Universidad Nacional Autónoma
De México

Facultad de Ciencias

Coinducción: de la Teoría de Categorías a la Programación Funcional.

T E S I S

Que para obtener el título de:
Licenciada en Ciencias de la Computación

P R E S E N T A :

Nombre de la Alumna:
Lourdes Del Carmen González Huesca

Tutor
Dr. Favio Ezequiel Miranda Perea

2007

Universidad Nacional Autónoma de México
UNAM
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
González
Huesca
Lourdes Del Carmen
53 73 50 43
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
402056830

2. Datos del tutor
Dr.
Favio Ezequiel
Miranda
Perea

3. Datos del sinodal 1
Dr.
Carlos
Torres
Alcaraz

4. Datos del sinodal 2
Dr.
Francisco
Hernández
Quiroz

5. Datos del sinodal 3
M. en C.
Araceli Liliana
Reyes
Cabello

6. Datos del sinodal 4
M. en C.
Miguel
Carrillo
Barajas

7. Datos del trabajo escrito
Coinducción: de la Teoría de Categorías a la Programación Funcional
72 p
2007

Índice general

Introducción

Las matemáticas y las ciencias de la computación siempre han estado relacionadas de una manera muy cercana, distintas ramas de las ciencias computacionales son derivadas y fundamentadas de las matemáticas. La inducción y la recursión forman parte de la vida diaria de un científico de la computación, para aprovecharlas mejor y comprender su funcionamiento es necesario identificar en ellas la parte matemática o formal que las sustentan. Diseñar programas es otra parte importante en las ciencias de la computación, para llevar a cabo el diseño de un programa existen diferentes paradigmas de programación, uno de ellos es el paradigma de programación funcional cuya idea principal es la evaluación de funciones. El cálculo lambda es una base matemática importante para la programación funcional ya que es un formalismo para expresar funciones y la evaluación de las mismas.
Al trabajar con una computadora siempre se tiene la limitante de sus recursos físicos, además, a cualquier científico de la computación, enfocado en la programación y alejado del formalismo matemático, le es difícil hablar de conjuntos infinitos; es más fácil y factible representar en una computadora objetos finitos de conjuntos infinitos como son: un número natural, una lista finita de objetos, un árbol con profundidad finita, etc. Cuando se diseñan programas se ven involucrados los tipos de datos, que son parte fundamental de un programa. Es común que los objetos como los antes mencionados, son los más utilizados en el diseño de un programa, formalmente se les denomina tipos de datos inductivos. Pero ¿qué sucede si los objetos a manejar en un programa son flujos constantes de datos, por ejemplo una lista infinita de información que será procesada?, en este momento el diseñador del programa tendrá que pensar en una forma de representación y manejo de un objeto "infinito", a estos objetos se les denomina tipos de datos coinductivos, objetos que pueden llegar a ser infinitos. El cálculo lambda, permite la descripción y uso de objetos tanto finitos como infinitos, además de trabajar con ellos de manera abstracta también pueden ser llevarlos a la práctica ya que puede verse al cálculo lambda como el lenguaje básico para el paradigma funcional.

En este momento el lector se preguntará el significado de algunas palabras o frases nombradas arriba: ¿porqué se denominan tipos de datos inductivos, si la inducción es una herramienta matemática usada para demostrar?, ¿qué es o cómo es un tipo de dato coinductivo?, ¿objetos infinitos dentro de una computadora?, etc. Todas estas preguntas pretenden ser respondidas en este trabajo; a continuación se presentan algunos términos en forma general. Un tipo de dato inductivo se refiere a objetos definidos mediante funciones constructoras, es decir, se expone una forma de construir nuevos objetos a partir de ciertos objetos básicos dados o de objetos construidos anteriormente que tienen una estructura más simple. Un tipo de dato coinductivo contiene objetos potencialmente infi-

nitos los cuales, por su naturaleza infinita, solamente pueden ser observados en partes, es decir, es necesario dividir o destruir el objeto coinductivo para poder observarlo; en pocas palabras la inducción construye un nuevo objeto a partir de uno existente y la coinducción destruye un objeto para poder observarlo, la coinducción es el dual de la inducción y viceversa.

En este trabajo se estudiará la base matemática de la inducción y la coinducción, como parte de la teoría de tipos de datos para lenguajes de programación. La idea principal es integrar un mecanismo para definir tipos de datos (co)inductivos mediante una extensión del sistema F del cálculo lambda, finalizando con un prototipo de lenguaje de programación funcional; para llevar este prototipo a una implementación en una computadora, se requiere de un lenguaje funcional que permita una evaluación no estricta o perezosa, es decir, soló evalúa un objeto hasta que su valor es necesario. Al trabajar en una computadora con objetos infinitos como son los coinductivos, no es posible realizar una evaluación completa del objeto, Haskell es un lenguaje de programación que, debido a sus características y a la aproximación sintáctica con la extensión propuesta en este trabajo permite manejar estructuras tanto finitas como infinitas y en el cual se puede llevar a la práctica el prototipo propuesto.

A lo largo de este trabajo se encuentran diversos ejemplos, basados en la experiencia que he adquirido como alumna de ciencias de la computación; buscando sentar las bases teóricas en un ambiente estrictamente matemático de los términos utilizados a lo largo de la formación de la carrera de ciencias de la computación.

Coinducción: De la teoría de Categorías a la Programación Funcional, cómo pasar de las bases teóricas, es decir, la teoría de puntos fijos y la de categorías, hasta un programa en un prototipo de sistema de tipos. Para conjuntar las bases teóricas y llevarlo a la práctica, es necesario asentar todo en el cálculo lambda: la inducción ya está incorporada de una manera "natural" en él, siendo necesario aumentar el poder que brinda para manejar estructuras de datos coinductivas mediante la incorporación de un mecanismo para obtener y trabajar con objetos inductivos y coinductivos de una manera fácil y legible.

En el primer capítulo se da una reseña de la teoría necesaria para comprender este trabajo, se recomienda tener conocimientos sobre conceptos de lógica de primer y segundo orden, así como nociones del cálculo lambda y teoría de categorías, aunque a lo largo de este trabajo se introducen los conceptos básicos necesarios para la comprensión del mismo. En el segundo capítulo se muestra el primer intento por programar con tipos de datos inductivos y coinductivos utilizando la teoría de categorías. En el tercer capítulo se presenta la primer propuesta para trabajar con estos tipos de datos, mostrando las desventajas que tiene y proponiendo una solución más clara e intuitiva para su manejo. Se introduce una extensión del sistema F que utiliza los conceptos presentados en el primer capítulo, y que es dirigido a un prototipo de lenguaje de programación funcional. Por último se presentan las conclusiones y las posibles direcciones que se pueden tomar en base a este trabajo.

Capítulo 1

Preliminares

En este capítulo se dan las bases teóricas para el desarrollo del trabajo. Se explicarán y darán ejemplos tanto de inducción como de coinducción, la primera es ampliamente conocida y será explicada nuevamente para dar paso a la segunda, la coinducción, que es el tema principal de este trabajo.

También se explican y se dan definiciones de los formalismos que serán la base para la formalización de la inducción y coinducción: la teoría de puntos fijos y la teoría de categorías. Finalmente se da una introducción al cálculo lambda que será de gran ayuda para poder entender el sistema de tipos de datos presentado en el capítulo 3.

1.1. El Reino Inductivo

Cuando se trata de programar u obtener un algoritmo, primero se debe tener un diseño en general que después se empieza a detallar para definir la estructura que tendrá; dentro de esta estructura es común usar la palabra inducción, recursión o iteración para referirnos a la forma de algún componente del programa. Generalmente usamos la palabra recursión cuando expresamos cierta manera de definir objetos, así también se usa la palabra iteración para los comportamientos de funciones.

La recursión permite construir objetos y las propiedades definidas para esos objetos, que por lo general son funciones, se demuestran con inducción. Se puede decir que siempre que se habla de inducción va implícita la palabra recursión ya que el procedimiento de demostración contiene inherentemente a los objetos recursivos. La inducción en matemáticas es un proceso que permite demostrar propiedades sobre objetos definidos recursivamente, la inducción en números naturales es la más usada cuando queremos demostrar propiedades de ellos, comúnmente llamada inducción matemática, aunque también se utiliza la inducción estructural para demostrar propiedades de estructuras bien fundadas.

1.1.1. Recursión

Se usa la palabra *recursión* para denotar al mecanismo de definición de funciones cuyo valor en cierto punto se define mediante otros valores de la misma función. Los valores usados para calcular un nuevo valor de la función deben ser los valores obtenidos anteriormente y estructuralmente más simples al que se desea calcular.

Como ejemplo de definiciones recursivas tenemos a los números naturales, sabemos que los números naturales pueden ser formados a partir del número cero y dado cualquier número n se puede obtener el número que le sigue si al número se le suma uno $n + 1$:

Definición 1.1 (Números Naturales)**.** *El conjunto de los números naturales* $\mathbb{N}$*, se define recursivamente mediante las siguientes cláusulas:*

i) $0 \in \mathbb{N}$

ii) Si $n \in \mathbb{N}$ *entonces* $succ(n) \in \mathbb{N}$

iii) Son todos

La tercera cláusula es una manera de aclarar que si un elemento está en el conjunto $\mathbb{N}$ solamente es porque se genera mediante alguna de las dos cláusulas anteriores. En otras palabras, la cláusula *iii*) nos dice que el conjunto $\mathbb{N}$ es el más pequeño que cumple tanto con *i*) como con *ii*).

Otro ejemplo de definición recursiva es el conjunto de listas con objetos sobre un conjunto dado: la lista vacía nil, es la lista que no tiene elementos, podemos afirmar que esta lista es la más pequeña y para construir cualquier lista a partir de una lista ya creada l solo es necesario agregar un elemento del conjunto al inicio de la lista existente (a, l).

Definición 1.2 (Listas sobre un conjunto A)**.** *El conjunto de listas de elementos sobre un conjunto* A*,* $List(A)$*, está definido recursivamente mediante las siguientes cláusulas:*

i) $nil \in List(A)$

ii) Si $a \in A$ *y* $l \in List(A)$ *entonces* $cons(a, l) \in List(A)$

iii) Son todas

La recursión define objetos finitos que pueden conformar un conjunto infinito, éstos objetos están definidos mediante constructores y objetos básicos, en los ejemplos anteriores los *objetos básicos* son el número cero y nil respectivamente mientras que los *constructores* son: en los naturales el sucesor de un número n, $succ(n)$ y en las listas dados un elemento a y una lista existente l la función que crea una nueva lista es $cons(a, l)$.
Estos conjuntos son llamados conjuntos inductivos y los abreviaremos como $\mathcal{I}$, están definidos con objetos básicos c_i, que siempre pertenecen al conjunto $c_i \in \mathcal{I}$, y constructores que son funciones cuyo codominio siempre es el conjunto inductivo y el dominio está en función del conjunto inductivo $c_j : F(\mathcal{I}) \to \mathcal{I}$, el dominio es el resultado de una función que actúa sobre el conjunto inductivo, se aclarará cuál es y cómo se define en la sección dedicada a teoría de categorías. Los elementos de estos conjuntos inductivos son definidos recursivamente.

Las definiciones recursivas de conjuntos, tales como las definiciones 1.1 y 1.2, se conocen en ciencias de la computación como definiciones inductivas, aunque estrictamente la inducción es un principio de demostración mientras que la recursión es un principio de definición, seguiremos los usos y costumbres y hablaremos de definiciones inductivas,

reservando el uso de la palabra *recursión* exclusivamente para hacer referencia a los principios para definir funciones.

En matemáticas hay muchos principios de recursión, en realidad todos éstos son teoremas que son demostrados en teoría de recursión; para este trabajo se usará el principio de iteración para definiciones de funciones que involucren conjuntos inductivos al igual que el principio de recursión primitiva.
La iteración es la forma más simple de recursión, consiste en aplicar repetidamente una función, es decir, dada una función f iterarla $n+1$ veces es obtener $f^{n+1}(x) = f(f^n(x))$. Podemos ejemplificar la iteración con funciones en los números naturales, si queremos definir una función $f : \mathbb{N} \to A$ entonces es necesario definir el valor de la función para el cero y también para el caso general, es decir cualquier número que haya sido obtenido usando la función sucesor:

Definición 1.3 (Principio de Iteración para Naturales)**.** *Dados $a \in A$ y $s : A \to A$ existe una única función $f : \mathbb{N} \to A$ tal que:*

$$f(0) = a$$
$$f(succ(n)) = s(f(n))$$

Por ejemplo si queremos definir la función que determine si un número es par o no, $odd(x)$, tenemos a:

$$odd : \mathbb{N} \to Bool$$
$$odd(0) = True \quad odd(succ(n)) = not(odd(n))$$

Donde $Bool = \{True, False\}$ y la función not está determinada por $not(True) = False$ y $not(False) = True$.

Análogamente definimos el principio de iteración de una función para listas, definiendo el valor de la función aplicada a la lista vacía y a cualquier otra lista construida mediante el operador *cons*:

Definición 1.4 (Principio de Iteración para Listas)**.** *Dados $a \in A$ y $s : B \times A \to A$ existe una única función $f : List(B) \to A$ tal que:*

$$f(nil) = a$$
$$f(cons(b, l)) = s(b, f(l))$$

Por ejemplo si se quiere definir una función sobre las listas como la longitud de una lista dada sólo es necesario especificar lo siguiente:

$$long : List(A) \to \mathbb{N}$$
$$long(nil) = 0 \quad long(cons(a, l)) = succ(long(l))$$

Otra función sobre las listas es la que aplica una función a cada elemento de la lista, si los elementos de ésta son objetos del conjunto A y la función a aplicar es $g : A \to B$ tenemos la función:

$$map\ g : List(A) \to List(B)$$
$$map\ g\ nil = nil \quad map\ g\ (cons(a, l)) = cons(g(a), (map\ g\ (l)))$$

La definición inductiva de una función f está determinada cuando se da el valor de la función para cada uno de los objetos básicos y los constructores del conjunto de objetos recursivos.

En general para cualquier función aplicable a un conjunto definido recursivamente se tiene el siguiente principio de iteración:

Definición 1.5 (Principio de Iteración)**.** *Dados un conjunto de objetos definidos recursivamente $\mathcal{I}$ (con objetos básicos $c_i \in \mathcal{I}$ y constructores $c_j : F_j(\mathcal{I}) \to \mathcal{I}$), $b_i \in B$ y $s_j : F_j(B) \to B$ existe una única función $f : \mathcal{I} \to B$ tal que:*

$$f(c_i) = b_i$$
$$f(c_j\, x) = s_j(\dots f(x) \dots),\ x \in \mathcal{I}$$

donde la expresión $''\dots f(x)\dots''$ denota a un elemento del dominio de s_j que involucra a $f(x)$.
Basta definir el valor de la función para cada objeto básico y para cada constructor, la función s_j es la función de paso correspondiente al constructor c_j.

Del ejemplo de las listas sabemos que la función a definir es $f = map : (A \to B) \times List(A) \to List(B)$, tenemos como dominio la función a aplicar a cada elemento de una lista y el conjunto inductivo de las listas sobre un conjunto A en donde el objeto básico es la lista vacía: $c_1 = nil$ y el constructor es la función *cons*: $c_2(a, l) = cons(a, l)$. Para definir la función *map* se tiene el caso del objeto básico y se define al objeto: $a_1 = nil$ y para el caso del constructor la función de paso $s_2 = cons$.
El elemento misterioso que involucra a $f(x) = map\ g\ (l)$ es $''\dots f(x)\dots'' = (g(a), map\ g\ (l)$.

La función de paso es un elemento importante para el principio anterior, ya que permite ir del paso anterior al que se está definiendo, es el puente que hace que cualquier definición descrita mediante el principio de iteración sobre un conjunto inductivo sea posible. Esta función de paso puede ser distinta para cada función a definir, en los ejemplos anteriores para la función *odd* la función *not* es la función de paso, en la función *long* que obtiene la longitud de una lista la función de paso es *succ* y en la función *map* que aplica una función a los objetos de la lista la función de paso es *cons*. Pero si queremos definir una función que determine si una lista es vacía o no, $empty(l)$ tenemos:

$$empty : List(A) \to Bool$$
$$empty(nil) = True \quad empty(cons(a, l)) = False$$

La función de paso es la constante $False$, es decir una función con cero argumentos. Es importante observar que en una definición mediante iteración, el valor de la función en un constructor depende exclusivamente de un valor de la función en un argumento estrictamente más simple que el actual. Es decir, un argumento construido previamente.

El principio de recursión primitiva es un principio que puede ser particularizado en el de iteración, tienen por diferencia un argumento extra: la recursión primitiva agrega como argumento esencial al objeto con el que se construye el nuevo objeto, es decir el argumento del constructor:

Definición 1.6 (Principio de recursión primitiva)**.** *Dados $a_i \in A$ y $s_j : F_j(\mathcal{I} \times A) \to A$ existe una única función $f : \mathcal{I} \to A$ tal que:*

$$f(c_i) = a_i$$
$$f(c_j x) = s_j(\dots(x, f(x))\dots),\ x \in \mathcal{I}$$

donde la expresión $''\dots(x, f(x))\dots''$ denota a un elemento del dominio de s_j que involucra a $(x, f(x))$. La función s_j es la función de paso.

La función de paso tiene el mismo fin que la función de paso explicada para el principio de iteración, solamente incluye un argumento más que es el del constructor, es decir, un objeto que ya ha sido construido.

Hasta este punto hemos formalizado los conceptos referentes a la inducción y la recursión, no hemos agregado nada nuevo, solamente es un recordatorio que nos servirá para explicar la coinducción.

1.2. El Reino Coinductivo

Esta sección está dedicada a un tema poco conocido y estudiado pero aplicado al ámbito computacional: la coinducción. Será explicada en base a su dual la inducción, para hacer más fácil su reconocimiento y comprensión.

Para iniciar tomemos un ejemplo, adaptado del ejemplo que aparece en [1]. Supongamos que tenemos una máquina con dos botones, *value* y *next* y cada vez que se aprieta el botón *next* la máquina realiza una serie de operaciones y cada vez que se aprieta el botón *value* devuelve un valor que describe cuál es el estado actual de la máquina.
No sabemos cómo funciona la máquina, es decir qué operaciones realiza para cambiar de estado, sólo podemos ver el valor que devuelve después de haber apretado repetidas veces el botón de *next*, digamos unas n veces, seguido de apretar el botón *value*.
Podemos apretar cuantas veces queramos el botón de *next* y el de *value* generando una lista de los valores que describen el estado de la máquina, si nunca nos detenemos tendremos una lista infinita de valores, un flujo continuo de valores que describen el estado de la máquina.

La lista infinita de valores o flujo de datos generada es llamada en ciencias de la computación *stream* por su nombre en inglés, debido a que se parece al comportamiento de un sistema de transición determinístico. Si tenemos dos estados de la máquina s y s' podemos decir que pasamos del estado s al s' con el valor a:

$$s \xrightarrow{a} s' \quad \text{si y sólo si} \quad value(s) = a \ \text{ y } \ next(s) = s'$$

La lista infinita tiene datos que indican el estado de la máquina, podemos decir que éstos son sus partes finitas y sólo son estos datos los que podemos conocer individualmente, la lista infinita es un objeto que no puede ser visto en su totalidad.

Pensemos ahora en poder conformar un conjunto de estos flujos de datos, debemos describir el conjunto de streams de una manera general o podemos decir cómo construir un

stream; luego entonces nos podemos preguntar varias cosas: ¿esta lista infinita puede ser construida de una manera semejante a la conocida, es decir, recursivamente?, ¿cúal sería el objeto básico del conjunto de las listas infinitas?, ¿es posible definir una lista infinita de una manera general?, ¿podemos dar un valor y a partir de él generar la lista infinita de los diferentes estados de la máquina que son resultado de apretar n veces el botón $next$?.

