Universidad Nacional Autónoma de México

Facultad de Ciencias

Mónadas en la programación funcional: una prueba formal de su equivalencia con las ternas de Kleisli

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

PRESENTA:

Cenobio Moisés Vázquez Reyes

TUTOR:

Dr. Favio Ezequiel Miranda Perea

Ciudad Universitaria, Cd. Mx., 2016

Universidad Nacional Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

1. Datos del alumno:
   Vázquez
   Reyes
   Cenobio Moisés
   54285746
   Universidad Nacional Autónoma de
   México
   Facultad de Ciencias
   Ciencias de la computación
   306328677
2. Datos del tutor:
   Dr
   Favio Ezequiel
   Miranda
   Perea
3. Datos del sinodal 1:
   Dra
   Diana
   Avella
   Alaminos
4. Datos del sinodal 2:
   Dr
   Hugo Alberto
   Rincón
   Mejía
5. Datos del sinodal 3:
   Dra
   Lourdes del Carmen
   González
   Huesca
6. Datos del sinodal 4:
   M en C
   Pilar Selene
   Linares
   Arévalo
7. Datos del trabajo escrito:
   Mónadas en la programación funcional:
   una prueba formal de su equivalencia con las ternas de Kleisli.
   82 p
   2016

AGRADECIMIENTOS

Haber llegado a la facultad de ciencias de la UNAM ha sido de las mejores cosas que me han ocurrido, he conocido gente muy valiosa, he crecido de manera intelectual y espiritual, y he vivido experiencias inolvidables. No cambiaría por nada el ser parte de esta gran comunidad que es Ciencias. Si bien este es el final de mi licenciatura, también es el comienzo de aventuras más emocionantes. Pero este viaje no lo estoy realizando solo, hay muchísimas personas que quiero y admiro que me brindan su apoyo y su cariño de manera incondicional. Voy a tratar de agradecer a todas esas personas que le dan sentido a mi vida y que me llenan de alegría con solo verlas. Aquí vamos.

Primero quiero agradecer a mi familia, hablo de mi madre, mi padre, y mis hermanitas. Ustedes son los seres que más amo, me han apoyado siempre, incluso cuando no merecía su cariño. Mamá, papá, gracias por ese inmenso amor que me dan, han hecho de este jovencito un hombre lleno de recuerdos hermosos, gracias por su paciencia, por su apoyo, por su invaluable educación. Ustedes me hacen sentir orgulloso de mí mismo. Hermanitas, ustedes son mis princesas, siempre cuentan con su hermanito, siempre. Gracias de verdad.

Recuerdo a todos mis amigos de la primaria, cómo olvidar todas las aventuras vividas (sí, por aventuras quiero decir travesuras). Amigos, siempre recuerdo el día en que nos quedamos al torneo de soccer e hicimos la casa de bancas, o la guerra de los desayunos, o los *conciertos* de bancas. ¿Verdad que fue genial? Gracias Irving, Alegrías, Víctor (Bikini), Ricardo (Tacher), Jorge (Pechán), Marcelino, Luis, Adrián, Marco (Zancudo), *Macisa*, Cañas, Jesús Iván, Jorge (tontín), Joel; gracias igual Vero, Marilú, Jaqueline, Itzarelli, Karen, Elizabeth. Sí, me sigo acordando de todos ustedes. También agradezco a todos mis profesores de primaria que me guiaron en ese extraño viaje que es la niñez.

La secundaria fue una etapa que tengo presente y que sueño a menudo con alegría. Gracias amigos, muchos de nosotros nos seguimos frecuentando y es genial recordar esos días en los que sin celulares, sin redes sociales, sin internet, éramos los más felices del planeta. Era genial irnos a jugar al parque, ir a jugar video juegos a casa de algún amigo, o simplemente salir a caminar. No saben cómo extraño todas las veces que cantamos con mi guitarra en el salón de clase. Gracias Raúl (frijol), Calzonci, Miguel (pinocho asesino), Chapu, *mamachas*, Irving, *llorón*, *mucha-lucha*, *pitufo*, Jorge *bravo*, *zaboo-mafoo*, Colín, y al que haya olvidado mencionar. También gracias a Sonia, Jessica, Ana, Esther, Alejandra, Mariana, Jeimy, Lizbeth, Geraldyne, Ivonne, Viridiana, Azu, Zayonara y todas las chicas que fueron mis super amigas. Gracias a todos. Y a todos mis profesores de la secundaria les sigo agradeciendo sus consejos.

A la par, estuve durante diez años con un grupo de amigos que quiero muchísimo: ¡El coro Vita Nova!. Gracias Laura, Manuel, y todos los integrantes de esa fabulosa agrupación. Logramos comunicar a través de las guitarras, las mandolinas y nuestras voces muchísimo amor. Amigos, no saben cuánto los quiero, y lo orgulloso que estoy de haber hecho música tan bella a su lado.

En el CCH viví experiencias únicas, inolvidables, de verdad la vida me empezó a mostrar colores que no conocía. Gracias amigos por seguir caminando conmigo, gracias Carlos, Éder, David, Lobos, César *Kikín*, *sexy*, Aroón, *Striker*, *Kamaleón*, *moco*, Ozumba, *Aldebarán*, Iván *el terrible*, Montzi, *chatarro*, *el hentais*, Rafa, Miguel, Mario Isaac, Juan,

Paco. También gracias a Arely, Hanae, Isabel, Rosario, Lau, Jazz, Karen, Lili, Érika, Ali, Adriana, Diocelín. Gracias también a mis profesores del CCH, eran la onda.

Llegando a la facultad me encontré gente muy valiosa, de hecho sigo conociendo gente muy valiosa. Primero quiero agradecer a todos mis amigos computólogos. Gracias Israel *Cullen*, Juan Camacho, *boss*, Hugo, Charly, Prince, Toñito, José Galindo, Martín, Vale, Armando, Diana, Aida, Najla, Clau, Eliza Beth, Lilian, Martha, Grace, Alan, Joshua, Diego, Emiliano, Juan José, Thalía, Sango, Opacho, Daniel Torres, Jonas, Víctor Segoviano, Miguel Piña, Lalo, Fer, Emmanuel, Diego Murillo, Rodrigo, Snake, Javier, Albert, Erick Iván, Monserrat, Pilar, Antonio, Isaac Pompa, Fer, Amilcar, Rafa, Laura, Luis, y todos los demás colegas. ¡Son geniales!

También quiero agradecer a Frank Pan y Vino, Yemi, Jesús, Rodrigo, Paco, Axel, Carmen, Cata, Eddy, Mindy, Miguel Alcubierre, Saúl, Ximena, Claus, Ángel Toledo, Antonio, Juan Orendaín, Osvaldo, Montse Torres, Hei, Jorge Vega, Gaby Pizita, y todos mis amigos de la facultad. Gracias por esas largas charlas.

También agradezco a mis amigos de Tekken Elite México, las veladas con ustedes siempre son divertidas, son geniales. ¡Somos los mejores tekkeneros del país!

Casi al final de la carrera descubrí que en C.U. hay clases gratis de Taekwondo, y yo siempre soñé con poder ejecutar patadas estilizadas y elegantes. Llevo tres años de entrenamiento y lo sigo soñando. Gracias a todos mis amigos de la clase del Profesor José Samano Hernández. Amigos, son personas que admiro y aprecio, todos ustedes ocupan un lugar muy especial en mi vida. Ojalá sigamos juntos en esta aventura. Gracias Gus, Sonia, Dalia, Angelito, Andrea, Ángel Aguilar, María, Esther, Arely, Rodrigo *Cuau*, Rodrigo *doc*, Lili, Haidé, Mariana, Eréndira, José Parra, Sandra, Sandy, Aldo, Samuel, Tomás, Andrés, Miguel, Tania, Diego, Lucero, René, Sofía, Zaid, Fer, Nico, Toribio y los que me faltaron. En especial quiero agradecer al Prof. Samano por todos sus consejos, sus enseñanzas, su ejemplo. Pero quiero dar un agradecimiento especial al Prof. Ernesto: usted es una de las personas que más admiro y respeto; gracias por formarnos con dedicación, por transmitir tantas enseñanzas en la clase. Gracias por su apoyo, profesor. Lo quiero mucho.

Ahora quiero agradecer a todos los profesores que me dieron clase a lo largo de mi carrera, de todos aprendí algo, pero quiero hacer una mención especial a aquellos que me dejaron enseñanzas realmente valiosas. Gracias (en el orden que los conocí) a mis queridos profesores Araceli Liliana, Lulú, Leonardo Faustino, Rocío del Pilar, Melisa Vivanco, Francisco Raggi †, Frank Patrick Murphy, Rodolfo Conde, Miguel Ángel Pérez León, Cris Camacho, Hugo Alberto Rincón, Jorge Luis Arjona, Susana León, Ernesto Yusmarg, Daniela Terán, Ilán Goldfeder, Alejandria Dosal, Héctor Méndez, Emily Sánchez, Juan Orendaín, Rafael Barbachano, Laura Leonides, Fabiola García, Hortensia Galeana, José Cruz Zagal, Ángel Zaldivar, Octavio Páez, Manuel Alcántara, Ángel Renato, Noé Hernández, David Peñaloza. Todos ustedes son mis héroes y mi ejemplo.

Gracias a mis sinodales, la Dra Diana Avella, el Dr Hugo Rincón, la Dra Lourdes González, y la M en C Selene Linares. Gracias por sus valiosas observaciones y sugerencias.

Por último; no por ser el menos importante, todo lo contrario, quiero agradecer al Dr Favio Miranda Perea por darme la oportunidad, primero de ser su ayudante, y luego su tesista. Muchísimas gracias, profesor, mi vida en la facultad cobró sentido cuando lo conocí: logré encontrar a alguien con quien conversar de temas de computación y matemáticas, descubrí que dar clase es algo que me apasiona y logré concretar este trabajo. Muchas gracias, profesor. Es el mejor.

CDMX, 27 de junio del 2016.

OBJETIVO

El objetivo de este trabajo se divide en dos partes:

- Mostrar la importancia de las mónadas en la programación puramente funcional.

  En la parte I de este trabajo se definen todos los conceptos provenientes de la teoría de las categorías necesarios para entender el concepto de mónada y terna de Kleisli, al final se muestra una prueba rigurosa de la equivalencia entre estos dos conceptos. En la parte II se habla a detalle de la importancia de las mónadas en la programación funcional, esto es, se muestra cómo es que, utilizando mónadas, es posible implementar características de la programación imperativa dentro de la programación puramente funcional sin perder la pureza del lenguaje. Conceptos como manejo de errores, continuaciones, estados, excepciones y más se implementan de manera sencilla y elegante en lenguajes puramente funcionales, como Haskell, utilizando conceptos categóricos que fueron definidos hace más de cuarenta años.

- Formalizar la prueba de la equivalencia entre mónadas y ternas de Kleisli.

  En la parte III se da una breve introducción a las matemáticas formalizadas, una línea de investigación que conecta a las matemáticas y a las ciencias de la computación a través de los asistentes de prueba: software que permite formalizar teoremas y sus demostraciones en la computadora para verificar de manera interactiva su correctud. También se da una breve introducción al asistente de prueba que usaremos a lo largo de este trabajo: Coq. Hecho esto, se formalizan todos los conceptos matemáticos necesarios dentro del asistente para precisar y verificar la correctud de la prueba dada en la primera parte de este trabajo.

Todo lo anterior con el fin de mostrar que las ciencias de la computación pueden ser de gran importancia en cualquier rama de las matemáticas: utilizando asistentes de prueba para verificar demostraciones matemáticas no triviales.

ÍNDICE GENERAL

I TEORÍA DE CATEGORÍAS 9
1. CONCEPTOS BÁSICOS 13
  1.1. Categorías 13
  1.2. Funtores 15
  1.3. Transformaciones naturales 19
  1.4. Mónadas 20
  1.5. Ternas de Kleisli 24
2. EQUIVALENCIA ENTRE MÓNADAS Y TERNAS DE KLEISLI. 27

II MÓNADAS EN LA PROGRAMACIÓN FUNCIONAL 31
3. MÓNADAS EN HASKELL 35
  3.1. ¿Qué es una mónada en un lenguaje de programación funcional? 35
  3.2. Manejo de errores con mónadas 36
  3.3. Notación *do* 37
  3.4. Listas 41
  3.5. Estado 45
  3.6. Continuaciones 48

III LA PRUEBA FORMAL 53
4. FORMALIZANDO LA PRUEBA EN COQ 59
  4.1. Coq 59
  4.2. Formalizando estructuras matemáticas en Coq 63
  4.3. Formalizando teoría de categorías en Coq 65
5. VERIFICANDO LA PRUEBA EN COQ 71

Al ser las matemáticas un lenguaje, no sólo pueden servir para informar, sino también para seducir.
Benoît Mandelbrot

No hay rama de las matemáticas, por abstracta que sea, que no pueda aplicarse algún día a los fenómenos del mundo real.
Nikolai Lobachevski

Parte I

TEORÍA DE CATEGORÍAS

INTRODUCCIÓN

Category theory provides an elegant and powerful means of expressing relationships across a wide area of mathematics.
Alfio Martini

Category theory is a relatively young branch of pure mathematics that most computer scientists would consider esoteric.
Benjamin C. Pierce

Samuel Eilenberg y Saunders Mac Lane introdujeron en sus trabajos de topología algebraica los primeros conceptos de la teoría de categorías [EM45]. El desarrollo de las categorías continuó en áreas como el álgebra homológica, geometría algebraica, lógica matemática, teoría matemática de la música y, hoy en día, es utilizada fuertemente en ciencias de la computación en áreas como lenguajes de programación, razonamiento automatizado, teoría de tipos, entre otras [Pie88]. Además, las categorías son una alternativa a la teoría de conjuntos como fundamento de las matemáticas utilizando ciertas categorías llamadas toposes [BW13]. El isomorfismo de Curry-Howard mostró la relación directa que tienen los programas de computadora y las demostraciones matemáticas [How80]. Más adelante, Joachim Lambek mostró que la relación entre programas y pruebas podía ser extendida a las categorías cartesianamente cerradas formando la correspondencia Curry-Howard-Lambek [LS88]. Una aplicación directa de este isomorfismo se hace en la verificación formal, específicamente, en los asistentes de pruebas, los cuales, reciben las demostraciones traducidas a un lenguaje de programación funcional para ser verificadas paso por paso y, finalmente, validadas [PCG+15].

Actualmente las categorías son utilizadas prácticamente en todas las ramas de las ciencias de la computación, por lo que es muy importante verificar esta teoría. Esto es tema de las llamadas *matemáticas formalizadas*, área de intersección entre las ciencias de la computación y las matemáticas, cuya importancia creciente se debe, entre otros, a Vladimir Voevodsky (medalla Fields 2002) que está convencido en que las computadoras redefinirán las raíces de las matemáticas [wir15]: los nuevos fundamentos serán teorías de tipos, y los asistentes de pruebas se encargarán de verificar que todas las pruebas sean correctas [Voe14b].

En esta parte se presentan todos los conceptos relacionados con teoría de categorías utilizados en este trabajo, primero dando su definición y luego varios ejemplos ilustrativos. Al final se presenta una prueba de la equivalencia entre mónadas y ternas de Kleisli.

1

CONCEPTOS BÁSICOS

CATEGORÍAS

What we are probably seeking is a "purer"view of functions: a theory of functions in themselves, not a theory of functions derived from sets. What, then, is a pure theory of functions? Answer: category theory.
Dana Scott

De manera intuitiva, una categoría está formada por puntos y flechas: las flechas van de un punto a otro, cada punto tiene una flecha que sale de él y llega a él; además, las flechas pueden unirse para formar otras flechas. Muchas propiedades u objetos matemáticos se pueden expresar fácilmente por medio de puntos y flechas, un ejemplo claro son los conjuntos y las funciones entre conjuntos. Ejemplos donde las categorías y las ciencias de la computación se conectan fuertemente son el diseño y especificación de lenguajes de programación funcional, semántica del cómputo concurrente, especificación y desarrollo de algoritmos, teoría de tipos y polimorfismo, entre muchas otras [Pie91, BW90, RB88].

La siguiente definición fue tomada de [SFSÁ07].

Definición 1. Una categoría $\mathcal{C}$ consiste de:

- Una clase[1] de objetos llamada $Ob(\mathcal{C})$.
- Por cada par de objetos a y b en $Ob(\mathcal{C})$, una clase de flechas de a en b denotada $Hom_{\mathcal{C}}(a,b)$. Si f es una flecha que está en $Hom_{\mathcal{C}}(a,b)$, la denotaremos $a \xrightarrow{f} b$.
- Por cada a en $Ob(\mathcal{C})$, la flecha id_a en $Hom_{\mathcal{C}}(a,a)$.
- Por cada terna a,b,c en $Ob(\mathcal{C})$, una operación de composición:

$$(_ \circ _)_{a,b,c} : Hom_{\mathcal{C}}(b,c) \times Hom_{\mathcal{C}}(a,b) \to Hom_{\mathcal{C}}(a,c)$$

tales que:

ASOCIATIVIDAD: Si a,b,c,d son objetos de $\mathcal{C}$; $a \xrightarrow{f} b, b \xrightarrow{g} c, c \xrightarrow{h} d$, se cumple que:

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

IDENTIDAD IZQUIERDA: Para cualesquiera a,b objetos en $Ob(\mathcal{C})$ y $a \xrightarrow{f} b$, se cumple que:

$$id_b \circ f = f$$

[1] Si φ es un predicado de conjuntos, $\{x|\varphi(x)\}$ es la clase de todos los conjuntos que cumplen φ. Todo conjunto es una clase, si A es conjunto, $\{x|x \in A\}$ es una clase que coincide con A. No toda clase es un conjunto, por ejemplo, las clases $\{x|x = x\}$ y $\{x|x \notin x\}$ no son conjuntos.

IDENTIDAD DERECHA: Para cualesquiera a, b objetos en $Ob(\mathcal{C})$ y $a \xrightarrow{f} b$, se cumple que:

$$f \circ id_a = f$$

Usualmente a las flechas también se les llama *morfismos*.

De acuerdo a [ML71], la idea fundamental de representar una función con una flecha apareció aproximadamente en el año 1940, posiblemente en [Hur41].

Las categorías están en todas partes, prueba de ello son los siguientes ejemplos:

- La categoría **Set** consiste en lo siguiente:
  - $Ob(\mathbf{Set})$ es la clase de todos los conjuntos.
  - Si A, B son conjuntos, definimos:

  $$Hom(A, B) := \{f \subseteq A \times B | f \text{ es función}\}$$

  - Sabemos que cada conjunto A tiene asociada la función identidad I_A.
  - La composición de morfismos es la composición usual de funciones.

  Es claro que la composición de funciones entre conjuntos es asociativa y que la función identidad en cada conjunto funciona como neutro de la composición.

  De manera similar tenemos las siguientes categorías:

- **Groups**: los objetos son los grupos y los morfismos los homomorfismos de grupos.
- $\mathbf{Vect_K}$: los objetos son los K-espacios vectoriales y los morfismos las transformaciones líneales.
- **Ab**: grupos abelianos y homomorfismos de grupos.
- **Top**: espacios topológicos y funciones continuas.
- **Poset**: conjuntos con un orden parcial y funciones monótonas.
- Un monoide es una terna $(M, *, e_M)$, tal que, M es un conjunto no vacío, $* : M \times M \to M$ es una función binaria en M que es asociativa y $e_M \in M$ es el neutro a izquierda y derecha de la operación $*$. Cada monoide $(M, *, e_M)$ puede ser visto como una categoría como sigue:
  - Definimos $Ob(M)$ como el conjunto $\{\bullet_M\}$, es decir, nuestra categoría sólo tiene un objeto.
  - Definimos $Hom(\bullet_M, \bullet_M) := M$, los morfismos son los elementos de M.
  - El morfismo identidad es e_M.
  - Por último; si $m_1, m_2 \in M$, definimos $m_1 \circ m_2 := m_1 * m_2$.

  Es inmediato a partir de los axiomas de monoide que los axiomas de categoría se cumplen. Este ejemplo muestra que los morfismos no necesariamente deben ser funciones.

- Consideremos un lenguaje de programación funcional con los siguientes tipos primitivos:

```
Int
Real
Bool
Unit
```

las operaciones:

```
iszero : Int->Bool
not : Bool->Bool
succ : Int->Int
sqrt : Real->Real
toInt : Real->Int
```

y las constantes:

```
zero : Int
true : Bool
false : Bool
unit : Unit
```

la categoría **FPL** *(Functional Programming Languaje)* consiste en:

  - Los objetos de **FPL** son los tipos del lenguaje.
  - Los morfismos son las operaciones `iszero, not,` etc.
  - Añadimos a cada tipo su función identidad.
  - La composición de morfismos es la composición usual de operaciones.

