_D$ y $F_{mor} : Mor_C \to Mor_D$ tales que, para cada $f : a \to b$, $g : b \to c$ en C:*

- $F_{mor}(f) : F_{ob}(a) \to F_{ob}(b)$
- $F_{mor}(g \circ f) = F_{mor}(g) \circ F_{mor}(f)$
- $F_{mor}(id_a) = id_{F_{ob}(a)}$

Observamos que un funtor F es una función que mapea cada objeto y morfismo de C a unos nuevos objetos y morfismos de D. Obviamente no podemos salirnos de nuestra categoría $Hask$ o la correspondiente a cada lenguaje de programación, por lo que los funtores que usaremos tendrán como codominio el propio dominio. Es decir, usaremos funtores $F : Hask \to Hask$.

Para crear un funtor sólo necesitamos un morfismo para cada tipo de dato y otro para cada función $f : a \to b$. Esto se consigue en Haskell, inicialmente, declarando cualquier tipo de dato:

```
1 data F a = ??
```

El constructor de este nuevo tipo de dato será la función que mapea cada objeto de $Hask$.

```
1 data F a = F a
```

Por último necesitamos la función que mapea cada morfismo. Esta función se denota en Haskell por fmap y es parte de la interfaz de la clase Functor.

```
1 class Functor f where
2    fmap :: (a -> b) -> f a -> f b
3
```

```haskell
-- Instanciamos F como funtor
instance Functor F where
   fmap f (F x) = F (f x)
```

Observa que fmap recibe dos argumentos, el morfismo y un objeto. Esto no contradice la definición, pues gracias a la currificación se verifica que `fmap f :: F a -> F b`. Es decir, el resultado es una función en la imagen del funtor.

Cabe destacar que la definición indica que necesitamos una función para los objetos y otra para los morfismos. Sin embargo, no hemos implementado niguna función en el sentido estricto.

```haskell
fob :: a -> b
```

En lugar de esto hemos dicho que la función es el constructor a un tipo de dato que representa a la imagen. Hay que destacar que el nombre que le demos a la imagen de una función nos es indiferente siempre y cuando el comportamiento sea idéntico. Es decir, en lugar de tener un funtor que nos devuelve un Int, obtenemos un $F(a)$, donde $F(a)$ se comportará igual que Int.

6.2. Funtores polinomiales

Es posible crear funtores más complejos utilizando más constructores y argumentos. A continuación vamos a ver ejemplos concretos de funtores hasta llegar a los funtores polinomiales que serán de especial interés en capítulos posteriores.

Ejemplo 6.2. *Sea C una categoría. El funtor identidad id_C se define como:*

- $id_C(a) = a, \forall a \in Ob_C$
- $id_C(f) = f, \forall f \in Mor_C$

En Haskell basta tener un constructor con un argumento y hacer que fmap deslice la función sobre el argumento del constructor.

```haskell
-- Funtor identidad
data Id a = Id a

instance Functor Id where
   fmap f (Id x) = Id (f x)
```

Ejemplo 6.3. *Sea C una categoría. El funtor constante Δ_c se define como:*

- $\Delta_c(a) = c, \forall a \in Ob_C$
- $\Delta_c(f) = id_c, \forall f \in Mor_C$

Observamos que independientemente del objeto de entrada la imagen siempre será la misma. Por esta razón, podemos obviar este argumento en el constructor. Es decir, un constructor sin argumentos se corresponde con el morfismo de objetos de un funtor constante.

```haskell
-- Funtor constante
data Const a = Const

instance Functor Const where
   fmap f Const = Const -- id Const == Const
```

Ejemplo 6.4. *Sea C una categoría y sean F y G dos funtores. El funtor producto $F \times G$ se define como:*

- $(F \times G)(a) = F(a) \times G(a), \forall a \in Ob_C$
- $(F \times G)(f) = F(f) \times G(f), \forall f \in Mor_C$

El funtor producto se corresponde con tipos de datos con constructores con dos o más argumentos. Hay que tener en cuenta que realizar el producto con un funtor constante equivale a no realizar el producto. Por eso en el siguiente ejemplo sólo se muestra el producto de dos funtores identidad.

```haskell
-- Funtor producto
data Prod a = Prod a a

instance Functor Prod where
   fmap f (Prod x y) = Prod (f x) (f y)
```

Ejemplo 6.5. *Sea C una categoría y sean F y G dos funtores. El funtor coproducto $F + G$ se define como:*

- $(F + G)(a) = F(a) + G(a), \forall a \in Ob_C$
- $(F + G)(f) = F(f) + G(f), \forall f \in Mor_C$

El coproducto de funtores se identifica con los tipos de datos con dos o más constructores.

```haskell
-- Funtor coproducto
data Coprod a = Const | Id a

instance Functor Coprod where
   fmap f Const = Const
   fmap f (Id x) = Id (f x)
```

Ejemplo 6.6. *Sea C una categoría y sean F y G dos funtores. El funtor composición $F \circ G$ se define como:*

- $(F \circ G)(a) = F(G(a)), \forall a \in Ob_C$
- $(F \circ G)(f) = F(G(f)), \forall f \in Mor_C$

Componer dos funtores no será otra cosa que utilizar un funtor existente en el constructor de un nuevo tipo de dato.

```haskell
-- Funtor ya existente
data G a = ...

instance Functor G where
```

```
5     fmap f ...
6
7  -- Nuevo funtor
8  data FoG a = F (G a)
9  instance Functor Coprod where
10    fmap f (F y) = F (fmap f y)
```

Con todos estos ejemplos ya tenemos los ingredientes suficientes para definir un funtor polinomial.

Ejemplo 6.7. *Sea C una categoría. Un funtor polinomial se define recursivamente como:*

- *El funtor identidad id_C es un funtor polinomial.*
- *El funtor constante Δ_a para todo $a \in Ob_C$ es un funtor polinomial.*
- *Si F y G son funtores polinomiales, entonces $F \times G$, $F + G$ y $F \circ G$ son funtores polinomiales.*

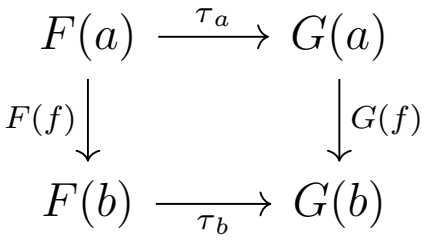
Ya disponemos de una batería bastante grande para crear tipos de datos nuevos. Más adelante veremos que los funtores polinomiales juegan un papel importante con los tipos de datos recursivos y nos abrirán las puertas a crear intérpretes básicos de forma sencilla.

6.3. Transformaciones naturales

Observando los ejemplos anteriores podemos interpretar los funtores como 'objetos' polimórficos a los que podemos cambiar su tipo subyacente. Las transformaciones naturales, en cambio, nos permitirán cambiar el funtor en sí.

Definición 6.8. *Sean $F, G : C \to D$ funtores. Entonces $\tau : F \to G$ es una transformación natural de F hacia G si:*

- $\forall a \in Ob_C$, $\tau_a \in D[F(a), G(a)]$
- $\forall f \in C[a, b]$, $\tau_b \circ F(f) = G(f) \circ \tau_a$



De la definición podemos decir que una transformación natural será una función polimórfica entre dos funtores. Pero más importante es la igualdad que la caracteriza, puediendo elegir dos formas de computar el mismo resultado.

```haskell
-- Transformacion natural entre [] y Maybe
car :: [a] -> Maybe a
car [] = Nothing
car [x:xs] = Just x

-- Queremos usar una funcion sobre el primer elemento de
-- la lista de los numeros naturales

-- Opcion 1
(car . (fmap (* 2))) [1..] -- Resultado: Just 2

-- Opcion 2
((fmap (* 2)) . car) [1..] -- Resultado: Just 2
```

Los funtores son una herramienta fundamental en la teoría de categorías y por eso la programación en Haskell gira continuamente alrededor de ellos. En los siguientes capítulos veremos construcciones basadas en estos conceptos, llegando a resultados bastante sorprendentes.