Sustituyamos los valores que describen el estado de la máquina por números naturales, entonces tenemos listas infinitas o streams de números naturales, a ese conjunto lo llamaremos $Stream(\mathbb{N})$. Si queremos dar una definición de función que construya una lista infinita de números naturales podemos pensar en la siguiente:

$$from : \mathbb{N} \to Stream(\mathbb{N})$$
$$from(n) = (n, n+1, n+2, n+3, \dots)$$

En otras palabras, a partir de un número natural se puede conformar una lista infinita de números naturales, el número dado y los que le suceden. Ésta puede ser una forma que permita "construir" un objeto infinito, pero qué pasa cuando queremos obtener $from(n+1)$:

$$from(n) = (n, n+1, n+2, n+3, \dots)$$
$$from(n+1) = (n+1, n+2, n+3, n+4, \dots)$$

Podemos observar que sucede lo siguiente:

$$from(n+1) = tail(from(n))$$

La función $tail$ elimina el primer elemento de una lista infinita. Al definir $from(n+1)$ podemos usar algo que hayamos construido antes, una parte de la lista infinita generada a partir de n. Aparentemente parece que el objeto $from(n)$, usado para definir $from(n+1)$, debió ser construido antes, usamos una parte de la lista infinita construida con un argumento más pequeño, pero observemos que la lista $from(n)$ no es estructuralmente más simple que la obtenida con $from(n+1)$ tiene un elemento más.
Esto asemeja a la recursión, usamos un objeto definido "antes", pero observemos otro detalle: para construir $from(n)$ debemos tener $from(n+1)$ ya que se encuentra contenido en el primero. Nos encontramos un ciclo, necesitamos de uno para construir el otro y viceversa; entonces no estamos usando la recursión para construir objetos nuevos del conjunto, sólo necesitamos una parte del objeto construido para obtener el siguiente.
Podemos pensar en escoger las partes de un objeto existente que sirven para obtener otro objeto.

En diversas partes de las matemáticas se puede encontrar el dual de algún objeto o propiedad, un caso particular de este fenómeno es la inducción y la coinducción. La coinducción es el dual de la inducción y es una herramienta para definir estructuras infinitas con objetos finitos, infinitos o potencialmente infinitos.
Hemos dicho que la inducción es el dual de la coinducción, así podemos pensar en dualizar todo lo que definimos en la sección pasada. Para construir objetos usamos la recursión y construimos conjuntos inductivos; al dualizar, a partir de un conjunto coinductivo de objetos es necesario destruir sus elementos para obtener otros que de igual forma pertenezcan

al conjunto. Debemos tratar de observar cómo se comporta un objeto estructuralmente infinito, ver sus partes y para esto hay que destruirlo en sus partes finitas o infinitas. En la sección anterior se enfatizó que para definir un conjunto de objetos inductivos se tiene dentro de ese conjunto a los objetos básicos y también se tienen a los constructores que son funciones que reciben como argumento un objeto dentro del conjunto y construyen uno nuevo; para definir un conjunto de objetos coinductivos se debe tratar de describir a los elementos de ese conjunto por medio de funciones destructoras que actúen sobre el conjunto, describir las observaciones que se pueden hacer a los objetos coinductivos.

Regresando al ejemplo de las listas infinitas de números naturales, no podemos saber exactamente qué elementos conforman al stream, ni en una hoja de papel ni en un monitor, es por eso que sólo podemos observar una parte del stream: a partir de una lista infinita de elementos de un conjunto A podemos obtener la parte inicial de la lista, es decir su cabeza y observarla, y al quitar la cabeza nos quedamos con el resto de ella, la cola, que sigue siendo una lista infinita.

La existencia de un objeto infinito está garantizada por cada una de sus partes finitas, las partes finitas de los streams de números naturales son cada uno de los números y con ayuda de las funciones que los destruyen podemos observarlos por partes, es decir, obtener cada número del stream: el primero lo obtenemos con la función destructora *head*, para obtener el siguiente número debemos tener la cabeza del resto de la lista esto es aplicar primero el destructor *tail* para tener la cola y luego *head*, si queremos al tercer elemento del stream obtenemos la cabeza de la cola de la cola del stream original y así sucesivamente.

Hemos visto que no podemos construir listas infinitas pero podemos garantizar su existencia gracias a todas sus partes finitas, entonces ¿qué sucede con la construcción del conjunto $Stream(\mathbb{N})$?
Supondremos la existencia del conjunto de listas infinitas de números naturales, y a estos objetos se les aplicarán las funciones destructoras. Esta suposición no es una idea "mágica" que soluciona el hecho de no poder conformar el conjunto de las listas infinitas de números naturales o de un conjunto coinductivo, sino que por el contrario está respaldado por la teoría de puntos fijos y la teoría de categorías que serán vistas en las siguientes secciones.

Veamos como queda la definición de streams:

Definición 1.7 (Listas Infinitas o Streams sobre un conjunto A)**.** *El conjunto de listas infinitas sobre un conjunto A, $Stream(A)$, se define mediante:*

i) Si $s \in Stream(A)$ entonces $head(s) \in A$

ii) Si $s \in Stream(A)$ entonces $tail(s) \in Stream(A)$

iii) Si X cumple con i) y con ii) entonces $X \subseteq Stream(A)$

La condición *iii*) enfatiza que el conjunto $Stream(A)$ es el conjunto más grande que cumple con las otras dos cláusulas.

Así como se pueden tener definiciones inductivas de funciones se pueden tener definiciones coinductivas; para poder exponer las observaciones hechas al objeto coinductivo se debe definir a la función f dando los valores de todos los destructores para cada $f(x)$ es decir, aplicar cada uno de los destructores a la función. En la sección anterior vimos como ejemplo las listas de números naturales y ejemplos de funciones, la longitud y la aplicación de una función a cada uno de sus elementos, en esta sección tenemos a la listas infinitas de números naturales, no podemos definir la longitud de cada stream pero si podemos definir la aplicación de una función a cada uno de los elementos de la lista, si los elementos del stream pertenecen al conjunto A entonces aplicar una función $g : A \to B$ a cada elemento de la lista se define como:

$$\begin{gathered} map\ g\ : Stream(A) \to Stream(B) \\ head(map\ g(l)) = g(head(l)) \\ tail(map\ g(l)) = map\ g(tail(l)) \end{gathered}$$

Para cualquier función que deseemos definir que de como resultado un stream, $f : D \to Stream(A)$, tenemos el siguiente principio de coiteración:

Definición 1.8 (Principio de coiteración para Stream(A))**.** *Dadas dos funciones $s_1 : D \to A$, $s_2 : D \to D$ y un stream x, existe una única función $f : D \to Stream(A)$ tal que:*

$$\begin{gathered} head(f(x)) = s_1(x) \\ tail(f(x)) = f(s_2(x)) \end{gathered}$$

Obsérvese la dualidad de las definiciones, en la definición coinductiva tenemos lo opuesto a la inductiva. En la inductiva se debe de aplicar la función a definir tanto a los objetos básicos como a los constructores del conjunto inductivo, en la coinductiva es aplicar los destructores a la imagen de la función. Las funciones destructoras o más sencillamente los destructores de un objeto coinductivo, son funciones que tienen como dominio al conjunto de objetos coinductivos $\mathcal{C}$, en este caso $Stream(A)$, y por codominio un conjunto que está en función del conjunto coinductivo, $d_k : \mathcal{C} \to F_k(\mathcal{C})$.
Para generalizar fácilmente el principio de coiteración para cualquier conjunto coinductivo $\mathcal{C}$, se debe realizar la dualización de éste necesariamente bajo el contexto de la teoría de categorías, esto será visto con más detalle en esa sección pero diremos que al definir una función que tenga como codominio un conjunto coinductivo $f : A \to \mathcal{C}$ se debe definir el resultado de aplicar cada destructor a la imagen de la función:

$$d_k(f(x)) = (\dots f \dots)(s_k x) \tag{1.1}$$

Esto es aplicar una función que involucra a f, $(\dots f \dots) : F_k(B) \to F_k(\mathcal{C})$, a la función de paso correspondiente a ese destructor.

1.3. Teoría de Puntos fijos

Una manera de formalizar lo visto en las secciones anteriores de este capítulo, es mediante la teoría de puntos fijos. Un punto fijo de una función es un punto que es mapeado así mismo bajo la función, es decir, una solución a la ecuación $X = F(X)$.

Para comenzar usaremos las definiciones e ideas propuestas por Tarski en [20], así como la de otros autores que trabajan teniendo como base la teoría de puntos fijos para el manejo de conjuntos de objetos inductivos o de objetos coinductivos y cómo están relacionadas con la inducción y coinducción como [14], [13] y [7].

La notación usada en esta sección es la adoptada por algunos autores para manejar los operadores monótonos y los puntos fijos, dado que es más fácil trabajar con ella.

1.3.1. Conceptos básicos

Definición 1.9 (Operador monótono)**.** *Sea $\mathscr{P}(A)$ el conjunto potencia de un conjunto A. Un operador sobre A es una función $F : \mathscr{P}(A) \to \mathscr{P}(A)$. Un operador es monótono si se cumple la siguiente implicación:*

$$si\ X \subseteq Y \subseteq A\ entonces\ F(X) \subseteq F(Y)$$

Es decir un operador monótono o función monótona preserva el orden.

Definición 1.10 (Puntos prefijo, postfijo y fijo)**.** *Sea $F : \mathscr{P}(A) \to \mathscr{P}(A)$ un operador. Un subconjunto $\mathcal{K} \subseteq A$ es*

- *F-cerrado o punto prefijo de F si $F(\mathcal{K}) \subseteq \mathcal{K}$.*
- *F-denso o punto postfijo de F si $\mathcal{K} \subseteq F(\mathcal{K})$.*
- *punto fijo de F si $F(\mathcal{K}) = \mathcal{K}$.*
- *F-inductivo si es incluido en cada punto prefijo de F, es decir, si $F(X) \subseteq X$ implica $\mathcal{K} \subseteq X$*
- *F-coinductivo si contiene a cada punto postfijo de F, es decir, si $X \subseteq F(X)$ implica $X \subseteq \mathcal{K}$*

Lema 1.1. *Se cumplen las siguientes afirmaciones:*

i) F tiene a lo más un punto prefijo inductivo y a lo más un punto postfijo coinductivo.

ii) Los puntos prefijos inductivos y postfijos coinductivos de operadores monótonos son puntos fijos.

Demostración:

i) Para demostrar que a lo más hay un punto prefijo inductivo supondremos que existen dos puntos prefijos inductivos $\mathcal{K}_1$ y $\mathcal{K}_2$ y veremos que son el mismo:
Sabemos que $\mathcal{K}_2$ es prefijo es decir $F(\mathcal{K}_2) \subseteq \mathcal{K}_2$ y dado que $\mathcal{K}_1$ es inductivo podemos concluir que $\mathcal{K}_1 \subseteq \mathcal{K}_2$.
Por otro lado como $\mathcal{K}_1$ es prefijo es decir $F(\mathcal{K}_1) \subseteq \mathcal{K}_1$ y $\mathcal{K}_2$ es inductivo entonces $\mathcal{K}_2 \subseteq \mathcal{K}_1$.
Por lo tanto $\mathcal{K}_1 = \mathcal{K}_2$, es decir a lo más hay un punto prefijo inductivo.
La demostración para el punto postfijo coinductivo es semejante.

ii) Se demostrará la doble contención para ver que $F(\mathcal{K})$ y $\mathcal{K}$ son iguales:
Sea $\mathcal{K}$ un punto prefijo e inductivo, se cumple que $F(\mathcal{K}) \subseteq \mathcal{K}$, ya que es prefijo.
Por otra parte partiendo del hecho de ser punto prefijo y ya que F cumple con la propiedad de ser monótono tenemos que $F(\mathcal{K}) \subseteq \mathcal{K} \Rightarrow F(F(\mathcal{K})) \subseteq F(\mathcal{K})$ y podemos concluir que $F(F(\mathcal{K})) \subseteq F(\mathcal{K})$, es decir $F(\mathcal{K})$ es prefijo. Como también sabemos que $\mathcal{K}$ es inductivo entonces podemos concluir que $\mathcal{K} \subseteq F(\mathcal{K})$.
Por lo tanto $F(\mathcal{K}) = \mathcal{K}$.
La demostración para el punto postfijo coinductivo es análoga.

□

Definición 1.11 (Ínfimo y supremo de los puntos prefijos y postfijos)**.** *Se definen los siguientes conjuntos:*

$$\mu X.F(X) := \bigcap\{S | F(S) \subseteq S\} \text{ (el ínfimo de los puntos prefijos)}$$
$$\nu X.F(X) := \bigcup\{S | S \subseteq F(S)\} \text{ (el supremo de los puntos postfijos)}$$

Estos conjuntos son puntos prefijos y postfijos de F, es decir: $F(\mu X.F(X)) \subseteq \mu X.F$ y $\nu X.F \subseteq F(\nu X.F)$, veamos porqué y para eso tenemos el siguiente lema:

Lema 1.2. *Si X_i es una colección de puntos prefijos de F entonces la intersección, $\bigcap_i X_i$, también lo es y si X_i es una colección de puntos postfijos de F entonces la unión, $\bigcup_i X_i$, también lo es.*

Demostración: La demostración se dividirá en dos partes, una para cada caso:

i) Para demostrar que la intersección de prefijos es prefija supondremos que X_i es prefijo para toda i, es decir $F(X_i) \subseteq X_i$ y por lo tanto podemos decir que $\bigcap_i F(X_i) \subseteq \bigcap_i X_i$.
Dado que F es monótono y que $\bigcap_i X_i \subseteq X_i$ para toda i también podemos decir que $F(\bigcap_i X_i) \subseteq F(X_i)$.
Entonces tenemos por transitividad que $F(\bigcap_i X_i) \subseteq \bigcap_i F(X_i) \subseteq \bigcap_i X_i$.
Por lo tanto $F(\bigcap_i X_i) \subseteq \bigcap X_i$, es decir $\bigcap_i X_i$ es prefijo.

ii) Ahora para demostrar que la unión de postfijos es postfija supondremos que X_i es postfijo entonces tenemos que $\bigcup_i X_i \subseteq \bigcup_i F(X_i)$.
Al ser F monótono y como $X_i \subseteq \bigcup_i X_i$ tenemos que $F(X_i) \subseteq F(\bigcup_i X_i)$ para cada i.
Por lo tanto $\bigcup_i F(X_i) \subseteq F(\bigcup_i X_i)$ y finalmente por transitividad $\bigcup_i X_i \subseteq F(\bigcup_i X_i)$, es decir, $\bigcup_i X_i$ es postfijo.

□

Teorema 1.1 (Knaster-Tarski)**.** *Todo operador monótono tiene un punto fijo inductivo y un punto fijo coinductivo. Más aún:*

- *$\mu X.F(X)$ es el mínimo punto fijo de F*
- *$\nu X.F(X)$ es el máximo punto fijo de F*

Demostración: Se demostrará el caso para $\mu X.F(X)$, el restante se puede demostrar por argumento dual. Así mismo se dividirá la demostración en dos partes, la primera para verificar que $\mu X.F(X)$ es fijo y la segunda par ver que es el mínimo.

a) Para ver que es fijo podemos usar el lema 1.1, debemos encontrar que es prefijo e inductivo.
Sabemos que $\mu X.F(X)$ es prefijo por definición es decir $F(\mu X.F(X)) \subseteq \mu X.F(X)$ y es inductivo porque está contenido en cada punto prefijo de F.
Por lo tanto $\mu X.F(X)$ es prefijo e inductivo, y por el lema anterior es punto fijo.

b) Para ver que es el mínimo partiremos de $F(X) \subseteq X \Rightarrow \mu X.F(X) \subseteq X$, es decir $\mu X.F(X)$ es fijo inductivo.
Sea J un punto fijo, $F(J) = J$ en particular $F(J) \subseteq J$, tal que $J \subseteq \mu X.F(X)$. Dado que $\mu X.F(X)$ es inductivo entonces $\mu X.F(X) \subseteq J$, por lo tanto son el mismo, $\mu X.F(X)$ es el mínimo punto fijo de F.

□

A partir de éste teorema podemos decir que $\mu X.F(X)$ es la solución más pequeña a $F(X) = X$ y que $\nu X.F(X)$ es la solución más grande a $X = F(X)$. Esto nos remonta a las secciones pasadas, habíamos dicho que un conjunto inductivo está determinado si es el conjunto más pequeño que cumple con ciertas cláusulas, las que definen a los objetos básicos y a los constructores de objetos de ese conjunto, y si existe un conjunto que haga verdaderas las cláusulas que definen el comportamiento de los destructores de objetos coinductivos ese conjunto será el más grande que contenga a esos objetos, es decir el conjunto coinductivo.

1.3.2. Inducción y Coinducción

La solución más pequeña y la más grande a $X = F(X)$ nos hace pensar en los conjuntos inductivos y coinductivos, es decir los conjuntos más pequeños y más grandes que cumplan ciertas cláusulas; con esta idea podemos enunciar el siguiente principio de demostración:

Definición 1.12 (Principio de Inducción y de Coinducción)**.** *Sea F un operador monótono entonces el Principio de Inducción para F es:*

$$F(X) \subseteq X \Rightarrow \mu X.F(X) \subseteq X$$

Y el Principio de Coinducción para F es:

$$X \subseteq F(X) \Rightarrow X \subseteq \nu X.F(X)$$

De aquí se sigue que para definir inductivamente un conjunto de objetos éste debe ser la solución más pequeña de una inecuación y para que un conjunto de objetos esté definido coinductivamente debe ser la solución más grande de una inecuación.

Veamos un ejemplo sencillo, considere un conjunto $X \subseteq U$ el cual contiene un elemento 0 y una función $succ : X \to X$, si se define una función monótona $F : \mathscr{P}(U) \to \mathscr{P}(U)$ mediante: $F(X) = \{0\} \bigcup \{succ(x) \mid x \in X\}$ entonces podemos definir a los naturales como el conjunto

$$\mathbb{N} = \mu X.F(X) \tag{1.2}$$

Es decir, los naturales son el conjunto más pequeño que contiene a 0 y a $\{succ(n)|n \in X\}$.

Normalmente cuando se requiere demostrar una propiedad P de los números naturales se usa como método de demostración la inducción, para esto es necesario demostrar que la propiedad se cumple para las cláusulas del principio de inducción, es decir se cumple para el cero $P(0)$ y si suponemos cierta la propiedad para un número n debe de cumplirse para el siguiente número $P(n) \to P(succ(n))$. Por lo tanto concluimos que todo número natural cumple con la propiedad: $\forall x(\mathbb{N}(x) \to P(x))$.
Esto lo podemos resumir en la siguiente fórmula de la lógica de segundo orden:

$$\forall P(\ P(0) \wedge \forall x(P(x) \to P(succ(x))) \Rightarrow \forall x(\mathbb{N}(x) \to P(x)))$$

Para toda propiedad, si se demuestra que la posee el cero y cada vez que un objeto $succ(n)$ cumple la propiedad es porque se tiene que n la cumple de antemano entonces se puede decir que todo número natural cumple con la propiedad.
Cuando usamos el principio de inducción en puntos fijos, es decir la definición 1.12, para demostrar una propiedad de los números naturales, debemos tener en cuenta que $\mathbb{N} = \mu X.F(X)$ y debemos formar un conjunto S que contenga a los objetos que cumplen la propiedad a demostrar. A continuación debemos demostrar que ese conjunto cumple con la inecuación $\{0\} \bigcup \{succ(x)|x \in S\} \subseteq S$ y así concluir que ese conjunto contiene a los números naturales $\mathbb{N} \subseteq X$.

Al demostrar la inecuación estamos diciendo que el conjunto de los números que cumplen con la propiedad en particular cumplen con las clausulas que definen a los naturales. Si la propiedad a demostrar es la definición de alguna función sobre los números naturales $f : \mathbb{N} \to B$, $S = \{x|x \in \mathbb{N}, f(x) \in B\}$ entonces estamos demostrando que el conjunto $F(S)$ cumple con las condiciones de los números naturales, es decir que se define a la función para el cero y para el sucesor de un número.
Esto nos lleva a pensar en la iteración, la definición del principio 1.5 en la página 4. Tenemos a la función de paso que permite definir la función para un objeto obtenido a partir de una función constructora $s_j : F_j(S) \to S$, esta función atestigua el hecho de que $F(S) \subseteq S$, partimos de un dominio que está en función del conjunto inductivo y nos lleva al mismo conjunto inductivo, estamos haciendo que el conjunto S cumpla con la inecuación que define a un conjunto inductivo mediante puntos fijos.