Un ejemplo de esto es la categoría **Hask**, que tiene por objetos a los tipos de Haskell y por morfismos a las funciones entre tipos.

FUNTORES

Functors arise naturally in algebra.
Saunders Mac Lane

Un funtor es una flecha entre dos categorías. La idea es relacionar sus objetos y sus morfismos de manera que se preserven aspectos importantes de ambas: sus composiciones y sus identidades. En muchas áreas de las matemáticas, como el álgebra, los funtores aparecen de manera natural. Las ciencias de la computación no son la excepción.

La siguiente definición también fue tomada de [SFSÁ07].

Definición 2. Un funtor de la categoría $\mathcal{C}$ a la categoría $\mathcal{D}$, denotado por $F : \mathcal{C} \to \mathcal{D}$, consiste de:

- Un funcional[2] entre las clases $Ob(\mathcal{C})$ y $Ob(\mathcal{D})$, denotado:

$$F_{Ob} : Ob(\mathcal{C}) \to Ob(\mathcal{D})$$

[2] Un funcional F es una asignación $F : \mathcal{C}_1 \to \mathcal{C}_2$ entre dos clases que se comporta como una función.

- Para cualesquiera objetos a, b en $Ob(\mathcal{C})$, un funcional que depende de los objetos a y b, denotado por:

$$F_{a,b} : Hom_{\mathcal{C}}(a,b) \rightarrow Hom_{\mathcal{D}}(F_{Ob}(a), F_{Ob}(b))$$

Es usual omitir los subíndices para el funtor al momento de aplicarlo en objetos o morfismos, por ejemplo, si a es un objeto de $\mathcal{C}$, en lugar de escribir $F_{Ob}(a)$ simplemente escribimos $F(a)$; si $a \xrightarrow{f} b$ es un morfismo en $\mathcal{C}$, en lugar de escribir $F_{a,b}(f)$ simplemente escribimos $F(f)$.

Estos funcionales deben cumplir:

PRESERVA COMPOSICIONES: Si $a \xrightarrow{f} b, b \xrightarrow{g} c$ son dos morfismos en $\mathcal{C}$, entonces:

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

PRESERVA IDENTIDADES: Para todo a en $Ob(\mathcal{C})$, se cumple que:

$$F(id_a) = id_{F(a)}$$

Ejemplos:

- Si $\mathcal{C}$ es una categoría, el funtor identidad $Id_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C}$ consiste en dejar fijos a los objetos y a los morfismos de $\mathcal{C}$. Trivialmente preserva composiciones e identidades.
- Si $F : \mathcal{C} \rightarrow \mathcal{D}$ y $G : \mathcal{D} \rightarrow \mathcal{E}$ son funtores, $G \circ F : \mathcal{C} \rightarrow \mathcal{E}$ es el funtor que resulta de componer los funtores F y G en los objetos y en los morfismos. Es inmediato verificar que la composición de funtores es un funtor.
- Hay un funtor F : **Set** $\rightarrow$ **Mon** que asigna a cada conjunto X el monoide X^* que consiste de todas las palabras sobre el alfabeto X con la operación de concatenación y la cadena vacía como neutro. Para definir F en los morfismos, si $f : X \rightarrow Y$, definimos $F(f) : X^* \rightarrow Y^*$ como sigue: si $s_1 s_2 ... s_n$ es una palabra en X^*, $F(f)(s_1 s_2 ... s_n) = f(s_1) f(s_2) ... f(s_n)$. A este funtor se le conoce como *funtor libre*. De igual manera hay un funtor libre para grupos, para espacios vectoriales, etc.
- El funtor G : **Mon** $\rightarrow$ **Set** que asigna a cada monoide $(M, *, e_M)$ el conjunto subyacente y a cada morfismo de monoides la misma función se le conoce como *funtor de olvido*, pues *olvida* la estructura algebraica del conjunto. También hay funtores de olvido para grupos, espacios vectoriales, etc.
- En varios lenguajes de programación se utiliza una estructura de datos fundamental llamada *lista*, la cual, sirve para agrupar varios objetos en uno. La definición en abstracto de una lista de tipo A es la siguiente:

$$List_A = \begin{cases} nil & \\ & \\ cons(x, \ell) & x \in A, \ell \in List_A \end{cases}$$

es decir, una lista es la constante *nil*, que representa a la lista vacía; o es la función constructora *cons* que recibe el par *(x,ℓ)*, cuya primera entrada es un objeto de tipo A y la segunda entrada es una lista de tipo A.

En Haskell, las listas agrupan objetos del mismo tipo y su representación es la siguiente:

$$\textbf{data}\ [a] = [\,]\ |\ a : [a]$$

La palabra *data* sirve para definir nuevos tipos en el lenguaje; la variable a es una variable de tipo. Así, podemos definir listas de cualquier tipo, es decir, listas polimórficas. Sabiendo esto, para crear una lista de números enteros que tenga por elementos del 1 al 5 hay que escribir 1:(2:(3:(4:(5:[])))), o simplemente [1,2,3,4,5]. En la parte II de este trabajo se hablará con más detalle de las listas en Haskell.

Una función muy útil para trabajar con listas es la función *map*:

```
map::(a → b)→[a]→[b]
map f [] = []
map f (x : xs) = (f x):(map f xs)
```

Básicamente, *map* aplica una función a cada elemento de una lista dada. Por ejemplo, al hacer:

$$\texttt{map}\ (\lambda n \to n\texttt{+3})^3\ \texttt{[1,2,3,4]}$$

obtenemos la lista [4,5,6,7].

Sabiendo todo lo anterior, vamos a mostrar que el constructor de listas en **Hask** es un funtor de la siguiente manera: a cada tipo a asigna el tipo $[a]$, y a cada función $f : a \to b$ le asigna la función $(\texttt{map}\ f) : [a] \to [b]$. Para probar esto haremos inducción estructural sobre listas. El principio de inducción para listas es el siguiente:

> Sea φ un predicado acerca de listas de tipo A, entonces, si se prueba:
>
> (I) $\varphi([])$.
>
> (II) Al suponer que xs es una lista de tipo A, x de tipo A, y $\varphi(xs)$; podemos concluir $\varphi(x : xs)$.
>
> Entonces podemos concluir que para cualquier lista ℓ de tipo A se cumple $\varphi(\ell)$.

Sea x de tipo a, debemos probar que el constructor de listas preserva composiciones e identidades, es decir, tenemos que probar que las igualdades:

```
map (g . f) x = map g (map f x)
map id x = id x
```

se cumplen.

El operador punto sirve para componer funciones dentro de Haskell y la función `id` está definida de la siguiente manera: $\texttt{id} = \lambda x \to x$.

[3] La notación $\lambda x \to t$ sirve para definir funciones anónimas dentro del cálculo lambda. En Haskell se utiliza el símbolo "\" en lugar de "λ", sin embargo, en todo este trabajo utilizaremos el símbolo "λ".

Para mostrar que se preservan las composiciones hay dos casos, si x es la lista vacía ambas igualdades se cumplen de manera trivial; en otro caso, x es de la forma $(y : ys)$. Supongamos válido para la lista ys; entonces, para la primera ecuación:

$$\begin{aligned}
\texttt{map}\ (g\ .\ f)\ (y:ys) &= (((g\ .\ f)\ y)\texttt{:}(\texttt{map}\ (g\ .\ f)\ ys)) && \texttt{(map)}\\
&= ((g\ .\ f)\ y)\texttt{:}(\texttt{map}\ g\ (\texttt{map}\ f\ ys)) && \text{(H.I.)}\\
&= (g\ (f\ y))\texttt{:}(\texttt{map}\ g\ (\texttt{map}\ f\ ys)) && \text{(comp)}\\
&= \texttt{map}\ g\ ((f\ y)\texttt{:}(\texttt{map}\ f\ ys)) && \texttt{(map)}\\
&= \texttt{map}\ g\ (\texttt{map}\ f\ (y:ys)) && \texttt{(map)}
\end{aligned}$$

Para mostrar que se preservan identidades, de manera análoga, si x es la lista vacía se cumple por definición. Supongamos válido para la lista ys, entonces:

$$\begin{aligned}
\texttt{map id}\ (y:ys) &= (\texttt{id}\ y)\texttt{:}(\texttt{map id}\ ys) && \texttt{(map)}\\
&= (\texttt{id}\ y)\texttt{:}ys && \text{(H.I.)}\\
&= y : ys && \text{(id)}\\
&= \texttt{id}\ (y:ys) && \text{(id)}
\end{aligned}$$

□

Describir las propiedades de los morfismos en una categoría puede complicarse muy fácilmente, especialmente cuando se involucran propiedades del estilo «$f_1 \circ g_1 = g_2 \circ f_2$». Para que las cosas sean más fáciles de entender utilizaremos una representación gráfica que simplifica notablemente las ideas en teoría de categorías.

Definición 3. Un *diagrama* en una categoría $\mathcal{C}$ es una gráfica dirigida cuyos vértices y aristas están etiquetados con objetos y morfismos de $\mathcal{C}$ de manera consistente, es decir, si una arista en el diagrama está etiquetada con un morfismo f y se tiene que $a \xrightarrow{f} b$ en $\mathcal{C}$, entonces la arista f va del vértice a al vértice b.

Los diagramas se utilizan para afirmar y demostrar propiedades de construcciones categóricas, la siguiente clase de diagramas nos será de gran utilidad.

Definición 4. Decimos que un diagrama *conmuta* si para cualesquiera par de vértices v, w en el diagrama, todos los caminos de v a w son iguales por composición.

Por ejemplo, decir que el siguiente diagrama conmuta:

$$\begin{array}{ccc}
X & \xrightarrow{f'} & Z \\
{\scriptstyle g'}\downarrow & & \downarrow{\scriptstyle g} \\
W & \xrightarrow{f} & Y
\end{array}$$

equivale a decir que $f \circ g' = g \circ f'$.

TRANSFORMACIONES NATURALES

I didn't invent categories to study functors; I invented them to study natural transformations.
Saunders Mac Lane

Así como los funtores pueden ser vistos como flechas entre categorías, una transformación natural puede pensarse como una flecha entre funtores. Hasta ahora, el símbolo de flecha ha sido utilizado para denotar morfismos entre objetos y funtores entre categorías; para no seguir abusando del lenguaje, utilizaremos la notación $F \Rightarrow G$ para referirnos a una transformación natural entre dos funtores.

De nuevo la definición fue tomada de [SFSÁ07].

Definición 5. Si $F, G : \mathcal{C} \to \mathcal{D}$ son dos funtores que van de la categoría $\mathcal{C}$ a la categoría $\mathcal{D}$, una transformación natural $\eta : F \Rightarrow G$ consiste en lo siguiente:

- Por cada objeto x en $Ob(\mathcal{C})$, un morfismo $\eta_x : F(x) \to G(x)$ tal que, si $a \xrightarrow{f} b$ es cualquier morfismo en $\mathcal{C}$, entonces:

$$G(f) \circ \eta_a = \eta_b \circ F(f)$$

  es decir, el siguiente diagrama conmuta:

$$\begin{array}{ccc} F(a) & \xrightarrow{F(f)} & F(b) \\ \downarrow{\scriptstyle \eta_a} & & \downarrow{\scriptstyle \eta_b} \\ G(a) & \xrightarrow{G(f)} & G(b) \end{array}$$

Ejemplos:

- Si $F : \mathcal{C} \to \mathcal{D}$ es un funtor, la transformación natural identidad $id_F : F \Rightarrow F$ queda definida por $\eta_x = id_{F(x)}$.
- El funtor $\mathcal{P}$: **Set** $\to$ **Set** asigna a cada conjunto su conjunto potencia. La transformación natural $pot : Id_{\mathbf{Set}} \to \mathcal{P}$ queda definida por $\eta_X = \{X\}$.
- Sea **R1ng** la categoría de los anillos con 1, sea $n \in \mathbb{N}$ y:

$$U, GL(n, _) : \mathbf{R1ng} \to \mathbf{Groups}$$

  los funtores tales que $U(R) = \{x \in R | x \text{ es unidad}\}$ y $GL(n, R)$ son todas las matrices invertibles de tamaño n por n con entradas en R. Así, $det : GL(n, _) \Rightarrow U$, donde $\eta_A = det(A)$. Es decir, si $f : R \to S$ es un morfismo de anillos, el siguiente diagrama:

$$\begin{array}{ccc} GL(n,R) & \xrightarrow{f} & GL(n,S) \\ \downarrow{\scriptstyle det} & & \downarrow{\scriptstyle det} \\ U(R) & \xrightarrow{f} & U(S) \end{array}$$

  conmuta.

- En Haskell la función *reverse* toma una lista con elementos de tipo x e invierte su orden, por ejemplo, al hacer `reverse [1,2,3,4]` obtenemos la lista [4,3,2,1]. La definición de *reverse* es la siguiente:

$$\begin{aligned}&\texttt{reverse::}[a] \to [a]\\&\texttt{reverse [] = []}\\&\texttt{reverse } (x : xs) \texttt{ = reverse } xs \texttt{ ++ } [x]\end{aligned}$$

Así, en la categoría **Hask** tenemos la transformación natural $rev : \texttt{[]} \Rightarrow \texttt{[]}$[4], donde η_x es la función $\texttt{reverse}:[x] \to [x]$. En efecto, si $f : a \to b$ y l es una lista de tipo $[a]$, es sencillo probar por inducción que:

$$\texttt{(map } f\texttt{) (reverse } l\texttt{) = reverse ((map } f\texttt{) } l\texttt{)}$$

CASO BASE: Por una parte:

$$\begin{aligned}\texttt{(map } f\texttt{) (reverse [])} &= \texttt{(map } f\texttt{) []} && \text{(def)}\\ &= \texttt{[]} && \text{(map)}\end{aligned}$$

por otra parte:

$$\begin{aligned}\texttt{reverse ((map } f\texttt{) [])} &= \texttt{reverse []} && \text{(map)}\\ &= \texttt{[]} && \text{(def)}\end{aligned}$$

HIPÓTESIS DE INDUCCIÓN: Supongamos válido el enunciado para la lista xs, es decir, se cumple que:

$$\texttt{(map } f\texttt{) (reverse xs) = reverse ((map } f\texttt{) xs)}$$

PASO INDUCTIVO: Para la lista $(x : xs)$, se tiene que:

$$\begin{aligned}(\texttt{map } f)\ (\texttt{reverse } (x : xs)) &= (\texttt{map } f)\ (\texttt{reverse } xs \texttt{ ++ } [x]) && \text{(reverse)}\\ &= (\texttt{map } f)\ (\texttt{reverse } xs) \texttt{ ++ } (\texttt{map } f)\ [x] && \text{(map)}\\ &= \texttt{reverse } ((\texttt{map } f)\ xs) \texttt{ ++ } (\texttt{map } f)\ [x] && \text{(H.I.)}\\ &= \texttt{reverse } ((\texttt{map } f)\ xs) \texttt{ ++ } [f\ x] && \text{(map)}\\ &= \texttt{reverse } ((f\ x) : ((\texttt{map } f)\ xs)) && \text{(reverse)}\\ &= \texttt{reverse } ((\texttt{map } f)\ (x : xs)) \quad \square\end{aligned}$$

Utilizando todos los conceptos anteriores vamos a definir el concepto más importante de este trabajo: las mónadas.

MÓNADAS

A monad is just a monoid in the category of endofunctors, what's the problem?
James Iry

Godement usó mónadas en sus trabajos de gavillas y cohomología [God58], él las llamó *construcciones estándar*. Huber mostró que dos funtores adjuntos crean una mónada [Hub61]. Por otra parte, Kleisli [Kle65]; y, de manera independiente, Eilenberg y Moore probaron la inversa [EM+65]. El término *mónada* lo introdujo Mac Lane [ML71].

[4] Nos estamos refiriendo a [] como el constructor de listas y no como la lista vacía.

Alrededor de 1990, Eugenio Moggi utilizó mónadas para dar semántica a los lenguajes de programación funcional [Mog91, Mog89]. Posteriormente, Philip Wadler se inspiró en el trabajo de Moggi para introducir las mónadas en la programación funcional con el objetivo de modelar características de la programación imperativa como estado, continuaciones y excepciones [Wad92a, Wad92b, Wad95, Wad98].

Esta vez la definición fue tomada de [Mog91].

Definición 6. Si $\mathcal{C}$ es una categoría, una mónada sobre la categoría $\mathcal{C}$ es una terna (T, η, μ) compuesta por:

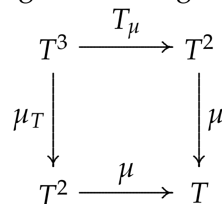
- $T : \mathcal{C} \to \mathcal{C}$ un funtor.
- $\eta : Id_{\mathcal{C}} \Rightarrow T$ una transformación natural.
- $\mu : T \circ T \Rightarrow T$ una transformación natural.

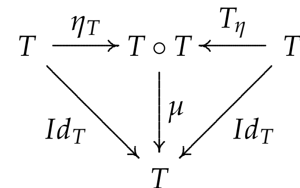
satisfaciendo las siguientes igualdades:

$$\mu \circ \mu_T = \mu \circ T_\mu \qquad (m_1)$$

$$\mu \circ \eta_T = Id_T = \mu \circ T_\eta \qquad (m_2)$$

es decir, los siguientes diagramas conmutan:



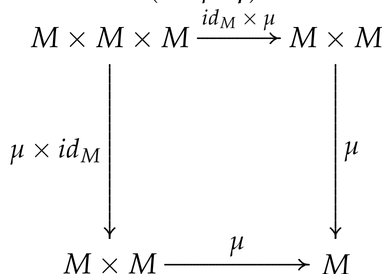


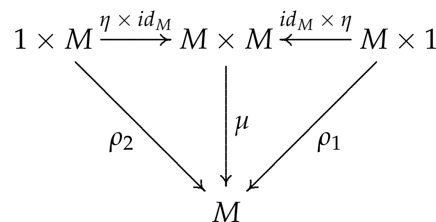
El primer diagrama expresa que μ es asociativa, el segundo muestra que η es neutro de μ a izquierda y derecha.

Se le llama *mónada* por su parecido con un monoide, como dijimos en la parte I, un monoide es una terna $(M, *, e)$, donde M es un conjunto no vacío, $* : M \times M \to M$ una operación binaria asociativa y $e \in M$ es neutro a izquierda y derecha de $*$. Podemos caracterizar un monoide usando diagramas conmutativos como sigue:

- Sean $\rho_1, \rho_2 : M \times M \to M$ definidas como $\rho_j(m_1, m_2) = m_j$
- $\mu : M \times M \to M$ definida como $\mu(m_1, m_2) = m_1 * m_2$
- $\eta : 1 \to M$, donde $1 = \{e\}$ y, $\eta(e) = e$

Así, la terna (M, μ, η) es un monoide si y sólo si los siguientes diagramas conmutan:





Es claro que estos diagramas y los diagramas de mónada son prácticamente los mismos.

Ejemplos:

- Si $\mathcal{C}$ es una categoría, el funtor $Id_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ es una mónada junto con las transformaciones $\eta_a = id_a$ y $\mu_a = id_a$.

- Si $(P, \leq)$ es un orden parcial podemos pensar en la categoría P que tiene por objetos a los elementos de P: si $a, b \in P$, existe un morfismo $a \to b$ si $a \leq b$. Claramente todo objeto en P tiene su morfismo identidad por ser una relación reflexiva y la asociatividad de la composición está forzada por la transitividad de la relación.

  Un funtor $T : P \to P$ debe cumplir que si $a \leq b$, entonces $T(a) \leq T(b)$; por tanto, T es una función monótona.

  Para que T sea una mónada, deben existir las transformaciones naturales $\{\eta_x : x \to T(x)\}_{x \in P}$ y $\{\mu_x : T(T(x)) \to T(x)\}_{x \in P}$, es decir, debe cumplirse que $x \leq T(x)$ y $T(T(x)) \leq T(x)$, para cada x en P. De las dos desigualdades se concluye que $T(T(x)) = T(x)$.

  Por tanto, una mónada en un orden parcial visto como categoría será cualquier función monótona idempotente. A estas funciones se les conoce como *operadores de clausura*.