Capítulo 7

Tipos de datos recursivos e intérpretes

Hemos hablado sobre productos, coproductos y exponenciales, que juntos forman los tipos de datos algebraicos, e incluso hemos visto una notación para describirlos.

$$MiTipo = 1 + a \times b + a \times b \times c$$

```
1 data MiTipo a b c = Nada | Prod a b | SuperProd a b c
```

Pero también hemos visto ejemplos usando el tipo de dato lista '[a]'. Si intentamos describir con la notación anterior este tipo de dato veremos que es imposible. Y es que una lista tiene una cantidad variable de datos y eso se tiene que reflejar de alguna manera en la descripción. La solución está en describirlo como un tipo recursivo, y la clave se encuentra en la idea de que eliminando el primer elemento obtenemos otra lista. O dicho de otra forma, una lista estará formada por un elemento y otra lista.

$$L = a \times L$$

Sin embargo, las listas son finitas, por lo que deben de tener una última lista especial, que será una lista vacía. Por tanto, una lista será una lista vacía (un valor constante) o un elemento junto con otra lista.

$$L = 1 + a \times L$$

```haskell
data Lista a = Null | Cons a (Lista a)
```

Es ampliamente conocido por los programadores cómo se deben usar estas construcciones. Una función que reciba una lista deberá llamarse recursivamente sobre la lista resultante de eliminar el primer elemento.

```haskell
-- Recibimos una lista (que esta formada)
-- por un elemento x y otra lista xs.
sumatorio :: [Int] -> Int
sumatorio [] = 0
sumatorio [x:xs] = x + (sumatorio xs)
```

Lo que veremos en este capítulo es cómo podemos simular tipos de datos recursivos en el supuesto de estar usando un lenguaje que no los soporte.

Además, daremos un paso más y generaremos intérpretes de forma sencilla. Es más, crearemos en Racket un generador de intérpretes con tan sólo 10 líneas de código.

7.1. F-álgebras

El término fundamental que estará presente en toda esta sección es la F-álgebra. En Haskell se corresponderá con un paso en concreto de un proceso recursivo y trataremos de ver cómo crear un morfismo general capaz de contener toda la recursión usando estos elementos.

Definición 7.1. *Sea $F : C \to C$ un funtor (concretamente, un endofuntor). Una F-álgebra es un par (a,α) donde $a \in Ob_C$ y $\alpha : Fa \to a$ es un morfismo en C.*

```haskell
-- Funtor F
data F a = Empty | F a a

instance Functor F where
fmap f Empty = Empty
fmap f (F x y) = F (f x) (f y)

```

```haskell
-- F-algebra (Int,alpha)
alpha :: F Int -> Int
alpha Empty = 0
alpha (F x y) = x - y
```

Claramente estaremos interesados en funtores $F : Hask \to Hask$. Estos F-álgebras van a ser los objetos de una categoría que construiremos en breve, pero aún necesitamos lo que serán los morfismos entre las F-álgebras.

Definición 7.2. *Un homomorfismo entre dos F-álgebras (a,α) y (b,β) es un morfismo $h : a \to b$ tal que:*

$$h \circ \alpha = \beta \circ Fh$$

$$\begin{array}{ccc} Fa & \xrightarrow{\alpha} & a \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle h} \\ Fb & \xrightarrow[\beta]{} & b \end{array}$$

En este momento ya podemos definir la categoría que necesitamos.

Definición 7.3. *Para cada funtor $F : C \to C$ se define la categoría Alg_F donde sus objetos son las F-álgebras y sus flechas son los homomorfismos entre éstas.*

Dentro de esta categoría nos interesan los denominados puntos fijos.

Definición 7.4. *Un punto fijo de F será una F-álgebra (a,α) tal que α es un isomorfismo.*

Más aún, dentro del conjunto de puntos fijos es posible que exista un elemento aún más interesante, el menor punto fijo.

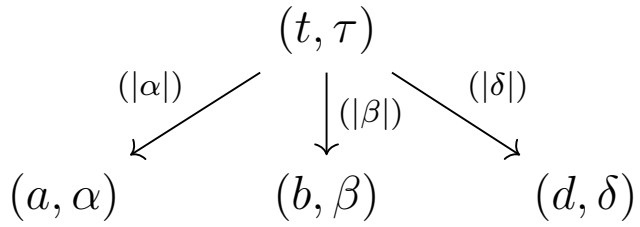
Definición 7.5. *Un menor punto fijo de F es una F-álgebra (a,α) que es un objeto inicial en la categoría Alg_F.*

Hay que observar que la definición no exige que la F-álgebra sea un punto fijo, y no es necesario, pues cualquier objeto inicial en Alg_F será un punto fijo, como demuestra el lema de Lambek.

Lema 7.6 (Lambek)**.** *Sea $F : C \to C$ un funtor. Si (t,τ) es un objeto inicial en Alg_F, entonces τ es un isomorfismo.*

Con todo esto, buscamos una categoría Alg_F a partir de un funtor $F : C \to C$ tal que contenga un menor punto fijo, es decir, debe contener una F-álgebra inicial (t,τ). Recordemos que si (t,τ) es inicial, debe existir un único morfismo que vaya de (t,τ) a cualquier otra F-álgebra (a,α).

Definición 7.7. *Sea (t,τ) una F-álgebra inicial. Para cualquier (a,α), el único morfismo entre (t,τ) y (a,α) es denominado un catamorfismo y se denota por ($|\alpha|$).*



Más concretamente, dada una F-álgebra (a,α) generaremos el catamorfismo ($|\alpha|$). Sin embargo, en ningún momento hemos dicho ni siquiera que este menor punto fijo exista. Por suerte, la siguiente proposición nos ofrece todas las respuestas. El resultado necesita el concepto de $\omega - colímites$, pero podemos ignorarlo sin problemas para lo que necesitamos.

Proposición 7.8. *Sea C una categoría. Si C tiene un objeto inicial 0 y $\omega - colímites$, y el funtor $F : C \to C$ preserva $\omega - colímites$, entonces F posee una F-álgebra inicial.*

Necesitamos saber qué funtores en $Hask$ tienen la cualidad descrita en la proposición y así tener asegurada la existencia de un menor punto fijo. Aunque no entendamos el resultado anterior disponemos de un corolario que nos facilita muchísimo el trabajo.

Corolario 7.9. *Todo funtor polinomial $F : Set \to Set$ posee una F-álgebra inicial.*

Aunque el resultado utilice concretamente la categoría Set, sabemos cómo crear versiones equivalentes en Haskell como ya hemos visto en el capítulo anterior. Por tanto, como $Hask$ posee un objeto inicial (Void) deducimos que cualquier funtor polinomial $F : Hask \to Hask$ genera una categoría Alg_F que posee un menor punto fijo.