Del lado coinductivo, al dualizar lo anterior, tomaremos el ejemplo de los streams de la definición 1.7. Sabemos que podemos destruirlos en cabeza y cola, *head* y *tail* y que esas dos condiciones permiten que el conjunto más grande que las cumpla, X, sea el de $Stream(A)$. Por lo tanto tenemos que la solución más grande a la inecuación: $X \subseteq \{s \in X \mid head(s) \in A\} \bigcap \{s \in X \mid tail(s) \in X\}$ es el punto fijo

$$\nu X.F(X) = Stream(A) \tag{1.3}$$

Podemos ver reflejada la coinducción en este ejemplo pero si tomamos el principio de coiteración para streams, de la definición 1.8, al definir una función que nos lleve al conjunto $Stream(A)$, $f : D \to Stream(A)$ sabemos que hay que definir cómo actúan los destructores sobre la imagen de la función. Haremos que un conjunto S contenga a las listas que son resultado de aplicarles la función a cada elemento de la función, este conjunto también cumple con las condiciones que determinan el conjunto $Stream(A)$; en otras palabras, hacer que el conjunto S cumpla con la inecuación que define el punto fijo $\nu X.F(X) = Stream(A)$.

De éstos ejemplos es claro ver cómo pasamos del principio de inducción y de coinducción al principio de iteración y coiteración respectivamente; en las secciones pasadas también hemos definido los principios de recursión y correcursión para definir funciones que involucran objetos inductivos o coinductivos pero en esta teoría ¿cómo es posible fundamentarlos?
Para esto veamos un ejemplo con naturales, demostraremos una propiedad de ellos para ver cómo funciona la definición 1.12 como método de demostración.
Sabemos que $\mathbb{N} = \mu X.F(X)$ donde $F(X) = \{0\} \bigcup \{succ(x) \mid x \in X\}$, suponer que queremos demostrar la propiedad: si $n \in \mathbb{N}$ entonces $n^2 \in \mathbb{N}$. Según el principio definiremos el conjunto de los números cuyo cuadrado es un número natural: $S := \{r | r^2 \in \mathbb{N}\}$ y debemos llegar a la conclusión: si $F(S) \subseteq S$ entonces $\mathbb{N} \subseteq S$. Para efectos de la demostración supondremos que las operaciones sobre los números naturales son cerradas bajo ellos, esto puede probarse por inducción.

Al demostrar la inecuación debemos ver que $0 \in S$, pero esto es claro. También debemos demostrar el caso para el sucesor, supongamos que $r \in S$, esto derivado de que $r^2 \in \mathbb{N}$. Ahora hay que demostrar que $succ(r) \in S$, es decir $succ(r)^2 \in \mathbb{N}$, sabemos que $succ(r)^2 = r^2 + r + r + 1$ solamente tenemos que asegurar que cada parte esta en $\mathbb{N}$. Pero solo sabemos que $r^2 \in \mathbb{N}$, no podemos asegurar nada a cerca de r, es decir no podemos concluir si $r \in S$.

El principio utilizado tiene fallas, no es posible tener una demostración; para poder corregir este detalle debemos suponer que $r \in \mathbb{N}$ es decir hay que construir un nuevo conjunto $S' = S \bigcap \mathbb{N}$, al asegurar la pertenencia de r en $\mathbb{N}$ se puede tener que $F(S') \subseteq S$. Entonces si tenemos que $F(S') \subseteq S'$ tendremos que $\mathbb{N} \subseteq S'$ que en realidad lo podemos interpretar como que $\mathbb{N} \subseteq S$.
Derivado de esta solución tenemos el siguiente principio:

Definición 1.13 (Principio de Inducción y de Coinducción extendida)**.** *Sea F un operador monótono, los principios de inducción y coinducción extendidos son:*

$$F(\mu X.F(X) \bigcap X) \subseteq X \Rightarrow \mu X.F(X) \subseteq X$$

$$X \subseteq F(\nu X.F(X) \bigcup X) \Rightarrow X \subseteq \nu X.F(X)$$

Del ejemplo de la propiedad si $n \in \mathbb{N}$ entonces $n^2 \in \mathbb{N}$ podemos ver hay que agregar el conjunto inductivo como parte de la hipótesis para que la demostración sea posible.

A partir de él podemos generar el principio de recursión de manera semejante a como fue inferido el de iteración a partir del principio de inducción. Si queremos demostrar una propiedad de los números naturales tenemos la siguiente fórmula:

$$\forall P(\ P(0) \wedge \forall x(\mathbb{N}(x) \wedge P(x) \rightarrow P(succ(x))) \Rightarrow \forall x(\mathbb{N}(x) \rightarrow P(x)))$$

Únicamente se agrega como condición importante, en el antecedente para demostrar que $P(succ(x))$, el que x cumpla con la propiedad de ser un número natural. Parece innecesaria esta condición pero es de gran utilidad, además de tener el conjunto que se desea cumpla con la propiedad se tiene al propio conjunto inductivo.

El principio 1.13, su parte inductiva $F(\mu X.F(X) \bigcap X) \subseteq X \Rightarrow \mu X.F(X) \subseteq X$, puede ser derivada como se muestra a continuación:
Tomaremos como hipótesis $F(\mu X.F(X) \bigcap X) \subseteq X$ y consideremos las siguientes implicaciones:

1. $(\mu X.F(X)) \bigcap X \subseteq X \quad \Rightarrow F(\mu X.F(X) \bigcap X) \subseteq F(X)$
2. $(\mu X.F(X)) \bigcap X \subseteq \mu X.F(X) \quad \Rightarrow F(\mu X.F(X) \bigcap X) \subseteq F(\mu X.F(X))$

De éstas podemos concluir que:
$F(\mu X.F(X) \bigcap X) \subseteq F(X) \bigcap F(\mu X.F(X)) \subseteq F(\mu X.F(X)) \subseteq \mu X.F(X)$.
Si sabemos que $\mu X.F(X)$ es fijo entonces nuestra conclusión previa se convierte en: $F(\mu X.F(X) \bigcap X) \subseteq F(X) \bigcap \mu X.F(X)$ y en particular agregando la hipótesis:

$$F(\mu X.F(X) \bigcap X) \subseteq \mu X.F(X) \bigcap X \subseteq X$$

obtenemos $\mu X.F(X) \subseteq X$ ya que $\mu X.F(X) \subseteq \mu X.F(X) \bigcap X \Rightarrow \mu X.F(X) \subseteq X$.
Finalmente concluimos que a partir de $F(\mu X.F(X) \bigcap X) \subseteq X$ llegamos a $\mu X.F(X) \subseteq X$.
Análogamente se puede derivar la parte del principio 1.13 correspondiente a la coinducción extendida.

Podemos resumir, como es hecho en [7], que $\mu X.F(X)$ es la solución más pequeña a $X = F(X)$ y que es el conjunto definido inductivamente por F, que $\nu X.F(X)$ es la solución más grande a $X = F(X)$ es el conjunto definido coinductivamente por F. Para demostrar una propiedad de algún conjunto inductivo $\mathcal{I}$ o coinductivo $\mathcal{C}$, es necesario encontrar la solución a una inecuación, es decir para poder usar la inducción o la coinducción se debe tener un conjunto S que será el que contenga a los objetos que cumplen la propiedad a demostrar y ese conjunto deberá satisfacer la inecuación que define al conjunto inductivo o coinductivo. De esta manera se demostrará que $\mathcal{I} \subseteq S$ e $\mathcal{I}$ será al menos ese conjunto, y para el conjunto coinductivo $\mathcal{C}$ se demostrará que $S \subseteq \mathcal{C}$ para que $\mathcal{C}$ sea la solución más grande. El principio de inducción y coinducción permite generar al mecanismo de definición que hemos usado, la iteración y coiteración; así mismo el principio de inducción y coinducción extendida genera el de recursión y correcursión.

1.4. Teoría de Categorías

La Teoría de las Categorías es una forma abstracta de tratar objetos matemáticos y las relaciones entre ellos; trabajaremos con los conjuntos de objetos inductivos y coinductivos que pueden ser vistos como objetos en una categoría.

A continuación se dan algunas definiciones básicas sobre teoría de categorías suficientes para el trabajo aquí realizado para más detalle se puede consultar [12] y [1]. Además se introducen algunas convenciones notacionales que se usarán a lo largo del trabajo desarrollado.

1.4.1. Conceptos Básicos

Definición 1.14 (Categoría)**.** *Una categoría C es un par $C = \langle \mathcal{O}, \mathcal{M} \rangle$ donde $\mathcal{O}$ es una colección de objetos y $\mathcal{M}$ es una colección de morfismos los cuales son funciones $f : X \to Y$ para cada par de objetos X, Y tales que:*

- *Si f es un morfismo del objeto X al objeto B y g es un morfismo del objeto X al objeto Z entonces la composición $g \circ f$ es un morfismo del objeto X al objeto Z*
- *Para cada objeto $X \in C$ existe un morfismo identidad: $\mathsf{Id}_X : X \to X$*
- *Para cada $f : W \to X$, $g : X \to Y$ y $h : Y \to Z$ se tiene que la función $\circ$ es asociativa:*

$$[(h \circ g) \circ f](X) = [h \circ g](f(X)) = h(g(f(X))) = h([g \circ f](X)) = [h \circ (g \circ h)](X)$$

- *Para cada $f : X \to Y$ se tiene que $(f \circ \mathsf{Id}_X)(X) = f(X)$ y $\mathsf{Id}_Y \circ f(X) = f(X)$*

Definición 1.15 (Funtor)**.** *Dadas dos categorías C y D, un funtor $F : C \to D$ consiste de dos operaciones que:*

- *envían objetos de la primera categoría en objetos de la segunda, si $X \in C$ entonces $F(X) \in D$*
- *envían morfismos de C en morfismos de D, si $f : X \to Y \in C$ entonces $F(f) : F(X) \to F(Y) \in D$*

tales que se cumplen las siguientes propiedades:

1. *Para todo objeto $X \in C$ sucede que $F(\mathsf{Id}_X) = \mathsf{Id}_{F(X)}$*
2. *Para los morfismos en C, $f : X \to Y$ y $g : Y \to Z$ se tiene que $F(g \circ f) = F(g) \circ F(f)$*

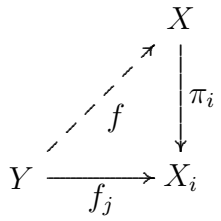
Las definiciones que a continuación se presentan están relacionadas directamente con el trabajo que se realiza aquí y para esto enunciaremos un principio muy importante en teoría de categorías:
Principio de dualidad Si alguna afirmación de categorías y funtores es verdadera o es un teorema entonces la afirmación dual también lo es. La afirmación dual puede ser obtenida, informalmente, al cambiar el sentido de las flechas usadas en la afirmación inicial, es decir cambiar el dominio de la afirmación inicial por el codominio de ella y viceversa.
El dual de una categoría C se denomina C^{op}, ésta categoría tiene los mismos objetos que la primera pero los morfismos o flechas cambian de sentido, es decir si en la categoría C hay un morfismo $f : X \to Y$ en la categoría opuesta está el morfismo $\bar{f} : Y \to X$. El dual de una propiedad en la categoría C tiene por opuesto una propiedad en la categoría C^{op}, cualquier teorema verdadero para las categorías C es verdadero para las categorías C^{op}.

Además podemos notar que $(C^{op})^{op} = C$.
Otra observación pertinente es la siguiente, al trabajar con categorías se debe tener en cuenta que se usarán diagramas conmutativos y que las propiedades derivadas de éstos en categorías implican una propiedad universal.

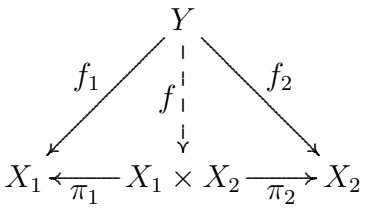
A continuación se dan las definiciones de producto y coproducto, útiles para poder incluirlos en las categorías que usaremos ya que para describir los objetos inductivos y coinductivos es necesario el uso de funtores.
Un ejemplo muy conocido de un producto es el producto cartesiano $A \times B$ que permite tener dos conjuntos y relacionar los elementos de ellos, necesariamente en el orden A seguido de B para obtener $A \times B$:

$$A \times B = \{(a, b) \mid a \in A \text{ y } b \in B\}$$

Definición 1.16 (Producto y par)**.** *Sean C una categoría y $\{X_i : i \in I\}$ una familia de objetos indexados en C, el producto de la familia $\{X_i\}$ es un objeto X junto con una colección de morfismos $\pi_i : X \to X_i$, llamados proyecciones, que satisfacen la siguiente propiedad universal: para todo objeto Y y una colección de morfismos $f_i : Y \to X_i$ existe un único morfismo $f : Y \to X$ de tal forma que para todo $i \in I$ se da el caso que $f_i = \pi_i f$, haciendo que el siguiente diagrama conmute:*



Si la familia de los objetos indexados sólo tiene dos elementos tenemos al producto como usualmente lo conocemos $x_1 \times x_2$ y el siguiente diagrama:

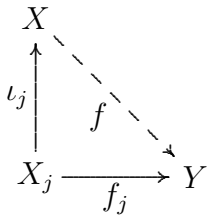


En este diagrama podemos ver que el producto se forma con las proyecciones: π_1, π_2.
El par es denotado por $\langle f_1, f_2 \rangle \in X_1 \times X_2$

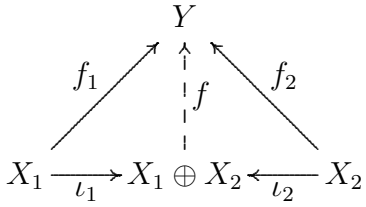
Aplicando el principio de dualidad, un coproducto, más usualmente llamado suma, es una construcción que sintetiza objetos, su evaluación depende del objeto:

$$B + C \to A$$
$$[f, g](x) \longmapsto \text{case x of} \begin{cases} inl(b) \Rightarrow f(b) \\ inr(c) \Rightarrow g(c) \end{cases}$$

Definición 1.17 (Coproducto y copar)**.** *Categóricamente: Sea C una categoría y sea $\{X_j : j \in J\}$ una familia de objetos indexados en C, el coproducto de la familia $\{X_j\}$ es un objeto X junto con una colección de morfismos $\iota_j : X_j \to X$, llamados inyecciones, que satisfacen la siguiente propiedad universal: para todo objeto Y y una colección de morfismos $f_j : X_j \to Y$ existe un único morfismo f que va de X a Y de tal forma que $f_j = f \circ \iota_j$, haciendo que el siguiente diagrama conmute:*



Si la familia de objetos consta de solo dos entonces el diagrama es el siguiente:

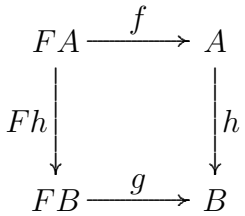


En este diagrama podemos ver claramente que el coproducto usual se forma con las inyecciones: ι_1, ι_2.
El copar es $[f_1, f_2] \in X_1 \oplus X_2$

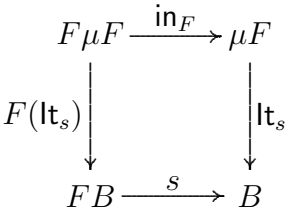
En adelante se omnitirán los paréntesis que acompañan la sintaxis de un funtor, es decir $F(X)$ cambiará por FX para facilitar su escritura.

Definición 1.18 (F-álgebra)**.** *Dado un funtor $F : C \to C$ en una categoría C, una F-álgebra es un par $\langle A, f\rangle$ tal que $f : FA \to A$.*

Definición 1.19 (Morfismos entre F-álgebras)**.** *Dadas dos F-álgebras $\langle A, f\rangle$ y $\langle B, g\rangle$ un morfismo de la primera a la segunda, $h : A \to B$, es un morfismo en C tal que el siguiente diagrama conmuta:*



Definición 1.20 (Álgebra inicial)**.** *Un álgebra inicial, $\langle A, f\rangle$, es un objeto inicial en la categoría de las F-álgebras, es decir, existe un único morfismo It_s, del álgebra inicial a cualquier otra $\langle B, s\rangle$. Si existe, el álgebra inicial es única y se denota como $(\mu F, \mathsf{in}_F)$:*



Las F-álgebras al tener en su par a $f : FA \to A$ nos hace recordar a los constructores de un conjunto inductivo: $c_j : F(\mathcal{I}) \to \mathcal{I}$, es por esto que las álgebras iniciales se usan para describir conjuntos inductivos. De la misma manera nos recuerda a los principios de inducción vistos en la sección de puntos fijos ya que se debe suponer que $F(X) \subseteq X$ para demostrar que $\mu X.F(X) \subseteq X$. Para poder explicar la transformación del símbolo $\subseteq$ al de $\to$ haremos referencia a una categoría importante, la categoría de conjuntos denotada por *Set*. Esta categoría tiene por objetos a conjuntos y por morfismos a funciones entre conjuntos, en particular podemos usar como función u operación a la contención. Si regresamos a las definiciones de la sección pasada, es decir los puntos fijos, podemos imaginar que todas las proposiciones en donde aparece $\subseteq$ será cambiado por $\to$ y de esa manera transportarnos al mundo de las categorías.
Dualmente para los conjuntos coinductivos se trabaja con las coálgebras finales, los destructores $d_j : \mathcal{C} \to F(\mathcal{C})$ se ven reflejados en $f : B \to FB$. Así mismo el principio de coinducción se puede traducir a categorías cambiando $\subseteq$ por $\to$ en $X \subseteq F(X) \Rightarrow X \subseteq \nu X.F(X)$.
A continuación dualizamos las definiciones anteriores:

Definición 1.21 (F-coálgebra)**.** *Dado un funtor $F : C \to C$ en una categoría C, una F-coálgebra es un par $\langle B, g\rangle$ tal que $f : B \to FB$.*

Definición 1.22 (Morfismos entre coálgebras)**.** *Dadas dos coálgebras $\langle B, g\rangle$ y $\langle A, f\rangle$, un morfismo de la primera a la segunda $h : B \to A$ es un morfismo en C tal que el siguiente diagrama conmuta:*

$$\begin{array}{ccc} B & \xrightarrow{g} & FB \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle Fh} \\ A & \xrightarrow{f} & FA \end{array}$$

Definición 1.23 (Coálgebra final)**.** *Una coálgebra final, $\langle B, g\rangle$, es un objeto terminal en la categoría de las F-coálgebras, es decir, existe un único morfismo Colt_s, de cualquier coálgebra $\langle A, s\rangle$ a la coálgebra final. Si existe, la coálgebra final es única y se denota como $(\nu F, \mathsf{out}_F)$:*

$$\begin{array}{ccc} A & \xrightarrow{s} & FA \\ {\scriptstyle \mathsf{Colt}_s}\downarrow & & \downarrow{\scriptstyle F(\mathsf{Colt}_s)} \\ \nu F & \xrightarrow{\mathsf{out}_F} & F\nu F \end{array}$$

Las notaciones para las álgebras iniciales y las coálgebras finales utilizan los símbolos que fueron usados en la sección de puntos fijos para referirse a la solución más pequeña, $\mu X.F(X)$, y a la solución más grande $\nu X.F(X)$.

Para los principios de recursión y correcursión es necesario nombrar dos definiciones más, éstas son las definiciones de las álgebras que son recursivas y correcursivas. La primera refleja el uso de productos y la segunda el de coproductos.

Definición 1.24 (F-álgebra recursiva)**.** *Sea $\Pi_D : C \to C$ el funtor definido como $\Pi_D C := C \times D$. Una F-álgebra $\langle A, f\rangle$ es recursiva si para toda $F(\Pi_A)$-álgebra $\langle B, g\rangle$ existe un morfismo $h : A \to B$ tal que:*

$$\begin{array}{ccc} FA & \xrightarrow{f} & A \\ {\scriptstyle F\langle \mathsf{Id}, h\rangle}\downarrow & & \downarrow{\scriptstyle h} \\ F(A \times B) & \xrightarrow{g} & B \end{array}$$

Definición 1.25 (F-coálgebra correcursiva)**.** *Sea $\Sigma_D : C \to C$ el funtor definido como $\Sigma_D C := C + D$. Una F-coálgebra $\langle A, f\rangle$ es correcursiva si para toda $F(\Sigma_A))$-coálgebra $\langle B, g\rangle$ existe un morfismo $h : B \to A$ tal que:*

$$\begin{array}{ccc} B & \xrightarrow{g} & F(A + B) \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle F[\mathsf{Id}, h]} \\ A & \xrightarrow{f} & FA \end{array}$$

Finalmente se nombrarán algunas definiciones necesarias para el trabajo posterior, éstas son las diálgebras. Serán útiles en el momento de justificar el sistema de tipos del último capítulo, más adelante se explica el porqué de su uso y la ventaja de trabajar con ellas.