- En la categoría **Hask**, el funtor $[\,]$: **Hask**$\to$**Hask** es una mónada. Para mostrarlo, sea a un tipo de Haskell:

  - $\eta_a : a \to [a]$ está definida por $\eta_a\ (x) = [x]$
  - $\mu_a : [[a]] \to [a]$ está definida por $\mu_a = \texttt{concat}$

  concat es una función que colapsa una lista de listas en una sola lista, por ejemplo, `concat` $[[1],[2],[3]]$ = $[1,2,3]$. La definición de *concat* es la siguiente:

```
concat::[[a]]→[a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

  Ahora, debemos probar que η y μ son transformaciones naturales. Para esto, haremos inducción estructural sobre listas.

  En adelante, sea L el funtor $L = [\,]$. Entonces, si $f : a \to b$, debemos probar que $\mu_b \circ L^2(f) = L(f) \circ \mu_a$. Sea x una lista de tipo $[[a]]$; si $x = [\,]$ trivialmente se cumple la igualdad; si x es de la forma $y : ys$, entonces:

$$
\begin{aligned}
(\mu_b \circ L^2(f))\ (y : ys) &= \texttt{concat}\ (L^2(f)\ (y : ys)) && \text{(def)} \\
&= \texttt{concat}\ ((\texttt{map}\ f\ y) : (L^2(f)\ ys)) && \text{(map)} \\
&= (\texttt{map}\ f\ y) +\!+ (\texttt{concat}\ (L^2(f)\ ys)) && \text{(concat)} \\
&= ((L(f)\ y) +\!+ (\mu_b\ (L^2(f)\ ys)) && \text{(def)} \\
&= (L(f)\ y) +\!+ ((\mu_b \circ (L^2(f)))\ ys) && \text{(comp)} \\
&= (L(f)\ y) +\!+ ((L(f) \circ \mu_a)\ ys) && \text{(H.I.)} \\
&= (L(f)\ y) +\!+ (L(f)\ (\mu_a\ ys)) && \text{(comp)} \\
&= L(f)\ (y +\!+ (\mu_a\ ys)) && \text{(map)} \\
&= L(f)\ (\mu_a\ (y : ys)) && \text{(concat)} \\
&= (L(f) \circ \mu_a)\ (y : ys) \quad \square
\end{aligned}
$$

Para ver que η es una transformación natural debemos probar que $\eta_b \circ f = L(f) \circ \eta_a$. Sea x de tipo a, entonces:

$$\begin{aligned}
(\eta_b \circ f)\ x &= \eta_b\ (f\ x) & \\
&= [f\ x] & \text{(def)} \\
&= \texttt{map}\ f\ [x] & \text{(map)} \\
&= L(f)\ (\eta_a\ x) & \text{(def)} \\
&= (L(f) \circ \eta_a)\ x \quad \square &
\end{aligned}$$

Para mostrar que se cumplen las igualdades m_1 y m_2, sea x una lista de tipo $[[[a]]]$; si $x = []$ entonces la igualdad se cumple, si x es de la forma $y : ys$, entonces:

$$\begin{aligned}
(\mu_a \circ L_{\mu_a})(y : ys) &= \mu_a\ (L_{\mu_a}\ (y : ys)) & \text{(def)} \\
&= \mu_a\ ((\mu_a\ y) : (L_{\mu_a}\ ys)) & \text{(map)} \\
&= (\mu_a\ y) +\!\!+ ((\mu_a \circ L_{\mu_a})\ ys) & \text{(concat)} \\
&= (\mu_a\ y) +\!\!+ ((\mu_a \circ \mu_{L_a})\ ys) & \text{(H.I.)} \\
&= \mu_a\ (y +\!\!+ ((\mu_{L_a})\ ys)) & \text{(concat)} \\
&= \mu_a\ (\mu_{L_a}\ (y : ys)) & \text{(concat)} \\
&= (\mu_a \circ \mu_{L_a})\ (y : ys) \quad \square &
\end{aligned}$$

Por otra parte, queremos probar que $\mu \circ L_\eta = Id_L = \mu \circ \eta_L$; sea x de tipo $[a]$, si $x = []$ ambas igualdades se cumplen; si x es de la forma $(y : ys)$:

$$\begin{aligned}
(\mu_a \circ L_{\eta_a})\ (y : ys) &= \mu_a\ (L_{\eta_a}\ (y : ys)) & \text{(def)} \\
&= \mu_a\ ((\eta_a\ y) : (L_{\eta_a}\ ys)) & \text{(map)} \\
&= (\eta_a\ y) +\!\!+ ((\mu_a \circ L_{\eta_a})\ ys) & \text{(concat)} \\
&= (\eta_a\ y) +\!\!+ (Id_{L_a}\ ys) & \text{(H.I.)} \\
&= (\eta_a\ y) +\!\!+ ys & \text{(id)} \\
&= [y] +\!\!+ ys & \text{(def)} \\
&= y : ys & \\
&= Id_{L_a}\ (y : ys) \quad \square &
\end{aligned}$$

para la otra igualdad:

$$\begin{aligned}
(\mu_a \circ \eta_{L_a})\ x &= \mu_a\ (\eta_{L_a}\ x) & \\
&= \mu_a\ [x] & \text{(def)} \\
&= x & \text{(concat)} \\
&= Id_{L_a}\ x \quad \square &
\end{aligned}$$

Las mónadas en Haskell no siguen esta definición categórica, en su lugar, utilizan la idea de *noción de cómputo*, idea que se puede expresar utilizando ternas de Kleisli.

TERNAS DE KLEISLI

Kleisli triples are just an alternative description for monads.
Eugenio Moggi

La idea de Moggi [Mog91] para dar semántica a un lenguaje de programación dentro de una categoría $\mathcal{C}$ es distinguir a un objeto A que denota valores (de tipo A), y al objeto $T(A)$ que denota cómputos (que generan valores de tipo A), donde $T : Ob(\mathcal{C}) \to Ob(\mathcal{C})$ es un funcional. A T se le llama *noción de cómputo*. Dentro de la teoría de conjuntos podemos pensar en la noción de cómputo parcial como sigue:

$$T(A) = A + \{\emptyset\}$$

Cuando $T(A)$ es un cómputo exitoso devolvemos un $a \in A$, en otro caso, devolvemos $\emptyset$. Moggi buscó las propiedades que comparten todas las nociones de cómputo y pidió que los programas formen una categoría, lo último equivale a pedir que T sea una terna de Kleisli.

La siguiente definición, al igual que la anterior, viene de [Mog91].

Definición 7. Una terna de Kleisli sobre una categoría $\mathcal{C}$ es la terna $(T, \eta, _^*)$ que consiste de:

- $T : Ob(\mathcal{C}) \to Ob(\mathcal{C})$ un funcional.
- Por cada a en $Ob(\mathcal{C})$, un morfismo $\eta_a : a \to T(a)$.
- Por cada $f : a \to T(b)$ en $\mathcal{C}$, el morfismo $f^* : T(a) \to T(b)$.

que satisfacen las siguientes igualdades:

K1) ${\eta_a}^* = id_{T(a)}$

K2) $f^* \circ \eta_a = f$, donde $f : a \to T(b)$

K3) $g^* \circ f^* = (g^* \circ f)^*$, donde $f : a \to T(b)$ y $g : b \to T(c)$

Intuitivamente, η_a es la inclusión de valores en cómputos y f^* es la extensión de una función de valores en cómputos en una función de cómputos en cómputos, la cual, primero evalúa un cómputo y luego aplica f al resultado. Cabe resaltar que no necesariamente T es un funtor ni η una transformación natural.

Ahora presentamos la categoría cuyos morfismos son los programas con la noción de cómputo T.

Definición 8. Si $(T, \eta, _^*)$ es una terna de Kleisli sobre la categoría $\mathcal{C}$, la *categoría de Kleisli* denotada $\mathcal{C}_T$ se define como:

- $Ob(\mathcal{C}_T) = Ob(\mathcal{C})$
- $Hom_{\mathcal{C}_T}(a, b) = Hom_{\mathcal{C}}(a, T(b))$
- $id_a = \eta_a$
- Para f en $Hom_{\mathcal{C}_T}(a, b)$, g en $Hom_{\mathcal{C}_T}(b, c)$, definimos $g \circ f$ como $g^* \circ f : a \to T(c)$

Así, de los axiomas de terna de Kleisli se sigue inmediatamente que, para
$f : a \to T(b), g : b \to T(c), h : c \to T(d)$:

asociatividad: $h^* \circ (g^* \circ f) = (h^* \circ g)^* \circ f$

identidad izquierda: $\eta_b^* \circ f = f$

identidad derecha: $f^* \circ \eta_a = f$

Ejemplos:

- La noción de cómputo $T(A) = A + \{\emptyset\}$ es una terna de Kleisli, donde $\eta_A = inl_{\{\emptyset\}}$ y, si $f : A \to T(B)$, entonces:

$$f^*(x) = \begin{cases} \emptyset & \text{si} \quad x = \emptyset \\ f(x) & \text{si} \quad x \in A \end{cases}$$

  En la categoría **Hask** esta noción de cómputo es el tipo *Maybe*, definido como:

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

  donde $\eta_a = \texttt{Just}$; si $f : a \to b$, $f^*(\texttt{Nothing}) = \texttt{Nothing}$ y $f^*(\texttt{Just}\ \ a) = f(a)$

  En la parte II hablaremos a detalle del tipo *Maybe*.

- Podemos pensar en nociones de cómputo con listas en **Hask** como sigue:

  - $\eta_a(x) = [x]$
  - Si $f : a \to [b]$ y l es una lista de tipo $[a]$, $f^*(l) = \texttt{concat}\ \ (\texttt{map}\ \ f\ l)$

  donde, como dijimos antes:

```
concat::[[a]]→[a]
concat [] = []
concat (xs : xss) = xs ++ concat xss
```

  Es sencillo probar que la noción de listas en Haskell es una terna de Kleisli.

  k1) Para la lista vacía trivialmente se cumple. Supongamos válido para la lista *ys*, entonces:

$$\begin{aligned} \eta_a(y:ys)^* &= \texttt{concat}\ \ (\texttt{map}\ \ \eta_a\ (y:ys)) && \text{(def)} \\ &= \texttt{concat}\ \ (\eta_a\ y) : (\texttt{map}\ \ \eta_a\ ys) && \text{(map)} \\ &= [y]{+}{+}(\texttt{concat}\ \ (\texttt{map}\ \ \eta_a\ ys)) && \text{(concat)} \\ &= [y]{+}{+}ys && \text{(H.I.)} \\ &= y : ys \end{aligned}$$

  k2) Sea $f : a \to [b]$ y sea x de tipo a, entonces:

$$\begin{aligned} (f^* \circ \eta_a)(x) &= \texttt{concat}\ \ (\texttt{map}\ \ f\ [x]) && \text{(def)} \\ &= \texttt{concat}\ \ [f\ x] && \text{(map)} \\ &= f\ x && \text{(concat)} \end{aligned}$$

K3) Sean $f : a \to [b], g : b \to [c]$. Para la lista vacía el resultado es inmediato. Supongamos válido para la lista ys, entonces:
$(g^* \circ f^*)\ (y : ys) =$ `concat (map` g `(concat (map` f $(y : ys)$`)))`
(map)
$=$ `concat (map` g `(concat (`$f\ y$`) : (map` $f\ ys$`)))`
(concat)
$=$ `concat (map` g `((`$f\ y$`)++concat (map` $f\ ys$`)))`
(map)
$=$ `concat ((map` g `(`$f\ y$`))++(map` g `(concat (map` $f\ ys$`))))`
(concat)
$=$ `(concat (map` g `(`$f\ y$`)))++(concat (map` g `(concat (map` $f\ ys$`))))`
(def)
$= ((g^* \circ f)\ y)$++$((g^* \circ f^*)(ys))$
(H.I.)
$= ((g^* \circ f)\ y)$++$((g^* \circ f)^*(ys))$
(def)
$= ((g^* \circ f)\ y)$++`(concat (map` $(g^* \circ f)\ ys$`))`
(concat)
$=$ `concat (((`$g^* \circ f$`)` y`) : (map` $(g^* \circ f)\ ys$`))`
(map)
$=$ `concat (map` $(g^* \circ f)\ (y : ys)$`)`
(def)
$= (g^* \circ f)^*(y : ys) \quad \square$

Las ternas de Kleisli son más fáciles de usar a la hora de justificar nociones de cómputo. Por otra parte, las mónadas poseen muchas ventajas matemáticas y hay más referencias en la literatura sobre éstas. Por fortuna, son conceptos equivalentes.

2

EQUIVALENCIA ENTRE MÓNADAS Y TERNAS DE KLEISLI.

Incidently, the equivalence explains the other name "triples"for monads that appears in much of the literature.
Shauna C. A . Gammon

Manes se refirió a las mónadas como *teorías algebraicas en forma monoidal* y mostró que son equivalentes a las *teorías algebraicas en forma de clon* [Man76]. Posteriormente, siguiendo la prueba de Manes, Moggi dio un esquema de la prueba para mostrar que las mónadas son equivalentes a las ternas de Kleisli [Mog91].

Dicha prueba no figura en la literatura usual y sólo pudimos encontrarla en [Gam05]. Al leerla notamos que no era una prueba inmediata, lo que nos llevó al objetivo final de este trabajo: realizar una verificación formal de la demostración; la cual, hasta donde pudimos averiguar, no ha sido verificada formalmente en ningún otro sitio hasta ahora.

A continuación presentamos la prueba de manera detallada:

Teorema: *Hay una correspondencia uno-a-uno entre mónadas y ternas de Kleisli.*

Demostración:

Sea $(T, \eta, _^*)$ una terna de Kleisli sobre una categoría $\mathcal{C}$, para construir la mónada (T, η, μ) extendemos T en los morfismos de $\mathcal{C}$ como sigue: si $a \xrightarrow{f} b$, $T(f) = (\eta_b \circ f)^*$

Para completar esta prueba, antes probaremos algunas afirmaciones:

Afirmación: *T es un funtor*.

Por una parte:

$$\begin{aligned} T(id_a) &= (\eta_a \circ id_a)^* && \text{(def)} \\ &= (\eta_a)^* && \text{(id)} \\ &= id_{T(a)} && \text{(k1)} \end{aligned}$$

ahora, sean $f : a \to b, g : b \to c$; entonces:

$$\begin{aligned} T(g \circ f) &= (\eta_c \circ (g \circ f))^* && \text{(def)} \\ &= ((\eta_c \circ g) \circ f)^* && \text{(asoc)} \\ &= (((\eta_c \circ g)^* \circ \eta_b) \circ f)^* && \text{(k2)} \\ &= ((\eta_c \circ g)^* \circ (\eta_b \circ f))^* && \text{(asoc)} \\ &= (\eta_c \circ g)^* \circ (\eta_b \circ f)^* && \text{(k3)} \\ &= T(g) \circ T(f) \end{aligned}$$

Afirmación: *η es una transformación natural.*

Sea $a \xrightarrow{f} b$, debemos mostrar que el siguiente diagrama conmuta:

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ \eta_a \downarrow & & \downarrow \eta_b \\ T(a) & \xrightarrow{T(f)} & T(b) \end{array}$$

entonces:

$$\begin{aligned} T(f) \circ \eta_a &= (\eta_b \circ f)^* \circ \eta_a && \text{(def)} \\ &= \eta_b \circ f && \text{(k2)} \end{aligned}$$

Finalmente, sea $\mu_a = (id_{T(a)})^*$

Afirmación: *μ es una transformación natural.*

Nuevamente, para $a \xrightarrow{f} b$ debemos mostrar que el diagrama:

$$\begin{array}{ccc} T^2(a) & \xrightarrow{T^2(f)} & T^2(b) \\ \mu_a \downarrow & & \downarrow \mu_b \\ T(a) & \xrightarrow{T(f)} & T(b) \end{array}$$

conmuta. Entonces:

$$\begin{aligned} T(f) \circ \mu_a &= (\eta_b \circ f)^* \circ (id_{T(a)})^* && \text{(def)} \\ &= ((\eta_b \circ f)^* \circ id_{T(a)})^* && \text{(k3)} \\ &= ((\eta_b \circ f)^*)^* && \text{(id)} \\ &= (id_{T(b)} \circ (\eta_b \circ f)^*)^* && \text{(id)} \\ &= (((id_{T(b)})^* \circ \eta_{T(b)}) \circ (\eta_b \circ f)^*)^* && \text{(k2)} \\ &= ((id_{T(b)})^* \circ (\eta_{T(b)} \circ (\eta_b \circ f)^*))^* && \text{(asoc)} \\ &= (id_{T(b)})^* \circ (\eta_{T(b)} \circ (\eta_b \circ f)^*)^* && \text{(k3)} \\ &= \mu_b \circ (\eta_{T(b)} \circ (\eta_b \circ f)^*)^* && \text{(def)} \\ &= \mu_b \circ (T(\eta_b \circ f))^* && \text{(def)} \\ &= \mu_b \circ T(T(f)) = \mu_b \circ T^2(f) && \text{(def)} \end{aligned}$$

Ahora, debemos mostrar que se cumplen las leyes monádicas. Por una parte, para el diagrama:

$$\begin{array}{ccc} T^3(a) & \xrightarrow{T_{\mu_a}} & T^2(a) \\ \mu_{T(a)} \downarrow & & \downarrow \mu_a \\ T^2(a) & \xrightarrow{\mu_a} & T(a) \end{array}$$

tenemos que:

$$\begin{aligned} \mu_a \circ \mu_{T(a)} &= (id_{T(a)})^* \circ (id_{T^2(a)})^* && \text{(def)} \\ &= ((id_{T(a)})^* \circ (id_{T^2(a)}))^* && \text{(k3)} \\ &= ((id_{T(a)})^*)^* && \text{(id)} \\ &= (id_{T(a)} \circ (id_{T(a)})^*)^* && \text{(id)} \\ &= (((id_{T(a)})^* \circ \eta_{T(a)}) \circ (id_{T(a)})^*)^* && \text{(k2)} \\ &= ((id_{T(a)})^* \circ (\eta_{T(a)} \circ (id_{T(a)})^*))^* && \text{(asoc)} \\ &= (id_{T(a)})^* \circ (\eta_{T(a)} \circ (id_{T(a)})^*)^* && \text{(k3)} \\ &= \mu_a \circ (\eta_{T(a)} \circ \mu_a)^* && \text{(def)} \\ &= \mu_a \circ T_{\mu_a} && \text{(def)} \end{aligned}$$

por otra parte, para el diagrama:

$$\begin{array}{ccccc} T(a) & \xrightarrow{\eta_{T(a)}} & T^2(a) & \xleftarrow{T_{\eta_a}} & T(a) \\ & \searrow^{Id_{T(a)}} & \downarrow \mu_a & \swarrow^{Id_{T(a)}} & \\ & & T(a) & & \end{array}$$

tenemos que:

$$\begin{aligned} \mu_a \circ \eta_{T(a)} &= (id_{T(a)})^* \circ \eta_{T(a)} && \text{(def)} \\ &= id_{T(a)} && \text{(k2)} \\ &= {\eta_a}^* && \text{(k1)} \\ &= (id_{T(a)} \circ \eta_a)^* && \text{(id)} \\ &= (((id_{T(a)})^* \circ \eta_{T(a)}) \circ \eta_a)^* && \text{(k2)} \\ &= ((id_{T(a)})^* \circ (\eta_{T(a)} \circ \eta_a))^* && \text{(asoc)} \\ &= (id_{T(a)})^* \circ (\eta_{T(a)} \circ \eta_a)^* && \text{(k3)} \\ &= (id_{T(a)})^* \circ T_{\eta_a} && \text{(def)} \\ &= \mu_a \circ T_{\eta_a} && \text{(def)} \end{aligned}$$

Ahora, sea (T, η, μ) una mónada, para construir la terna de Kleisli $(T, \eta, _^*)$ definimos la extensión de $f : a \to T(b)$ como $f^* = \mu_b \circ T(f)$, así:
k1:

$$\begin{aligned} \eta_a^* &= \mu_a \circ T(\eta_a) && \text{(def)} \\ &= id_{T(a)} && \text{(m2)} \end{aligned}$$

k2: Sea $f : a \to T(b)$, entonces:

$$f^* \circ \eta_a = (\mu_b \circ T(f)) \circ \eta_a \qquad \text{(def)}$$

por la naturalidad de η, el diagrama:

$$\begin{array}{ccc} a & \xrightarrow{f} & T(b) \\ \eta_a \downarrow & & \downarrow \eta_{T(b)} \\ T(a) & \xrightarrow{T(f)} & T^2(b) \end{array}$$

conmuta, entonces:

$$\begin{aligned} \mu_b \circ (T(f) \circ \eta_a) &= \mu_b \circ (\eta_{T(b)} \circ f) \\ &= (\mu_b \circ \eta_{T(b)}) \circ f && \text{(asoc)} \\ &= id_{T(b)} \circ f && \text{(m2)} \\ &= f && \text{(id)} \end{aligned}$$

k3: Sean $f : a \to T(b), g : b \to T(c)$; entonces:

$$\begin{aligned}
g^* \circ f^* &= \mu_c \circ T(\mu_c \circ T(g) \circ f) && \text{(def)}\\
&= \mu_c \circ T(\mu_c) \circ T^2(g) \circ T(f) && \text{(funtor)}\\
&= (\mu_c \circ T(\mu_c)) \circ T^2(g) \circ T(f) && \text{(asoc)}\\
&= (\mu_c \circ \mu_{T(c)}) \circ T^2(g) \circ T(f) && \text{(m1)}\\
&= \mu_c \circ (\mu_{T(c)} \circ T^2(g)) \circ T(f) && \text{(asoc)}
\end{aligned}$$

por la naturalidad de μ, el diagrama:

$$\begin{array}{ccc}
T^2(b) & \xrightarrow{T^2(g)} & T^3(c)\\
\mu_b \downarrow & & \downarrow \mu_{T(c)}\\
T(b) & \xrightarrow{T(g)} & T^2(c)
\end{array}$$

es conmutativo, entonces:

$$\begin{aligned}
\mu_c \circ (\mu_{T(c)} \circ T^2(g)) \circ T(f) &= \mu_c \circ (T(g) \circ \mu_b) \circ T(f)\\
&= (\mu_c \circ T(g)) \circ (\mu_b \circ T(f)) && \text{(asoc)}\\
&= g^* \circ f^* && \text{(def)}
\end{aligned}$$

□

Es claro que no es una prueba inmediata, se utilizan todos los conceptos definidos anteriormente y muchas de las igualdades se logran completar de forma *truculenta*. Fue esto lo que más nos motivo a realizar una verificación formal de esta demostración.

En la parte III se realiza la verificación formal de esta prueba en Coq, para esto, también definiremos formalmente todos los conceptos involucrados.

Ahora vamos a hablar de la importancia de las mónadas en la programación puramente funcional.

Parte II

MÓNADAS EN LA PROGRAMACIÓN FUNCIONAL

INTRODUCCIÓN

En definitiva, las mónadas eran la cuadratura del círculo: permitían un estilo imperativo, con costes imperativos en ciertas partes del programa, a la vez que permitían mantener un estilo funcional en el resto y todo ello de un modo coherente.
Ricardo Peña Marí

Haskell is faster than C++, more concise than Perl, more regular than Python, more flexible than Ruby, more typeful than C#, more robust than Java, and has absolutely nothing in common with PHP.
Audrey Tang

Un lenguaje de programación puramente funcional, como Haskell, ofrece características como la evaluación perezosa, funciones de orden superior y el poder escribir programas de manera transparente utilizando sólo especificaciones de funciones matemáticas. Un lenguaje de programación imperativo e impuro, como C, ofrece características como los ciclos *while* y *for*, excepciones, continuaciones y el poder escribir programas como una secuencia de instrucciones que alteran el estado de sus variables.

Un factor que podría influenciar qué estilo de programación elegir es la facilidad con la cual un programa puede ser modificado. Muchas veces en un programa escrito en un lenguaje puramente funcional, un cambio aparentemente pequeño requiere una extensa reestructuración, mientras que en un contexto imperativo e impuro se puede hacer alterando unas cuantas líneas de código. Por ejemplo, si implementáramos un intérprete de expresiones aritméticas en un lenguaje puramente funcional y quisiéramos manejar los errores, como la división entre cero, necesitaríamos modificar el resultado incluyendo un tipo de dato *error* y en cada llamada recursiva verificar y manejar los errores apropiadamente. En cambio, en un lenguaje imperativo e impuro no es necesario reestructurar el programa, basta usar excepciones y estado. Si quisiéramos añadir un contador o un ambiente de entrada y salida, las modificaciones al código puro serían muchas, mientras que en un lenguaje imperativo e impuro bastaría usar una variable global y manipular el estado del programa.

Las mónadas permiten incorporar muchos aspectos de la programación imperativa dentro de los lenguajes puramente funcionales como ambientes de entrada/salida, estado, excepciones, continuaciones, entre otras. [PJW93, Lau93].

En toda la segunda parte usaremos el estilo de programación de Haskell, sin embargo, las mónadas también existen en otros lenguajes de programación como C, Java, Prolog, Python, entre otros [mon15].

3

MÓNADAS EN HASKELL

¿QUÉ ES UNA MÓNADA EN UN LENGUAJE DE PROGRAMACIÓN FUNCIONAL?

The sheer number of different monad tutorials on the internet is a good indication of the difficulty many people have understanding the concept.
[all15]

Wadler trasladó el concepto categórico de mónada a la programación funcional [Wad92a] pensando en un constructor de tipo M junto con tres funciones:

$$\begin{aligned} \texttt{map} &: (x \to y) \to (M\ x \to M\ y) \\ \texttt{unit} &: x \to M\ x \\ \texttt{join} &: M\ (M\ x) \to x \end{aligned}$$

satisfaciendo los axiomas:

$$\begin{aligned} \texttt{map id} &= \texttt{id} && \text{(i)} \\ \texttt{map}\ (g \cdot f) &= \texttt{map}\ g\ \cdot\ \texttt{map}\ f && \text{(ii)} \\ \texttt{map}\ f\ \cdot\ \texttt{unit} &= \texttt{unit}\ \cdot\ f && \text{(iii)} \\ \texttt{map}\ f\ \cdot\ \texttt{join} &= \texttt{join}\ \cdot\ \texttt{map (map}\ f) && \text{(iv)} \\ \texttt{join}\ \cdot\ \texttt{unit} &= \texttt{id} && \text{(I)} \\ \texttt{join}\ \cdot\ \texttt{map unit} &= \texttt{id} && \text{(II)} \\ \texttt{join}\ \cdot\ \texttt{join} &= \texttt{join}\ \cdot\ \texttt{map join} && \text{(III)} \end{aligned}$$

donde (i) y (ii) garantizan que M es un funtor; (iii) y (iv) que `unit` y `join` son transformaciones naturales; finalmente, (I), (II) y (III) que M es una mónada.

El enfoque actual que se le da a las mónadas en la programación funcional está apegado al concepto de terna de Kleisli [Hug00, PJW93, Wad92b, Gib13], el cual, como se demostró en la parte I, es equivalente al concepto de mónada y es más sencillo de manejar.

Haskell utiliza el concepto de *clase* para implementar mónadas. Una clase es una especificación en abstracto de lo que un tipo debe implementar para formar parte de esa clase. Es el análogo a las definiciones de estructuras algebraicas en matemáticas, por ejemplo, para que un conjunto sea un monoide debe estar equipado con una operación binaria que cumpla los axiomas correspondientes. También se puede pensar como la versión funcional de las *interfaces* en la programación orientada a objetos.

La definición de mónada, en Haskell, es la siguiente:

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

donde m es un constructor de tipos, *return* coincide con η y el operador infijo $(\gg=)$ llamado *bind* funge como el operador de extensión. La función *return* toma un valor y lo encapsula dentro del constructor monádico creando un cómputo trivial. El operador *bind* combina dos nociones de cómputo en secuencia, pasando el resultado del primero como argumento al segundo.

Las funciones deben cumplir los axiomas de terna de Kleisli adaptados como sigue:

$$x \gg= \textbf{return} = x \tag{k1}$$

$$(\textbf{return}\ x) \gg= f = f\ x \tag{k2}$$

$$(x \gg= f) \gg= g = x \gg= (\lambda z \rightarrow (f\ z \gg= g)) \tag{k3}$$

El ejemplo más sencillo de mónada en Haskell es considerar el constructor de tipos:

```
data Id a = Ident a
```

el cual, simplemente toma un valor y lo encapsula con el constructor *Ident*. Para que *Id* forme parte de la clase *Monad* debemos dar implementaciones para las funciones *return* y $\gg=$. Para esto, escribimos:

```
instance Monad Id where
    return = Ident
    Ident x >>= f = f x
```

Es inmediato que los axiomas de terna de Kleisli se cumplen. De esta manera obtenemos la mónada identidad.

A continuación vamos a discutir cómo es que utilizando mónadas podemos capturar y simular diversos mecanismos de la programación imperativa dentro de la programación funcional sin perder la pureza en el lenguaje y de una manera muy sencilla y eficiente.

MANEJO DE ERRORES CON MÓNADAS

Under lazy evaluation, exceptions can be implemented as an abstract data type, rather than as a feature of the programming language.

Philip Wadler

Algunas veces los programas terminan su ejecución satisfactoriamente y producen una salida; otras veces terminan de manera abrupta sin producir resultados, ejemplos de esto podrían ser efectuar una división entre cero, pedir la cabeza o la cola de una lista vacía, entre otras. En los lenguajes imperativos, u orientados a objetos, las excepciones permiten manejar errores cambiando el flujo del programa a un manejador; en un ambiente funcional se pueden manejar los errores de manera sencilla sin necesidad de lanzar una excepción.

El tipo:

```
data Maybe[1] a = Just a | Nothing
```

sirve para manejar posibles fallos en la evaluación de un programa. La idea es devolver algo de la forma *Just* x, x de tipo a cuando el cómputo es exitoso; o bien, devolver *Nothing* en caso de que el cómputo falle. Este tipo hace que una función indique de manera explícita en su firma que podría producir un error.

Por ejemplo, podemos pensar en una función que toma un número y calcula su raíz cuadrada pero falla cuando el número es negativo:

[1] El nombre *Maybe* fue dado en [Spi90].

```
raizSegura::Double→Maybe Double
raizSegura z = if z < 0 then Nothing else Just (sqrt z)
```

Ahora pensemos en el siguiente tipo que representa expresiones aritméticas:

```
data Exp = Num Int | Div Exp Exp
```

Una expresión puede ser un número entero o la división entera de dos expresiones.

Utilizando el tipo *Maybe* se puede implementar un evaluador que maneje el error al hacer una división entre cero:

```
eval::Exp→Maybe Int
eval (Num n) = Just n
eval (Div e1 e2) = case eval e1 of
          Nothing → Nothing
          Just v1 → case eval e2 of
                    Nothing → Nothing
                    Just v2 → if v2 == 0 then Nothing
                              else Just (v1 ÷ v2)
```

Si bien el código se mantiene puro, resulta muy tedioso verificar de manera explícita el error para manejarlo.

Esto puede hacerse más sencillo utilizando mónadas.

El tipo *Maybe* forma parte de la clase *Monad* de la siguiente manera:

```
instance Monad Maybe where
  return = Just
  x >>= f = case x of
              Just y → f y
              Nothing → Nothing
```

Ahora podemos definir funciones que manejen errores sin hacer una verificación explícita de *Just* y *Nothing*.

El evaluador queda adaptado en su versión monádica de la siguiente manera:

```
eval::Exp→Maybe Int
eval (Num n) = Just n
eval (Div e1 e2) = eval e1 >>= λv1 →
                   eval e2 >>= λv2 → if v2 == 0 then Nothing
                                     else Just (v1 ÷ v2)
```

Es posible que a primera vista resulte confuso usar el operador *bind* junto con abstracciones lambda para manejar los errores. Por fortuna las mónadas permiten implementar una notación alternativa muy fácil de leer.

NOTACIÓN *do*

Pure functional languages have this advantage: all flow of data is made explicit. And this disadvantage: sometimes it is painfully explicit.
Philip Wadler

Pensemos en un tipo de dato *Person* el cual sirve para representar una pequeña base de datos familiar, y en las funciones:

```
father::Person→Maybe Person
mother::Person→Maybe Person
```

como consultas que nos devuelven la información de nuestros padres o *Nothing* si no están registrados en la base de datos.

Sabiendo esto, para consultar la información de nuestro abuelo materno, sin usar el operador *bind*, basta la función:

```
maternalGrandfather::Person→Maybe Person
maternalGrandfather p = case mother p of
                             Nothing → Nothing
                             Just m → father m
```

Por otra parte, si quisiéramos obtener la información de ambos abuelos paternos y sin usar el operador *bind*, habría que hacer:

```
bothGrandfathers::Person→Maybe (Person,Person)
bothGrandfathers p = case father p of
                  Nothing → Nothing
                  Just f → case mother f of
                        Nothing → Nothing
                        Just gf → case mother p of
                              Nothing → Nothing
                              Just m → case father m of
                                    Nothing → Nothing
                                    Just gm → Just (gf, gm)
```

Si no usamos el operador *bind*, tenemos que verificar explícitamente todos los posibles errores de cómputo. Por otra parte, si aprovechamos el poder monádico del tipo *Maybe*, el código se reduce bastante.

Veamos la implementación de las dos consultas anteriores utilizando el operador *bind*; primero, la consulta de nuestro abuelo materno:

```
maternalGrandfather::Person→Maybe Person
maternalGrandfather p = mother p >>= father
```

ahora la consulta de nuestros abuelos:

```
bothGrandfathers::Person→Maybe (Person,Person)
bothGrandfathers p = father p >>=
                       λf → father f >>=
                         λgf → mother p >>=
                           λm → father m >>=
                             λgm → return (gf,gm)
```

Las líneas de código se reducen y no hay que hacer verificaciones explícitas para manejar el posible error, pero usar tantas abstracciones lambda puede resultar confuso en la práctica.

Es común que los desarrolladores que utilizan lenguajes imperativos, como C; u orientados a objetos, como Java; sufran con la sintaxis de un lenguaje funcional ya que en esos lenguajes los programas se especifican mediante una secuencia de instrucciones del estilo:

```
action1;
action2;
action3;
⋮
actionN
```

Para incorporar esta sintaxis con estilo imperativo dentro de un lenguaje puramente funcional no es necesario tener que redefinir la estructuración interna de éste ni añadir características extras: basta tener mónadas [Gib13].

En Haskell esto se conoce como *notación do* y está implementada de la siguiente manera:

$$\begin{aligned}
&\textbf{do}\ \{v\} = v\\
&\textbf{do}\ \{x \leftarrow m; p\} = m \gg= \lambda x \rightarrow \textbf{do}\ \{p\}\\
&\textbf{do}\ \{m_1; m_2\} = m_1 \gg \textbf{do}\ \{m_2\}\\
&\textbf{do}\ \{\textbf{let}\ \{x_1 = y_1; \ldots; x_n = y_n\}; p\} = \textbf{let}\ \{x_1 = y_1; \ldots; x_n = y_n\}\ \textbf{in}\ \textbf{do}\ \{p\}
\end{aligned}$$

Analizemos detenidamente cada caso:

- v es un valor monádico y simplemente lo regresamos. Es decir, v es de la forma $m\ x$, donde m es un constructor mónadico y x es un valor.
- Extraemos el contenido de m y lo ligamos a x, finalmente lo pasamos al resultado de evaluar de manera recursiva la expresión **do** $\{p\}$.
- Al definir el operador *bind* automáticamente ganamos el operador *then*:

  $$\begin{aligned}
  &(\gg) :: (\texttt{Monad}\ m) \texttt{=>} m\ a \rightarrow m\ b \rightarrow m\ b\\
  &x \gg y = x \gg= \lambda_ \rightarrow y
  \end{aligned}$$

  Este operador es un caso especial del operador *bind*, simplemente el resultado del segundo cómputo es independiente del primero.
- Hacemos una sustitución simultánea en p, donde $x_i \neq x_j$ siempre que $i \neq j$.

Así, la *notación do* sólo es *azúcar sintáctica*[2] de los operadores monádicos. Sabiendo esto, podemos pasar de la consulta:

```
bothGrandfathers::Person→Maybe (Person,Person)
bothGrandfathers p = father p >>=
                       λf → father f >>=
                         λgf → mother p >>=
                           λm → father m >>=
                             λgm → return (gf,gm)
```

a la siguiente expresión:

```
bothGrandfathers::Person→Maybe (Person,Person)
bothGrandfathers p = do
       f ← father p;
       gf ← father f;
       m ← mother p;
       gm ← father m;
       return (gf, gm)
```

que resulta ser una versión imperativa muy fácil de leer.

La notación *do* es tan cómoda que permite omitir los ";"para encadenar nuestros cómputos, basta dar un salto de línea e indentar cada línea correctamente.

[2] La azúcar sintáctica en un lenguaje de programación es sintaxis que permite expresar ideas de manera más clara y sencilla a la hora de programar.

Podemos reescribir los axiomas de terna de Kleisli utilizando la notación *do* como sigue:

$$\mathbf{do}\ \{m\} = \mathbf{do}\ \{v \leftarrow m; \mathbf{return}\ v\} \quad \text{(k1)}$$
$$\mathbf{do}\ \{f\ x\} = \mathbf{do}\ \{v \leftarrow \mathbf{return}\ x;\ f\ v\} \quad \text{(k2)}$$
$$\mathbf{do}\ \{x \leftarrow m; y \leftarrow f\ x; g\ y\} = \mathbf{do}\ \{y \leftarrow \mathbf{do}\ \{x \leftarrow m; f\ x\}; g\ y\} \quad \text{(k3)}$$

Los lenguajes de programación funcional se caracterizan por ser lenguajes en los que se escribe poco y se hace mucho. En Haskell el clásico *Hello World!* puede ser escrito en una sóla línea así:

```
main = putStrLn ``Hello World!''
```

Haskell maneja internamente el ambiente de Entrada/Salida con la mónada *IO*. La notación *do* es especialmente popular con esta mónada ya que podemos crear sencillos programas interactivos fácilmente. Por ejemplo, un programa que nos pregunta nuestro nombre para luego saludarnos:

```
doName::IO ()
doName = do
     putStr ``What is your first name?''
     first ← getLine
     putStr ``And your last name?''
     last ← getLine
     let full = first ++ `` '' ++ last
     putStrLn (``Pleased to meet you, ''++ full ++ ``!'')
     return ()
```

El tipo *void*, denotado por (), contiene un único elemento que también está denotado por (). Es un valor trivial que devolvemos cuando no nos importa producir una salida en un cómputo, en este caso, imprimir un mensaje en pantalla.

Uno podría pensar que al igual que en C o en Java la palabra *return* termina el cómputo de una función pero no es así. Veamos el siguiente ejemplo:

```
nameReturn::IO ()
nameReturn = do putStr ``What is your first name?''
             first ← getLine
             putStr ``And your last name?''
             last ← getLine
             let full = first++`` ''++last
             putStrLn (``Pleased to meet you, ''++full++``!'')
             return full
             putStrLn ``I am not finished yet!''
```

La última línea siempre aparecerá en pantalla. La función *return* simplemente encapsula la cadena *full* dentro de un valor monádico de tipo *IO String*, este valor es independiente de la última línea, por tanto, esta secuencia de cómputo se lleva a cabo utilizando el operador $\gg$.

Ahora veamos el tipo de las siguientes funciones:

```
readFile::FilePath→IO String
writeFile::FilePath→String→IO ()
```

El tipo *FilePath* sólo es un alias del tipo *String*. Ambas funciones se encargan de toda la tarea que implica manejar archivos: abrir archivo, leer archivo, crear archivo, cerrar archivo, etc.

Veamos el siguiente ejemplo:

```
import Data.Char(toUpper)
main = do
     inpStr ← readFile ‘‘input.txt’’
     writeFile ‘‘output.txt’’ (map toUpper inpStr)
```

en dos líneas hicimos lo siguiente: todo el contenido de *input.txt* fue transferido a un nuevo archivo *output.txt* pero en mayúsculas. ¿Cuántas líneas habría tomado hacerlo en C o en Java?

Existe una inmensa cantidad de funciones para hacernos la vida más sencilla en cuanto a *entrada/salida*.

También existen trabajos realizados en Haskell utilizando la mónada *IO* que sencillamente son impresionantes, por ejemplo, una versión del famoso juego *Super Mario Bros* llamada *Super Nario Bros*. El video que muestra un *in-game* completo se encuentra aquí [nar15].

Para profundizar más en esta mónada recomendamos visitar [rea15, all15, apr15].

listas

Wadler also observed that the methods of the Monad interface are sufficient to implement a notation based on the set comprehensions of Zermelo–Fraenkel set theory.
Jeremy Gibbons

Las listas son un tipo de dato el cual sirve para agrupar valores. En Haskell las listas están representadas recursivamente de la siguiente manera:

$$\textbf{data}\ [a] = [\,]\ \mid\ a : [a]$$

es decir; una lista o bien es vacía, o es un elemento de tipo *a* seguido de una lista. Por ejemplo, una lista con tres números enteros puede escribirse como 1:(2:(3:[])), o usando azúcar sintáctica simplemente como [1,2,3]. Si bien esta notación es más clara y ahorra paréntesis no nos libra de tener que escribir de manera explícita cada elemento en una lista.

En Haskell existe la clase *Enum*, los tipos dentro de esta clase están enumerados y eso permite escribir listas de manera muy compacta. El tipo *Int* es parte de la clase *Enum* con la relación de orden usual; así, para obtener la lista que va del 1 al 100 basta escribir [1..100]. Esta notación a manera de intervalos combinada con la evaluación perezosa permite definir y manipular listas potencialmente infinitas, un ejemplo de esto es la lista [1..] que contiene a todos los enteros positivos.

Además, podemos calcular listas a manera de intervalos especificando la distancia entre sus elementos, es decir, si queremos la lista de todos los números pares entre el 0 y el 100 basta escribir [0,2..100]. Haskell calcula la relación aritmética de los dos primeros elementos de la lista para obtener los elementos restantes, de esta manera, al escribir [5,4..1] obtenemos la lista [5,4,3,2,1].

Todo lo anterior hace que las listas sean un recurso poderoso y fácil de usar. Sin embargo, aún no hemos dicho nada acerca de filtrar elementos en una lista. Una posible opción es considerar la siguiente función:

```
filter::(a → Bool)→[a]→[a]
filter _ [] = []
filter p (x:xs) = if (p x) then p:(filter p xs) else filter p xs
```

que devuelve los elementos en una lista que cumplen con el predicado dado. Sabiendo esto, para calcular todos los números pares entre cero y cien basta escribir: `filter even [0..100]`. Esta función es útil pero limita a filtrar elementos únicamente de una lista y muchas veces se requiere filtrar elementos de varias listas en una sola.

De la teoría de conjuntos sabemos que si φ es un predicado y A es un conjunto, entonces:

$$\{x|x \in A, \varphi(x)\}$$

es un conjunto. A esto se le conoce como *axioma de comprensión* y sirve para especificar los elementos de un conjunto. El análogo en listas son las *listas por comprensión* las cuales existen en varios lenguajes de programación [Lis15].

Nuestro objetivo, inspirado en las *comprensiones*, es que al escribir:

```
[(i,j)|i ←[1,2],j ←[3,4]]
```

nos de por resultado:

```
[(1,3),(1,4),(2,3),(2,4)]
```

Esta idea puede implementarse utilizando mónadas.

En general, una comprensión es de la forma:

$$[t(x_1, x_2, \ldots, x_n) \mid x_1 \leftarrow l_1, x_2 \leftarrow l_2, \ldots, \; x_n \leftarrow l_n]$$

donde $t(x_i)$ es un término cerrado y cada l_i es una lista.

Para esto, hacemos que el tipo lista forme parte de la clase *Monad*:

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
```

Usando azúcar sintáctica definimos una comprensión:

$$[t(x_1, x_2, \ldots, x_n) \mid x_1 \leftarrow l_1, x_2 \leftarrow l_2, \ldots, \; x_n \leftarrow l_n]$$

como:

$$l_1 \gg= \lambda x_1 \rightarrow l_2 \gg= \lambda x_2 \rightarrow \ldots l_n \gg= \lambda x_n \rightarrow \textbf{return} \; t(x_1, x_2, \ldots, x_n)$$

o usando la notación *do*:

$$\textbf{do} \; \{x_1 \leftarrow l_1; x_2 \leftarrow l_2; \ldots; x_n \leftarrow l_n; \; \textbf{return} \; t(x_1, x_2, \ldots, x_n)\}$$

Así, las siguientes expresiones son equivalentes:

- `[(x,y) | x ←[1,2], y ←[3,4]]`
- `[1,2] >>= λx → [3,4] >>= λy → return (x,y)`
- `do {x ←[1,2]; y ←[3,4]; return (x,y)}`

y nos devuelven la lista `[(1,3),(1,4),(2,3),(2,4)]`.

Lo que sigue es poder filtrar elementos dentro de una comprensión, es decir, que al escribir:

$$[(a,b) \mid a \leftarrow [1,2,3], b \leftarrow [4,5,6], \texttt{even}\ (a+b)]$$

obtengamos la lista:

$$[(1,5),(2,4),(2,6),(3,5)]$$

Un filtro es una función de tipo $a \rightarrow \texttt{Bool}$. Queremos poder construir comprensiones del estilo:

$$[t(x_i) \mid x_1 \rightarrow l_1, \ldots, x_n \rightarrow l_n, p_1(x_i), \ldots, p_m(x_i)] \qquad 1 \leq i \leq n$$

donde cada p_j es un filtro.

Las listas no son la única mónada con la cual se pueden hacer comprensiones, Wadler mostró pueden extenderse a cualquier mónada [Wad92a]. Siguiendo esa idea, para que una mónada en Haskell pueda filtrar elementos dentro de una comprensión primero debe formar parte de la clase:

```
class Monad m ⇒ MonadPlus m where
    mzero:: m a
    mplus:: m a → m a → m a
```

Esta clase pide definir una operación binaria y un neutro para esa operación. Hacemos que el tipo lista forme parte de la clase *MonadPlus* de la siguiente manera:

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

la cual resulta natural pues las listas forman un monoide con la concatenación y la lista vacía.

Una vez que un tipo monádico forma parte de la clase *MonadPlus* gana automáticamente la función:

```
guard::(MonadPlus m)⇒Bool→ m ()
guard True = return ()
guard False = mzero
```

Veamos un ejemplo, el siguiente código:

```
do {x ←[1,2,3,4];guard (even x);return x}
```

devuelve [2,4]. Es una función muy conveniente: cuando un elemento no cumple con el predicado dentro de la función *guard*, se aborta el cómputo para ese elemento con la lista vacía; en otro caso, el cómputo continúa gracias a la lista unitaria de tipo *void*.

Siguiendo esta idea, definimos las listas por comprensión con filtros:

$$[t(x_i) \mid x_1 \leftarrow l_1, \ldots, x_n \leftarrow l_n, p_1(x_i), p_2(x_i), \ldots, p_m(x_i)]$$

como sigue:

```
do
    x1 ← l1
    x2 ← l2
    ⋮
    xn ← ln
    guard (p1(xi))
    guard (p2(xi))
    ⋮
    guard (pm(xi))
    return t(xi)
```

Sabiendo esto, las siguientes tres expresiones son equivalentes:

- `[(x,y,z)|x ← [1..10],y ← [1..10],z ← [1..10],x^2 + y^2 == z^2]`
- `[1..10] >>= λx → [1..10] >>= λy → [1..10] >>= λz → guard (x^2 + y^2 == z^2) >> return (x,y,z)`
- **do**

```
do
  x ← [1..10]
  y ← [1..10]
  z ← [1..10]
  guard (x^2 + y^2 == z^2)
  return (x,y,z)
```

y devuelven como resultado [(3,4,5),(4,3,5),(6,8,10),(8,6,10)].

Ahora, mostramos ejemplos donde se utiliza el poder de las listas por comprensión:

- Dado un número entero n, una función que calcula todos los números primos menores o iguales que n:

```
primos::Int → [Int]
primos n = [p|p ← [2..n], esPrimo p] where
                esPrimo n = null (divisores n) where
                divisores n = [m|m ← [2..(n÷2)],mod n m == 0]
```

- El algoritmo de ordenamiento *quicksort*:

```
quickSort::Ord a ⇒ [a] → [a]
quickSort [] = []
quickSort (x : xs) = quickSort [y|y ← xs, y ≤ x] ++
                            [x] ++ quickSort [y|y ← xs, y > x]
```

  Este algoritmo es de los más complicados de implementar en un lenguaje imperativo; aquí resultó sencillo utilizando el poder de las listas por comprensión.

- Dentro de la biblioteca *Control.Monad* existe la función:

```
filterM::(Monad m)⇒(a → m Bool)→[a]→ m [a]
```

  la cual generaliza a la función *filter* definida en listas, por ejemplo, la expresión:

```
filter even [1..4]
```

  es equivalente a la expresión:

```
filterM (λx → return $ even x) [1..4]
```

  y ambas devuelven [2,4].

  Esta función es muy conveniente pues al hacer:

```
filterM (λx → [False,True]) [1,2,3]
```

nos devuelve [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]. La idea detrás de esta función es generar todas las sublistas de la lista [1,2,3] con los valores *False* y *True*, esto es:

- Todos los valores van a dar a *False*: esto genera la lista vacía.
- 3 va a dar a *True*, los demás a *False*: [3].
- 1 y 2 van a dar a *True*, 3 a *False*: [1,2].

y así sucesivamente.

Siguiendo esta idea, podemos calcular la potencia de una lista en una línea:

```
potencia::[a]→[[a]]
potencia xs = filterM (λx → [False,True]) xs
```

Lo bello de esta función es que coincide con el teorema que dice que para obtener el conjunto potencia de un conjunto basta obtener todas las funciones de ese conjunto en un conjunto con dos elementos.

ESTADO

Functional programmers restrict themselves to manipulating expressions, rather than statements.
Jeremy Gibbons

Al ser un lenguaje de programación puro, la salida de cualquier programa escrito en Haskell depende únicamente de sus argumentos de entrada. En un lenguaje impuro, como C, la salida depende de sus argumentos de entrada y del estado de sus variables.

Si bien los lenguajes puramente funcionales ofrecen la ventaja de escribir de manera transparente cualquier programa, los lenguajes impuros ofrecen características como la asignación de variables, excepciones o continuaciones, que muchas veces permiten resolver un problema de manera más sencilla o reducir bastante el código.

Un ejemplo concreto es pensar en un evaluador para las siguientes expresiones aritméticas:

```
data Exp = Num Int | Div Exp Exp
```

Implementarlo en Haskell es muy sencillo:

```
eval::Exp→Int
eval (Num n) = n
eval (Div e1 e2) = (eval e1)÷(eval e2)
```

Ahora, si queremos contar cuántas divisiones se efectúan al evaluar una expresión, en un lenguaje impuro, basta añadir una variable local e irla incrementando con cada división efectuada. Para hacer esto en un lenguaje puramente funcional debemos simular la idea de cómputo con estado. La idea es la siguiente: un cómputo con estado es una función que recibe un estado inicial y devuelve una tupla que contiene el resultado de efectuar el cómputo y el estado final, en otras palabras, es una función de tipo $s \to (a,s)$; donde s indica el tipo del estado y a el tipo de los valores que devuelve.

El evaluador queda adaptado para recibir una expresión y devolver un cómputo con estado de la siguiente manera:

```
eval::Exp→(Int→(Int,Int))
eval (Num n) = λs →(n,s)
eval (Div e1 e2) = λs → let (v1,s1) = eval e1 s in
                        let (v2,s2) = eval e2 s1 in (v1 ÷ v2,s2+1)
```

Resultó relativamente sencillo manejar los cómputos con estado, pero es fácil que las cosas se compliquen si manejamos el estado de manera explícita.

Una *pila* es una estructura de datos abstracta similar a una lista, la diferencia radica en que sólo tenemos acceso al último elemento apilado, para llegar a un elemento que está en medio o al fondo de la pila primero debemos retirar todos los elementos apilados encima de éste. Podemos modelar una pila utilizando listas de la siguiente manera:

```
type Stack a = [a]
```

es decir, una pila con elementos de tipo *a* simplemente es una lista de tipo *a*. La palabra reservada *type* sólo crea un sobrenombre para un tipo ya existente.

Las siguientes funciones alteran el estado de una pila:

```
pop::Stack a →(a,Stack a)
pop (x : xs) = (x,xs)

push::a →Stack a →((),Stack a)
push x xs = ((),x : xs)
```

La primera función toma una pila y devuelve el elemento en el tope, el resto de la pila queda como estado final. La segunda función inserta un elemento en el tope de una pila, como no esperamos un resultado al hacer esta operación devolvemos () y el estado final es la pila con el elemento insertado en el tope.

Vamos a implementar una función llamada *king*[3], la cual, toma una pila de números enteros y retira los primeros tres números apilados; si estos suman 19, se insertan de regreso en la pila el primer y el tercer número y en ese orden; en otro caso, se insertan en la pila, y en ese orden, los números 11, 7 y 2.

La función queda implementada así:

```
king::Stack Int→((),Stack Int)
king = λs →let (x1,s1) = pop s in
           let (x2,s2) = pop s1 in
           let (x3,s3) = pop s2 in
           if (x1+x2+x3 == 19) then
               let ((),t) = push x1 s3 in push x3 t else
                   let ((),t1) = push 11 s3 in
                   let ((),t2) = push 7 t1 in
                   push 2 t2
```

Como vemos en este ejemplo: manejar explícitamente el estado es tedioso.

Las mónadas permiten trabajar con nociones de cómputo con estado sin tener que manejarlo de manera explícita. Siguiendo la idea anterior, definimos el tipo:

```
newtype State s a = ST (s →(a,s))
```

que representa cómputos con estado en *s* y valores en *a*. La palabra reservada *newtype* sirve, al igual que la palabra reservada *data*, para crear nuevos tipos, la diferencia es que *newtype* sólo puede ser usada con tipos que tienen únicamente un constructor. Esto ayuda al compilador a que las tareas de extraer y envolver el contenido del constructor *ST* sea trivial.

Junto con el tipo *State* utilizaremos la función:

```
runState::State s a → s →(a,s)
runState (ST s) s' = s s'
```

[3] En honor a Stephen King. El número 19 es importante en muchas de sus novelas.

que recibe un cómputo con estado y lo evalúa con un estado inicial.

Hacemos que *State* forme parte de la clase *Monad*:

```
instance Monad (State a) where
    return x = ST (λs →(x,s))
    s >>= k = ST (λt → let (v1,s1) = runState s t in runState (k v1) s1)
```

Para *return*, dado un valor, creamos un cómputo con estado que toma un estado inicial y devuelve el mismo valor manteniendo el estado inicial. El operador *bind* crea un cómputo con estado de la siguiente manera: evalúa el estado s, el valor que éste produce lo pasa como argumento a la función k produciendo un cómputo con estado, el cual, evalúa con el estado que produjo la función k.

Al hacer que *State* forme parte de la clase *Monad* manejamos explícitamente el estado sólo una vez, en adelante, la notación *do* nos dará una sintaxis imperativa sin perder la pureza del lenguaje.

Ahora podemos reescribir el evaluador de expresiones aritméticas utilizando la mónada de estado, para esto, la función:

```
tick::State Int ()
tick = ST (λn →((),n+1))
```

incrementará en uno el estado de nuestro evaluador por cada división que se efectúe.

El evaluador con estado queda de la siguiente manera:

```
evalState::Exp→State Int Int
evalState (Num n) = return n
evalState (Div e1 e2) = do
                            v1 ←evalState e1
                            v2 ←evalState e2
                            tick
                            return (v1 ÷ v2)
```

Ahora nuestro evaluador es muy sencillo de leer y entender.

Probemos el nuevo evaluador. Al hacer:

```
runState (evalState (Num 23)) 0
```

nos da como resultado (23,0). Al hacer:

```
(runState (evalState ((Div (Div (Num 1972 ) (Num 2 )) (Num 23 ))))) 0
```

el resultado es (42,2).

Para manejar pilas con estado las funciones quedan reescritas así:

```
pop::State (Stack a) a
pop = ST (λ(x : xs)→(x,xs))

push::a →State (Stack a) ()
push x = ST (λxs →((),x : xs))
```

Finalmente, la función *king* queda reescrita de la siguiente manera:

```
kingState::State (Stack Int) ()
kingState = do
                x1 ←pop; x2 ←pop; x3 ←pop
                if (x1+x2+x3==19) then do
                                push x1; push x3
                                  else do
                                push 11; push 7; push 2
```

En todo momento el estado queda oculto.

Así, al hacer:

```
runState kingState [3,0,1,4,5,1]
```

obtenemos ((),[2,7,11,4,5,1]). Al hacer:

```
runState kingState [10,3,6,2,0,9,9]
```

obtenemos ((),[6,10,2,0,9,9]).

Es fácil añadir arreglos que se manipulan eficientemente a un lenguaje funcional utilizando la mónada de estado. También es posible implementar analizadores sintácticos, renombre de variables, entre otras cosas [Wad95, Wad92b, Wad92a].

CONTINUACIONES

Abuse of the continuation monad can produce code that is impossible to understand and maintain.
Jeff Newbern

Uno de los aspectos más útiles de la programación imperativa es el poder cambiar el flujo de un programa de manera arbitraria, como en C, que podemos abortar un ciclo *for* o *while* con la instrucción *break*. Las continuaciones son la versión funcional para cambiar el flujo de un programa y han sido utilizadas para modelar muchos conceptos como excepciones, co-rutinas, generadores, entre otros [Rey93, con15]. También han sido fuertemente utilizadas en el desarrollo web [web15, Que03]

Una continuación representa un cómputo a futuro. La idea central es que una función no devuelva un valor, en su lugar, debe pasar el resultado a la siguiente continuación; es decir, que una función de tipo $a \to r$ pase a tener tipo $a \to (r \to r') \to r'$. A este estilo de programación se le conoce como CPS (*Continuation Passing Style*) e inicialmente se desarrolló para trabajar con *semántica denotativa* [Rey72, Plo75]. Este estilo también se ha usado para optimizar compiladores [SSJ15].

Veamos unos ejemplos. La función identidad normalmente se define como:

```
id::a → a
id x = x
```

su respectiva definición en CPS queda así:

```
cpsid::a →(a → r)→ r
cpsid x k = k x
```

Ahora nuestra función no se limita a devolver el valor que recibe, devuelve el valor que recibió evaluado con la siguiente continuación. Así, al hacer:

```
(cpsid [6,1,6]) head
```

nos da por resultado 6, al hacer:

```
(cpsid [9,9,0,2]) reverse
```

obtenemos la lista [2,0,9,9].

Ahora pensemos en una función recursiva que toma una lista de enteros y multiplica sus elementos:

```
mult::[Int]→Int
mult [] = 1
mult (x : xs) = x*(mult xs)
```

su respectiva versión en CPS:

```
cpsmult::[Int]→(Int→ r)→ r
cpsmult [] k = k 1
cpsmult (x : xs) k = k $ (cpsmult xs) (λy → x*y)
```

El operador $ sirve para ahorrar paréntesis, lo que hace es asociar a la derecha una aplicación, por ejemplo, es equivalente escribir $f\ (x\ y)$ a escribir f $ $x\ y$.

Por último, pensemos en nuestro evaluador de expresiones aritméticas, recordemos que su implementación era sencilla:

```
eval::Exp→Int eval (Num n) = n
eval (Div e1 e2) = (eval e1)÷(eval e2)
```

en cambio, su versión en CPS:

```
evalcps::Exp→(Int→ r)→ r
evalcps (Num n) k = k n
evalcps (Div e1 e2) k = let v1 = evalcps e1
                            v2 = evalcps e2 in
                            v1 $ λx → v2 $ λy → k $ x ÷ y
```

no lo es.

La mónada de continuaciones ofrece una interfaz para manipular nociones de cómputo en estilo CPS. El tipo para representar continuaciones es el siguiente:

```
newtype Continuation r a = Cont ((a → r)→ r)
```

Una continuación es una noción de cómputo que espera valores de tipo a y produce valores de tipo r.

Hacemos que el tipo *Continuation* forme parte de la clase *Monad*:

```
instance Monad (Continuation r) where
    return x = Cont $ λk → k x
    (Cont c) >>= f = Cont $ λk → c $ λa → runCont (f a) k
```

La función *return* crea una continuación, a la cual, le pasa el mismo valor que recibe. El operador *bind* funciona de la siguiente manera: crea la continuación k, primero, evaluando la continuación c y ligando su resultado al valor a, luego, pasando ese valor a la función f y haciendo que todo continúe con k.

Ahora podemos reescribir nuestras funciones como sigue:

```
cpsid::a →Continuation r a
cpsid x = return x

cpsmult::[Int]→Continuation r Int
cpsmult [] = return 1
cpsmult (x : xs) = do
                        y ←cpsmult xs
                        return $ x*y

evalcps::Exp→Continuation r Int
evalcps (Num n) = return n
evalcps (Div e1 e2) = do
                        v1 ←evalcps e1
                        v2 ←evalcps e2
                        return $ v1 ÷ v2
```

Todas son muy fáciles de leer.

Hasta ahora la mónada de continuaciones únicamente nos permite transformar nuestros programas de manera elegante a su versión en CPS, nos falta decir cómo esta mónada sirve para alterar arbitrariamente el flujo de un programa.