Este menor punto fijo aún nos es desconocido, pero la demostración de la proposición anterior nos da una manera de construirlo. Dado F un funtor polinomial, su menor punto fijo viene dado por el objeto

$$\lim_{n\to\infty} F^n(0) = l$$

y un morfismo τ. Además, por una propiedad que presentan los funtores polinomiales se tiene

$$l = \lim_{n\to\infty} F^n(0) \simeq \lim_{n\to\infty} F(F^n(0)) \simeq F(\lim_{n\to\infty} F^n(0)) = F(l)$$

Es decir,

$$F(l) \simeq l$$

De manera más intuitiva, buscamos un tipo que es isomorfo a él mismo aplicando de nuevo el funtor en cuestión. En Haskell necesitamos algo como esto.

```
-- Representamos el menor punto fijo como Fix f
type Fix f = f (Fix f)
```

Desgraciadamente Haskell no permite esta forma de construir un nuevo tipo debido a la aparición de un bucle infinito, pues el nuevo tipo se utiliza a sí mismo de forma recursiva. Pero ya vimos con el tipo Lista que sí somos capaces de representar la recursividad. Por tanto, podemos representar el menor punto fijo como sigue.

```
-- In es el constructor del menor punto fijo
newtype Fix f = In (f (Fix f))
```

Podemos observar además que `In` es un morfismo que recibe un dato de tipo $F(l)$ y devuelve un dato de tipo l. Es decir, el propio constructor es el morfismo τ de la F-álgebra inicial. Este morfismo en realidad es un

isomorfismo, y su inversa suele llamarse unfix, y basta con que devuelva el contenido del constructor In.

```haskell
newtype Fix f = In (f (Fix f))

unfix :: Fix f -> f (Fix f)
unfix (In x) = x
```

Por último, sólo queda averiguar el catamorfismo entre este menor punto fijo y cualquier otra F-álgebra.

$$(Fixf, unfix) \xrightarrow{(|\alpha|)} (a, \alpha)$$

Recordemos que si $(|\alpha|)$ es un homomorfismo en Alg_f se debe verificar que $(|\alpha|) \circ In = \alpha \circ f((|\alpha|))$

$$\begin{array}{ccc} f(Fixf) & \xrightarrow{In} & Fixf \\ {\scriptstyle f((|\alpha|))}\downarrow & & \downarrow{\scriptstyle (|\alpha|)} \\ fa & \xrightarrow[\alpha]{} & a \end{array}$$

Además, `In` tiene como inversa unfix, por lo que obtenemos que $(|\alpha|) = \alpha \circ f((|\alpha|)) \circ unfix$.

$$\begin{array}{ccc} f(Fixf) & \xleftarrow{unfix} & Fixf \\ {\scriptstyle f((|\alpha|))}\downarrow & & \downarrow{\scriptstyle (|\alpha|)} \\ fa & \xrightarrow[\alpha]{} & a \end{array}$$

Esta última igualdad es lo que buscábamos. Dado α, podemos obtener su correspondiente catamorfismo. Es decir, podemos crear una función, que llamaremos `cata`, capaz de obtener el catamorfismo a partir del morfismo de una F-álgebra.

```
cata :: (f a -> a) -> Fix f -> a
cata alpha = alpha . fmap (cata alpha) . unfix
```

Ejemplo 7.10. *Veamos un ejemplo sencillo empezando con un tipo de dato recursivo que representará al conjunto de los números naturales.*

$$Nat = 1 + Nat$$

Recordemos que 1 representa a un morfismo constante, un constructor sin argumentos. En Haskell se tendría lo siguiente.

```
data Nat = Uno | Sig Nat
```

Debemos convertirlo en un funtor polinomial. Para ello basta sustituir `Nat` *en el miembro de la derecha por un tipo dependiente e indicar que, efectivamente, es un funtor.*

```
data Nat b = Uno | Sig b

instance Functor Nat where
   fmap f Uno = Uno
   fmap f (Sig x) = Sig (f x)
```

Una F-álgebra de `Nat` *dada por (Int,count) puede escribirse en Haskell como sigue.*

```
count :: Nat Int -> Int
count Uno = 1
count (Sig x) = x + 1
```

Idealmente, un elemento de tipo `Fix Nat` *se escribiría de la siguiente forma.*

```
cuatro :: Fix Nat -- Igual a (Nat (Nat (Nat ...)))
cuatro = (Sig (Sig (Sig Uno)))
```

Observemos que a pesar de tener algo finito como `(Sig (Sig (Sig Uno)))`*, sirve como elemento del menor punto fijo que, a priori, es algo infinito. La clave está en que* `Uno` *es el morfismo de un funtor constante. Recordemos que en el funtor constante se obvia el argumento, que en este caso sería del tipo (Nat (Nat (Nat ...))). Es decir, los constructores sin argumentos sirven como un 'final' para el menor punto fijo.*

Como ya dijimos anteriormente, esta forma de escribir elementos del menor punto fijo está prohibido en Haskell. Necesitamos el constructor `In`*.*

```
cuatro :: Fix Nat
cuatro = In (Sig (In (Sig (In (Sig Uno)))))
```

Ahora generamos el catamorfismo a partir de la función `count`*.*

```
catacount :: Fix Nat -> Int
catacount = cata count
```

Otro apunte necesario es la explicación de por qué termina la llamada a `cata`*. Su definición era la siguiente.*

```
cata :: (f a -> a) -> Fix f -> a
cata alpha = alpha . fmap (cata alpha) . unfix
```

Podemos ver que `cata` *se llama recursivamente. La clave de nuevo está en el funtor constante. El funtor constante convertía, mediante* `fmap`*, cualquier morfismo en el morfismo identidad. Esto, junto con la evaluación perezosa que presenta Haskell conseguimos entender que la recursión termine.*

Ahora podemos usar el catamorfismo.

```haskell
cuatro :: Fix Nat
cuatro = In (Sig (In (Sig (In (Sig Uno)))))

catacount cuatro -- Resultado: 4
```

Claramente, esto en la práctica es inviable debido a las limitaciones del propio lenguaje. Sin embargo, podemos usar las mismas ideas en un lenguaje con menos restricciones como Racket, pues es de tipado dinámico.

7.2. Intérpretes

Siguiendo con el ejemplo de los números naturales, observemos que dado el dato `Uno`, devolvemos mediante la función `count` el valor 1. Es decir, estamos interpretando la estructura `Uno`. Podemos explotar esta idea en Racket aprovechándonos de su tipado dinámico. Para representar estructuras como `Uno` o `Sig a` usaremos las listas.

```racket
; Uno
'(Uno)

; Siguiente
'(Sig a)

(define cuatro '(Sig (Sig (Sig (Uno)))))
```

Podemos expandir la funcionalidad descrita en Haskell realizando las siguientes suposiciones.

- Cualquier elemento que no sea una lista se supondrá que es la imagen de un funtor constante.
- Cualquier lista se supondrá un elemento del menor punto fijo, por lo que habrá que realizar la correspondiente llamada recursiva.
- EL elemento inicial de una lista también puede ser otra lista.