Definición 1.26 (Diálgebra)**.** *Sean $F, G : C \to D$ dos funtores covariantes entre las categorías C, D. Una F, G-diálgebra es un par $\langle A, f\rangle$ donde A es un objeto en C y $f : FA \to GA$ es un morfismo en D.*

El término covariante se refiere a la invarianza de forma, es decir de la permanencia sin cambios; esto permite relacionarse con la monotonía de un funtor y muchas veces se les toma como sinónimos.

Definición 1.27 (Morfismo entre diálgebras)**.** *Un morfismo entre dos $F - G$-diálgebras $\langle A, f\rangle$ y $\langle B, g\rangle$ es un morfismo en C, $h : A \to B$ tal que el siguiente diagrama conmuta.*

$$\begin{array}{ccc} FA & \xrightarrow{f} & GA \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle Gh} \\ FB & \xrightarrow{g} & GB \end{array}$$

Si sustituimos alguno de los dos funtores por el de identidad, I, tenemos por un lado al sustituir el segundo funtor, una F, I-diálgebra que en realidad es una F-álgebra y si sustituimos el primero tendremos una I, F-diálgebra que es una F-coálgebra.
Se considerarán las diálgebras cuyos funtores $F, G : C \to C^n$ sean de la forma $F \equiv \langle F_1, \ldots, F_n\rangle$ y $G \equiv \langle I, \ldots I\rangle$ con $F_i : C \to C$.

Definición 1.28 (F, G-diálgebra inicial)**.** *En caso de existir la F, G-diálgebra inicial, será denotada por $\langle \mu(F_1, \ldots, F_n), \mathsf{in}_n \rangle$. En tal caso se cumple la siguiente propiedad universal:*

$$\begin{array}{ccc} \langle F_1 U, \ldots, F_n U \rangle & \xrightarrow{\mathsf{in}_n} & \langle U, \ldots, U \rangle \\ \Big\downarrow{\scriptstyle \langle F_1 h, \ldots, F_n h \rangle} & & \Big\downarrow{\scriptstyle \langle h, \ldots, h \rangle} \\ \langle F_i B, \ldots, F_n B \rangle & \xrightarrow{s} & \langle B, \ldots, B \rangle \end{array}$$

Donde $U := \mu(F_1, \ldots, F_n)$ y el morfismo $h : U \to B$ es el único que permite que el diagrama conmute, es decir:

$$h \circ \mathsf{in}_n = s \circ \langle F_1 h, \ldots, F_n h \rangle$$

.

Definición 1.29 (G,F-diálgebra final)**.** *En caso de existir, la G, F-diálgebra final será denotada por $\langle \nu(F_1, \ldots, F_n), \mathsf{out}_n \rangle$. En tal caso se cumple la siguiente propiedad universal:*

$$\begin{array}{ccc} \langle B, \ldots, B \rangle & \xrightarrow{s} & \langle F_1 B, \ldots, F_n B \rangle \\ \Big\downarrow{\scriptstyle \langle h, \ldots, h \rangle} & & \Big\downarrow{\scriptstyle \langle F_1 h, \ldots, F_n h \rangle} \\ \langle V, \ldots, V \rangle & \xrightarrow{\mathsf{out}_n} & \langle F_1 V, \ldots, F_n V \rangle \end{array}$$

Donde $V := \nu(F_1, \ldots, F_n)$ y la función $h : B \to V$ es la única tal que hace que el diagrama conmute, es decir:

$$\mathsf{out}_n \circ \langle h, \ldots, h \rangle = \langle F_1 h, \ldots, F_n h \rangle \circ s$$

.

1.4.2. (Co)Iteración y (Co)Recursión

Los principios que se enuncian a continuación ya han sido discutidos, pero son presentados en el contexto de esta sección y se derivan a partir de las propiedades universales tanto de las álgebras iniciales y coálgebras finales así como de las álgebras recursivas y las coálgebras correcursivas, es decir la conmutatividad de los diagramas mencionados en las definiciones anteriores.

Teorema 1.2 (Principio de Iteración y de Coiteración)**.** *Categóricamente, el principio de iteración está dado por la propiedad universal de la definición 1.20:*

$$\mathsf{It}_s \circ \mathsf{in}_F \;=\; s \circ F(\mathsf{It}_s)$$

Y el principio de coiteración mediante la propiedad universal de la definición 1.23:

$$\mathsf{out}_F \circ \mathsf{CoIt}_s \;=\; F(\mathsf{CoIt}_s)$$

En ambos se tiene a la función f como el morfismo entre las álgebras o coálgebras, ésta función a definir mediante iteración es llamada It_s y para la coiteración la función es llamada CoIt_s. La función s es la función de paso que hemos definido desde secciones anteriores, esta función de paso es especial para cada función iterativa o coiterativa es por eso que se indica como subíndice de cada una de ellas. Los constructores están codificados en la estructura in_F es decir en el morfismo del álgebra inicial, $\mathsf{in}_F = [c_i, c_j]$. De la misma manera los destructores de un conjunto coinductivo pertenecen a la estructura $\mathsf{out}_F = \langle d_k \rangle$. Del ejemplo de los números naturales, en la definición 1.1 y ecuación (1.2), $Nat := \mu X{,}1 + X$, podemos notar que el álgebra inicial que define a los números naturales es: $(Nat, [0, succ])$, el álgebra inicial del funtor $F(X) = 1 + X$. Y del ejemplo de los streams, de la definición 1.7 y ecuación (1.3) podemos definir a los streams de números naturales como la coálgebra final $(Stream(Nat), \langle head, tail \rangle)$, la coálgebra final del funtor $F(X) = Nat \times X$ quedando como $Stream(Nat) := \nu X.Nat \times X$.

En las dos primeras secciones se habló de unas funciones que no estaban del todo definidas dentro de los principios de iteración y coiteración, solamente se dijo que tenían algunos términos involucrados. En el principio de iteración 1.5, la función $(\dots f(x) \dots)$ corresponde a la aplicación del funtor a la función a definir: $F(\mathsf{It}_s)$; de la misma manera al hacer referencia del principio de coiteración con la fórmula (1.1) corresponde en el diagrama anterior a $F(\mathsf{CoIt}_s)$.

Los principios de recursión y correcursión primitivas se derivan de las definiciones de álgebras y coálgebras recursivas, 1.24 y 1.25:

Definición 1.30 (Principio de Recursión y de Correcursión primitiva)**.** *El principio de recursión está dado por la conmutatividad del siguiente diagrama:*

$$\begin{array}{ccc} F\mu F & \xrightarrow{\mathsf{in}_F} & \mu F \\ {\scriptstyle F\langle \mathsf{Id}, \mathsf{Rec}_s \rangle}\downarrow & & \downarrow{\scriptstyle \mathsf{Rec}_s} \\ F(\mu F \times B) & \xrightarrow{s} & B \end{array}$$

Y el principio de correcursión mediante la conmutatividad del diagrama:

$$\begin{array}{ccc} B & \xrightarrow{s} & F(\nu F + B) \\ {\scriptstyle \mathsf{CoRec}_s}\downarrow & & \downarrow{\scriptstyle F[\mathsf{Id}, \mathsf{CoRec}_s]} \\ \nu F & \xrightarrow{\mathsf{out}_F} & F\nu F \end{array}$$

La recursión y la correcursión primitivas definen funciones mediante las siguientes igualdades:

$$\begin{aligned} \mathsf{Rec}_s \circ \mathsf{in}_F &= s \circ F(\langle \mathsf{Id}, \mathsf{Rec}_s \rangle) \\ \mathsf{out}_F \circ \mathsf{CoRec}_s &= F([\mathsf{Id}, \mathsf{CoRec}_s]) \circ s \end{aligned}$$

De las últimas definiciones, las de diálgebras, enunciaremos los principios de (co)iteración y (co)recursión derivados de las definiciones 1.28 y 1.29. En estas definiciones, los diagramas pueden ser separados en los siguientes para $1 \leq i \leq n$:

$$\begin{array}{ccc} F_i(\mu(F_1,\ldots,F_n)) & \xrightarrow{\mathsf{in}_{n,i}} & \mu(F_1,\ldots,F_n) \\ \downarrow{\scriptstyle F_i(\mathsf{It}^n_s)} & & \downarrow{\scriptstyle \mathsf{It}^n_s} \\ F_iB & \xrightarrow{s_i} & B \end{array} \qquad \begin{array}{ccc} B & \xrightarrow{s_i} & F_iB \\ \downarrow{\scriptstyle \mathsf{CoIt}^n_s} & & \downarrow{\scriptstyle F_i(\mathsf{CoIt}^n_s)} \\ \nu(F_1,\ldots,F_n) & \xrightarrow{\mathsf{out}_{n,i}} & F_i(\nu(F_1,\ldots,F_n)) \end{array}$$

Entonces los principios se pueden enunciar de la siguiente manera:

Definición 1.31 (Principio de (Co)Iteración)**.** *Los principios de (co)iteración están dados por la conmutatividad de los diagramas anteriores, es decir:*

$$\mathsf{It}^n_s \circ \mathsf{in}_{n,i} = s_i \circ F_i(\mathsf{It}^n_s) \qquad \mathsf{out}_{n,i} \circ \mathsf{CoIt}^n_s = F_i(\mathsf{CoIt}^n_s) \circ s_i$$

Para los principios de (co)recursión tenemos los siguientes diagramas:

$$\begin{array}{ccc} F_i(\mu(F_1,\ldots,F_n)) & \xrightarrow{\mathsf{in}_{n,i}} & \mu(F_1,\ldots,F_n) \\ \downarrow{\scriptstyle F_i(\langle \mathsf{Id}, \mathsf{Rec}^n_s \rangle)} & & \downarrow{\scriptstyle \mathsf{Rec}^n_s} \\ F_i(\mu(F_1,\ldots,F_n) \times B) & \xrightarrow{s_i} & B \end{array} \qquad \begin{array}{ccc} B & \xrightarrow{s_i} & F_i(\nu(F_1,\ldots F_n) + B) \\ \downarrow{\scriptstyle \mathsf{CoRec}^n_s} & & \downarrow{\scriptstyle F_i([\mathsf{Id}, \mathsf{CoRec}^n_s])} \\ \nu(F_1,\ldots,F_n) & \xrightarrow{\mathsf{out}_{n,i}} & F_i(\nu(F_1,\ldots,F_n)) \end{array}$$

Definición 1.32 (Principio de (Co)Recursión)**.** *Los principios de (co)recursión están dados por la conmutatividad de los diagramas anteriores a decir:*

$$\mathsf{Rec}^n_s \circ \mathsf{in}_{n,i} = s_i \circ F_i(\langle \mathsf{Id}, \mathsf{Rec}^n_s \rangle) \qquad \mathsf{out}_{n,i} \circ \mathsf{CoRec}^n_s = F_i([\mathsf{Id}, \mathsf{CoRec}^n_s]) \circ s_i$$

Resumiendo podemos decir que las álgebras iniciales formalizan lo que se vio en el reino inductivo y las coálgebras lo correspondiente al reino coinductivo. El principio de iteración permite definir una función mediante iteración, es decir aplicar la función a definir a los objetos básicos y a los constructores. El principio de coinducción permite definir funciones mediante la aplicación de los destructores a la imagen de la función.
Recordemos siempre que bajo estas definiciones se encuentran las funciones de paso específicas para cada una de ellas.

1.5. Cálculo Lambda

Es un sistema formal que involucra todo lo referente a las funciones, definiciones, aplicaciones, recursión, etc., además se ocupa de formalizar un prototipo de un lenguaje de programación. Es bien conocido entre las personas que se dedican al estudio de los lenguajes de programación, en particular a la programación funcional; por lo que me lleva a incluirlo en este trabajo, será utilizado para pasar de la teoría de categorías a la programación funcional. En esta sección solamente se dará una breve introducción a la historia de éste y a los conceptos básicos, así como una descripción del sistema F que es la base para el desarrollo del sistema de tipos presentado en el último capítulo.

1.5.1. Historia y el Cálculo lambda puro

Alrededor de 1930, con base en el cálculo lambda y la lógica combinatoria se pretendía desarrollar una teoría general para las funciones y extenderla con lógica para tener una base para las matemáticas. Schönfinkel y Curry desarrollaron la lógica combinatoria y Alonzo Church se dedicó a desarrollar el cálculo lambda en un intento para proponer un sistema formal completo que fundamentara las matemáticas, pero por desgracia no pudo lograr ese fin, debido a que fueron encontradas algunas inconsistencias por Stephen Kleene y Rosser. En las décadas siguientes se siguió trabajando con el cálculo lambda y se encontraron otros usos para esta teoría.

Church encontró que usando el cálculo lambda a manera de funciones podía desarrollar la formalización de un "cómputo efectivo" mediante el concepto de una definición lambda. Kleene, partiendo de esta formalización, pudo demostrar que la definición lambda es equivalente a la recursividad de Gödel y Herbrand y finalmente Church demostró que la recursividad es una formalización de un cómputo efectivo.
Al mismo tiempo Turing al dar un análisis de la computabilidad de una máquina, demostró que el ser Turing-computable es equivalente a una definición lambda.

Si describimos el cálculo lambda podemos decir que es una teoría de funciones, las funciones son reglas, es decir, el uso de funciones como el proceso para ir de un argumento a un valor. Al ser las funciones el objeto de estudio dentro del cálculo lambda, es importante hacer la observación que las funciones son objetos de primera clase, es decir pueden ser al mismo tiempo funciones, argumentos de funciones y hasta resultado de alguna aplicación.
El alma del cálculo lambda son las expresiones lambda que permiten definir de una forma distinta a la conocida una función, estas expresiones siempre tienen un solo argumento seguido de la acción a la que se someterá dicho argumento.
Veamos la definición de una expresión de este formalismo:

Definición 1.33 (Expresión del Cálculo lambda)**.** *Una expresión del cálculo lambda, o un término lambda* $\langle \lambda - term \rangle$ *está determinado por:*

$$\begin{array}{ll} \langle \lambda - term \rangle & ::= \langle var \rangle \mid \lambda \langle var \rangle \,.\, \langle \lambda - term \rangle \mid \langle \lambda - term \rangle \langle \lambda - term \rangle. \\ \langle var \rangle & ::= x \mid y \mid z \mid \ \ldots \end{array}$$

Es decir un término lambda, t, puede ser una variable, una abstracción lambda o una aplicación. Al tener una sintaxis tan simple se podría pensar que no es fuerte pero por el contrario tiene una facilidad para describir las funciones computables y es por esto que es considerado un lenguaje ensamblador en el paradigma funcional. La abstracción corresponde a lo que conocemos como función o computacionalmente como un procedimiento en algún lenguaje de programación, recibimos información como entrada y devolvemos una salida. Debido a que cualquier término puede ser visto como una función y un argumento también puede ser una función nos encontramos frente al paradigma de los lenguajes funcionales.

Estudiemos con atención las dos operaciones del cálculo lambda, la abstracción y la aplicación. La aplicación es una operación primitiva, la aplicación rs se traduce como: el término r recibe como argumento al término s. Si hay más de dos términos siempre se asociará a la izquierda, es decir un término rst es equivalente a $(rs)t$.
La abstracción denota la atracción de una variable hacia un término, $\lambda x.t$, es decir $f(x) = t$; el punto dentro de este término ayuda a indicar el alcance de la abstracción que siempre será a la derecha del punto y lo más lejano que sea posible. Por ejemplo el término $\lambda x.xy$ en realidad es $\lambda x.(xy)$. Dijimos que las funciones tienen un sólo argumento así que un término como éste: $\lambda x_1.\lambda x_2.\lambda x_3.\ldots.\lambda x_n.t$ podemos cambiarlo por $\lambda x_1 x_2 x_3 \ldots x_n.t$ teniendo en cuenta que siempre se hablará de funciones con un solo argumento. Por ejemplo la expresión $(\lambda xyz.xz)yz$ es equivalente a $((\lambda x.\lambda y.\lambda z.xz)y)z$.
Al restringir el uso de un sólo argumento para las funciones debemos pensar cómo solucionar el problema que se presenta cuando una función tiene dos o más argumentos, como por ejemplo supongamos que queremos definir la función que suma dos números: $add : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, tenemos como argumento un par de números y la función devuelve un número, la suma del par. En el cálculo lambda sería traducido como $add = \lambda m \lambda n.t$ si tuviéramos definidos los números y el término t sería la operación de sumar ambos números. Al trabajar con funciones de un sólo argumento obliga al cálculo lambda a incorporar un fenómeno llamado curryficación, debido a Haskell Curry. La curryficación se presenta cuando se transforma una función que tiene varios argumentos en una secuencia o serie de funciones con un solo argumento, es decir los argumentos de las funciones de esta secuencia son funciones, las cuales reciben como argumento los argumentos que faltan. Por lo tanto nuestra función de suma se trasformaría en $add : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$, recibe primero un número y devuelve una función que espera como argumento el segundo número,el resultado final es la suma de los dos argumentos de las funciones. Así podemos extender la operación de suma a tantos argumentos como queramos: $add : \mathbb{N} \to (\mathbb{N} \to (\mathbb{N} \to \ldots) \ldots) \to \mathbb{N})$. Podemos generalizar de una manera informal este procedimiento al cambiar los símbolos $\times$ por $\to$ en el dominio de la función a curryficar, transformando todo a funciones unarias.

Veamos unos ejemplos para aclarar la definición, si queremos sumarle dos a un número tenemos la siguiente función: $f(x) = x + 2$, traducida al cálculo lambda tenemos la expresión $\lambda x.x + 2$. Observemos que siempre hemos trabajado con funciones con nombre, podemos decir que la función f es la que suma dos a un número, pero al tener esta función en el cálculo lambda aparece sin nombre, no hay forma de anotar su nombre y de ahí que no podemos distinguirla de las demás, son funciones anónimas.

Tomando este mismo ejemplo, para aplicar una expresión a un argumento sólo es necesario colocar la expresión antes del argumento: $(\lambda x.x + 2)\ 1 = 3$.

En las abstracciones tenemos una noción importante como la que es utilizada en lógica de predicados, cuando se tienen cuantificadores, ligados a éstos se encuentran las variables. De la misma manera se ligan las variables al operador λ:

Definición 1.34 (Variables libres y ligadas)**.** *Dado un término, $t = \lambda x.r$, la variable x es libre en el término si x no está en el alcance de $\lambda x.r$. De lo contrario está ligada, es decir, x aparece en el término r.*
El conjunto de variables libres de un término t se define de la siguiente manera:

$$vl(t) := \begin{cases} vl(x) = \{x\} \\ vl(\lambda x.t) = vl(t) - \{x\} \\ vl(rs) = vl(r) \bigcup vl(s) \end{cases}$$

Así mismo un término que no contiene variables libres es llamado término cerrado o combinador.

El alcance de una variable en un término permite determinar hasta qué punto la variable en cuestión se ocupa, por ejemplo en el término $(\lambda x.x(\lambda x.x))x)$ la primer x afecta a la inmediata, el siguiente término se comporta de la misma manera y la última es libre; ese término se puede reescribir como $(\lambda x.x(\lambda y.y))z)$. Otros ejemplos de la definición anterior son los siguientes, en el término $(\lambda xy.xyz)(\lambda z.zy)w$ las variables libres son: la primer z, la segunda y y por último w y no es un combinador, mientras que el término $\lambda xyz.xz(yz)$ es un término cerrado.

Cuando queremos comparar dos términos, es decir si ambos codifican una misma función podemos hacerlo si al compararlos solamente difieren en el nombre de las variables ligadas. Podemos renombrar variables de la manera conocida en lógica, entonces si tenemos dos términos r y s decimos que son α equivalentes, $r \equiv_\alpha s$, si r resulta de s al final de varios cambios de variables; trabajaremos con esta equivalencia y consideraremos a $r \equiv_\alpha s$ como una forma de decir que r y s son idénticos.

De la misma manera como el renombre de variables es un concepto conocido el de sustitución también lo debe ser:

Definición 1.35 (Sustitución)**.** *Se define la sustitución en términos del cálculo lambda de la siguiente manera:*

- $x[x := r] = r$.
- $y[x := r] = y \ \ si \ \ x \neq y$.
- $(ts)[x := r] = t[x := r]s[x := r]$.
- $(\lambda y.t)[x := r] = \lambda y.t[x := r] \ \ donde \ y \notin \{x\} \cup VL(r)$.