La siguiente función:

```
callCC::((a →Continuation r b)->Continuation r a)→Continuation r a
callCC f = Cont $ λk →runCont (f (λa →Cont $ λ_ → k a)) k
```

simula la operación *call-with-current-continuation*, implementada en Scheme [RC86] como *call/cc* y en Standard ML [DHM91] como *callcc*. Esta función sirve como un mecanismo de escape, el cual, nos permite abortar el cómputo actual y regresar un valor inmediatamente.

Por ejemplo, la función *cpsmult* toma una lista de enteros y multiplica todos sus elementos, sin embargo, es una implementación un tanto ineficiente: si la lista contiene un cero (al inicio de la lista, por ejemplo), innecesariamente se llevarán a cabo todas las multiplicaciones restantes.

Usando el operador *callCC*, es sencillo que la función *cpsmult* aborte el cómputo con la continuación en curso en cuanto reciba un cero:

```
cpsprod0::[Int]→Continuation r Int
cpsprod0 [] = return 1
cpsprod0 (x : xs) = callCC $ λk → do
                                    when (x==0) $ k 0
                                    y ←cpsprod0 xs
                                    return $ x*y
```

El operador *when::Monad m ⇒Bool→ m ()→ m ()* se encuentra dentro de la biblioteca *Control.Monad* y, combinado con el operador *callCC*, nos permite elegir cuándo cambiar el flujo de nuestro programa.

Ahora pensemos en una función llamada *akrag*[4], la cual, toma una cadena y la pasa a mayúsculas; si *akrag* ve un número, se deshace de él y llama a la continuación actual con el resto de la cadena, pero «volteada». La función queda implementada así:

```
import Data.Char(toUpper)

akrag::String→Continuation r String
akrag [] = return []
akrag (x : xs) = callCC $ λk → do
                         when (esNum x) $ k $ reverse xs
                         xs' ←akrag xs
                         return $ (toUpper x):xs' where
                               esNum c = elem c ['0'..'9']
```

toUpper toma un carácter y lo transforma a mayúscula. Entonces:

```
runCont (akrag ‘‘orenanab’’) reverse
```

devuelve "BANANERO". Por otra parte, al hacer:

```
runCont (akrag ‘‘acm2tp1’’) id
```

devuelve "ACM1pt". Por último:

[4] En honor al amigo *bananero*, pionero de los video-bloggers.

```
runCont (akrag ''nev3r forev3r 5'') (++ ''again'')
```

devuelve "NEV5 r3verof ragain".

Las continuaciones también pueden modelar el manejo de excepciones, para mostrarlo, pensemos en nuestro evaluador y en una manera de arrojar una excepción cuando se efectúe una división entre cero. Para esto, el evaluador recibirá como argumento extra un manejador de errores con tipo String→Continuation *r* Int.

El evaluador queda adaptado de la siguiente manera:

```
evalCont::Exp→(String→Continuation r Int)→Continuation r Int
evalCont (Num n) _ = return n
evalCont (Div e1 e2) handler = callCC $ λok → do
                       v2 ←evalCont e2 handler
                       err ←callCC $ λnotOk → do
                          when (v2==0) $ notOk ''denominator 0''
                          v1 ←evalCont e1 handler
                          ok $ v1 ÷ v2
                       handler err
```

Estamos usando dos continuaciones anidadas: la primer continuación, nombrada *ok*, se usará cuando no haya problemas; la segunda, nombrada *notOk*, cuando se efectúe una división entre cero. El evaluador funciona de la siguiente manera: si v_2 no es cero, la función cae dentro de la continuación *ok* y esta nos lleva de vuelta a la entrada del evaluador; cuando v_2 es igual a cero, la continuación *notOk* crea un escape del bloque *do* actual para pasar el error depositado en *err* al manejador.

Finalmente, presentamos la función *newbern*[5], la cual, cuenta con una complicada estructura de control:

```
import Control.Monad(when)
import Data.Char(toUpper,digitToInt,intToDigit)

newbern::Int→String
newbern n = (`runCont` id) $ do
       str ← callCC $ λexit1 → do
             when (n<10) (exit1 (show n))
             let ns = map digitToInt (show (n÷2))
             n' <- callCC $ λexit2 → do
                   when ((length ns)<3) (exit2 (length ns))
                   when ((length ns)<5) (exit2 n)
                   when ((length ns)<7) $ do
                         let ns' = map intToDigit (reverse ns)
                         exit1 (dropWhile (=='0') ns')
                   return $ sum ns
             return $ ''(ns = '' ++ (show ns) ++ '') '' ++ (show n')
       return $ ''Answer: '' ++ str
```

[5] Jeff Newbern presentó originalmente esta función en http://www.nomaware.com/monads/html/index.html. Actualmente esta función se encuentra en [all15] o en [con15].

La función se comporta de esta manera:

Entrada n:	Salida:	Lista de salida (ns):
0-9	n	ninguna
10-199	número de dígitos en $(n/2)$	los dígitos de $(n/2)$
200-19999	n	los dígitos de $(n/2)$
20000-1999999	(n/2) en reversa	ninguna
$n \geq$2000000	la suma de los dígitos de $(n/2)$	los dígitos de $(n/2)$

Algunos ejemplos de salida:

Entrada n	Salida
7	Answer: 7
157	Answer: (ns = [7,8]) 2
250	Answer: (ns = [1,2,5]) 250
10299	Answer: (ns = [5,1,4,9]) 10299
2099	Answer: (ns = [1,0,4,9]) 2099
5402793	Answer: (ns = [2,7,0,1,3,9,6]) 28

Hasta aquí hemos mostrado que las mónadas en Haskell son una poderosa herramienta para construir programas sin sufrir por una notación carente de estilo imperativo.

Para ver ejemplos de cómo se utilizan las mónadas en el mundo real recomendamos visitar [all15].

Para todos los interesados en el desarrollo web recomendamos encarecidamente el entorno de trabajo Yesod[6], el cual utiliza todo el poder de Haskell y otras herramientas para crear aplicaciones web robustas.

Ahora pasamos a la parte final de este trabajo, la cual consiste en presentar una verificación formal de la equivalencia entre mónadas y ternas de Kleisli.

[6] http://www.yesodweb.com/

Parte III

LA PRUEBA FORMAL

¿POR QUÉ FORMALIZAR?

Formalization of mathematics is feasible with modern computer technology and sofware.
John Harrison

Mathematical research currently relies on a complex system of mutual trust based on reputations.
Vladimir Voevodsky

Verification can be time consuming and painful.
Jeremy Avigad

Científicos de la computación, matemáticos, físicos, actuarios, incluso ingenieros, utilizamos lenguaje formal para comunicar ideas: teoremas, algoritmos, especificaciones de sistemas, nuevas teorías, entre muchas otras cosas. Sin embargo, entre más especializados son los temas que se manejan, más ambiguo tiende a ser el lenguaje empleado para definir o explicar los conceptos que se utilizan. En muchos artículos o libros, por ejemplo, las relaciones lógicas o de conjuntos se expresan en lenguaje natural. En áreas del álgebra abstracta, como teoría de categorías, la sobrecarga de notación puede ser confusa, hasta intimidante. Incluso se llegan a hacer demostraciones utilizando frases del estilo: *la prueba es inmediata, es claro que..., se sigue de..., es trivial*, etc.

Algo es cierto: nadie escribe matemáticas formales en su totalidad, no es práctico. La verificación de las demostraciones no siempre se lleva a cabo de manera rigurosa, muchas veces sólo se lee el bosquejo y uno se termina convenciendo de que las cosas son ciertas.

Hay demostraciones que pueden verificarse muy fácilmente y, a veces el mismo enunciado nos parece tan cierto que la prueba se hace más por compromiso que por necesidad. Por desgracia los teoremas más importantes o los más interesantes suelen tener pruebas tan complicadas que requieren de un equipo de personas para comprobar su validez.

La mala noticia es que verificar demostraciones a mano, aunque sea entre varias personas, puede acarrear muchísimos errores que pueden terminar en aprobar resultados completamente falsos; o en nuestro caso: hacer una mala verificación de un software puede resultar fatal, incluso mortal [bug15]:

- *La Mariner 1 (1962): Esta fue la primera misión de la NASA para sobrevolar Venus. El cohete no duró más de 5 minutos en vuelo cuando se desvió de su trayectoria y fue autodestruido por los responsables. El motivo: la omisión de un guión '-' en el programa que controlaba el cohete.*

- *Therac-25 (1985-1987): La Atomic Energy of Canada Limited(AECL) creó una máquina de radioterapia utilizada para entornos médicos. Durante un periodo existieron seis accidentes en el que los pacientes recibieron alta sobredosis de radiación. En las investigaciones se responsabilizó al software tanto en el diseño por ser un código indocumentado y ofuscado como en fallos concretos detectados.*

- *MIM-104 Patriot (1991): Es un misil antiaéreo que puede ser utilizado para interceptar misiles balísticos a modo de defensa. Durante la Guerra del Golfo en 1991 un Scud iraquí mató a 28 soldados al alcanzar un cuartel norteamericano ya que estos misiles fallaron. Se dictaminó que fue un error de software en el reloj del sistema que se había retrasado un tercio de segundo por haber estado activado 100 horas.*

Volvamos a las demostraciones. El teorema de Feit–Thompson, de *teoría de grupos*, afirma lo siguiente: cualquier grupo finito de orden impar es soluble. El enunciado fue demostrado en 1963 por Walter Feit y John Griggs Thompson [FT63]. La prueba requirió 255 páginas. ¿Cuánto tiempo y esfuerzo requeriría verificar esta prueba a mano?

Verificar pruebas puede convertirse en un problema tedioso no sólo por su tamaño, las hay tan complicadas que terminan siendo aceptadas porque vienen de gente a la que se le atribuye una capacidad intelectual sobrenatural.

Vladimir Voevodsky, medalla Fields en 2002, lo vivió [Voe14a]:

> *[...] In 1999/2000, again at the IAS, I was giving a series of lectures, and Pierre Deligne was taking notes and checking every step of my arguments. Only then did I discover that the proof of a key lemma in "Cohomological Theory" contained a mistake and that the lemma, as stated, could not be salvaged. [...]*
>
> *[...] This story got me scared. Starting from 1993 multiple groups of mathematicians studied the "Cohomological Theory" paper at seminars and used it in their work and none of them noticed the mistake. And it clearly was not an accident. A technical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail. [...]*

Actualmente Voevodsky utiliza el asistente de pruebas Coq para verificar sus demostraciones e invita a la comunidad a formalizar matemáticas utilizando asistentes de prueba [Voe11].

Verificar formalmente una prueba nos garantiza su correctud, la pregunta es: ¿para qué verificar una prueba que ya se sabe que es correcta? La respuesta es sencilla: la prueba podría ser dudosa. Ejemplos sobran: el teorema de los cuatro colores, los teoremas de incompletud de Gödel, la infinitud de los números primos, la conjetura de Kepler, el lema de la serpiente, la irracionalidad de e, el lema de Yoneda y muchos otros más.

Formalizar matemáticas consiste en trasladar los teoremas y las demostraciones a un lenguaje formal con el suficiente detalle para que puedan ser verificadas paso a paso de manera algorítmica y, así, probar su correctud. Los asistentes de prueba son la herramienta encargada de verificar las pruebas de manera interactiva.

Formalizar pruebas no sólo nos da la certeza de que las cosas están bien hechas, también nos ayuda a profundizar los conceptos involucrados, dando paso a una nueva manera de enseñar matemáticas, y a tener una interacción directa con las matemáticas a través del software.

En 2012, luego de seis años de esfuerzo, Georges Gonthier y su equipo lograron formalizar el teorema de Feit-Thompson [GAA+13] en Coq. El resultado: cerca de 170,000 líneas de código, donde hubo cerca de 4,200 definiciones, 15,000 teoremas y, de acuerdo a Laurent Théry, muchísima diversión.

También se ha verificado el teorema de los cuatro colores, los teoremas de incompletud de Gödel, el teorema de Ramsey, el teorema fundamental de la teoría de Galois, entre otros. En [Wie15] hay una lista con al menos cien teoremas relevantes que ya han sido verificados.

La verificación formal no es algo que se queda atrapado en la academia, también tiene impacto en la industria.

En 1994, Intel perdió cerca de 475 millones de dólares en reemplazar procesadores que contenían un error en la división de punto flotante. Desde entonces, Intel dedica tiempo a verificar sus procesadores de manera formal [Har99].

El projecto *CompCert* se encargó de verificar un compilador realista y de alta seguridad para el lenguaje C [Ler09]; compatible en un 99 % para el estandard ISO C90 / ANSI C, encargado de generar código para las arquitecturas PowerPC, ARM y x86.

Y hay más ejemplos donde se utilizan métodos formales [Avi94]: Microsoft utiliza herramientas como *Boogie* y *SLAM* para verificar programas y controladores, Airbus para verificar software de aviones, Toyota los utiliza en sistemas híbridos para verificar sistemas de control, París los utilizó para verificar la línea 14 del metro sin conductor, la NSA los utiliza para verificar algoritmos criptográficos.

¿Por qué formalizar? Porque lo necesitamos.

4

FORMALIZANDO LA PRUEBA EN COQ

A lo largo de este capítulo daremos una breve introducción a Coq y definiremos toda la herramienta necesaria para verificar la equivalencia entre mónadas y ternas de Kleisli. Todo está escrito a manera de tutorial para hacer la lectura más llevadera.

COQ

Some French computer scientists have a tradition of naming their software as animal species: Caml, Elan, Foc or Phox are examples of this tacit convention. In French, "coq" means rooster, and it sounds like the initials of the Calculus of Constructions (CoC) on which it is based.

The Coq Development Team

You have to trust that the implementation of the Coq kernel mirrors the theory behind Coq. The kernel is intentionally small to limit the risk of conceptual or accidental implementation bugs.

The Coq Development Team

Usualmente los enunciados en matemáticas tienen un rigor informal, ejemplos de esto son: *hay una infinidad de números primos, derivar es lo opuesto a integrar, los reales no son numerables,* entre otros. Las matemáticas usualmente siguen el esquema:

Teorema. Si Γ*, entonces* A.

Demostración:

$\vdots$

$\therefore A \quad \square$

donde Γ es un conjunto de hipótesis descritas de manera informal, A es un enunciado descrito de manera informal y la prueba que se da, aunque podría ser informal, es rigurosa.

Una manera de darle formalidad a esto es utilizar lógica de primer orden:

Teorema. $\Gamma \vdash_L A$

Demostración:

$1.B_1$

$2.B_2$

$\vdots$

$k.B_k$

$\vdots$

$n.A \quad \square$

donde $\Gamma = \{A_1, A_2, \dots, A_m\}$ y A son fórmulas bien formadas y la demostración es una sucesión que se obtiene aplicando deducción natural. Si bien la prueba está formalmente definida por la relación $\vdash_L$ esta puede contener elementos informales.

Las *teorías de tipos* permiten formalizar en un mismo lenguaje a los teoremas y a sus demostraciones. En la teoría de tipos todos los términos tienen un tipo, para decir que t es de tipo T escribimos $t : T$. Por ejemplo, $2 : Nat$, $\mathsf{suc} : Nat \to Nat$, $Nat : \mathsf{Type}$.

La aplicación de una función a un término dentro de la teoría de tipos se hace simplemente poniendo un espacio en blanco entre la función y el argumento que recibe: suc 3, sum 5 9; aunque también pueden usarse paréntesis para indicar la aplicación de una función. Además, dentro de la teoría de tipos existen reglas de reescritura de términos conocidas como *reducción de términos*. Por ejemplo, (suc 1) y 2 son términos sintácticamente diferentes, pero (suc 1) se *reduce* a 2.

La idea para formalizar matemáticas utilizando teoría de tipos es definir la relación:

$$\Gamma \vdash_T p : A$$

léase: bajo el contexto Γ, el término p tiene tipo A. En esta relación el contexto Γ es una lista $[x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n]$ de variables tipadas; y, tanto p como A, son objetos formales. Utilizando el *isomorfismo de Curry-Howard* [SU06] hacemos que las fórmulas de la lógica correspondan a tipos del lenguaje.

Por ejemplo, mostramos algunos enunciados de manera informal y su formalización con tipos:

- *La suma de naturales es cerrada.*

  La formalización es la siguiente:

  $$[n : Nat, m : Nat] \vdash_T (n + m) : Nat$$

  Básicamente estamos diciendo que, bajo el supuesto de que n y m son números naturales, su suma también es un número natural.

- *La potencia de un conjunto es un conjunto.*

  $$[x : Set] \vdash_T \mathcal{P}\ x : Set$$

- *Para todo natural n, S n = n+1.*

  $$[] \vdash_T \forall n : Nat\ ((\mathsf{suc}\ n = n + 1) : Prop)$$

  A diferencia de los ejemplos anteriores, esta es una proposición que puede probarse.

La relación $\Gamma \vdash_T p : A$ es equivalente a:

$$\text{TYPE}_\Gamma(p) = A$$

donde $\text{TYPE}_{_}(_)$ es una función que encuentra un tipo para p bajo el contexto Γ. En resumen: utilizando teoría de tipos, la verificación de una prueba corresponde a una verificación de tipos.

En la práctica, los asistentes de prueba realizan la verificación de tipos de manera interactiva, esto es, el usuario guía al asistente en la verificación de la prueba utilizando *tácticas* hasta terminar dicha verificación.

Para profundizar más en teoría de tipos y su uso en dentro de los asistentes de prueba recomendamos consultar [BG01].

El asistente de prueba que usaremos en este trabajo será Coq[1].

Coq es un asistente de prueba basado en el *cálculo de construcciones inductivas*, esto es, un cálculo-λ con un suntuoso sistema de tipos. Como se mencionó arriba, a través del

[1] https://coq.inria.fr/

isomorfismo de Curry-Howard las pruebas son identificadas con términos del lenguaje y la verificación de pruebas se convierte en verificación de tipos. Además de permitir construir pruebas formales y verificarlas de manera interactiva, permite escribir programas funcionales consistentes con sus especificaciones; todo esto a través del lenguaje *Gallina*, que es el lenguaje utilizado por Coq. La forma en la que las pruebas se escriben en Coq es constructiva, la principal característica de esta forma de razonar es la ausencia de pruebas por contradicción.

Para ilustrar cómo se interactúa con Coq presentamos de manera detallada el siguiente ejemplo.

Primero recordemos la definición recursiva de los números naturales:

$$\mathbb{N} = \begin{cases} 0 \in \mathbb{N} \\ \\ n \in \mathbb{N} \quad \text{entonces} \quad \mathsf{S}\ n \in \mathbb{N} \end{cases}$$

y la definición recursiva de la suma:

$$\begin{aligned} &0 + n = n \\ &\mathsf{S}\ m + n = \mathsf{S}(m + n) \end{aligned}$$

Ahora consideremos el siguiente enunciado y su demostración:

Proposición. $\forall n \in \mathbb{N}(\mathsf{S}\ n = n + 1)$.

Demostración *(Inducción).*
$n = 0$

$$\begin{aligned} \mathsf{S}\ 0 &= 1 \\ &= 0 + 1 \qquad \text{(def. suma)} \end{aligned}$$

Hipótesis de inducción: $\mathsf{S}\ n = n + 1$
Paso inductivo:

$$\begin{aligned} \mathsf{S}(\mathsf{S}\ n) &= \mathsf{S}(n + 1) \qquad \text{(H.I.)} \\ &= (\mathsf{S}\ n) + 1 \qquad \text{(def. suma)} \end{aligned}$$

□

Vamos a verificar esta demostración en Coq.

Dentro del asistente existe el tipo `nat`, el cual consiste de lo siguiente:

```
0:nat
S:nat->nat
```

también tenemos la función `plus:nat->nat->nat`, la cual, cuenta con una notación infija utilizando el símbolo '+'.

Primero traducimos la proposición dentro del asistente:

```
Proposition suc_esSumar1 : forall n, S n = n+1.
```

Por una parte, la palabra reservada *Proposition* indica al asistente que vamos a escribir un enunciado matemático. También hay otras palabras reservadas como *Definition, Theorem, Lemma*, etc; es indistinto para Coq cuál etiqueta utilizemos. Por otra parte, *suc_esSumar1* es un identificador para nuestro enunciado.