Con esto podemos crear la siguiente función para generar catamorfismos.

```
(define (cata alpha)
  (lambda (l)
    (if (list? l)
        (alpha (map (lambda (x) ((cata alpha) x)) l))
        l)))
```

A diferencia de la función `cata` definida en Haskell, aquí no necesitamos ninguna función unfix. Además, teniendo en cuenta que un objeto de tipo l también es un objeto de tipo $F(l)$ podemos usar la identidad como isomorfismo del F-álgebra inicial. Por otro lado, disponemos de la función `match` para realizar pattern matching de forma similar que Haskell.

```
(define (count l)
   (match l
      [`(Uno) 1]
      [`(Sig ,x) (+ x 1)]))
```

Y así usamos el catamorfismo deseado.

```
(define cuatro '(Sig (Sig (Sig (Uno)))))

(define catacount (cata count))

(catacount cuatro) ; Resultado: 4
```

Pero aún podemos ir más lejos. Siempre que creemos un morfismo como `count` generaremos seguidamente su catamorfismo correspondiente. Además, la función `match` siempre valdrá la pena usarlo, por lo que podemos crear una macro que junte todo el proceso y obvie la información redundante.

```
(define-syntax (interpreter stx)
  (syntax-case stx ()
    [(_ name pat ...) #`(define name
                          (cata (lambda (v)
                              (match v pat ...))))]))
```

Con esto, podemos crear intérpretes de forma sencilla y cómoda.

```
(interpreter catacount
   ['(Uno) 1]
   [`(Sig ,x) (+ x 1)])

(catacount '(Sig (Sig (Sig (Uno))))) ; Resultado: 4

; ----------------------------

(interpreter fibo
   ['(Uno) '(0 1)]
   [`(Sig (,n ,m)) `(m (+ n m))])

(fibo '(Sig (Sig (Sig (Uno))))) ; Resultado: '(2 3)
```

Capítulo 8

Computación auxiliar y E/S

Uno de los problemas más importantes de la programación funcional es lidiar con la entrada y salida. Funciones para leer el contenido de un fichero están presentes en la mayoría de lenguajes de programación, pero hay que tener en cuenta que el resultado de esas funciones puede variar independientemente de los argumentos que le proporcionemos. Es decir, no se corresponde con ninguna función matemática, y por tanto no puede estar presente en un lenguaje funcional.

Por otro lado, un lenguaje sin E/S es un lenguaje inútil, por lo que necesitamos algún mecanismo que nos de esta funcionalidad manteniendo la pureza de un programa funcional. La solución que ofrece Haskell son las mónadas.

8.1. Mónadas

Una idea intuitiva de lo que nos permiten las mónadas es realizar cálculos auxiliares a partir de una computación principal. Cada mónada estará identificada con qué cálculos auxiliares realiza. Ejemplos importantes que veremos son la mónada `Writer`, que irá guardando una lista de cadenas de texto a modo de registro, mientras que la mónada `IO` realizará operaciones de E/S.

Definición 8.1. *Sea C una categoría. Una mónada es una terna (T, μ, η) donde $T : C \to C$ es un funtor de C, y tanto $\mu : T^2 \to T$ como $\eta : Id_C \to T$ son dos transformaciones naturales en C (Id_C denota el funtor identidad en C). Además, se deben verificar las siguientes igualdades:*

- $\mu \circ T\mu = \mu \circ \mu T$

- $\mu \circ T\eta = \mu \circ \eta T = Id_T$

Hay que recordar que una transformación natural en Haskell no será otra cosa que una función polimórfica. Es decir, para crear una mónada necesitaremos un funtor y dos funciones polimórficas que deben verificar las igualdades de la definición. Veamos con detalle qué significan estas igualdades.

La transformación natural $\mu : T^2 \to T$ se puede interpretar como una operación de 'combinación'. Es por esto que esta función polimórfica en Haskell se denomina `join`. La igualdad

$$\mu \circ T\mu = \mu \circ \mu T$$

relaciona dos transformaciones naturales, por lo que para cada objeto $X \in C$ se debe cumplir

$$\mu_X \circ T(\mu_X) = \mu_X \circ \mu_{T(X)}$$

$$\begin{array}{ccc} T(T(T(X))) & \xrightarrow{T(\mu_X)} & T(T(X)) \\ \downarrow{\scriptstyle \mu_{T(X)}} & & \downarrow{\scriptstyle \mu_X} \\ T(T(X)) & \xrightarrow[\mu_X]{} & T(X) \end{array}$$

Con la ayuda del diagrama, podemos observar que partiendo del objeto T(T(T(X))) podemos obtener T(X) de dos formas equivalentes. Por un lado, podríamos 'combinar' el objeto T(T(X)) usando el morfismo $T(\mu_X)$ y seguidamente 'combinar' T(T(X)); o podríamos 'combinar' el objeto T(T(Y)) donde Y=T(X) usando el morfismo $\mu_Y = \mu_{T(X)}$ y finalmente 'combinar' el objeto T(T(X)). Con esto ya podemos decir que la primera igualdad nos está asegurando una asociatividad con respecto a la transformación natural μ. O en el caso de Haskell, una asociatividad con respecto a la función polimórfica `join`.

La transformación natural η nos va a proporcionar un elemento 'identidad' con respecto a μ. En Haskell, la función polimórfica correspondiente se denomina `return`. Al igual que antes, la igualdad

$$\mu \circ T\eta = \mu \circ \eta T$$

relaciona dos transformaciones naturales, por lo que para cada $X \in C$ se debe verificar

$$\mu_X \circ T(\eta_X) = \mu_X \circ \eta_{T(X)} = Id_T$$

$$\begin{array}{ccc} T(X) & \xrightarrow{\eta_{T(X)}} & T(T(X)) \\ {\scriptstyle T(\eta_X)}\downarrow & & \downarrow{\scriptstyle \mu_X} \\ T(T(X)) & \xrightarrow[\mu_X]{} & T(X) \end{array}$$

Con la ayuda del diagrama, podemos observar que la igualdad nos obliga a que η debe proporcionar un elemento que 'combinado' con un segundo elemento, ya sea por la izquierda o por la derecha, debe dar como resultado ese mismo segundo elemento.

Ejemplo 8.2. *Vamos a mostrar la construcción de una versión simplificada de la mónada* `Writer`*. Necesitamos que el constructor del tipo de dato contenga un valor polimórfico y un* `String`*.*

```
data Writer a = Writer a String
```

Por definición, la mónada debe ser un funtor.

```
data Writer a = Writer a String

instance Functor Writer where
   fmap (Writer x str) = Writer (f x) str
```

Para crear la mónada necesitamos implementar las funciones polimórficas `join` *y* `return`*. La función* `join` *'combinará' dos objetos Writer concatenando los* `String` *que posee cada uno. Por otro lado,* `return` *devolverá un objeto* `Writer` *cuyo* `String` *será una cadena vacía.*

```
instance Monad Writer where
   join (Writer (Writer x str2) str1) =
                                Writer x (str1 ++ str2)
   return x = Writer x ""
```

Por último podemos comprobar la validez de las igualdades de la definición de Mónada. Para cada $X \in C$ se deben verificar

$$\mu_X \circ T(\mu_X) = \mu_X \circ \mu_{T(X)}$$

$$\mu_X \circ T(\eta_X) = \mu_X \circ \eta_{T(X)} = Id_T$$

que en Haskell significa

```
-- Igualdades:
-- join . (fmap join) = join . join
-- join . (fmap return) = join . return = id


-- Comprobamos la asociatividad de join
holamundo :: Writer (Writer (Writer Int))
holamundo =
   Writer (Writer (Writer 5 "mundo.") "nuevo ") "Hola "

-- (fmap join) concatena "nuevo " y "mundo."
-- Finalmente join concatena "Hola " y "nuevo mundo."
join (fmap join holamundo)
-- Resultado: (Writer 5 "Hola nuevo mundo.")

-- join concatena "Hola " y "nuevo "
-- Finalmente join concatena "Hola nuevo " y "mundo."
join (join holamundo)
-- Resultado: (Writer 5 "Hola nuevo mundo.")


-- Comprobamos la validez del elemento identidad
unafrase :: Writer Int
unafrase = Writer 8 "Una frase cualquiera."

-- (fmap return) genera el elemento identidad dentro de
   'unafrase'
-- Finalente join concatena "Una frase cuelquiera" y ""
join (fmap return unafrase)
-- Resultado: (Writer 8 "Una frase cualquiera")

-- return genera el elemento identidad fuera de '
   unafrase'
-- Finalmente join concatena "" y "Una frase cualquiera
   ."
join (return unafrase)
```

```
34 -- Resultado: (Writer 8 "Una frase cualquiera.")
```

La función `join` es la responsable de todo el cálculo auxiliar que hemos mencionado al inicio del capítulo. Sin embargo, aún no sabemos qué es ni cómo manipular el flujo principal de computación. Para ello necesitamos el concepto de categoría de Kleisli.