Como observación importante, la restricción usada en la última cláusula de la definición anterior no es tal por el uso de la $\equiv_\alpha$.

Semántica Operacional

Dar un significado a un programa mediante las matemáticas es hablar de la semántica operacional, a partir de la semántica operacional de un lenguaje de programación es posible describir una serie de computos que permiten interpretar a un programa válido, es decir, dar el significado del programa. Dentro del contexto funcional, después de procesar la secuencia obtendremos el valor del programa, para procesarla es necesario hacerlo mediante transiciones entre los términos a reducir, y para ello se dará una regla de inferencia para definir las transiciones válidas.
La reducción β es una regla que permitirá pasar de un término a otro al hacer una sustitución:

$$(\lambda x.r)s \rightarrow_\beta r[x := s]$$

Cuando se tiene un término de la forma del lado izquierdo de la regla de reducción anterior se dice que es un *redex*, es decir una expresión reducible (reducible expression). Cuando un término no tiene un se puede decir que esta en forma normal.

A continuación veremos unos ejemplos de estos términos, éstos los llamaremos programas. Son la representación de booleanos, $True$ y $False$ con algunas funciones, los pares de objetos y por supuesto los conocidos números naturales llamados numerales de Church.

1. Booleanos
   - $\mathsf{true} := \lambda x \lambda y.x$
   - $\mathsf{false} := \lambda x \lambda y.y$
   - $\mathsf{test} := \lambda b \lambda t \lambda e.bte$
   - $\mathsf{not} := \lambda z.z\,\mathsf{false}\,\mathsf{true}$
   - $\mathsf{and} := \lambda x \lambda y.xy\,\mathsf{false}$
2. Pares
   - $\mathsf{pair} := \lambda f \lambda s \lambda b.bfs$
   - $\mathsf{fst} := \lambda p.p\,\mathsf{true}$
   - $\mathsf{snd} := \lambda p.p\,\mathsf{false}$
3. Numerales de Church
   - $\overline{0} := \lambda s \lambda z.z$
   - $\overline{1} := \lambda s \lambda z.sz$
   - $\overline{2} := \lambda s \lambda z.s(sz)$
   - $\overline{3} := \lambda s \lambda z.s(s(sz))$
   - $\overline{n} := \lambda s \lambda z.s(\ldots(sz)\ldots)$
   - $\mathsf{succ} := \lambda n \lambda s \lambda z.s(nsz)$
   - $\mathsf{add} := \lambda m \lambda n \lambda s \lambda z.ms(nsz)$
   - $\mathsf{prod} := \lambda m \lambda n.m(\mathsf{add}\,n)0$

- $\mathsf{iszero} := \lambda m.m(\lambda x.\,\mathsf{false})\,\mathsf{true}$
- $\mathsf{pred} := \lambda m.fst(m\,\mathsf{ss}\,\mathsf{zz})$ donde:
  - $\mathsf{zz} := \mathsf{pair}\,00$
  - $\mathsf{ss} := \lambda p.\,\mathsf{pair}(\mathsf{snd}\,p)(\mathsf{succ}(\mathsf{snd}\,p))$

El número $\overline{n}$ es tener n veces el sucesor del cero z.
Estos programas pueden combinarse, por ejemplo si se quiere resolver la operación $2+1$ al traducirla al cálculo lambda tendremos $add\ \overline{2}\ \overline{1}$. Siguiendo la reducción:

$$(\lambda m\lambda n\lambda s\lambda z.ms(nsz))\overline{2}\ \overline{1} \to \lambda s\lambda z.\overline{2}s(\overline{1}sz) \to \lambda s\lambda z.\overline{2}s(\lambda s\lambda z.sz(sz)) \to$$

$$\lambda s\lambda z.(\lambda s\lambda z.ssz(s(sz))) \to \lambda s\lambda z.sssz = \overline{3}$$

Hasta este momento hemos visto que este cálculo posee propiedades importantes ya que se puede expresar cualquier función computable en un término lambda pero tiene unos pequeños defectos. Para empezar tenemos sucesiones infinitas de reducción esto quiere decir que el cálculo lambda no es terminal por ejemplo si tenemos los términos $\omega := \lambda x.xx$ y $\Omega := \omega\omega$ bajo la reducción β tenemos: $\Omega \to_\beta \Omega$, estos términos no pueden ser evaluados a una forma normal por lo que se les denomina términos divergentes.
Cuando aplicamos la reducción β podríamos pensar en que al sustituir los términos se simplifican pero no es de esa manera, considerar el término $\omega' := \lambda x.xxx$ al tener el término $\omega'\omega'$ se reduce en

$$\omega'\omega' \to_\beta \omega'\omega'\omega \to_\beta (\omega'\omega'\omega')\omega' \to_\beta (\omega'\omega'\omega'\omega')\omega' \to_\beta \dots$$

También puede darse el caso que la reducción β nos lleve a sucesiones infinitas de reducción a pesar de que el término en cuestión esté en forma normal.

Para poder acotar el comportamiento de la reducción β es necesario escoger una de las siguientes estrategias de evaluación:

- Orden Normal: se reduce en primer lugar el redex que se encuentre más a la izquierda y más afuera.
- Llamada por nombre: también llamada *evaluación perezosa* debido a que sigue la misma idea del orden normal pero sin reducir abstracciones.
- Llamada por valor: a ésta se le conoce como *evaluación ansiosa* debido a que la reducción sólo es en los redex cuyo argumento es un valor, es decir una abstracción.

Cuando un término tenga una forma normal, independientemente de la estrategia que se escoja, ¿puede ser que sea única esta forma normal?. En un sentido computacional podemos preguntar si un programa, al terminar ¿tendrá siempre el mismo resultado?. Para responder estas preguntas tenemos el siguiente teorema:

Teorema 1.3 (Teorema de Church-Rosser)**.** *Si $t \to_\beta t'$ y $t \to_\beta t''$ entonces existe r tal que $t' \to_\beta r$ y $t'' \to_\beta r$.*

Corolario 1.1 (Unicidad de las formas normales)**.** *Si $t \to_\beta^* t'$ y $t \to_\beta^* t''$ con t' y t'' formas normales entonces $t' = t''$.*

Representación de funciones recursivas

Como hemos visto en las secciones anteriores, la recursión es una herramienta muy importante para definir funciones que involucran objetos inductivos. Con lo que hemos descrito del cálculo lambda puro sería útil poder definir términos que definan una función recursiva, para ésto se utiliza el combinador divergente $\omega = (\lambda x.xx)(\lambda x.xx)$; cada vez que se reduce el redex de este término obtenemos a él mismo. Este combinador es una generalización de un operador de punto fijo, es una función de orden superior que calcula el punto fijo de otra función

$$fix = \lambda f.\ (\lambda x.\ f\,x\,x)\ (\lambda x.\ f\,x\,x)$$

Es decir, la expresión $fix\ g$ diverge para cualquier g Éste término tiene un carácter repetitivo, lo usamos para definir una función recursiva h tal que ésta vuelva a llamarse así misma dentro de su cuerpo $h = \ldots h \ldots$. Obsérvese que este término no es normalizable, pues no termina de evaluarse.
Por ejemplo si queremos definir la función factorial:

```
If n = 0 then  1
 else  n * (If n-1 = 0 then 1 else
             (n-1) * (If n-2 = 0 then 1 else
                       (n-2) * ...
                     )
           )
```

Para definirla en el cálculo lambda con ayuda del operador fix debemos definir la función $g = \lambda f.(...f...)$ y luego la función $factorial = fix\,g$. Teniendo finalmente a:

$$g = \lambda fct.\lambda n.\ \mathsf{if\ iszero}\ n\ \ \mathsf{then}\ \overline{1}\ \ \mathsf{else}(\mathsf{prod}\,n(fct(\mathsf{pred}\,n)))$$

Aquí fct es sólo un nombre para una variable utilizado para facilitar la definición.

1.5.2. Cálculo lambda tipificado

Lo que hemos descrito del cálculo lambda ha sido la parte llamada pura o no tipificada, sólo tenemos una forma nueva de describir funciones mediante abstracciones λ. En esta parte veremos el refinamiento del cálculo lambda puro al agregar *tipos*, éstos definen la pertenencia de un término o una expresión a una clase[1] de objetos que cumplen cierta característica. En ciencias de la computación se le asigna un tipo a un dato, es decir se determina la pertenencia de una variable, una constante o una función a un dominio. Un tipo de dato describe objetos básicos, su representación, estructura y cómo van a ser entendidos dentro de un algoritmo o en una computadora.
En el cálculo lambda tipificado se consideran tanto el dominio como el codominio de las funciones para poder trabajar con ellas.

Generalmente cuando realizamos una función no nos detenemos a pensar en los argumentos que recibe, por ejemplo si deseamos evaluar la función *and* en los booleanos

[1]La cual no necesariamente es un conjunto

esperamos que los argumentos sean booleanos, es decir $true$ o $false$, por que si fueran de cualquier otro tipo el resultado pudiera no ser un booleano como estamos acostumbrados a devolver o simplemente no estaría definido. Es por esto que el cálculo lambda tipificado permite eliminar errores de evaluación, aseguramos que los argumentos y las evaluaciones de los términos lambda sean correctos y no haya inconsistencias.

Dentro de la tipificación existen tres estilos, cualquier término está denotado por t y cualquier tipo con letras mayúsculas:

1. Tipificación explícita o *à la Church*
   Este estilo de tipificación expone los tipos en cada parte de los términos lambda, es decir están anotados:

$$t ::= x : T \mid (\lambda x : \mathsf{T} . t : \mathsf{S}) \ : \ \mathsf{T} \to \mathsf{S} \mid (t : \mathsf{T} \to \mathsf{S})(t : \mathsf{T}) : \mathsf{S}$$

2. Tipificación parcial o *à la Church*
   Esta tipificación sólo anota a las variables de las abtracciones λ:

$$t ::= x \mid \lambda x : \mathsf{T} . t \mid tt$$

3. Tipificación implícita o *à la Curry*
   Aquí no hay tipificación, el tipo puede ser deducido a partir de los tipos de las variables dadas, los cuales se guardan en un contexto:

$$t ::= x \mid \lambda x.t \mid tt$$

Un ejemplo de un sistema del cálculo lambda tipificado parcialmente es el simple denotado por $\lambda^{\to}$ en el que aparecen tipos básicos predefinidos y el tipo función $\to$:

$$\begin{aligned} \mathsf{T} &::= C \mid \mathsf{T} \to \mathsf{T} \\ t &::= x \mid \lambda x : \mathsf{T} . t \mid tt \end{aligned}$$

Algunos tipos básicos, son por ejemplo los booleanos $\mathsf{bool} = \{True, False\}$ y los naturales $\mathsf{nat} = \{0, 1, \dots\}$, etc. Éstos son los tipos C y las constantes son los elementos de los tipos básicos.

En este sistema también es importante agregar valores, los cuales son términos que representan posibles resultados de evaluaciones. Y para poder conocer los tipos de las variables existentes es necesario agregar contextos, éstos contienen las declaraciones de variables $\Gamma = \{x_1 : \mathsf{T}_1, x_2 : \mathsf{T}_2, \dots x_n : \mathsf{T}_n\}$ y se definen como:

$$\Gamma ::= \emptyset \mid \Gamma, x : \mathsf{T}$$

Donde se supone que en Γ, la variable x no ha sido definida.

Cuando es necesario referirse al tipo de un término o a la deducción del tipo de un término, dado un contexto, usamos la notación $\Gamma \vdash \ t : \mathsf{S}$ que se lee como: el término t recibe o habita el tipo S bajo el contexto Γ. A éstas deducciones se les conoce como juicios de tipo.

Por ejemplo si al sistema $\lambda^{\rightarrow}$ se le agrega como tipo básico los booleanos los juicios de tipos serían los siguientes:

$$\Gamma \vdash \mathsf{true} : \mathsf{bool} \quad \Gamma \vdash \mathsf{false} : \mathsf{bool} \quad \Gamma, x : \mathsf{T} \vdash x : \mathsf{T}$$

$$\frac{\Gamma \vdash t : \mathsf{T} \to \mathsf{S} \quad \Gamma \vdash s : \mathsf{S}}{\Gamma \vdash ts \to \mathsf{S}} \qquad \frac{\Gamma, x : \mathsf{T} \vdash t : \mathsf{S}}{\Gamma \vdash \lambda x : \mathsf{T}.t \ : \mathsf{T} \to \mathsf{S}}$$

Este sistema tiene propiedades que es importante nombrar, no serán demostradas pero se puede consultar [2] para más detalles:

Lema 1.3 (Inversión)**.** *Se cumple lo siguiente:*

- *Si* $\Gamma \vdash \mathsf{true} : \mathsf{T}$ *entonces* $\mathsf{T} = bool$
- *Si* $\Gamma \vdash \mathsf{false} : \mathsf{T}$ *entonces* $\mathsf{T} = bool$
- *Si* $\Gamma \vdash x : \mathsf{T}$ *entonces* $x : \mathsf{T} \in \Gamma$
- *Si* $\Gamma \vdash rs : \mathsf{T}$ *entonces existe un tipo* S *tal que* $\Gamma \vdash r : \mathsf{S} \to \mathsf{T}$ *y* $\Gamma \vdash s : \mathsf{S}$
- *Si* $\Gamma \vdash \lambda x : \mathsf{T}.t \ : \mathsf{R}$ *entonces* $\mathsf{R} = \mathsf{T} \to \mathsf{R}_2$ *para alguna* R_2 *con* $\Gamma, x : \mathsf{T} \vdash t : \mathsf{R}_2$

Proposición 1.1 (Determinismo)**.** *En un contexto dado* Γ *un término* t *recibe a lo más un tipo, es decir, si un término es tipificable en* Γ *su tipo es único. Más aún, la derivación de dicho tipo es única.*

Lema 1.4 (Formas Canónicas)**.** *Sea* v *un valor, es decir un término cerrado:*

- *Si* v *es booleano, es decir* $\vdash v : \mathsf{bool}$*, entonces* $v = \mathsf{true}$ *o* $v = \mathsf{false}$
- *Si* v *es un valor del tipo* $\mathsf{T} \to \mathsf{S}$*, es decir* $\vdash t : \mathsf{T} \to \mathsf{S}$*, entonces* $v = \lambda x : \mathsf{T}.t$

Teorema 1.4 (Progreso)**.** *Si* t *es cerrado y tipificable, es decir* $\vdash t : \mathsf{T}$*, entonces* t *es un valor o* $t \to t'$ *para algún* t'*.*

Lema 1.5 (Preservación de tipos)**.** *Si* $\Gamma \vdash t : \mathsf{T}$ *y* $t \to t'$ *entonces* $\Gamma \vdash t' : \mathsf{T}$

Cabe destacar que en el sistema $\lambda^{\rightarrow}$ se pueden definir únicamente funciones que terminan, a esto se le llama normalización fuerte.

Definición 1.36 (Conjunto de términos fuertemente normalizables)**.** *El conjunto de términos fuertemente normalizables,* fn *se define recursivamente como sigue:*

$$\textit{Si para cada } r' \textit{ tal que } r \to_\beta r' \textit{ y } r' \in fn \textit{ entonces } r \in fn$$

Si $r \in fn$ *decimos que* r *es fuertemente normalizable.*

Este conjunto es el conjunto de términos r tales que no existe una sucesión infinita de β-reducciones a partir de r.

Teorema 1.5 (Normalización para $\lambda^{\rightarrow}$)**.** *Si* $\vdash t : \mathsf{T}$ *entonces* t *es fuertemente normalizable.*

En el momento agregar tipos a un programa y poder tipificarlo aseguramos que los subtérminos también están tipificados, si existe un término que pueda bloquearse, es decir computacionalmente hablando, que cause un error en la ejecución entonces ese término no puede ser tipificado. La tipificación ayuda a controlar los programas pero a su vez un término no bloqueado no siempre asegura una tipificación correcta. Es por esto que los lenguajes de programación traen consigo un sistema de tipos para revisar que cada parte de algún programa esté correctamente tipificada y no ocurran errores a lo largo de su ejecución. Para tener esta seguridad o correctud en el lenguaje son necesarios los teoremas de progreso y preservación de tipos.

1.5.3. Sistema F

Como hemos dicho los tipos son una forma de asegurar que los programas no caigan en errores y que las funciones se evalúen correctamente, pero el uso de los tipos puede llegar a ser molesto.
Por ejemplo si para cada término de un programa debe de anotarse el tipo para poder indicarlo explícitamente, entonces puede volverse difícil leer el código del programa, para solucionar este problema podemos dejar de anotar los tipos, sólo trabajar con las funciones y en el momento que se requiera obtener o verificar el tipo de algún término se puede hacer una inferencia de tipos, podemos escoger entre los diferentes estilos de tipificación. Otro ejemplo es cuando una función es la misma pero se define para cada tipo, por ejemplo la suma de números siempre es la misma pero es distinta si los argumentos están limitados a un tipo en específico, es decir es la misma por que realiza la suma en todos los casos, pero al mismo tiempo distinta por que es definida para diferentes tipos.
Una solución al problema de tener varias funciones que hacen lo mismo pero con diferentes tipos es el polimorfismo. Una función es llamada polimórfica si el tipo de alguno de sus argumentos puede variar cada vez que se usa esa función, es decir si puede ser aplicada y/o evaluada a valores de diferentes tipos. En el ejemplo de la suma para diferentes tipos, la función que suma dos números naturales es la misma función que suma dos números enteros solamente que estamos tratando con diferentes tipos, los naturales y los enteros; es conveniente definir una función que sume cualquier tipo de números y así tener una sóla y no una para cada tipo, ahorramos definiciones y el estar recordando cuál es la función que se debe usar en cada caso.

Un sistema polimórfico importante en el cálculo lambda es el sistema F. Este sistema fue desarrollado separadamente por Jean-Yves Girard y John Reynolds, es una extensión del sistema $\lambda^{\rightarrow}$ y tiene la característica de poder cuantificar universalmente a los tipos, a lo que hemos llamado *polimorfismo* y de ahí que también se le nombre como el cálculo polimórfico de segundo orden.
Hay diversos tipos de polimorfismo y a su vez ellos tienen una clasificación propia, pero el sistema F formaliza el llamado universal paramétrico impredicativo, se parametrizan los valores y se pueden sustituir a los tipos por cualquiera incluyendo tipos polimórficos, la idea principal es agregar la abstracción y la aplicación de tipos.

Existen lenguajes de programación que manejan el polimorfismo como Haskell y ML, pero tienen la desventaja de solamente tener un fragmento; el polimorfismo en el sistema

F tiene el poder suficiente para estudiar aspectos no triviales de ahí que sea de gran importancia dentro de la teoría de lenguajes de programación.

Definición 1.37 (Sistema F)**.** *El sistema* F *se presenta bajo la siguiente sintaxis, tipificado* à la Curry*:*

$$\begin{array}{lll} \Gamma & ::= & \emptyset \mid \Gamma, x : \mathsf{T} \\ \mathsf{T}, \mathsf{R}, \mathsf{S} & ::= & X \mid \mathsf{T} \to \mathsf{S} \mid \forall X.\, \mathsf{T} \mid \mathsf{T} + \mathsf{R} \mid \mathsf{T} \times \mathsf{R} \\ t, r, s & ::= & x \mid \lambda x.t \mid rs \mid \mathsf{inl}\, r \mid \mathsf{inr}\, s \mid \mathsf{case}(r, x.s, y.t) \mid \langle r, s\rangle \mid \mathsf{fst}\, r \mid \mathsf{snd}\, r \end{array}$$

Usualmente los tipos producto y coproducto así como los términos para los pares y copares no son parte del sistema F, pueden definirse al ser agregados como primitivos, lo que se hará por comodidad.