Una vez que introducimos un enunciado a nuestro asistente, este genera una *meta*, es decir, nos indica textualmente qué es lo que tenemos que demostrar, en este caso, la meta generada es la siguiente:

```
1 subgoals
______________________________________(1/1)
forall n : nat, S n = n + 1
```

además, el asistente entra en *modo-demostración*, en este ambiente es donde interactuamos con el asistente utilizando *tácticas*, esto es, estrategias que sirven para guiar al asistente a lo largo de la verificación de la prueba.

Para comenzar a demostrar un enunciado dentro de Coq se comienza con la palabra reservada *Proof*, la cual, indica al asistente que vamos a comenzar una prueba.

Lo siguiente es indicar al asistente que nuestra prueba será por inducción, para esto, escribimos la táctica *induction n*. Una vez hecho esto se generan las metas:

```
2 subgoal
______________________________________(1/2)
1 = 0 + 1
______________________________________(2/2)
S (S n) = S n + 1
```

que corresponden al caso base y al paso inductivo. La primer meta es muy sencilla, tanto que existe una táctica muy especial que sirve para estos casos: *trivial*. Hecho esto, la primer meta queda demostrada, resta probar:

```
1 subgoals
n : nat
IHn : S n = n + 1
______________________________________(1/1)
S (S n) = S n + 1
```

Ahora tenemos dos hipótesis: la primera nos dice que *n* es de tipo *nat*; la segunda es nuestra hipótesis de inducción. La táctica *simpl* reduce términos dentro de la meta, al utilizarla la meta generada es la siguiente:

```
1 subgoals
n : nat
IHn : S n = n + 1
______________________________________(1/1)
S (S n) = S (n + 1)
```

Es decir, se redujo el término $S\ n + 1$ utilizando la definición de suma de naturales al término $S\ (n{+}1)$. Es momento de usar la hipótesis de inducción: la táctica *rewrite IHn* le indica al asistente buscar el término $S\ n$ en nuestra meta y reescribirlo por el término $n{+}1$. La meta generada es:

```
1 subgoals
n : nat
IHn : S n = n + 1
______________________________________(1/1)
S (n + 1) = S (n + 1)
```

Finalmente, utilizamos la táctica *trivial*, a lo que el asistente nos responde:

```
No more subgoals.
```

Para finalizar nuestra prueba, escribimos la palabra reservada *Qed*, que permite al asistente usar este resultado, ya sea para reescribir términos o para *aplicarlo* a una meta. Más adelante explicaremos el uso de la táctica *apply*.

Coq puede cargar archivos con nuestros enunciados y pruebas para ser verificadas, por ejemplo, todo lo anterior se puede guardar en un archivo de texto plano llamado *ejemplo.v* de la siguiente manera:

```
Proposition suc_esSumar1 : forall n, S n = n+1.
Proof.
induction n.
trivial.
simpl.
rewrite IHn.
trivial.
Qed.
```

Hay que notar que las sentencias en Coq se separan con un punto.

Para profundizar más en todo lo relacionado con este asistente, se recomienda visitar [HKPM14, Tea15, coq15, BC13]

formalizando estructuras matemáticas en coq

Type classes are a nice way to formalize (mathematical) structures.
Pierre Castéran & Matthieu Sozeau

Los teoremas en matemáticas se formulan de la manera más general posible. En *teoría de grupos*, por ejemplo, los teoremas se enuncian en abstracto, cualquier resultado automáticamente es válido en cualquier instancia de grupo que a uno se le ocurra: los números enteros, los reales, los racionales; matrices, polinomios, etc. Generalmente estas teorías se valen de estructuras para clasificar objetos, y dichas estructuras tienen una jerarquía: en *teoría de conjuntos* se puede hablar de la relación de *subconjunto*, dentro del álgebra se puede hablar de la relación de *subgrupo, subanillo, subcampo*, etc.

Pensemos en la siguiente función recursiva, la cual, recibe dos números naturales como argumento:

$$f(x,n) = \begin{cases} 1 & \text{si } n = 0 \\ \\ x * f(x,m) & \text{si } n \text{ es de la forma } \quad \mathsf{S}\ m \end{cases}$$

Es inmediato probar por inducción que $\forall n \in \mathbb{N}, f(1,n) = 1$.

Si n es cero, se cumple por definición.

Supongamos que $f(1,m) = 1$, entonces:

$$\begin{aligned} f(1,\mathsf{S}\ m) &= 1 * f(1,m) && \text{(def)} \\ &= 1 * 1 && \text{(H.I.)} \\ &= 1 \quad \square \end{aligned}$$

Es claro que $f(x,n)$ es la función x^n y que se puede definir en otras estructuras como *matrices*, *polinomios*, *los números reales*, y más. También el resultado previo es válido en las estructuras antes mencionadas.

Sería tedioso verificar esta prueba por cada estructura en la que descubramos que es válida.

En lugar de hacer eso, podemos re-definir la función como sigue:
Sea $f : M \times \mathbb{N} \to M$, con $(M, *, e)$ un monoide:

$$f(x,n) = \begin{cases} e & \text{si } n = 0 \\ \\ x * f(x,m) & \text{si } n \text{ es de la forma } \ \mathtt{S}\ m \end{cases}$$

y enunciar el resultado como sigue: $\forall n \in \mathbb{N}, f(e,n) = e$. La prueba es exactamente la misma, basta cambiar «1» por «e».

Se puede razonar de la misma manera en Coq utilizando *clases de tipos*.

Una clase en Coq es un mecanismo que permite definir estructuras matemáticas. Las clases reciben argumentos que serán usados dentro de su cuerpo, el cual, contiene proposiciones que definen a la estructura.

El siguiente código define un monoide en Coq:

```
Class Monoide (M:Type) (dot:M->M->M) (e:M) := {

dot_assoc : forall x y z:M, dot x (dot y z) = dot (dot x y) z;
unit_left : forall x, dot e x = x;
unit_right : forall x, dot x e = x

}.
```

En este ejemplo, la clase *Monoide* recibe tres argumentos: el primero, *M*, es un tipo que simula un conjunto no vacío; el segundo, *dot* es una función binaria definida en *M*; el último, *e*, será el neutro de nuestro monoide. Dentro de la clase escribimos los axiomas que deben cumplirse, Coq maneja estos enunciados como proposiciones con el tipo *Prop*. A cada axioma se le debe dar un nombre para que el usuario pueda referirse a él.

Ahora, para definir f en Coq, escribimos:

```
Generalizable Variables M dot e.
Fixpoint f `{Mon:Monoide M dot e} (x:M) (n:nat) :=
            match n with 0=>e
                   | S m =>dot x (f x m)
end.
```

Vamos a detallar lo anterior:

- El mecanismo *Generalizable Variables* permite que el asistente infiera el tipo de las variables *M, dot, e* para que sean argumentos de la clase *Monoide*.

- La palabra reservada *Fixpoint* permite definir funciones recursivas. *f* será el nombre de nuestra función.

- Lo que sigue es un poco más complicado: estamos pasando argumentos implícitos a nuestra función. Intuitivamente, el asistente recibe la siguiente información: *sabiendo que (M,dot,e) es un monoide...*
  Los argumentos implícitos van entre llaves, en este caso, nuestro argumento implícito es una instancia de la clase *Monoide* llamada *Mon*. El símbolo (‘) se utiliza para indicar que dentro de nuestro argumento hay variables generalizadas.

- x y n son los argumentos que realmente recibe nuestra función, estos son argumentos explícitos.

- La palabra reservada *match* hace un análisis de casos del tipo *nat*, es el análogo al *case of* de Haskell.

Para verificar formalmente el resultado mostrado anteriormente, escribimos:

```
Lemma e_a_la_n `{M :Monoide A dot e} : forall n:nat, f e n = e.
Proof.
induction n.
(*Caso base*)
trivial.
(*Paso inductivo*)
simpl.
(*Usamos la hip. de inducción*)
rewrite IHn.
rewrite unit_left.
trivial.
Qed.
```

Los comentarios en Coq se encierran entre (* *).

El mecanismo de clases será fuertemente utilizado en nuestro proceso de formalización, para profundizar más en clases en Coq recomendamos consultar [SO08, SVdW11].

FORMALIZANDO TEORÍA DE CATEGORÍAS EN COQ

The type class system in Coq is useful both for developing elegant programs and concise mathematical formalizations on abstract structures.
Matthieu Sozeau & Nicolas Oury

El sistema de tipos que utiliza Coq es un cálculo-λ equipado con *tipos dependientes*, es decir, los tipos pueden depender de valores del lenguaje. Esto permite expresar propiedades matemáticas de manera concisa y sencilla dentro de Coq.

La definición de categoría mostrada en la parte I de este trabajo puede ser implementada directamente en el sistema de tipos de Coq utilizando clases.

La siguiente definición de categoría en Coq se basa en [SO08]:

```
Generalizable All Variables.
Class Categoria (obj:Type) (hom:obj->obj->Type) := {

comp: forall {a b c:obj}, hom b c->hom a b->hom a c;
id: forall a:obj, hom a a;
id_izq: forall {a b:obj} (f:hom a b), comp (id b) f = f;
id_der: forall {a b:obj} (f:hom a b), comp f (id a) = f;
asoc: forall {a b c d:obj} (f:hom a b) (g:hom b c) (h:hom c d),
                comp h (comp g f) = comp (comp h g) f

}.
```

Nuestra clase *Categoria* se compone de lo siguiente:

- El argumento *obj* es un tipo que representará a los objetos de nuestra categoría. El argumento *hom* es una función que recibe dos objetos de nuestra categoría y nos devuelve un tipo que representará los morfismos entre esos objetos.

  Ahora, dentro de nuestra clase:

- La función *comp* realizará la composición de morfismos; esta función depende de tres objetos y dos morfismos, por cómo se definió la clase, los primeros tres argumentos son implícitos, es decir, Coq se encargará de inferirlos.

- La función *id* dará un morfismo de tipo *hom a a* por cada objeto *a* en la categoría. Este morfismo es especial pues será el morfismo identidad de cada objeto.
- Por último, especificamos los axiomas de categoría que deben cumplirse. Es importante notar que los axiomas pueden tener argumentos implícitos.

Utilizando esta definición es muy sencillo probar que los tipos de Coq junto con las funciones entre tipos forman una categoría.

```
Program Instance Cat_tipos : Categoria Type (fun (a b:Type) =>a->b) := {
id a := fun x:a =>x;
comp a b c := fun (g:b->c) (f:a->b) =>fun x:a =>g (f x)
}.
```

El mecanismo *Program Instance* verifica de manera automática que, con los argumentos dados, hay una instancia de la clase *Categoria*. En este caso hay dos argumentos:

- El primer argumento le indica al asistente que nuestros objetos serán todos los términos x tales que $x : Type$.
- El segundo argumento debe ser una función que reciba dos objetos y nos devuelva los morfismos entre ellos. La palabra reservada *fun* sirve para definir funciones anónimas, en este caso estamos diciendo que dados los objetos *a* y *b*, todo morfismo de *a* en *b* tendrá tipo `a->b`.
- Por último definimos los componentes de nuestra clase *id* y *comp* utilizando funciones anónimas.

Una vez hecho esto, el mecanismo *Program Instance* unifica la información brindada con la clase *Categoria*, hecho esto, nos indica que tenemos *obligaciones* por completar de manera interactiva, estas obligaciones son demostrar que, con la información dada, se cumplen los axiomas de la clase *Categoria*:

```
Cat_tipos has type-checked, generating 3 obligation(s)
Solving obligations automatically...
3 obligations remaining
Obligation 1 of Cat_tipos:
forall (a b : Type) (f : a -> b), (fun x : a => f x) = f.

Obligation 2 of Cat_tipos:
forall (a b : Type) (f : a -> b), (fun x : a => f x) = f.

Obligation 3 of Cat_tipos:
forall (a b c d : Type) (f : a -> b) (g : b -> c) (h : c -> d),
(fun x : a => h (g (f x))) = (fun x : a => h (g (f x))).
```

Las tres obligaciones son obvias a la vista pues simplemente son igualdades de términos, pero son igualdades obvias.

Para que el asistente pruebe los tres axiomas en un solo golpe utilizando la táctica *trivial* escribimos:

```
Solve Obligations using trivial.
```

Lo que sigue es menos intuitivo: utilizando el ejemplo visto en la parte I, vamos a verificar que un monoide puede ser visto como una categoría.

Para esto, definimos el tipo:

```
Inductive P := Punto.
```

el cual, es un tipo trivial que contiene un único objeto: *un punto*. La palabra reservada *Inductive* sirve para definir nuestros propios tipos. En este caso, *P* es un tipo cuyo único habitante es *Punto*.

Entonces, para ver que una instancia de la clase *Monoide* es una instancia de la clase *Categoria*, escribimos:

```
Program Instance Monoide_Cat `{Mon:Monoide M dot e} :
          Categoria P (fun P P =>M) := {
comp := fun P P P m1 m2 =>dot m1 m2;
id := fun P =>e
}.
```

Sabiendo que *(M,dot,e)* forman un monoide, creamos la categoría cuyo único objeto es de tipo *P*. Al haber sólo un objeto, *Punto:P*, basta definir los morfismos de *P* en *P*, nuestra elección son los elementos de *M*. Además, especificamos que la composición de morfismos y el morfismo identidad son, respectivamente, la multiplicación y el neutro del monoide.

Al declarar un monoide instancia de la clase *Categoria* se generan tres obligaciones, para indicar a Coq que vamos a comenzar a resolverlas utilizamos el comando *Solve Obligations*, para resolver cada obligación usamos el comando *Next Obligation*. A continuación presentamos las obligaciones generadas y sus respectivas pruebas.

- `forall (M : Type) (dot : M ->M ->M) (e : M),`

  `Monoide M dot e ->P ->P ->forall f : M, dot e f = f`

  ```
  Proof.
  intros.
  apply neutro_izq.
  Qed.
  ```

  La táctica *intros* inicializa todas las hipótesis, es decir, los antecedentes de la implición y las variables cuantificadas de manera universal. Una vez aplicada la táctica *intros* la meta que queda es la siguiente:

  `dot e f = f`

  la cual, es idéntica al axioma *neutro_izq* de la clase *Monoide*. La táctica *apply* aplica un enunciado en nuestra meta; en este caso, para terminar basta aplicar la táctica *apply neutro_izq*.

- `forall (M : Type) (dot : M ->M ->M) (e : M),`

  `Monoide M dot e ->P ->P ->forall f : M, dot f e = f`

  ```
  Proof.
  intros.
  apply neutro_der.
  Qed.
  ```

- ```
  forall (M : Type) (dot : M ->M ->M) (e : M),
  Monoide M dot e ->
  P ->P ->P ->P ->forall f g h : M,
  dot h (dot g f) = dot (dot h g) f
  Proof.
  intros.
  apply asocc.
  Qed.
  ```

Formalizar puede ser tedioso debido a que todos los enunciados están escritos en un lenguaje de programación sofisticado, podemos aligerar esto utilizando notación especial para nuestro código, por ejemplo, al hacer:

Infix "°":= comp (at level 40).

le indicamos a Coq que reemplace la notación «comp g f» por la notación «g°f»; el comando *(at level 40)* indica el nivel de precedencia del operador.

Utilizando nuestra nueva notación, definimos la clase para funtores como sigue:

```
Class Funtor `{C:Categoria objC homC, D:Categoria objD homD}
    (Fobj: objC->objD)
    (F: forall {a b:objC}, homC a b->homD (Fobj a) (Fobj b)) := {
preser_id: forall a:objC, F (id a) = id (Fobj a);
preser_comp: forall {a b c:objC} (f:homC a b) (g:homC b c),
                        F(g°f) = (F g)°(F f)
}.
```

De manera implícita, recibimos dos instancias de categorías. El primer argumento, llamado *Fobj*, es el funcional entre los objetos de ambas categorías; el segundo argumento, llamado *F*, es el funcional entre la clase de morfismos, este argumento depende de otros dos.

Dentro de la definición de la clase *Funtor* podemos inferir los primeros dos argumentos de *F*, sin embargo, al no ser un componente dentro de la clase, en futuras instancias tendremos que especificarlos de manera explícita. Esto es, al aplicar el funtor al morfismo `f:a->b`, se tiene que `F a b f:Fobj a->Fobj b`.

Ahora damos dos definiciones que nos serán de mucha utilidad a la hora de definir mónadas dentro de Coq: la de funtor identidad y la de composición de funtores:

- El funtor identidad deja fijos a los objetos y a los morfismos.

  ```
  Definition funtId `(C:Categoria obj hom) :
      Funtor (fun x:obj =>x) (fun (a b:obj) (f:hom a b) =>f).
  ```

- La composición se define de manera natural: componiendo las componentes de los funtores.

  ```
  Definition compFunt `{C:Categoria objC homC,
                        D:Categoria objD homD,
                        E:Categoria objE homE}
                     `(Gfunt: ! Funtor (Gobj:objD->objE) G,
                       Ffunt: ! Funtor (Fobj:objC->objD) F):
  Funtor (fun x:objC =>Gobj (Fobj x))
  (fun (a b:objC) (f:homC a b) =>G (Fobj a) (Fobj b) (F a b f)).
  ```
  ```

El operador ! cambia la forma en la que el asistente interpreta el uso de estructuras compartidas. En el caso de *Gfunt*, se tiene una instancia de funtor, que a su vez, recibe componentes de las categorías D y E. Este operador hace que Coq lea de manera normal lo que está viendo, sin generalizarlo.

Definimos una notación especial para la composición de funtores:

```
Infix "•":= compFunt (at level 40).
```

De manera análoga traducimos directamente la definición de transformación natural:

```
Class TransfNat `{C:Categoria objC homC, D:Categoria objD homD}
                `(F1: ! Funtor (Fobj:objC->objD) F,
                  F2: ! Funtor (Gobj:objC->objD) G)
                 (η : forall x : objC, homD (Fobj x) (Gobj x)) := {
ley_transfNat: forall {a b:objC} (f : homC a b),
                         (G a b f)°(η a)=(η b)°(F a b f)
}.
```

Nuestra clase recibe dos instancias de funtores y la componente que define la transformación natural. Dentro de la clase sólo se especifica el axioma que debe cumplir la componente η.

Ahora, la clase para mónadas tiene la siguiente definición:

```
Class Monada `{C:Categoria obj hom}
             `(TFunt: ! Funtor (Tobj:obj->obj) T,
               etaTN: ! TransfNat (funtId C) TFunt η,
               muTN: ! TransfNat (TFunt•TFunt) TFunt μ) := {
mu_asoc: forall x:obj,
(μ x)°(T (Tobj (Tobj x)) (Tobj x) (μ x)) = (μ x)°(μ (Tobj x));

eta_neutroI: forall x:obj, (μ x)°(η (Tobj x)) = id (Tobj x);

eta_neutroD: forall x:obj, (μ x)°(T x (Tobj x) (η x)) = id (Tobj x)
}.
```

La clase recibe una instancia de funtor y dos instancias de transformación natural, es importante notar la manera en la que estamos construyendo las transformaciones naturales: la instancia de categoría C es un argumento implícito pero se utiliza para generar el funtor identidad. Usualmente en matemáticas nos referimos a un funtor sólo por F, pero esto puede llegar a ser confuso pues la misma F se usa tanto para objetos como para morfismos; en nuestra formalización le estamos dando un nombre a la instancia completa de la clase *Funtor*, por ejemplo, para referirnos al funtor T hacemos referencia a la instancia *TFunt*. Dentro de la clase sólo se especifican los axiomas de mónada.

Finalmente, la clase para ternas de Kleisli:

```
Class TernaKleisli `{C:Categoria obj hom}
                   (Tobj:obj->obj)
                   (η:forall x:obj, hom x (Tobj x)) := {
ext: forall {a b:obj}, hom a (Tobj b) ->hom (Tobj a) (Tobj b);
ki: forall a:obj, ext (η a) = id (Tobj a);
kii: forall {a b} (f:hom a (Tobj b)), (ext f)°(η a) = f;
kiii: forall {a b c} (f:hom a (Tobj b)) (g:hom b (Tobj c)),
          (ext g)°(ext f) = ext ((ext g)°f)
}.
```

La clase recibe un funcional entre los objetos de la categoría C, la componente η y dentro de la clase especificamos que se debe de dar la función de extensión y los axiomas de terna de Kleisli.

Añadimos una notación especial para la extensión de morfismos de Kleisli:

```
Notation "f ⋆":= (ext f) (at level 40).
```

así, los axiomas de terna de Kleisli pueden reescribirse de la siguiente manera:

K1) `(η a)⋆ = id (Tobj a)`

K2) `f ⋆ ° (η a) = f`

K3) `g ⋆ ° f ⋆ = (g ⋆ ° f)⋆`

Obsérvese que fue necesario definir formalmente, es decir, dentro de Coq, todos los conceptos necesarios para enunciar el teorema deseado.

Con esto ya tenemos toda la herramienta necesaria para formalizar la equivalencia entre mónadas y ternas de Kleisli.

5

VERIFICANDO LA PRUEBA EN COQ

El desarrollo de la matemática hacia una mayor precisión ha llevado, como es bien sabido, a la formalización de diversas de sus áreas, de manera que las demostraciones pueden realizarse de acuerdo a unas pocas reglas mecánicas.
Kurt Gödel

Como vimos en el capítulo anterior, para formalizar una prueba en Coq simplemente hay que traducirla a tácticas que pueda verificar el asistente hasta completar todas las metas generadas.

Como vimos en la parte I, el enunciado original de la equivalencia es el siguiente:

Hay una correspondencia uno-a-uno entre mónadas y ternas de Kleisli.

Para formalizar este teorema en Coq vamos a separarlo en dos, el primero:

Teorema. *Si* $(T, \eta : Id_{\mathcal{C}} \Rightarrow T, \mu : T \circ T \Rightarrow T)$ *es una mónada, entonces* $(T, \eta, _^*)$ *es una terna de Kleisli; donde el operador de extensión se obtiene haciendo* $f^* = \mu_b \circ T(f)$.

De manera implícita estamos diciendo que $\mathcal{C}$ es una categoría, que T es un funtor, y que η y μ son transformaciones naturales.

La respectiva formalización de lo anterior dentro del asistente es lo siguiente:

```
Program Instance Monada_es_TernaKleisli `{C:Categoria obj hom}
                         `{TFunt: ! Funtor (Tobj:obj->obj) T}
                     `{etaTN: ! TransfNat (funtId C) TFunt η}
                   `{muTN: ! TransfNat (TFunt•TFunt) TFunt μ}
`(Mon: ! Monada TFunt etaTN muTN) : TernaKleisli Tobj η := {
ext a b := fun (f:hom a (Tobj b)) =>(μ b)°(T a (Tobj b) f)
}.
```

Al igual que en la prueba de la parte I, para construir una terna de Kleisli a partir de una mónada utilizamos la componente del funtor T definida en los objetos y la misma componente de la transformación natural η, por último, indicamos cómo extender los morfismos de la forma $f : a \to T(b)$ utilizando la componente μ.

El mecanismo *Program Instance* genera tres obligaciones que corresponden a probar los axiomas de terna de Kleisli, presentamos las metas a probar y sus respectivas pruebas:

K1) `μ a ° T a (Tobj a) (η a) = id (Tobj a)`

```
Proof.
rewrite eta_neutroD; trivial.
Qed.
```

Cuando usamos «;» entre dos tácticas, Coq se encarga de componerlas: al hacer $t_1; t2$ el asistente se encarga de aplicar la táctica t_2 a todas las metas generadas

por la táctica t_1. Esto es muy útil cuando queremos omitir detalles técnicos, como asociar muchas veces, realizar operaciones aritméticas, etc.

K2) `μ b ° T a (Tobj b) f) ° η a = f`

Proof.
```
rewrite <- asoc;destruct etaTN; rewrite ley_transfNat0.
rewrite asoc;destruct Mon; rewrite eta_neutroI0.
rewrite id_izq;trivial.
```
Qed.

La táctica *rewrite* <- *asoc* le indica al asistente que reescriba la meta usando el axioma *asoc* pero cambiando la parte derecha por la parte izquierda de la igualdad.

En la meta tanto η como μ están presentes, al mandar llamar al axioma *ley_transfNat* Coq podría entrar en conflicto al no saber a cuál transformación natural nos estamos refiriendo. Al usar la táctica *destruct etaTN* le indicamos al asistente que despliegue la definición de transformación natural con la instancia η.

De igual manera, *destruct Mon* despliega la definición de mónada para utilizar el axioma *eta_neutroI*.

K3) `(μ c ° T b (Tobj c) g) ° (μ b ° T a (Tobj b) f) =`
`μ c ° T a (Tobj c) ((μ c ° T b (Tobj c) g) ° f)`

Proof.
```
rewrite preser_comp; rewrite preser_comp.
rewrite (asoc (T a (Tobj b) f) (T (Tobj (Tobj c)) (Tobj c)
(μ c)° T (Tobj b) (Tobj (Tobj c)) (T b (Tobj c) g)) (μ c));
      rewrite (asoc (T (Tobj b) (Tobj (Tobj c)) (T b (Tobj c) g))
      (T (Tobj (Tobj c)) (Tobj c) (μ c)) (μ c)).
rewrite mu_asoc;
     rewrite <- (asoc (T a (Tobj b) f) (T (Tobj b) (Tobj (Tobj c))
     (T b (Tobj c) g)) (μ c ° μ (Tobj c)));
     rewrite (asoc (T a (Tobj b) f) (T (Tobj b) (Tobj (Tobj c))
     (T b (Tobj c) g)) (μ c ° μ (Tobj c)));
     rewrite <- (asoc (T (Tobj b) (Tobj (Tobj c)) (T b (Tobj c) g))
     (μ (Tobj c)) (μ c)). destruct muTN; rewrite <- ley_transfNat0.
rewrite <- asoc; rewrite asoc; rewrite asoc; rewrite asoc;trivial.
```
Qed.

La parte complicada en esta parte de la prueba es que la meta tiene muchas composiciones: si usamos la táctica *rewrite* con el axioma *asoc*, Coq podría reescribir una composición que no nos interesa cambiar. Recordemos que el axioma *asoc* está formalizado de la siguiente manera:

```
asoc: forall {a b c d:obj} (f:hom a b) (g:hom b c) (h:hom c d),
                comp h (comp g f) = comp (comp h g) f
```

Así, *asoc* es un axioma que recibe tres funciones como argumentos, por eso, para indicar a Coq sobre cuál composición debe enfocarse, basta pasar como argumento las funciones sobre las cuáles debe aplicar la reescritura.

Hemos completado la primera parte de la prueba.

Para la segunda parte de la prueba vamos a probar tres lemas que serán de mucha utilidad: el primero mostrará que una terna de Kleisli es un funtor; el segundo que la componente η de la terna de Kleisli se puede extender a una transformación natural; y el tercero mostrará que podemos construir la componente μ para nuestra mónada a partir de una terna de Kleisli.

- **Lema.** *Si* $(T, \eta, _^*)$ *es una terna de Kleisli sobre la categoría* $\mathcal{C}$*, entonces podemos crear el funtor* $T : \mathcal{C} \to \mathcal{C}$ *de la siguiente manera: si* $f : a \to b$*, entonces* $T(f) = (\eta_b \circ f)^*$.

  Su formalización en el asistente:

```
Lemma TernaKleisli_es_funtor `{C:Categoria obj hom}
                        `(TK: ! TernaKleisli Tobj η) :
     Funtor Tobj (fun a b (f:hom a b) =>((η b)°f)⋆).
Proof.
      (*Para probar que preserva identidades*)
split;intros.
rewrite id_der.
rewrite ki;trivial.
     (*Para probar que T preserva la composición*)
rewrite asoc;pattern ((η c)°g) at 1;
      rewrite <- (kii ((η c)°g)).
rewrite <- asoc;rewrite <- kiii; trivial.
Qed.
```

  Al comenzar la prueba, la meta original es:

```
Funtor Tobj (fun (a b : obj) (f : hom a b) =>(η b ° f) ⋆)
```

  La táctica *split;intros* se encarga de desglozar la meta utilizando la clase *Funtor* para crear dos metas:

```
--------------------------------------(1/2)
(η a ° id a) ⋆ = id (Tobj a)
--------------------------------------(2/2)
(η c ° (g ° f)) ⋆ = ((η c ° g) ⋆) ° ((η b ° f) ⋆)
```

  las cuales, podemos atacar fácilmente.

  Ahora, la táctica *pattern* se encarga de enfocar una expresión que aparece muchas veces dentro de la meta, el argumento que recibe es un número entero con el que indicamos cuál aparición debe enfocar. En este caso, la meta:

```
((η c ° g) ° f) ⋆ = ((η c ° g) ⋆) ° ((η b ° f) ⋆)
```

  contiene la expresión *(η c ° g)* en ambos lados lados de la igualdad. Al hacer *pattern ((η c)°g) at 1* estamos diciendo al asistente que enfoque la primer ocurrencia de la expresión *((η c)°g)* que encuentre de izquierda a derecha dentro de la meta, al componer lo anterior con la táctica *rewrite* <- *(kii ((η c)°g))* el asistente sólo reescribe la primer ocurrencia de la expresión *((η c)°g)* utilizando el axioma *kii*.

- **Lema.** *Si* $(T, \eta, _^*)$ *es una terna de Kleisli sobre la categoría* $\mathcal{C}$*, entonces se puede construir la transformación natural* $\eta : Id_{\mathcal{C}} \Rightarrow T$*, donde* T *es el funtor definido en el lema anterior.*

  Su formalización:

```
Lemma TernaKleisli_es_etaTN `{C:Categoria obj hom}
                            `(TK: ! TernaKleisli Tobj η) :
 TransfNat (funtId C) (TernaKleisli_es_funtor TK) η.
```

  Hay que notar que estamos indicando al asistente que la segunda instancia de *Funtor* que espera la clase *TransfNat* es el funtor que creamos con el lema anterior.

```
Proof.
split;intros.
rewrite kii;
trivial.
Qed.
```

- **Lema.** *Si* $(T, \eta, _^*)$ *es una terna de Kleisli sobre la categoría* $\mathcal{C}$*, podemos crear la transformación natural* $\mu : T \circ T \Rightarrow T$*; donde el funtor* T *se construye de la misma manera que en el lema anterior, y* $\mu_x = (id_{T(x)})^*$*.*

  Su formalización:

```
Lemma TernaKleisli_es_muTN `{C:Categoria obj hom}
                            `(TK: ! TernaKleisli Tobj η) :
TransfNat ((TernaKleisli_es_funtor TK)•(TernaKleisli_es_funtor TK))
        (TernaKleisli_es_funtor TK) (fun x:obj =>(id (Tobj x))⋆).
Proof.
split;intros.
rewrite kiii.
rewrite id_der.
pattern (((η b)°f)⋆) at 1;rewrite <- id_izq.
pattern (id (Tobj b)) at 1; rewrite <- kii.
rewrite <- asoc;rewrite <- kiii; trivial.
Qed.
```

Usando estos tres lemas concluiremos la segunda parte de la prueba.

> **Teorema.** *Si* $(T, \eta, _^*)$ *es una terna de Kleisli sobre la categoría* $\mathcal{C}$*, entonces podemos crear la mónada* (T, η, μ)*, donde el funtor* T*, y las transformaciones naturales* η *y* μ *se obtienen como en los lemas anteriores.*

La respectiva formalización en Coq es la siguiente:

```
Theorem TernaKleisli_es_Monada `{C:Categoria obj hom}
                          `(TK: ! TernaKleisli Tobj η) :
                       Monada (TernaKleisli_es_funtor TK)
                              (TernaKleisli_es_etaTN TK)
                              (TernaKleisli_es_muTN TK).

Proof.
      (*Para probar la asociatividad de μ*)
split;intros.
rewrite kiii.
rewrite asoc;rewrite (kii (id (Tobj x))).
rewrite id_izq;pattern ((id (Tobj x))⋆) at 1;
      rewrite <- (id_der ((id (Tobj x))⋆)).
rewrite <- kiii;trivial.
      (*Para probar que η es neutro por la izquierda*)
rewrite kii;trivial.
      (*Para probar que η es neutro por la derecha*)
rewrite kiii.
rewrite asoc; rewrite kii.
rewrite id_izq; rewrite <- ki; trivial.
Qed.
```

Con esto, hemos verificado formalmente la equivalencia entre mónadas y ternas de Kleisli.

CONCLUSIONES Y TRABAJO FUTURO

Las ciencias de la computación se distinguen de otras disciplinas relacionadas a la computación por su fuerte e íntima conexión con las matemáticas, específicamente el vínculo con la teoría de categorías ha revolucionado la manera en la que programamos en los lenguajes funcionales. En Haskell, las mónadas han aportado muchísimas herramientas que facilitan el desarrollo de software robusto y legible. En la teoría de lenguajes de programación, las categorías han permitido profundizar la semántica de éstos, incluso aportar herramientas poderosas, por ejemplo, un algoritmo categórico de unificación de términos [RB86].

Los lenguajes de programación funcional son cada vez más populares en la industria por ser rápidos de manejar, por su robustez y por lo fácil que es, en comparación con los lenguajes orientados a objetos, dar mantenimiento a un código escrito de manera funcional. Ejemplo de esto son el lenguaje Swift, que fue creado por Apple para el desarrollo de aplicaciones para iOS y Mac OS X; y Java, que es utilizado para crear aplicaciones para el sistema Android y que, desde la versión 8, incluye programación funcional. Por supuesto, tanto Swift como Java pueden implementar mónadas. Es nuestro deber seguir investigando para continuar haciendo crecer el poder y la popularidad de este paradigma de programación.

Por otra parte, los asistentes de prueba son evidencia del vínculo entre las matemáticas y las ciencias de la computación y, posiblemente, serán una pieza fundamental en la investigación, la formalización y la enseñanza de éstas. El reto es acercar a más computólogos a formar parte de este cambio, ya sea formalizando y verificando matemáticas o haciendo verificación formal de software. Aunque no sólo debemos lograr que otros computólogos conozcan esto, hay matemáticos que no están enterados de este tipo de software, o que simplemente no les interesa formalizar sus resultados porque les puede llegar a parecer aburrido y muy tedioso. Un posible reto es acercanos a nuestros colegas matemáticos y mostrarles lo apasionante y emocionante que puede ser formalizar resultados en la computadora.

La inmersión en las mónadas para este trabajo fue emocionante, en el futuro se espera conocer más de los *transformadores monádicos*. En el ámbito de la verificación formal, si bien se descubrió mucho del poder de Coq, no se usó ni la mitad de su poder de abstracción. Un reto a futuro es profundizar más en todo lo relacionado a este asistente, el cual, es una herramienta verdaderamente poderosa.

REFERENCIAS

[all15] Wiki haskellAll About Monads. https://wiki.haskell.org/All_About_Monads/, 2015.

[apr15] ¡Aprende Haskell por el bien de todos! http://aprendehaskell.es/, 2015.

[Avi94] Jeremy Avigad. In *Formal verification, interactive theorem proving, and automated reasoning*. Department of Philosophy and Department of Mathematical Sciences Carnegie Mellon University, 1994.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BG01] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. *Handbook of automated reasoning*, 2:1149–1238, 2001.

[bug15] Bugs en el software i: Accidentes y pérdidas graves por fallos en el software. http://www.genbetadev.com/seguridad-informatica/bugs-en-el-software-i-accidentes-epicos-por-fallos-en-el-software, 2015.

[BW90] Michael Barr and Charles Wells. Category theory for computer scientists, 1990.

[BW13] Michael Barr and Charles Wells. *Toposes, triples and theories*, volume 278. Springer Science & Business Media, 2013.

[con15] Haskell/continuation passing style. https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style, 2015.

[coq15] Frequently asked questions. https://coq.inria.fr/faq, 2015.

[DHM91] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ml. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM, 1991.

[EM45] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, pages 231–294, 1945.

[EM+65] Samuel Eilenberg, John C Moore, et al. Adjoint functors and triples. *Illinois Journal of Mathematics*, 9(3):381–398, 1965.

[FT63] Walter Feit and John Thompson. Chapter i, from solvability of groups of odd order, pacific j. math, vol. 13, no. 3 (1963. *Pacific journal of mathematics*, 13(3):775–787, 1963.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[Gam05] Shauna C. A . Gammon. Notions of category theory in functional programming. Master's thesis, B.Sc, Memorial University of Newfoundland, 2005.

[Gib13] Jeremy Gibbons. Unifying theories of programming with monads. In *Unifying Theories of Programming*, pages 23–67. Springer, 2013.

[God58] Roger Godement. Théorie des faisceaux, hermann, paris. 1958.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics*, pages 113–130. Springer, 1999.

[has15] do notation Haskell. https://en.wikibooks.org/wiki/Haskell/do_notation, 2015.

[HKPM14] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant a tutorial. *Rapport Technique*, 2014.

[How80] William A Howard. The formulae-as-types notion of construction. 1980.

[Hub61] Peter J Huber. Homotopy theory in general categories. *Mathematische Annalen*, 144(5):361–385, 1961.

[Hug00] John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.

[Hur41] Witold Hurewicz. On duality theorems. *Bull. Am. Math. Soc*, 47:562–563, 1941.

[Kle65] Heinrich Kleisli. Every standard construction is induced by a pair of adjoint functors. *Proceedings of the American Mathematical Society*, pages 544–546, 1965.

[Lau93] John Launchbury. Lazy imperative programming. In *Workshop on State in Programming Languages, Copenhagen, Denmark, ACM*, 1993.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Lis15] List comprehension. https://en.wikipedia.org/wiki/List_comprehension, 2015.

[LS88] Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

[Man76] Ernest G Manes. Algebraic theories. 1976.

[ML71] Saunders Mac Lane. Category theory for the working mathematician, 1971.

[Mog89] E Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS\'89, Proceedings., Fourth Annual Symposium on*, pages 14–23, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[mon15] Monad class. https://wiki.haskell.org/Monad/, 2015.

[nar15] Making 'Super Nario Bros.' in Haskell . `https://youtu.be/gVLFGQGRsDw`, 2015.

[PCG+15] Benjamin C Pierce, Chris Casinghino, Michael Greenberg, Catalin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. Software foundations. electronic textbook, 2015.

[Pie88] Benjamin C Pierce. A taste of category theory for computer scientists. 1988.

[Pie91] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

[PJW93] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[Que03] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *ACM SIGPLAN Notices*, 38(2):57–64, 2003.

[RB86] David E Rydeheard and Rod M Burstall. A categorical unification algorithm. In *Category Theory and Computer Programming*, pages 493–505. Springer, 1986.

[RB88] David E Rydeheard and Rod M Burstall. *Computational category theory*, volume 152. Prentice Hall Englewood Cliffs, 1988.

[RC86] Jonathan Rees and William Clinger. Revised report on the algorithmic language scheme. *ACM Sigplan Notices*, 21(12):37–79, 1986.

[rea15] Real world haskell Classic I/O in Haskell. `http://book.realworldhaskell.org/read/io.html`, 2015.

[Rey72] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.

[Rey93] John C Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

[SFSÁ07] Andrea Solotar, Marco Farinati, and Mariano Suárez-Álvarez. Anillos y sus categorías de representaciones. *Cuadernos de Matemática y Mecánica, IMAL (CONICET-UNL)-CIMEC (INTEC, CONICET-UNL)*, 2007.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008.

[Spi90] Mike Spivey. A functional theory of exceptions. *Science of computer programming*, 14(1):25–42, 1990.

[SSJ15] Gerald Jay Sussman and Guy Lewis Steele Jr. Lambda: The ultimate imperative. 2015.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

[SVdW11] Bas Spitters and Eelis Van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(04):795–825, 2011.

[Tea15] The Coq Development Team. Reference manual. `https://coq.inria.fr/distrib/current/refman/`, 2015.

[und15] Understanding monads Haskell. `https://en.wikibooks.org/wiki/Haskell/Understanding_monads`, 2015.

[Voe11] VA Voevodsky. Univalent foundations. In *Mathematisches Forschungsinstitut Oberwolfach, mini-workshop: the homotopy interpretation of constructive type theory, Report*, number 11, page 2011, 2011.

[Voe14a] V Voevodsky. The origins and motivations of univalent foundations, ias-the institute letter. summer 2014. *Institute for Advanced Study, Princeton, NJ, USA*, pages 8–9, 2014.

[Voe14b] Vladimir Voevodsky. Computer proof assistants - the future of mathematics. Lecture at the NUS, Singapore., 2014.

[Wad92a] Philip Wadler. Comprehending monads. *Mathematical structures in computer science*, 2(04):461–493, 1992.

[Wad92b] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[Wad98] Philip Wadler. The marriage of effects and monads. In *ACM SIGPLAN Notices*, volume 34, pages 63–74. ACM, 1998.

[web15] Reading list on xml and web programming. `http://readscheme.org/xml-web/`, 2015.

[Wie15] Freek Wiedijk. Formalizing 100 theorems. `http://www.cs.ru.nl/~freek/100/`, 2015.

[wir15] Will computers redefine the roots of math? *http://www.wired.com/2015/05/will-computers-redefine-roots-math/*, 2015.