8.2. Categoría de Kleisli y el operador de extensión

Un programa en cualquier lenguaje de programación se crea a partir de la composición de funciones. Buscamos ahora una forma de componer mónadas, pues al unirlas con la 'combinación' obtendremos el resultado deseado. Es decir, la composición de funciones entre mónadas nos dará los cálculos principales que serán adornados mediante el cálculo auxiliar que ofrece la 'combinación' antes vista.

Definición 8.3. *Sea (T, μ, η) una mónada en C. La categoría de Kleisli asociada a la mónada es la categoría C_T cuyos objetos y morfismos vienen dados por*

$$Obj_{C_T} := Obj_C$$

$$Mor_{C_T}(X, Y) := Mor_C(X, T(Y))$$

donde $X, Y \in Obj_C$. Sean $f : X \to T(Y)$ y $g : Y \to T(Z)$, la composición de morfismos viene dada por la expresión

$$g \circ_T f := \mu_Z \circ T(g) \circ f : X \to T(Z)$$

Por último, para cada objeto $X \in Obj_C$, su morfismo identidad correspondiente es

$$id_X := \eta_X : X \to T(X)$$

Ejemplo 8.4. *Siguiendo con la mónada `Writer` podemos ver algunos de los elementos de su categoría de Kleisli asociada.*

```haskell
-- Funcion que esta en la categoria de Kleisli
positivo :: Int -> Writer Bool
positivo x = if (x > 0)
                Writer (x > 0)
                       ((show x) ++ " es positivo."
             else
                Writer (x > 0)
                       ((show x) ++ " no es positivo."

-- Otra funcion en la categoria de Kleisli
por2 :: Int -> Writer Int
por2 x = Writer (x * 2)
         ("Guardamos el valor " ++ (show (x * 2)) ++ ".")
```

Por definición, la composición de dos funciones f,g viene dada por

$$\mu_Z \circ T(g) \circ f$$

y en Haskell obtenemos lo siguiente:

```haskell
-- Componemos las dos funciones
(join . (fmap positivo) . por2) (Writer 7 "Valor 7.")
-- Resultado:
-- (Writer True "Valor 7.Guardamos el valor 14.14 es
   positivo.")
```

Con este ejemplo ya podemos ver a qué nos referimos con computación principal y auxiliar. Por un lado, hemos multiplicado por 2 el valor 7 y luego comprobamos si el resultado es positivo. Mientras tanto, de manera auxiliar, se han ido anotando en un `String` los pasos que se han seguido.

Sin embargo, usar de esta manera la composición de mónadas es incómodo e inviable si hablamos de operaciones más complejas. En primer lugar, estamos usando funciones de la forma

$$h : X \to T(Y)$$

pero sería más conveniente tener algo como

$$H : T(X) \to T(Y)$$

En concreto buscamos generar una nueva mónada a partir de otra. De esta forma podríamos tener mayor control sobre qué se va a computar. La solución está en usar el denominado operador de extensión y la notación do.

8.3. Notación do

Definición 8.5. *Sea C una categoría y (T, μ, η) una mónada y C_T su categoría de Kleisli asociada. Para cada $f : X \to T(Y) \in C_T$, el operador de extensión se define como*

$$(\_)^* : (X \to T(Y)) \to (T(X) \to T(Y))$$

$$f^* := \mu_Y \circ T(f)$$

El operador de extensión es comúnmente conocido en Haskell como `bind` y se escribe `>>=`.

```
-- Funcion bind
>>= :: m a -> (a -> m b) -> m b
```

Nótese que los argumentos de `>>=` están en distinto orden. Aunque esta sea la versión oficial, utilizaremos en los ejemplos sucesivos nuestra propia función `bind` por motivos de claridad. Siguiendo la definición, podemos declarar la función `bind` como sigue.

```
-- Funcion bind
bind :: (Functor T) => (a -> T b) -> (T a -> T b)
bind f = join . (fmap f)
```

Por la currificación de funciones y la precedencia por la derecha del operador '-¿' podemos eliminar los paréntesis en la expresión `(T a -> T b)` obteniendo equivalentemente

```
bind :: (Functor T) => (a -> T b) -> T a -> T b
bind f = join . (fmap f)
```

Es decir, `bind` es una función que acepta otra función de tipo `(a -> T b)`, un objeto de tipo `T a` y devuelve un objeto de tipo `T b`.

Ahora, en lugar de usar una función predefinida, le proporcionamos a `bind` una función lambda.

```
1 bind (\x -> cuerpo_lambda) (T a)
```

Si nos fijamos en la definición de la función `bind` apreciamos que la primera función en ejecutarse es `(fmap f)`, donde `f` en este caso es la función lambda. Además, se ejecutará sobre el objeto de tipo `(T a)`. Por tanto, `f` se ejecutará sobre un elemento de tipo `a` relacionado de alguna forma con el objeto `(T a)`. Con esto diremos que el argumento `x` de la expresión lambda representa a un valor que es 'retornado' por el objeto de tipo `(T a)`.

Dicho todo esto, consideremos la siguiente transformación:

$$
\begin{array}{l}
(bind \quad (\backslash x \rightarrow cuerpo\_lambda) \quad (T\ a)) \\
\quad \downarrow\ \downarrow\ \downarrow \\
(x \leftarrow (T\ a)) \\
(cuerpo\_lambda)
\end{array}
$$

Pongamos ahora que `cuerpo_lambda` llama de nuevo a la función `bind`, con un nuevo lambda y un nuevo objeto `T`. La transformación sería:

$$
\begin{array}{l}
(bind \quad (\backslash x \rightarrow cuerpo\_lambda) \quad (T\ a)) \\
\quad \downarrow\ \downarrow\ \downarrow \\
(x \leftarrow (T\ a)) \\
(bind \quad (\backslash y \rightarrow cuerpo\_lambda2) \quad (T\ b)) \\
\quad \downarrow\ \downarrow\ \downarrow \\
(x \leftarrow (T\ a)) \\
(y \leftarrow (T\ b)) \\
(cuerpo\_lambda2)
\end{array}
$$

Cabe destacar el estilo imperativo al que nos recuerda este tipo de transformación. Además, los valores 'generados' como `x` o `y` del ejemplo anterior

pueden ser usados en la creación de las siguientes mónadas. Esto se puede apreciar con más detalle en el ejemplo mostrado más abajo.

Si el cuerpo del último lambda que creemos no llama a la función `bind` se terminará la transformación. Para terminar el ejemplo anterior, podemos sustituir `cuerpo_lambda2` por cualquier espresión que devuelva una mónada de tipo `T`, como la función `return`.

$$\begin{aligned}&(x \leftarrow (T\ a))\\&(y \leftarrow (T\ b))\\&(return \quad g(x,y))\end{aligned}$$

Ejemplo 8.6. *En Haskell podemos usar esta nueva forma de programar mediante el uso de la notación do.*

```haskell
-- Creamos una monada Writer usando do
por2_positivo :: Writer Int -> Writer Bool
por2_positivo mv =
  do  x <- mv
      y <- (Writer (x * 2)
           "Guardamos el valor "++(show (x * 2))++".")
      if (y > 0)
         Writer (y > 0) ((show y) ++ " es positivo."
      else
         Writer (y > 0) ((show y) ++ " no es positivo."

por2_positivo (Writer 7 "Valor 7.")
-- Resultado:
-- (Writer True "Valor 7.Guardamos el valor 14.14 es
   positivo.")
```

8.4. Entrada y salida en Haskell. La mónada IO

La mónada `IO` se usa de manera similar a `Writer`. Mediante la notación do podemos especificar un cálculo principal que será acompañado de una computación auxiliar que realizará el trabajo de la entrada y salida de datos.