Se presentan las reglas de tipificación extendiendo las que se dieron anteriormente para el sistema $\lambda^{\to}$:

$$\Gamma, x : \mathsf{T} \vdash x : \mathsf{T}$$

$$\frac{\Gamma, x : \mathsf{T} \vdash t : \mathsf{S}}{\Gamma \vdash \lambda x.t \ : \mathsf{T} \to \mathsf{S}} \qquad \frac{\Gamma \vdash t : \mathsf{T} \to \mathsf{S} \quad \Gamma \vdash s : \mathsf{S}}{\Gamma \vdash ts \to \mathsf{S}}$$

$$\frac{\Gamma \vdash t : \mathsf{T}}{\Gamma \vdash t : \forall X.\, \mathsf{T}} \qquad \frac{\Gamma \vdash t : \forall X.\, \mathsf{T}}{\Gamma \vdash t : \mathsf{T}[X := \mathsf{S}]}$$

$$\frac{\Gamma \vdash t : \mathsf{T}}{\Gamma \vdash \mathsf{inl}\, t : \mathsf{T} + \mathsf{R}} \qquad \frac{\Gamma \vdash t : \mathsf{R}}{\Gamma \vdash \mathsf{inl}\, t : \mathsf{T} + \mathsf{R}}$$

$$\frac{\Gamma \vdash r : \mathsf{T} + \mathsf{R} \quad \Gamma, x : \mathsf{T} \vdash s : \mathsf{S} \quad \Gamma, y : \mathsf{R} \vdash t : \mathsf{S}}{\Gamma \vdash \mathsf{case}(r, x.s, y.t) : \mathsf{S}}$$

$$\frac{\Gamma \vdash r : \mathsf{R} \quad \Gamma \vdash s : \mathsf{S}}{\Gamma \vdash \langle r, s\rangle : \mathsf{R} \times \mathsf{S}} \qquad \frac{\Gamma \vdash t : \mathsf{R} \times \mathsf{S}}{\Gamma \vdash \mathsf{fst}\, t : \mathsf{R}} \qquad \frac{\Gamma \vdash t : \mathsf{R} \times \mathsf{S}}{\Gamma \vdash \mathsf{snd}\, t : \mathsf{S}}$$

La semántica operacional está dada por las reglas de reducción basadas en la reducción β, definida como la cerradura de las siguientes:

$$\begin{array}{rcl} (\lambda x.t)s & \mapsto_\beta & t[x := s] \\ \mathsf{case}(\mathsf{inl}\, t, x.r, y.s) & \mapsto_\beta & r[x := t] \\ \mathsf{case}(\mathsf{inr}\, t, x.r, y.s) & \mapsto_\beta & s[x := t] \\ \mathsf{fst}\langle r, s\rangle & \mapsto_\beta & r \\ \mathsf{snd}\langle r, s\rangle & \mapsto_\beta & s \end{array}$$

Tipos recursivos, Iteración y Recursión

Se presentan algunos ejemplos de tipos de datos, éstos son derivados del poder expresivo que tiene el sistema F, algunos fueron vistos en secciones pasadas:

- El tipo unitario
  Éste tipo está dado por el término

  $$Unit := \forall X.X \to X$$

  Y cuyo único elemento o habitante es la identidad polimórfica $\lambda x.x$ denotada $\star$.

- Productos
  Este tipo de dato está agregado en la sintaxis del lenguaje y también puede definirse de la siguiente manera:

  $$\mathsf{T} \times \mathsf{S} := \forall X.(\mathsf{T} \to \mathsf{S} \to X) \to X$$

  Lógicamente, esta definición quiere decir que el producto es el conjunto más pequeño cerrado bajo el par y si un objeto pertence al conjunto cerrado bajo el par entonces pertenece al producto es decir:

  $$y \in \mathsf{T} \times \mathsf{S} \Leftrightarrow \forall X.\forall x, z(\mathsf{T}\, x \to \mathsf{S}\, z \to X\langle x, z\rangle) \to Xy$$

  Los términos vistos en la parte del cálculo lambda no tipificado se convierten en los siguientes, del lado derecho se indica el tipo del término:

  $$\begin{array}{rcll} \mathsf{pair} & := & \lambda f.\lambda s.\lambda p.pfs & : \mathsf{T} \to \mathsf{S} \to \mathsf{T} \times \mathsf{S} \\ \mathsf{fst} & := & \lambda p.p(\lambda x.\lambda y.x) & : \mathsf{T} \times \mathsf{S} \to \mathsf{T} \\ \mathsf{snd} & := & \lambda p.p(\lambda x.\lambda y.y) & : \mathsf{T} \times \mathsf{S} \to \mathsf{S} \end{array}$$

- Sumas
  Igualmente la suma ésta agregada en los tipos pero puede definirse de la siguiente manera:

  $$\mathsf{T} + \mathsf{S} := \forall Z.(\mathsf{T} \to Z) \to (\mathsf{S} \to Z) \to Z$$

  Los términos vistos anteriormente se transforman en:

  $$\begin{array}{rcl} \mathsf{inl} & := & \lambda x.\lambda y.\lambda u.yx \\ \mathsf{inr} & := & \lambda x\lambda y.\lambda u.ux \\ \mathsf{case} & := & \lambda x.\lambda f.\lambda f.\lambda g.xfg \end{array}$$

- El tipo vacío
  Para este tipo no debe existir un término cerrado que lo habite:

  $$\mathsf{void} := \forall X.X$$

- Booleanos
  Éste tipo básico es definido mediante:

  $$\mathsf{bool} := \forall X.X \to X \to X$$

  Los términos se convierten en:

$$\begin{array}{rcl} \mathsf{true} & := & \lambda y.\lambda z.y \\ \mathsf{false} & := & \lambda y.\lambda z.z \\ \mathsf{test} & := & \lambda x.\lambda y.\lambda z.xyz \end{array}$$

- Números Naturales
  Como recordamos de las subsecciones del Reino Inductivo y de Puntos Fijos, los números naturales son un conjunto cerrado bajo el cero y el sucesor:

$$y \in \mathsf{nat} \Leftrightarrow \forall X.(\forall z.Xz \to X\,\mathsf{succ}\,z) \to X0 \to Xy$$

$$\mathsf{nat} := \forall X.(X \to X) \to X \to X$$

  Los numerales de Church se transforman en:

$$\begin{array}{rcl} 0 & := & \lambda s.\lambda z.z \\ \mathsf{succ} & := & \lambda n.\lambda s.\lambda z.s(nsz) \\ \overline{n} & := & \lambda s.\lambda z.s^n(z) \end{array}$$

- Listas infinitas o Streams
  Los streams de objetos del tipo T están definidos mediante:

$$\mathsf{stream}(\mathsf{T}) := \forall X(\forall Z.(Z \to \mathsf{T}) \to (Z \to Z) \to Z \to X) \to X$$

  con destructores $\mathsf{head}, \mathsf{tail}$ definidos como:

$$\begin{array}{rcl} \mathsf{head} & := & \lambda s.s(\lambda h\lambda t\lambda x.hx) \\ \mathsf{tail} & := & \lambda s.s(\lambda h\lambda t\lambda x.\mathsf{build}\,h\,t(tx)) \end{array}$$

  donde $\mathsf{build} := \lambda h.\lambda t.\lambda x.\lambda f.(fhtx)$.

Finalmente se describe la forma en que puede codificarse la iteración y la recursión para naturales en el sistema F.
La iteración se lleva a cabo mediante el uso del siguiente término, se realiza mediante los argumentos n que es un número natural sobre el que se va a iterar, el objeto $z : \mathsf{T}$ que devuelve en caso de que $n = 0$ y una función de paso $f : \mathsf{T} \to \mathsf{T}$ la cual se va a iterar:

$$\mathrm{It}(n, z, f) := nfz$$

En particular tenemos: $\mathrm{It}(\overline{0}, z, f)) \to z$ e $\mathrm{It}(\mathsf{succ}\ \overline{n}, z, f) \to f(\mathrm{It}(\overline{n}, z, f))$.

La recursión es una forma más fuerte para definir funciones y dado que hemos agregado los productos definiremos el operador de recursión de la siguiente manera, recibe como argumentos a un número natural n que será sobre el que se realice la recursión, un objeto $z : \mathsf{T}$ que devuelve en caso de que $n = 0$ y una función de paso $f : \mathsf{nat} \to \mathsf{T} \to \mathsf{T}$ que es el paso recursivo:

$$\mathrm{Rec}(n, z, f) := \mathrm{snd}(\mathrm{It}(n, \langle\overline{0}, z\rangle, g_f))$$

Donde $g_f : \mathsf{nat} \times \mathsf{T} \to \mathsf{nat} \times \mathsf{T}$, $g := \lambda x.\langle \mathrm{succ}(\mathrm{fst}x), f(\mathrm{fst}x)(\mathrm{snd}x)\rangle$.

Este sistema también tiene propiedades como las del cálculo lambda simple, el sistema F es un lenguaje seguro y terminal debido a los siguientes teoremas:

Teorema 1.6 (Progreso). *Si t es cerrado y tipificable, es decir $\vdash t : \mathsf{T}$, entonces t es un valor o $t \to t'$ para algún t'.*

Lema 1.6 (Preservación de tipos). *Si $\Gamma \vdash t : \mathsf{T}$ y $t \to t'$ entonces $\Gamma \vdash t' : \mathsf{T}$*

Teorema 1.7 (Normalización fuerte para F). *Si $\vdash t : \mathsf{T}$ entonces t es fuertemente normalizable.*

Hasta este momento hemos formalizado los conocimientos necesarios para este trabajo; las categorías y el cálculo lambda son herramientas muy poderosas que permiten el uso de objetos (co)inductivos de manera que se puede pensar en programar con ellas. Además se han relacionado los ejemplos más usados para la explicación de la inducción y la recursión con las definiciones y la dualización de ellos mismos en los principios de coinducción y correcursión, respectivamente. Con este capítulo será más fácil continuar con la meta del trabajo, llegar a la programación funcional manteniendo el uso de los ejemplos para la comprensión y la dualización de los términos conocidos que permitan entender la coinducción.

Capítulo 2

Tipos de Datos Categóricos

Al momento de diseñar un programa, además de definir la estructura que tendrá, también es importante pensar en los datos que se manejarán a lo largo de él, tanto en las variables como en los datos que se puedan tener como información. Al analizarlos podremos saber qué tipo de dato son, es decir podremos identificar caracteres, cadenas, enteros o flotantes, un tipo primitivo o uno más complejo como un árbol o una lista de objetos incluso un tipo de dato abstracto.

A lo largo de las secciones del capítulo anterior se ha hablado de objetos inductivos o coinductivos y se han estudiado sus diferentes formas de representación de acuerdo a las teorías expuestas, computacionalmente se han definido a esos objetos como tipos de datos. Podemos decir que un tipo de dato inductivo requiere de elementos básicos y las funciones constructoras para su definición y que un tipo de dato coinductivo consta de funciones destructoras por definición.
Usualmente se hace referencia a los tipos de datos inductivos o más bien son los usados con mayor frecuencia en la práctica, son pocas las veces que se trabaja con los coinductivos, es común generalizar el concepto de tipo de dato sin hacer distinción real del tipo de dato en cuestión, ambos, los inductivos y los coinductivos, son solamente nombrados tipos de datos sin hacer la aclaración correspondiente. La aclaración del tipo de dato, ya sea inductivo o coinductivo, debe ser hecha en cada momento para diferenciarlos y así contextualizar las referencias que se les hacen, en lo que resta de este trabajo se nombrarán por el tipo de dato que son.

En este capítulo se dan ejemplos de tipos de datos inductivos y coinductivos, como por ejemplo los números naturales, listas con elementos sobre un conjunto dado, diferentes tipos de árboles, los números conaturales, listas infinitas, etc, todos éstos tipos de datos son definidos categóricamente con el uso de funtores, álgebras iniciales o coálgebras terminales según sea el caso del tipo inductivo o coinductivo. Las funciones como los constructores o destructores así como las de paso son términos escritos en cálculo lambda para facilitar su comprensión y manejo además de que es una forma de expresar a las funciones como programas.

En estos ejemplos sólo se trabaja con F-álgebras o F-coálgebras, el uso de funtores dentro de la definición de objetos inductivos y coinductivos es importante ya que éstos

son una forma de ver las descripciones o cláusulas necesarias para definir objetos que pertenecen a un conjunto inductivo o las que hacen que se cumpla que un conjunto sea coinductivo; cada vez que se hace referencia al tipo 1 o al conjunto sobre el que se están definiendo las partes de los objetos, generalmente llamado A, se están describiendo a los objetos básicos de cada tipo de dato como el cero, la lista vacía, etc., o a los objetos que resultan después de haber aplicado algún destructor a un objeto coinductivo.

El trabajo del functor, ya sea en una función que parte de objetos inductivos o en la función que tiene como codominio a objetos coinductivos, es principalmente definir el dominio o el codominio de la función según sea el caso de los objetos a tratar, es decir, propiamente definir a los objetos inductivos o coinductivos. Además, el funtor permite definir el dominio de la función de paso para el caso inductivo o en el caso inductivo define la función que recibe como argumento el resultado de la función de paso aplicada a un objeto del dominio.

Los ejemplos de funciones sobre tipos inductivos o coinductivos están basados en el principio de iteración y coiteración respectivamente. Resumiendo, las estructuras que manejaremos para definir a los tipos de datos inductivos y a los coinductivos son las siguientes:

- Tipos inductivos $\mathcal{I} = \mu X.F(X)$
  objetos básicos y constructores: $\mathsf{in} : F\mu F \to \mu F$
  funciones con dominio inductivo $f : \mathcal{I} \to B$
- Tipos coinductivos: $\mathcal{C} = \nu X.F(X)$
  destructores: $\mathsf{out} : \nu F \to F\nu F$
  funciones con codominio coinductivo: $f : A \to \mathcal{C}$

Y las funciones f, sobre los tipos de datos, se definirán dando la función de paso φ correspondiente, dentro de los diagramas conmutativos para cada tipo de dato.

2.1. Tipos de datos inductivos

Se presentan ejemplos de tipos de datos inductivos, algunos ya se han descrito en el capítulo anterior. Aquí se da, nuevamente, una descripción intuitiva y recursiva del tipo de dato en cuestión y se define categóricamente con ayuda de las álgebras iniciales, es decir el conjunto de objetos inductivos determinado por el funtor y los morfismos in.

2.1.1. Números naturales

Uno de los tipos clásicos y básicos son los números naturales, éstos están descritos por medio de la siguiente definición recursiva:

$$0 \in \mathsf{nat}$$
$$\text{Si } n \in \mathsf{nat} \text{ entonces } \mathsf{succ}(n) \in \mathsf{nat}$$

donde succ es la función sucesor que obtiene el siguiente número a partir de un número. La definición categórica mediante el álgebra inicial $(\mathsf{nat}, \mathsf{in})$ es:

$$\begin{aligned} \mathsf{nat} &= \mu X, 1 + X \\ \mathsf{in} &= [0, \mathsf{succ}] \end{aligned}$$

Donde el morfismo in se define mediante:

$$\begin{array}{rcll} 0 & = & \mathsf{in}(\mathsf{inl}(\star)) & : 1 \to \mathsf{nat} \\ \mathsf{succ} & = & \lambda n.\, \mathsf{in}(\mathsf{inr}(n)) & : \mathsf{nat} \to \mathsf{nat} \end{array}$$

Se pueden definir varias operaciones sobre este tipo de datos utilizando el principio de iteración, es decir el siguiente diagrama conmutativo representando el morfismo entre el álgebra inicial, es decir los números naturales y cualquier otra álgebra C:

$$\begin{array}{ccc} 1 + \mathsf{nat} & \xrightarrow{\mathsf{in}} & \mathsf{nat} \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ FC & \xrightarrow{\varphi} & C \end{array}$$

donde f es la función que queremos definir sobre los números naturales, como por ejemplo:

- Suma de dos números naturales $\mathsf{add} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
  Sabemos que la suma puede ser definida como:

  $$\begin{array}{lllcl} \mathsf{add} & m & 0 & = & 0 \\ \mathsf{add} & m & (\mathsf{succ}\ n_1) & = & \mathsf{succ}\ (\mathsf{add}\ \ m\ \ n_1) \end{array}$$

  En nuestro diagrama conmutativo tenemos a $\varphi = [\lambda x.m, \mathsf{succ}]$

- Producto de dos números naturales $\mathsf{prod} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$
  Definiendo el producto de la siguiente manera:

  $$\begin{array}{lllcl} \mathsf{prod} & m & 0 & = & 0 \\ \mathsf{prod} & m & (\mathsf{succ}\ n) & = & \mathsf{add}\ \ m\ \ (\mathsf{prod}\ \ m\ \ n) \end{array}$$

  Sustituyendo en el diagrama conmutativo $\varphi = [\lambda x.m, \lambda x.\, \mathsf{add}(m, x)]$

Una función más interesante sería el factorial de un número, pero ésta no puede ser definida utilizando el principio de iteración, dado que la definición intuitiva de factorial está hecha con recursión:

$$\begin{array}{lllcl} \mathsf{fact} & 0 & & = & 1 \\ \mathsf{fact} & (\mathsf{succ}\ n) & & = & \mathsf{prod}\ (\mathsf{succ}\ n)\ (\mathsf{fact}\ n) \end{array}$$

Es decir tenemos un argumento extra en la función de paso, pertenece a nat y es $\mathsf{succ}\, n$, además de la aplicación de la función a un elemento de menor complejidad. Por lo tanto usaremos el siguiente diagrama conmutativo que ilustra el principio de recursión primitiva:

$$\begin{array}{ccc} 1 + \mathsf{nat} & \xrightarrow{\mathsf{in}} & \mathsf{nat} \\ {\scriptstyle F\langle \mathsf{Id}, f\rangle}\downarrow & & \downarrow{\scriptstyle f} \\ F(\mathsf{nat} \times C) & \xrightarrow{\varphi} & C \end{array}$$

Finalmente la función factorial queda definida con $\varphi = [1, \lambda f, n.\, \mathsf{prod}\ \ (\mathsf{succ}\ \ n\ \ \ f)]$

2.1.2. Listas

Otro tipo básico son las listas sobre un conjunto A, $\mathsf{list}(A)$; una lista es una sucesión de elementos del conjunto A, como por ejemplo una lista sobre el conjunto de números naturales es: $[3, 7, 2, 9, 1, 5]$, y se define recursivamente como:

$$\mathsf{nil} \in \mathsf{list}(A), \text{ la lista vacía}$$
$$\text{Si } a \in A \text{ y } l \in \mathsf{list}(A) \text{ entonces } \mathsf{cons}(a, l) \in \mathsf{list}(A)$$

donde $\mathsf{cons}(a, l)$ es la función constructora de listas.
En la categoría de las álgebras tenemos a las listas como un álgebra inicial $(\mathsf{list}(A), \mathsf{in})$ definida mediante:

$$\begin{array}{rcl} \mathsf{list}(A) & = & \mu X, 1 + A \times X \\ \mathsf{in} & = & [\mathsf{nil}, \mathsf{cons}] \end{array}$$

Donde el morfismo in se define mediante:

$$\begin{array}{rcll} \mathsf{nil} & = & \mathsf{in}(\mathsf{inl}(*)) & : 1 \to \mathsf{list}(A) \\ \mathsf{cons} & = & \lambda x, y.\, \mathsf{in}(\mathsf{inr}(x, y)) & : A \times \mathsf{list}(A) \to \mathsf{list}(A) \end{array}$$

Con el siguiente diagrama conmutativo podemos ver los morfismos entre las listas y una categoría C para obtener funciones sobre las listas:

$$\begin{array}{ccc} 1 + A \times \mathsf{list}(A) & \xrightarrow{\mathsf{in}} & \mathsf{list}(A) \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ FC & \xrightarrow{\varphi} & C \end{array}$$

Algunas funciones sobre listas:

- Longitud de una lista $\mathsf{length} : \mathsf{list}(A) \to Nat$
  Con esta función podemos obtener el número de elementos de la lista:

  $$\begin{array}{rcl} \mathsf{length}\ \ \mathsf{nil} & = & 0 \\ \mathsf{length}\ \ (\mathsf{cons}\ a\ l) & = & 1 + \mathsf{length}(l) \end{array}$$

  Y podemos ver a $\varphi = [0, \lambda x.\, \mathsf{succ}\ \ \mathsf{snd}\, x]$ dentro del diagrama.

- Concatenación de dos listas $\mathsf{concat} : \mathsf{list}(A) \to \mathsf{list}(A) \to \mathsf{list}(A)$
  Concatenar dos listas es simplemente colocar a los elementos de la segunda al final de los de la primera para así obtener una sóla lista:

  $$\begin{array}{rcl} \mathsf{concat}\ \ \mathsf{nil}\ \ l & = & l \\ \mathsf{concat}\ \ (\mathsf{cons}\, a\ l_1)\ \ l_2 & = & \mathsf{cons}\ a\ (\mathsf{concat}\ l_1\ l_2) \end{array}$$

  En el diagrama se tiene a $\varphi = [\lambda x.x, \mathsf{cons}]$

- Reversa de una lista $\mathsf{rev} : \mathsf{list}(A) \to \mathsf{list}(A)$
  La reversa de una lista también es una operación básica muy utilizada, consiste en " voltear" a la lista, es decir, colocar los elementos al revés como por ejemplo la

lista de naturales $l = [2, 5, 1, 7, 3, 0]$ tiene por reversa a $\mathsf{rev}(l) = [0, 3, 7, 1, 5, 2]$ y la podemos definir de la siguiente manera:

$$\begin{array}{lcl} \mathsf{rev} \quad \mathsf{nil} & = & \mathsf{nil} \\ \mathsf{rev} \quad (\mathsf{cons}\ a\ l) & = & \mathsf{concat}\ \ (\mathsf{rev}\ l)\ \ (\mathsf{cons}\ a\ \ \mathsf{nil}) \end{array}$$

Tenemos a $\varphi = [\mathsf{nil}, \mathsf{concat}]$

2.1.3. Árboles

Un tipo de datos más complejo son los árboles, éstos pueden ser de diferentes tipos dependiendo de la información que contengan o dependiendo de su organización:

1. Árboles con información sólo en las hojas
   La información de las hojas pertenece a un tipo A, están definidos recursivamente como:

   Si $a \in A$ entonces $\mathsf{leaf}(a)$ es una hoja y es un árbol.
   Si t_1 y t_2 son árboles entonces $\mathsf{tree}(t_1, t_2)$ es un árbol.

   Categóricamente están definidos como un álgebra inicial $(BTree(A), \mathsf{in})$ donde:

   $$\begin{array}{rcl} \mathsf{BTree}(A) & = & \mu X.A + X \times X \\ \mathsf{in} & = & [\mathsf{leaf}, \mathsf{tree}] \end{array}$$

   El morfismo in está definido mediante:

   $$\begin{array}{lcll} \mathsf{leaf} & = & \mathsf{in}(\mathsf{inl}(a)) & : A \to \mathsf{BTree(A)} \\ \mathsf{tree} & = & \lambda t, t.\, \mathsf{in}(\mathsf{inr}(t, t)) & : \mathsf{BTree(A)} \times \mathsf{BTree(A)} \to \mathsf{BTree(A)} \end{array}$$

   Para definir funciones sobre este tipo de árboles usamos el siguiente diagrama conmutativo:

   $$\begin{array}{ccc} A + BTree(A) \times BTree(A) & \xrightarrow{\mathsf{in}} & BTree(A) \\ \Big\downarrow{\scriptstyle Ff} & & \Big\downarrow{\scriptstyle f} \\ FC & \xrightarrow{\varphi} & C \end{array}$$

   Una función sobre estos árboles:

   - Lista de los elementos del árbol $\mathsf{BTreeList} : \mathsf{BTree(A)} \to \mathsf{list}(A)$
     Obtener la lista de elementos del árbol es recorrer las hojas y recuperar la información de ellas, se puede definir de la siguiente manera con ayuda de las funciones para listas:

     $$\begin{array}{lcl} \mathsf{BTreeList} \quad (\mathsf{leaf}\ a) & = & \mathsf{cons}\ a\ \ \mathsf{nil} \\ \mathsf{BTreeList} \quad (\mathsf{tree}\, t_1\ t_2) & = & \mathsf{concat}\ (\mathsf{BTreeList}\ t_1)\ (\mathsf{BTreeList}\ t_2) \end{array}$$

     En el diagrama conmutativo tenemos a $\varphi = [\mathsf{uniq}, \mathsf{concat}]$ donde la función uniq se define como $\mathsf{uniq}(x) = \mathsf{cons}(x, \mathsf{nil})$, es decir, es la lista unitaria $\mathsf{uniq}(x) = [x]$.

2. Árboles con información en todos los nodos
   Éstos árboles se definen recursivamente como sigue:

   Si $a \in A$ entonces $\mathsf{leaf}(a)$ es un árbol.
   Si $a \in A, t_1$ y t_2 son árboles entonces $\mathsf{tree}(a, t_1, t_2)$ es un árbol.

   Categóricamente están definidos como un álgebra inicial $(\mathsf{BTree(A)}, \mathsf{in})$:

$$\begin{aligned} \mathsf{BTree(A)} &= \mu X.A + A \times X \times X \\ \mathsf{in} &= [\mathsf{leaf}, \mathsf{tree}] \end{aligned}$$

   Donde el morfismo in se define mediante:

$$\begin{aligned} \mathsf{leaf} &= \mathsf{in}(\mathsf{inl}(a)) && : A \to \mathsf{BTree}(A) \\ \mathsf{tree} &= \lambda x, t_1, t_2.\, \mathsf{in}(\mathsf{inr}(x, t_1, t_2)) && : A \times \mathsf{BTree}(A) \times \mathsf{BTree}(A) \to \mathsf{BTree}(A) \end{aligned}$$

   Para definir funciones sobre este tipo de árboles tenemos el siguiente diagrama conmutativo:

$$\begin{array}{ccc} A + A \times BTree(A) \times BTree(A) & \xrightarrow{\mathsf{in}} & BTree(A) \\ \Big\downarrow{\scriptstyle Ff} & & \Big\downarrow{\scriptstyle f} \\ FC & \xrightarrow{\quad\varphi\quad} & C \end{array}$$

   Algunas funciones clásicas sobre estos árboles son los diferentes recorridos para obtener una lista de sus elementos; nuevamente se utilizan algunas funciones de listas y la función $\mathsf{uniq}(a) = \mathsf{cons}(a, \mathsf{nil})$:

   - Recorrido en preorden $\mathsf{preorden} : \mathsf{BTree}(A) \to \mathsf{list}(A)$
     El recorrido de un árbol con un sólo nodo es muy sencillo mientras que el recorrido en preorden de un árbol que tiene más de un nodo se obtiene al recuperar primero la información de la raíz y luego el recorrido en preorden de los subárboles izquierdo y derecho respectivamente:

$$\begin{aligned} \mathsf{preorden}\ \ (\mathsf{leaf}\ a) &= \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{preorden}\ \ (\mathsf{tree}\ a\, t_1\, t_2) &= \mathsf{cons}\ a\ (\mathsf{concat}\ (\mathsf{preorden}\ t_1)\ (\mathsf{preorden}\ t_2)) \end{aligned}$$

     Y la función $\varphi = [\mathsf{uniq}, \mathsf{cons}]$

   - Recorrido en inorden $\mathsf{inorden} : \mathsf{BTree}(A) \to \mathsf{list}(A)$
     El recorrido de un árbol con más de un nodo en inorden coloca primero la información del recorrido en inorden del subárbol izquierdo, la información de la raíz y por último el recorrido en inorden del árbol derecho:

$$\begin{aligned} \mathsf{inorden}\ \ (\mathsf{leaf}\ a) &= \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{inorden}\ \ (\mathsf{tree}\ a\, t_1\, t_2) &= \mathsf{concat}(\mathsf{inorden}\ t_1)\ (\mathsf{cons}\ a\ (\mathsf{inorden}\ t_2)) \end{aligned}$$

     La función de paso en el diagrama conmutativo es $\varphi = [\mathsf{uniq}, \mathsf{cons}]$

- Recorrido en postorden $\mathsf{postorden} : \mathsf{BTree}(A) \to \mathsf{list}(A)$
  El recorrido en postorden de un árbol con más de un nodo coloca primero la información del recorrido en postorden del subárbol izquierdo, el recorrido en postorden del árbol derecho y por último la información de la raíz:

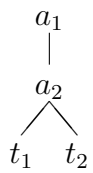
$$\begin{array}{lrcl} \mathsf{postorden} & (\mathsf{leaf}\ a) & = & \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{postorden} & (\mathsf{tree}\ a\, t_1\, t_2) & = & \mathsf{concat}(\mathsf{postorden}\ t_1)\ (\mathsf{concat}(\mathsf{postorden}\ t_2)\ (\mathsf{cons}\ a\ \mathsf{nil})) \end{array}$$

  Con $\varphi = [\mathsf{uniq}, \mathsf{cons}]$

3. (1,2)-Árboles
   Estos árboles tienen la forma de ser sólamente un nodo o un nodo raíz tiene un hijo el cual tiene sólo dos hijos, como se muestra en el siguiente diagrama:

   a) una hoja $\cdot$ a

   b) dos objetos en A y dos árboles unidos de esta manera forman un árbol:



   Recursivamente pueden ser definidos como:

$$\text{Si } a \in A \text{ entonces } \mathsf{leaf}(a) \text{ es un árbol}$$
$$\text{Si } a_1,\ a_2 \in A \text{ y } t_1,\ t_2 \text{ son árboles entonces } \mathsf{tree}(a_1, a_2, t_1, t_2) \text{ es un árbol.}$$

   Categóricamente están definidos como un álgebra inicial $(\mathsf{Tree}_{(1,2)}(A), \mathsf{in})$:

$$\begin{array}{rcl} \mathsf{Tree}_{(1,2)}(A) & = & \mu X.A + A \times A \times X \times X \\ \mathsf{in} & = & [\mathsf{leaf}, \mathsf{tree}] \end{array}$$

   El morfismo in se define mediante:

$$\begin{array}{rcl} \mathsf{leaf} & = & \mathsf{in}(\mathsf{inl}(a)) \quad : A \to \mathsf{Tree}_{(1,2)}(A) \\ \mathsf{tree} & = & \lambda x_1, x_2, t_1, t_2.\, \mathsf{in}(\mathsf{inr}(x_1, x_2, t_1, t_2)) \\ \mathsf{tree} & & : A \times A \times \mathsf{Tree}_{(1,2)}(A) \times \mathsf{Tree}_{(1,2)}(A) \to \mathsf{Tree}_{(1,2)}(A) \end{array}$$

   Para definir funciones sobre este tipo de árboles tenemos el siguiente diagrama conmutativo:

$$\begin{array}{ccc} A + A \times A \times Tree_{(1,2)}(A) \times Tree_{(1,2)}(A) & \xrightarrow{\quad\mathsf{in}\quad} & Tree_{(1,2)}(A) \\ \Big\downarrow Ff & & \Big\downarrow f \\ FC & \xrightarrow{\quad\varphi\quad} & C \end{array}$$

   Algunas funciones sobre éstos árboles son los recorridos, y se definen de manera semejante a los anteriores:

- Recorrido en preorden $\mathsf{preorden} : \mathsf{Tree}_{(1,2)}(A) \to \mathsf{list}(A)$
  El recorrido de un árbol en preorden coloca primero la información de la raíz, el nodo hijo y luego el recorrido en preorden de los subárboles izquierdo y derecho respectivamente:

$$\begin{array}{lrcl} \mathsf{preorden} & (\mathsf{leaf}\ a) & = & \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{preorden} & (\mathsf{tree}\ a_1\, a_2\, t_1\, t_2) & = & \mathsf{cons}\ a_1(\mathsf{cons}\ a_2\ (\mathsf{concat}\ (\mathsf{preorden}\, t_1)\ (\mathsf{preorden}\, t_2))) \end{array}$$

  Con $\varphi = [\mathsf{uniq}, \mathsf{cons}]$

- Recorrido en inorden $\mathsf{inorden} : \mathsf{Tree}_{(1,2)}(A) \to \mathsf{list}(A)$
  El recorrido de un árbol en inorden coloca primero la información del recorrido en inorden del subárbol izquierdo, la información de la raíz, luego la información del nodo inferior a la raíz y por último el recorrido en inorden del árbol derecho:

$$\begin{array}{lrcl} \mathsf{inorden} & (\mathsf{leaf}\ a) & = & \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{inorden} & (\mathsf{tree}\ a_1\, a_2\, t_1\, t_2) & = & \mathsf{concat}(\mathsf{inorden}\ t_1)\ (\mathsf{cons}\ a_1\ (\mathsf{cons}\ a_2\ (\mathsf{inorden}\ t_2))) \end{array}$$

  Con $\varphi = [\mathsf{uniq}, \mathsf{concat}]$

- Recorrido en postorden $\mathsf{postorden} : \mathsf{Tree}_{(1,2)}(A) \to \mathsf{list}(A)$
  El recorrido de un árbol en postorden coloca primero la información del recorrido en postorden del subárbol izquierdo, el recorrido en postorden del árbol derecho y por último la información de la raíz seguida de la información del nodo inferior a la raíz:

$$\begin{array}{lrcl} \mathsf{postorden} & (\mathsf{leaf}\ a) & = & \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{postorden} & (\mathsf{tree}\ a_1\, a_2\, t_1\, t_2) & = & \mathsf{concat}(\mathsf{postorden}\ t_1) \\ & & & \quad (\mathsf{concat}(\mathsf{postorden}\ t_2)\ (\mathsf{cons}\ a_1\, [a_2])) \end{array}$$

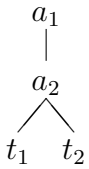
  Con $\varphi = [\mathsf{uniq}, \mathsf{concat}]$

4. (1,2)-Árboles con sucesor
   Los árboles pueden ser de la forma siguiente:

   a) un nodo solo, sin sucesor $\cdot$ a

   b) nodo con árbol sucesor

   c) nodo con un hijo con dos árboles sucesores



   Recursivamente pueden ser definidos como:

Si $a \in A$ entonces $\mathsf{leaf}(a)$ es una hoja y es un árbol
Si $a \in A$ y t es un árbol entonces el $\mathsf{succ}(a, t)$ es un árbol
Si $a_1,\ a_2 \in A$ y $t_1,\ t_2$ son árboles entonces $\mathsf{tree}(a_1, a_2, t_1, t_2)$ es un árbol

Categóricamente están definidos como un álgebra inicial $(\mathsf{Tree}_{(1,2)\mathsf{s}}(A), \mathsf{in})$:

$$\begin{aligned} \mathsf{Tree}_{(1,2)\mathsf{s}}(A) &= \mu X.A + A \times X + A \times A \times X \times X \\ \mathsf{in} &= [\mathsf{leaf}, \mathsf{succ}, \mathsf{branch}] \end{aligned}$$

Donde el morfismo in se define mediante:

$$\begin{aligned} \mathsf{leaf} &= \mathsf{in}(\mathsf{inl}(a)) \quad : A \to \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \\ \mathsf{succ} &= \lambda x, t_1.\, \mathsf{in}(\mathsf{inr}(x, t_1)) \quad : A \times \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \to \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \\ \mathsf{branch} &= \lambda x_1, x_2, t_1, t_2.\, \mathsf{in}(inr(x_1, x_2, t_1, t_2) \\ \mathsf{branch} &\quad : A \times A \times \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \times \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \to \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \end{aligned}$$

Para definir funciones sobre este tipo de árboles tenemos el siguiente diagrama conmutativo:

$$A + A \times Tree_{(1,2)s}(A) +$$

$$\begin{array}{ccc} A \times A \times Tree_{(1,2)s}(A) \times Tree_{(1,2)s}(A) & \xrightarrow{\quad\mathsf{in}\quad} & Tree_{(1,2)s}(A) \\ \Big\downarrow Ff & & \Big\downarrow f \\ FC & \xrightarrow{\quad\varphi\quad} & C \end{array}$$

Las funciones básicas sobre este tipo de árboles son la obtención de información de tres diferentes recorridos como ha sido visto en los árboles anteriores:

- Recorrido en preorden $\mathsf{preorden} : \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \to \mathsf{list}(A)$
  El recorrido de un árbol de la forma sucesor coloca primero la información del nodo y posteriormente la información en preorden del árbol que lo sucede, para los otros constructores se define de manera semejante a los ejemplos de árboles vistos anteriormente:

  $$\begin{aligned} \mathsf{preorden} \quad (\mathsf{leaf}\ a) \quad &= \quad \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{preorden} \quad (\mathsf{succ}\ a\, t) \quad &= \quad \mathsf{cons}\ a\, (\mathsf{preorden}\ t) \\ \mathsf{preorden} \quad (\mathsf{tree}\ a_1\, a_2\, t_1\, t_2) \quad &= \quad \mathsf{cons}\ a_1\ (\mathsf{cons}\ a_2\ (\mathsf{concat}\ (\mathsf{preorden}\ t_1)(\mathsf{preorden}\ t_2))) \end{aligned}$$

  Y la función $\varphi = [\mathsf{uniq}, \mathsf{cons}, \mathsf{cons}]$

- Recorrido en inorden $\mathsf{inorden} : \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \to \mathsf{list}(A)$
  El recorrido de un árbol en inorden para el caso de sucesor, coloca primero la información del nodo y luego el recorrido en inorden del árbol inferior:

  $$\begin{aligned} \mathsf{inorden} \quad (\mathsf{leaf}\ a) \quad &= \quad \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{inorden} \quad (\mathsf{succ}\ a\, t) \quad &= \quad \mathsf{concat}\, (\mathsf{inorden}\ t)\ (\mathsf{cons}\ a\ \mathsf{nil}) \\ \mathsf{inorden} \quad (\mathsf{tree}\ a_1\, a_2\, t_1\, t_2) \quad &= \quad \mathsf{concat}\, (\mathsf{inorden}\ t_1)\ (\mathsf{cons}\ a_1\ (\mathsf{cons}\ a_2\ (\mathsf{inorden}(t_2)))) \end{aligned}$$

  La función en el diagrama conmutativo es $\varphi = [\mathsf{uniq}, \mathsf{cons}, \mathsf{concat}]$

- Recorrido en postorden $\mathsf{postorden} : \mathsf{Tree}_{(1,2)\mathsf{s}}(A) \to \mathsf{list}(A)$
  El recorrido de un árbol en postorden coloca primero la información del recorrido en postorden del subárbol izquierdo, el recorrido en postorden del árbol derecho y por último la información de la raíz seguida de la información del nodo inferior a la raíz:

$$\begin{array}{lrcl} \mathsf{postorden} & (\mathsf{leaf}\ a) & = & \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{postorden} & (\mathsf{succ}\ a\, t) & = & \mathsf{concat}(\mathsf{postorden}\ t)\ (\mathsf{cons}\ a\ \mathsf{nil}) \\ \mathsf{postorden} & (\mathsf{tree}\ a_1\, a_2\, t_1\, t_2) & = & \mathsf{concat}\,(\mathsf{postorden}\ t_1) \\ & & & \qquad (\mathsf{concat}(\mathsf{postorden}\ t_2)\ (\mathsf{cons}\ a_1\, [a_2])) \end{array}$$

  Con $\varphi = [\mathsf{uniq}, \mathsf{concat}, \mathsf{concat}]$

2.2. Tipos de datos coinductivos

Se presentan ejemplos correspondientes a los tipos de datos coinductivos, contienen una descripción intuitiva y una correcursiva, después se definen categóricamente con ayuda de las coálgebras finales, es decir por el funtor correspondiente y el morfismo out.