Consideremos el término `getChar :: IO Char`, que 'lee' un caracter de la entrada y lo devuelve para su uso. Al ser una mónada podemos usar la notación do como sigue:

```
lectura :: IO String
lectura = do
   x <- getChar
   return ("Leemos " ++ (show x))
```

Recordemos que la notación do es sólo una transformación sintáctica, por lo que el código anterior es equivalente a:

```
lectura :: IO String
lectura =
   bind (\x -> return ("Leemos " ++ (show x))) getChar
```

Como la función lambda devuelve una mónada con valor principal `String`, `bind` devolverá un objeto de tipo `IO String`.

Como hemos dicho, la computación auxiliar de una mónada `IO` consiste en realizar operaciones de entrada y salida. En este caso estaríamos leyendo un caracter. Esto significa que en el momento que hemos definido el término `lectura` estaríamos leyendo el caracter. Claramente, esto sería un caos en cualquier proyecto, pues dependiendo del orden en el que se definen los valores realizaríamos unas operaciones diferentes de entrada y salida. Y por si fuera poco, una mónada debía cumplir una regla de asociatividad con respecto a la función `join`, lo que implicaba que alterando el orden en el que 'combinábamos' la computación auxiliar el resultado era el mismo. Como ya sabemos, esto no ocurre con las operaciones de entrada y salida.

Para solucionar todos estos problemas Haskell no ejecuta ninguna operación de entrada y salida al crear las mónadas `IO`, sino que va guardando y componiendo representaciones de las operaciones que se deben ejecutar. Es por esto que cualquier programa de Haskell es una mónada `IO`. Más concretamente, para compilar un programa en Haskell necesitamos un objeto `main` de tipo `IO ()`. Durante la compilación, estas representaciones son traducidas a código máquina para poder realizar las operaciones pertinentes.

Capítulo 9

Conclusión y líneas futuras

Durante todo el trabajo hemos demostrado el potencial de la teoría de categorías en el campo de la programación, y no sólo en lenguajes puramente funcionales. Identificar elementos propios de los lenguajes de programación con conceptos puramente matemáticos nos ha permitido construir tipos de datos complejos, estudiar el polimorfismo y la recursión, e incluso organizar una computación auxiliar a partir de un flujo principal y así investigar sobre la entrada y salida en el ámbito funcional puro.

Claramente, la teoría de categorías tiene mucho más que ofrecer. Uno de los resultados más importantes de esta rama matemática es el Lema de Yoneda, que nos ofrece un isomorfismo entre dos conjuntos que podemos aprovechar para obtener reflexiones muy interesantes. Otra de sus consecuencias es el denominado Continuation Passing Style que tiene el potencial de conseguir un lenguaje aparentemente imperativo con un comportamiento interno puramente funcional, pudiendo escoger lo mejor de los dos mundos. Hemos visto el concepto de funtor que contiene un argumento genérico. Sin embargo, podemos considerar tipos de datos con dos o más argumentos genéricos obteniendo bi-funtores, tri-funtores, o de forma general, n-funtores. Y por último, las ópticas comprenden una gran teoría sobre cómo acceder o modificar la información de complejas estructuras de datos. Merecen especial atención las lentes, los prismas y los traversals. Estos últimos han sido profundamente estudiados e implementados en Haskell por Edward Kmett.

Apéndice A

El lenguaje de programación Racket

Racket es un lenguaje derivado de Lisp, por lo que comparten las características más importantes que definen este tipo de lenguajes, como lo son la sintaxis y el sistema de macros.

Racket es considerado un lenguaje orientado a lenguajes, es decir, se creó con la idea de poder crear nuevos lenguajes de manera rápida y sencilla. Tanto es así que en un fichero de Racket debemos especificar qué lenguaje queremos usar mediante la sentencia `#lang <language>`. Lo más común es usar el propio lenguaje Racket.

```
1 #lang Racket
2
3 ; Codigo
```

A.1. S-expresiones

El código está formado principalmente por S-expresiones, que se definen como un identificador o una lista de sub-S-expresiones, donde las listas se escriben entre paréntesis.

```
1 #lang Racket
2
```

```
-- Identificador
5
-- Lista de sub-S-expresiones
(+ 3 (- 3 2))
```

Cada una de las S-expresiones es evaluada y su valor es impreso. En el caso anterior se imprime un 5 y luego un 4.

Para poder evaluar una lista correctamente debe verificarse que el primer elemento que contiene es una S-expresión (identificador o lista) que se evalúa a un procedimiento (una función). El resto de elementos de la lista serán sus argumentos. Cada uno de los argumentos de la función serán evaluados de izquierda a derecha antes de poder evaluar la función en cuestión.

```
#lang Racket

; El identificador + se evalua a un procedimiento
; Cada valor constante es evaluado a si mismo
(+ 3 (- 3 2))
; Resultado: 4
```

A.2. Formas especiales

Si todo el código es evaluado siguiendo las reglas anteriores no podríamos tener control de flujo en nuestro programa. En concreto, es imposible definir una función `if` que elija qué código evaluar, pues cada uno de los argumentos serían evaluados antes de evaluar la función `if`. Por tanto, existen excepciones a esta regla, las llamadas formas especiales. Como es de esperar, `if` es una de ellas. Otra de las más importantes es la forma especial define, con la que podemos definir nuevos identificadores.

```
#lang Racket

(define valor 5)

(if (> valor 3)
      "Valor mayor que 3"
```

```
      "Valor menor o igual que 3")
; Resultado: "Valor mayor que 3"
```

A.3. Procedimientos

Una de las formas especiales más importantes es `lambda`, que nos permite generar funciones nuevas.

```
#lang Racket

(define por2 (lambda (x) (* x 2)))

(por2 5)
; Resultado: 10
```

Usualmente se definen nuevos procedimientos usando una sintaxis más cómoda que nos ofrece el propio `define`.

```
#lang Racket

(define (por2 x)
   (* x 2))

(por2 5)
; Resultado: 10
```

Un procedimiento es un valor más, por lo que podemos aceptarlo como argumento de una función o retornarlo.

```
#lang Racket

(define (usar5 proc)
   (proc 5))

(usar5 (lambda (x) (+ 3 x)))
```

```
; Resultado: 8
```

A.4. Quotation

Una de las características más importantes de este tipo de lenguajes es la capacidad de desactivar la evaluación de una lista. Esto se consigue mediante la forma especial `quote`, que se abrevia utilizando una comilla simple `'`.

```
#lang Racket

; Desactivamos la evaluacion de (+ 3 5)
(quote (+ 3 5))

; Equivalente a
'(+ 3 5)
; Resultado en ambos casos: '(+ 3 5)
```

Como el código está formado por listas, estas expresiones sin evaluar son concretamente listas, desde el punto de vista de los datos. Además, Racket ofrece funciones para manejar estos datos. Las más importantes son `car` y `cdr`.

```
#lang Racket

; Obtenemos el primer elemento de una lista
(car '(+ 3 5))
; Resultado: '+

; Descartamos el primer elemento de una lista
(cdr '(+ 3 5))
; Resultado: '(3 5)
```

Otro detalle importante es que los valores constante son evaluados a sí mismos, por lo que un valor constante evaluado o no evaluado es algo equivalente.

```
#lang Racket

; Definimos una lista de numeros
(define numeros '(1 1 2 3 5 8))

; Utilizamos el primero
(+ 5 (car numeros))
; Resultado: 6
```

Por último cabe destacar que mediante la desactivación de S-expresiones somos capaces de representar cualquier código de Racket mediante una lista de datos para manejarlos a nuestro antojo. Basta añadir una comilla simple justo delante. Esto permite generar intérpretes y analizadores sintácticos complejos de manera muy sencilla. Tal es el poder de esta herramienta que existe la función `eval`, que dada una S-expresión desactivada, la evalúa y retorna su valor. Es decir, `eval` es un intérprete de Racket.

A.5. Macros

Si no tenemos suficiente con lo anterior, las macros nos permiten cambiar una S-expresión dada por otra S-expresión ya existente (durante la compilación). Pequeñas macros generan herramientas poderosas, grandes macros generan nuevos lenguajes de programación (manteniendo la sintaxis de Racket).

Como ejemplo, supongamos que queremos una forma especial como `define` que sea capaz de definir varios identificadores con el mismo valor. La manera de usarlo sería:

```
#lang Racket

(define-many (numero casa hola) 5)

numero
; Resultado: 5
(+ 3 casa)
; Resultado: 8
```

Podemos crear una macro que expanda la S-expresión anterior a varios `define` consecutivos.

```
#lang Racket

(define-syntax (define-many stx)
  (syntax-case stx ()
    [(define-many (ids ...) value)
       #'(begin (define ids value) ...)]))

(define-many (hola casa) 5)

(+ 3 casa)
; Resultado: 8
```

Como podemos observar, las macros nos permiten amoldar el lenguaje a nuestras necesidades. Lo más común es usar lenguajes como C++ o Java donde el programador debe adaptarse a las herramientas que se le ofrecen, pero en lenguajes derivados de Lisp como Racket es el lenguaje el que se adapta al programador.

Apéndice B

Mónadas en C++

En C++ podemos conseguir también hacer uso de las mónadas. El principal problema es la imposibilidad de crear una notación do como la existenete en Haskell. Por eso, debemos conformarnos con el operador de extensión o función `bind`.

Seguiremos con el ejemplo de mónada `Writer`. Debemos recordar que si queremos una mónada debemos primero crear un funtor.

```
// Struct Writer
template<typename A>
struct Writer{
    A value;
    std::string str;
}

// Declaramos la funcion fmap para Writer
template<typename A, typename Func>
auto fmap(Func&& f, const Writer<A>& wrt){
    return Writer<decltype(f(std::declval<A>()))>{
       f(wrt.value) , wrt.str
    };
}
```

Para generar la mónada basta definir las funciones `join` y `return`. Debemos tener en cuenta que si creamos diferentes tipos de mónadas, C++ no será capaz de distinguir entre las diferentes versiones de la función `return`. Por

ello, deberíamos definir esta función como `static` dentro del `struct Writer`. Además, como `return` es una palabra reservada de C++ utilizaremos `_return`.

```
// Struct Writer con return
template<typename A>
struct Writer{
    A value;
    std::string str;

    // Funcion return
    template<typename B>
    static Writer<B> _return(B&& value){
        return Writer<B>{ value, "" };
    }
}

// Funcion join
template<typename A>
Writer<A> join(const Writer<Writer<A>>& wrt){
   return Writer<A>{ wrt.value.value , wrt.str + wrt.
   value.srt };
}
```

Definimos la función `bind`, que se correspondía con la composición de `join` con `fmap`.

```
// Funcion bind
template<typename Func, typename Monad>
auto bind(Func&& f, const Monad& m){
   return join(fmap(f,m));
}
```

Una vez hemos llegado hasta aquí necesitamos usar una notación `do`. Por desgracia, C++ no tiene la suficiente flexibilidad como para dejar al programador generar este tipo de herramientas. Por ello debemos buscar una alternativa.

Recordemos que dada una categoría C y una mónada (T, μ, η), la categoría de Kleisli asociada a C, denotada por C_T verificaba que la composición

de dos morfismos $f : X \to T(Y)$ y $g : Y \to T(Z)$ se definía como

$$g \circ_T f := \mu_Z \circ T(g) \circ f = g^* \circ f$$

El resultado de la composición es otro morfismo de C_T, por lo que podemos aplicar el operador de extensión. Además, se verifican las siguientes igualdades.

$$\begin{aligned}(g \circ_T f)^* = (g^* \circ f)^* &= \mu_Z \circ T(\mu_Z \circ T(g) \circ f) \\ &= \mu_Z \circ T(\mu_Z) \circ T^2(g) \circ T(f) \\ &= \mu_Z \circ \mu_{T(Z)} \circ T^2(g) \circ T(f) \\ &= \mu_Z \circ T(g) \circ \mu_Y \circ T(f) \\ &= g^* \circ f^*\end{aligned}$$

Es decir, la extensión de la composición de morfismos es igual a la composición de la extensión de morfismos. Podemos usar esto para realizar de manera relativamente cómoda la composición de mónadas.

Necesitamos, por tanto, una función de composición. Esta función recibirá una mónada y un número variable de funciones a las que le aplicaremos la función `bind` junto con la mónada resultante de aplicar `bind` al resto de funciones.

```
1  // Funcion compose_with_bind
2  // Caso base
3  template<typename Monad, typename Func>
4  auto compose_with_bind(const Monad& m, Func&& f){
5      return bind(f,m);
6  }
7
8  // Caso recursivo
9  template<typename Monad, typename Func, typename...
       Funcs>
10 auto compose_with_bind(const Monad& m, Func&& f, Funcs
       &&... gs){
11     return compose_with_bind(bind(f,m),gs...);
12 }
```

Con estas funciones podemos manipular las mónadas de manera sencilla y cómoda.

```
// Morfismos en la categoria de Kleisli
Writer<int> por2(int x){
    return Writer<int>{ x*2, "Guardamos el valor " + std::to_string(x*2) + "." };
}

Writer<bool> positivo(int x){
    if (x > 0)
        return Writer<bool>{ x > 0, std::to_string(x) + " es positivo." };
    else
        return Writer<bool>{ x > 0, std::to_string(x) + " no es positivo." };
}

// Ejemplo de uso
int main(){

   // Monada inicial
   Writer<int> w1{ 7 , "Valor 7." };

   // Realizamos operaciones sobre la monada inicial
   Writer<bool> w2 = compose_with_bind(w1,
                                       por2,
                                       positivo);

   std::cout << "Valor de la monada: " << w2.value << std::endl;
   // Resultado: "Valor de la monada: 1"
   // 1 es true en C++
   std::cout << "Texto de la monada: " << w2.str << std::endl;
   // Resultado: "Texto de la monada: Valor7.Guardamos el valor 14.14 es positivo."

   return 0;

```

```
32 }
```

Apéndice C

Mónadas en Racket

De forma análoga al caso de C++, podemos generar mónadas en Racket usando `structs`. Además, las macros nos permitirán crear una notación `do` equivalente a la que encontramos en el lenguaje Haskell.

Siguiendo con el ejemplo de la mónada `writer`, creamos el tipo de dato correspondiente.

```
#lang Racket

(struct writer
   (value
    str)

    ; Metodos
    ;...
    )
```

Necesitamos ahora que `Writer` sea un funtor. Queremos que la función `fmap` distinga entre las diferentes mónadas que podamos crear, por lo que creamos un método genérico.

```
#lang Racket

(require racket/generic)

; Definimos el metodo generico fmap
```

```
(define-generics funtor
   (fmap f funtor))

(struct writer
   (value
    str)

    ; Implementamos el metodo fmap
    #:methods gen:funtor
    (define (fmap f F)
       (match F
          [(writer val str) (writer (f val) str)]))
    )
```

Finalmente sólo nos queda definir `join` y `return` para que `Writer` sea una mónada. Sin embargo, los métodos genéricos necesitan un argumento con el que poder deducir qué implementación usar. En el caso de `return`, debemos 'ligar' este método con la clase. Para ello usamos las `properties`.

```
#lang Racket

(require racket/generic)

; Definimos el metodo generico join
(define-generics monad
   (join monad))

; Definimos la propiedad return
(define-values (prop:return return? get-return)
  (make-struct-type-property 'return))

(struct writer
   (value
    str)

    ;...

    ; Implementamos el metodo join
    #:methods gen:monad
```

```
21      (define (join M)
22         (match M
23            [(writer (writer val str2) str)
24             (writer val (string-append str str2))]))
25
26      ; Implementamos la propiedad return
27      #:property
28      prop:return (lambda (z) (writer z ""))
29      )
```

La función se implementa de forma inmediata.

```
1  #lang Racket
2
3  (define (bind m f)
4     (join (fmap f m)))
```

Lo más cómodo es usar la notación `do`, por lo que creamos una macro para conseguirlo.

```
1  #lang Racket
2
3  (require (for-syntax syntax/parse))
4
5  (define-syntax (do stx)
6    (syntax-parse stx
7      #:datum-literals (<- return)
8      [(do struct-type:id (return arg:expr))
9          #'((get-return struct-type) arg)]
10     [(do struct-type:id body:expr) #'body]
11     [(do struct-type:id
12           (x:id <- (return arg:expr)) rest:expr ...)
13         #'(bind ((get-return struct-type) arg)
14                 (lambda (x)
15                    (do struct-type rest ...)))]
16     [(do struct-type:id (x:id <- body:expr) rest:expr
    ...)
17         #'(bind body (lambda (x)
```

```
                          (do struct-type rest ...)))]
    [(do struct-type:id body:expr rest:expr ...)
        #`(bind body (lambda (#,(gensym))
             (do struct-type rest ...)))]))
```

A pesar de la dificultad que supone leer la macro anterior, su uso es muy sencillo. La única diferencia con respecto a Haskell es la necesidad de proporcionar explícitamente el tipo de dato del `struct` que se va a usar.

```
#lang Racket

(define (positivo-str val)
   (if (> val 0)
      (format "~a es positivo.")
      (format "~a no es positivo")))

(define my-monad
   (do struct:writer
      (x <- (writer 7 "Valor 7."))
      (y <- (writer (* 2 x)
             (format "Guardamos el valor ~a." (* 2 x))))
      (writer (> val 0) (positivo-str val))))

(writer-value my-monad)
; Resultado: 14

(writer-str my-monad)
; Resultado: "Valor7.Guardamos el valor 14.14 es
   positivo ."
```

Apéndice D

La mónada IO en Racket

Racket no es un lenguaje funcional puro, por lo que no es necesario el uso de mónadas para las operaciones de entrada y salida. Sin embargo, construir las mónadas en este lenguaje nos permite entender muy bien el mecanismo utilizado por Haskell.

En primer lugar, recordemos que en lugar de ejecutar cada operación, Haskell generaba una representación de lo que se debe de ejecutar. En Racket podemos simular este comportamiento de forma muy sencilla. Para ello, necesitamos algo para representar cada una de las operaciones, pero basta usar las propias funciones de Racket de entrada y salida. En concreto, guardaremos una función `lambda` que debería ejecutar la correspondiente operación de entrada y salida, es decir, la función `lambda` será la representación de lo que queremos ejeturar. En el siguiente ejemplo definimos `getChar` y `putChar` usando las funciones de Racket `read-char` y `write-char`.

```
1  ; Definicion de la estructura IO
2  (struct IO
3     (repr))
4
5  ; Monada getChar (simulando a Haskell)
6  (define getChar
7     (IO (lambda () read-char)))
8
9  ; Funcion putChar
10 (define (putChar c)
11    (IO (lambda () (write-char c))))
```

Para completar la mónada necesitamos implementar los métodos `fmap`, `join` y `return`. Para ello vale la pena realizar la siguiente observación. Si un objeto es de tipo `IO Char` significa que la operación de entrada y salida que representa, en este caso una función `lambda`, debe retornar un `Char`. De forma más general, un objeto de tipo `IO a` guarda una representación de una operación que debe retornar un objeto de tipo `a`.

Con esto en mente, si `fmap` recibe una función de tipo `a->b` y un objeto de tipo `IO a`, podemos ver que para implementar `fmap` basta crear un `lambda` que ejecute dicha función sobre lo que retorne la función `lambda` de la mónada.

```
; Definicion de la estructura IO
(struct IO
   (repr)

   #:methods gen:funtor
   (define (fmap f F)
      (match F
         [(IO repr) (IO (lambda () (f (repr))))])))
```

El método `join` recibe un objeto de tipo `IO (IO a)`. En este objeto tenemos dos representaciones que debemos 'combinar'. Como la representación exterior debería ejecutarse antes que la interior, basta crear una representación que realice eso mismo.

```
; Definicion de la estructura IO
(struct IO
   (repr)

   ;...

   #:methods gen:monad
   (define (join M)
      (match M
         [(IO repr)
          (IO (lambda () (match (repr)
                           [(IO repr2) (repr2)])))])))
```

Por último, `return` debe ser una función de tipo `a->IO a` que no ejecuta

ninguna operación de entrada y salida. Basta crear una representación que retorna el argumento de `return`.

```
; Definicion de la estructura IO
(struct IO
   (repr)

   ;...

   #:property
   prop:return (lambda (z) (IO (lambda () z))))
```

Con esto ya tenemos creada la mónada IO. Podemos usar ya la notación `do` para generar nuevos objetos de este tipo.

```
; getChar
(define getChar
  (IO (lambda () (read-char))))

; putChar
(define (putChar c)
  (IO (lambda () (write-char c))))

; nextChar
(define (nextChar c)
  (integer->char (add1 (char->integer c))))

; Ejemplo con notacion do
(define io-next-char
   (do struct:IO
      (c <- getChar)
      (putChar (nextChar c))))

; Con IO-repr obtenemos la representacion
(define my-repr (IO-repr io-next-char))

; Ejecutamos la representacion
(my-repr)
Resultado: Lee un caracter e imprime el siguiente.
```

Bibliografía

[1] Lisp (Programming Language). https://en.wikipedia.org/wiki/Lisp_(programming_language)

[2] Alonzo Church. https://en.wikipedia.org/wiki/Alonzo_Church

[3] Lambda Calculus. https://en.wikipedia.org/wiki/Lambda_calculus

[4] Bartosz Milewski, *Category Theory For Programmers*, 2018.

[5] Jan-Willem Buurlage, *Categories and Haskell: An introduction to the mathematics behind modern functional programming*, 2018.

[6] Andrea Asperti, Giuseppe Longo, *Categories, Types and Structures: An Introduction to Category Theory for the working computer scientist*, 1991.

[7] Exponential object. https://ncatlab.org/nlab/show/exponential+object

[8] Distributive category. https://ncatlab.org/nlab/show/distributive+category

[9] Function curry in C++. https://stackoverflow.com/a/26768388/9612294

[10] Open Multi-Methods for C++11. https://www.codeproject.com/Articles/635264/Open-Multi-Methods-for-Cplusplus11-Part-1-The-Case

[11] Monads in Category Theory. https://en.wikipedia.org/wiki/Monad_(category_theory)

[12] Kleisli Category. https://en.wikipedia.org/wiki/Kleisli_category

[13] Do notation in Haskell. https://en.wikibooks.org/wiki/Haskell/do_notation

[14] Racket generics methods. https://docs.racket-lang.org/reference/struct-generics.html

[15] Racket structure types properties. https://docs.racket-lang.org/reference/structprops.html