2.2.1. Conaturales

Los números naturales, como un tipo inductivo, tienen su dual: los números conaturales. A este conjunto pertenecen los números naturales como los conocemos y un nuevo elemento ω que se explicará después. Se definen coinductivamente como se ve a continuación:

$$\text{Si } n \in \mathsf{conat} \text{ entonces } \mathsf{pred}(n) \in \mathsf{conat}$$

donde $\mathsf{pred}(n)$ es el predecesor de n y se define como:

$$\begin{array}{rcl} \mathsf{pred}(0) & = & 0 \\ \mathsf{pred}(n+1) & = & n \\ \mathsf{pred}(\omega) & = & \omega \end{array}$$

Coinductivamente podemos tomar la categoría de las coalgebras, el tipo de datos conat es una coalgebra terminal definida por (conat, out):

$$\begin{array}{rcl} \mathsf{conat} & = & \nu X, 1 + X \\ \mathsf{out} & = & \mathsf{pred} \end{array}$$

Donde el morfismo out se define mediante:

$$\begin{array}{rcl} \mathsf{pred} : & & \mathsf{conat} \to \mathsf{conat} \\ \mathsf{pred}(0) & = & \mathsf{inl}() \\ \mathsf{pred}(n+1) & = & \mathsf{inr}(n) \\ \mathsf{pred}(\omega) & = & \mathsf{inr}(\omega) \end{array}$$

Para definir funciones sobre este tipo de datos podemos basarnos en el siguiente diagrama en las coalgebras:

$$\begin{array}{ccc} C & \xrightarrow{\quad\varphi\quad} & FC \\ \Big\downarrow{\scriptstyle f} & & \Big\downarrow{\scriptstyle Ff} \\ \mathsf{conat} & \xrightarrow{\quad\mathsf{out}\quad} & 1+\mathsf{conat} \end{array}$$

Para definir funciones sobre los conaturales tenemos que:

$$\mathsf{pred}(f(x)) = \mathsf{case}\, f(x)\ \mathsf{of} \begin{cases} \mathsf{inl}(\star) \Rightarrow' inl(\star) \\ \mathsf{inr}(y) \Rightarrow \mathsf{inr}(f(y)) \end{cases}$$

- Suma de dos conaturales $\mathsf{add} : \mathsf{conat} \times \mathsf{conat} \to \mathsf{conat}$
  Podemos definir la suma como:

$$\mathsf{pred}(\mathsf{add}(x,y)) = \begin{cases} \mathsf{inl}(\star) & \text{si } \mathsf{pred}(x) = \mathsf{pred}(y) = \mathsf{inl}(\star) \\ \mathsf{inr}\ \mathsf{add}(x',y) & \text{si } \mathsf{pred}(x) = \mathsf{inr}(x') \\ \mathsf{inr}\ \mathsf{add}(x,y') & \text{si } \mathsf{pred}(x) = \mathsf{inl}(\star) \text{y } \mathsf{pred}(y) = \mathsf{inr}(y') \end{cases}$$

  Sustituimos en el diagrama conmutativo

$$\varphi(n,m) = \begin{cases} \mathsf{inl}(\star) & \text{si } \mathsf{pred}(n) = \mathsf{pred}(m) = \mathsf{inl}(\star) \\ \mathsf{inr}\ (n',m) & \text{si } \mathsf{pred}(n) = \mathsf{inr}(n') \\ \mathsf{inr}\ (n,m') & \text{si } \mathsf{pred}(n) = \mathsf{inl}(\star) \text{y } \mathsf{pred}(m) = \mathsf{inr}(m') \end{cases}$$

- Producto de dos conaturales $\mathsf{prod} : \mathsf{conat} \times \mathsf{conat} \to \mathsf{conat}$
  Podemos definir el producto como:

$$\mathsf{pred}(\mathsf{prod}(x,y)) =$$
$$\begin{cases} \mathsf{inl}(\star) & \text{si } \mathsf{pred}(x) = \mathsf{inl}(\star) \\ \mathsf{inr}\ y' & \text{si } \mathsf{pred}(x) = \mathsf{inr}(x'),\ \mathsf{pred}(x') = \mathsf{inl}(\star) \text{y } \mathsf{pred}(y) = \mathsf{inr}(y') \\ \mathsf{inr}\ \mathsf{add}(\mathsf{prod}(x',y),x') & \text{si } \mathsf{pred}(x) = \mathsf{inr}(x') \end{cases}$$

  Sustituimos en el diagrama conmutativo

$$\varphi(n,m) = \begin{cases} \mathsf{inl}(\star) & \text{si } \mathsf{pred}(n) = \mathsf{inl}(\star) \\ \mathsf{inr}\ (n,m') & \text{si } \mathsf{pred}(n) = \mathsf{inr}(n'), \mathsf{pred}(n') = \mathsf{inl}(\star) \text{y } \mathsf{pred}(m) = \mathsf{inr}(m') \\ \mathsf{inr}\ (n',m) & \text{si } \mathsf{pred}(n) = \mathsf{inr}(n') \end{cases}$$

La introducción del elemento ω en conat se debe a la necesidad de hacer a la coálgebra definida por el funtor $F(X) = 1 + X$, una coalgebra final. En otras palabras, los números naturales se pueden construir mediante succ en un proceso infinito; para los conaturales, al destruirlos con pred se requiere de una representación del "último" de los conaturales que pueda ser destruido para poder observarlo, intuitivamente si estamos en el infinito el

número que esté ahi es tan grande que al ser destruido queda el mismo. Este elemento es ω, a continuación se presenta la construcción de ω categóricamente.
Dado el destructor $\mathsf{out} = \mathsf{pred}$ podemos obtener su inverso y definir los constructores para los números conaturales:

$$\begin{array}{rcl} \mathsf{out}^{-1} & : & 1 + \mathsf{conat} \to \mathsf{conat} \\ 0 & = & out^{-1}(\mathsf{inl}(\star)) \\ \mathsf{succ}(x) & = & \mathsf{out}^{-1}(\mathsf{inr}(x))\ x \in \mathsf{conat} \end{array}$$

Consideremos el siguiente diagrama conmutativo:

$$\begin{array}{ccc} 1 & \xrightarrow{\quad\varphi\quad} & 1+1 \\ \Big\downarrow{\scriptstyle \omega^+} & & \Big\downarrow{\scriptstyle 1+\omega^+} \\ \mathsf{conat} & \xrightarrow{\quad\mathsf{out}\quad} & 1+\mathsf{conat} \end{array}$$

Se define a ω como $\omega^+(\star)$ y dado que el diagrama conmuta tenemos las siguientes observaciones, con $\varphi(x) = \mathsf{inr}(x)$:

$$\begin{array}{rcl} \mathsf{pred} \circ \omega^+ & = & (1 + \omega^+) \circ \varphi \\ \mathsf{pred}(\omega^+(x)) & = & \omega^+(x) \\ (1 + \omega^+) \circ \varphi)(x) & = & \omega^+(x) \end{array}$$

2.2.2. Streams

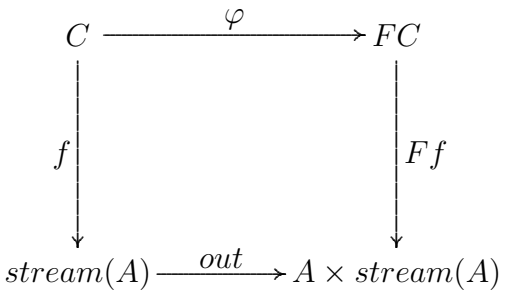
Un tipo semejante a las listas sobre un conjunto A son los streams sobre un conjunto A, $\mathsf{stream}(A)$; los elementos de este tipo de dato son listas infinitas sobre el conjunto dado:

$$\begin{array}{c} \text{Si } s \in \mathsf{stream}(A) \text{ entonces } \mathsf{head}(s) \in A \\ \text{Si } s \in \mathsf{stream}(A) \text{ entonces } \mathsf{tail}(s) \in \mathsf{stream}(A) \end{array}$$

donde head y tail son las funciones destructoras de los streams.
Los streams son una coalgebra terminal, $(\mathsf{stream}(A), \mathsf{out})$ donde:

$$\begin{array}{rcl} \mathsf{stream}(A) & = & \nu.A \times X \\ \mathsf{out} & = & < \mathsf{head}, \mathsf{tail} > \\ \mathsf{head} & : & \mathsf{stream}(A) \to A \\ \mathsf{tail} & : & \mathsf{stream}(A) \to \mathsf{stream}(A) \end{array}$$

Podemos definir funciones sobre los streams en base al siguiente diagrama de morfismos hacia los streams:



Algunas funciones:

- Un stream a partir de una constante $\mathsf{const} : A \to stream(A)$
  Este stream es muy básico, es la lista infinita que repite el número natural dado como argumento:

  $$\text{Si } a \in A \text{ entonces } \mathsf{const}(a) = [a, a, a, \dots]$$

  En el diagrama $\varphi = \langle \mathsf{Id}, \mathsf{Id} \rangle$:

  $$\begin{aligned} \mathsf{head}(\mathsf{const}\ a) &= a \\ \mathsf{tail}(\mathsf{const}\ a) &= \mathsf{const}\ (a) \end{aligned}$$

- Un stream a partir de un natural $\mathsf{from} : \mathsf{nat} \to \mathsf{stream}(\mathsf{nat})$
  Es una lista infinita de naturales a partir del número natural dado, n:

  $$\mathsf{from}(n) = (n, \mathsf{succ}(n), \mathsf{succ}(\mathsf{succ}(n)), \dots)$$

  Categóricamente, en el diagrama conmutativo tenemos a $\varphi = \langle \mathsf{Id}, \mathsf{succ} \rangle$:

  $$\begin{aligned} \mathsf{head}(\mathsf{from}\ n) &= n \\ \mathsf{tail}(\mathsf{from}\ n) &= \mathsf{from}(\mathsf{succ}\ n) \end{aligned}$$

- Empaquetado de dos streams $\mathsf{zip} : \mathsf{stream}(A) \times \mathsf{stream}(B) \to \mathsf{stream}(A \times B)$
  El empaquetado de dos streams obtiene un stream de pares, éstos tienen a sus elementos en cada uno de los conjuntos de los streams dados y se define como:

  $$\mathsf{zip}((a_1, a_2, \dots), (b_1, b_2, \dots)) = (\langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle, \dots)$$

  Con $\varphi = \langle \langle \mathsf{head}, \mathsf{head} \rangle, \langle \mathsf{tail}, \mathsf{tail} \rangle \rangle$:

  $$\begin{aligned} \mathsf{head}(\mathsf{zip}\ s_1\ s_2) &= \langle a_1, b_1 \rangle \\ \mathsf{tail}(\mathsf{zip}\ s_1\ s_2) &= \mathsf{zip}(\mathsf{tail}\ s_1)\ (\mathsf{tail}\ s_2) \end{aligned}$$

  Con $s_1 = (a_1, a_2, \dots)$ y $s_2 = (b_1, b_2, \dots)$

- Mezcla de dos streams $\mathsf{merge} : \mathsf{stream}(A) \times \mathsf{stream}(B) \to \mathsf{stream}(A \cup B)$
  Esta mezcla se hará combinando los elementos de un stream con el del otro de manera alternada:

$$\mathsf{merge}((a_1, a_2, \dots), (b_1, b_2, \dots)) = (a_1, b_1, a_2, b_2, a_3, b_3, \dots)$$

Con $\varphi = \langle \mathsf{head}\ \ \mathsf{fst}, (\mathsf{snd}, \mathsf{tail}\,\mathsf{fst}) \rangle$

$$\begin{array}{rcl} \mathsf{head}(\mathsf{merge}\ s_1\ s_2) & = & \mathsf{head}\ \ s_1 \\ \mathsf{tail}(\mathsf{merge}\ s_1\ s_2) & = & \mathsf{merge}\ s_2\ (\mathsf{tail}\ \ s_1) \end{array}$$

Veamos un último ejemplo, una función tomada de [22] en la cual aplicaremos una función, $g : A \to A$, solamente a la cabeza de un stream: $\mathsf{maphead}\ g : \mathsf{stream}(A) \to \mathsf{stream}(A)$. Esta función no hace ningún cambio a la cola del stream por lo que para definirla es necesario ocupar el principio de correcursión primitiva. En el siguiente diagrama conmutativo se aprecia la función definida mediante $\varphi = \langle g \circ \mathsf{head}, \mathsf{inr} \circ \mathsf{tail} \rangle$:

$$\begin{array}{ccc} B & \xrightarrow{\quad\varphi\quad} & A \times \mathsf{stream}(A) + B \\ \Big\downarrow{\scriptstyle \mathsf{CoRec}_s^n} & & \Big\downarrow{\scriptstyle F_i([\mathsf{Id}, \mathsf{CoRec}_s^n])} \\ \mathsf{stream}(A) & \xrightarrow{\mathsf{out}_{n,i}} & A \times \mathsf{stream}(A) \end{array}$$

Como se puede observar en los ejemplos propuestos, la forma de manipular tanto a los tipos de datos como a las funciones definidas es difícil debido a los diagramas conmutativos, si quisiéramos programar con ellos sería tedioso.
En el siguiente capítulo se propone una nueva forma de programar con tipos de datos (co)inductivos, manteniendo la idea de las categorías y en especial de las funciones definidas por los diagramas anteriores.

Capítulo 3

Un Sistema de Tipos (Co)inductivos

En el capítulo anterior, se expusieron ejemplos tanto de tipos de datos inductivos como coinductivos desde el punto de vista categórico, la codificación de funciones que involucran estos tipos de datos no es cómoda para trabajar, el hecho de realizar alguna operación con ellas es demasiado compleja en cuanto a la sintaxis y en el uso de diagramas.

En este capítulo se dará una extensión del sistema F que permitirá programar con tipos de datos (co)inductivos, al incluirlos en el sistema se simplificará la interacción con el usuario. Ésta inclusión puede ser hecha al agregar constructores para las álgebras iniciales, $\langle \mu XF, \mathsf{in}_F \rangle$ y las coálgebras finales $\langle \nu XF, \mathsf{out}_F \rangle$ pero se explicará que a pesar de ser una forma directa de extender el sistema F es mejor el uso de diálgebras.

Finalmente se dan ejemplos de tipos de datos bajo el sistema, así mismo se propone un prototipo de lenguaje de programación funcional.

3.1. Programación con categorías

Al ser un objetivo principal de esta tesis la incorporación de objetos (co)inductivos a un sistema en donde se pueda programar con estructuras de estos tipos, es necesario encontrar una manera adecuada para trabajar con ellos, teniendo como idea principal el uso de constructores primitivos para representar los objetos (co)inductivos y por supuesto expresar funciones por medio de (co)iteración y (co)recursión primitiva. Hemos visto ejemplos de este objetivo: en el capítulo pasado se dio una exposición de tipos de datos en el contexto categórico, con definiciones de algunas funciones mediante los principios de (co)iteración y y (co)recursión; en la sección dedicada al cálculo lambda también se dieron programas con los cuales se podía incorporar, tanto a los objetos inductivos como a los coinductivos.

En cada uno de estos formalismos se podía programar incluyendo ambos tipos de datos, pero si hacemos una comparación de estas dos formas de programar, podemos decir que la programación con categorías es más difícil, sintácticamente hablando, que la vista en el sistema F, ya que para programar es necesario utilizar los diagramas conmutativos. En el sistema F era posible agregar constructores que codificaran tipos de datos inductivos como por ejemplo codificar a los números naturales y modelar la inducción a través del tipo $\mu XF = \forall X(F(X) \to X \to X)$, así mismo codificar funciones iterativas pero

sólamente fue definido un operador que permitía manejar la recursión sobre los números naturales.
La codificación de la recursión primitiva en el sistema F es un problema abierto, en [23] podemos ver varios intentos por definir eliminadores de los operadores de puntos fijos, es decir iteradores y recursores dentro del sistema F, llegando a la conclusión que éstos no pueden ser implementados dentro de él mediante la reducción β. En otras palabras no es posible codificar una regla de recursión sin los términos F y G definidos dentro del sistema, éstos corresponden a términos que codifican un tipo de dato μXF, es decir in.
Para codificar un tipo de dato inductivo, a partir de la definición, sólo es necesario incorporar un cuantificador universal siendo esto inmediato al agregarlo en el sistema F, pero para uno coinductivo, dada la definición $\nu XF = \exists X(X \to F(X) \to X)$ habría que agregar el cuantificador existencial al sistema, en este trabajo se realiza por medio de cuantificadores universales como fue hecho en el ejemplo en el que se modela al tipo stream en la sección del cálculo lambda.

Para obtener un sistema que permita programar con tipos de datos (co)inductivos, se desarrollará la idea de incorporar la definición y manejo de tipos (co)inductivos de una manera natural, es decir definir un sistema de tipos que maneje tipos de datos inductivos y coinductivos. Para ésto podemos optar por combinar la teoría de categorías y el cálculo lambda dado que ambos representan una solución para incorporar estos objetos y así facilitar el trabajo: de las categorías sabemos que los conjuntos inductivos están representados por las álgebras iniciales y los coinductivos por las coálgebras finales proporcionando una forma clara en el manejo de éstos tipos, y del cálculo lambda sabemos que el sistema F permite trabajar con tipos además de que el polimorfismo agrega una ventaja importante al sistema. Se construirá un sistema que permita definir constructores primitivos de tipo para las álgebras iniciales y las coálgebras finales, por medio de las generalizaciones de las diálgebras, y para ello es necesario ver al sistema de tipos que definiremos como una categoría $\mathcal{C}$, que permita describir la semántica que nos ofrece la teoría de categorías; los objetos son tipos de datos y los morfismos son las funciones entre tipos, la composición está definida como la composición de funciones: $g \circ f := \lambda z.g(fz)$ y la semántica operacional del sistema está determinada por la conmutatividad de los diagramas que usamos para programar con categorías.
Al extender el sistema F ya no interesa hablar de categorías en el sentido estricto sino de trabajar con ellas como categorías de tipos, para ésto supondremos que existen las categorías de tipos pero para más detalles se puede revisar [16].

Un funtor $F : \mathcal{C} \to \mathcal{C}$ será utilizado para pasar de un tipo a otro, utilizaremos los funtores que transforman un tipo B en un tipo $F[X := B]$ y serán denotados mediante la abstracción λXF, ésta abstracción será la forma en que se representarán tanto a las álgebras iniciales mediante μXF como a las coálgebras finales con νXF. Éstas abstracciones no son formalmente funtores ya que sólo se puede describir su comportamiento en los objetos y no en los morfismos.
Al definir un sistema que incluye constructores para tipos (co)inductivos, también sería posible incluir la (co)recursión y la (co)iteración. Para definir funciones mediante estos principios, es necesario que las variables de tipo figuren positivamente en los términos μ y ν. Es decir que si en F, X figura sólo positivamente entonces se pueden construir los tipos μXF o νXF; normalmente ésta es la forma de trabajar, usar las presencias positivas de

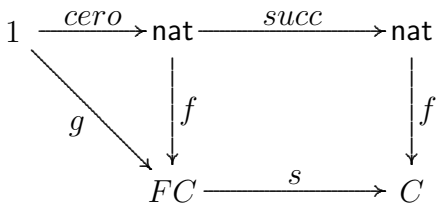
variables en F como es desarrollado, por ejemplo, en [21]. Una variable figura positivamente dentro de un tipo si ésta no figura a la izquierda de un número par de flechas de función, por ejemplo en los siguientes X es positiva: $X + R$, $S \times X$ y $T \to X$ y viceversa para un tipo negativo por ejemplo en: $X \to T$ y $(S + X) \to T$.

Las presencias positivas de una variable en el funtor garantizan la característica del propio funtor o en otras palabras la monotonía de él, como cuando se modeló a los objetos (co)inductivos mediante puntos fijos, ahí era necesario que el operador que definía a ese conjunto fuera monótono: $\forall X \forall Y.(X \to Y) \to FX \to FY$. Ésta segunda idea será la que adoptaremos en nuestra extensión del sistema F, para asegurar la monotonía del funtor será necesario el uso de términos que representan la propiedad del funtor λXF sobre morfismos, es decir se anexará un *testigo de monotonía* $\mathsf{map} : F \,\mathsf{mon}\, X$ en un contexto dado; este término se define mediante: $\mathsf{map} := \forall X \forall Y.(X \to Y) \to F \to F[X := Y]$ que expone el hecho de que el funtor λXF es monótono con respecto a X. Por lo tanto cada funtor debe tener un testigo de su monotonía $\langle \mu XF, \mathsf{map} \rangle$ y $\langle \nu XF, \mathsf{map} \rangle$, este testigo se encarga de hacer que la abstracción funcione adecuadamente sobre morfismos, haciendo que las abstracciones sean funtores.
Esta forma de definir un funtor se asemeja a la usada en el lenguaje de programación funcional Haskell, en este lenguaje es posible definir una clase que describa un funtor en donde se puede apreciar la capacidad del funtor, es una función entre categorías y una función que permite ver como trabaja sobre morfismos:

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

La correspondencia entre las álgebras iniciales con μXF y de las coálgebras finales con νXF y de éstas con las diálgebras permitirá que el sistema F anexe constructores para las diálgebras e incluir ambos tipos de datos.
En la sección dedicada a la teoría de categorías se dio la definición de las diálgebras y se explicó que pueden generalizar a las álgebras y las coálgebras, por lo tanto generalizaremos a las F-álgebras iniciales que corresponden a los tipos inductivos con las F, G-diálgebras y dualmente las F-coálgebras finales que corresponden a los coinductivos con las G, F-diálgebras. Una razón para utilizar las diálgebras radica en el hecho de que las F-álgebras iniciales no pueden definir un constructor del tipo coproducto, es decir, una suma no puede ser definida de manera natural, pero en cambio las F, G-diálgebras al ser más poderosas, definen tanto el tipo coproducto como el del producto como constructores, aunque serán utilizadas más adelante como funciones primitivas.
Para hacer notar la ventaja de utilizar las diálgebras veamos un ejemplo, el más claro y sencillo es el de los números naturales, en las F-álgebras éstos están definidos mediante $\mathsf{nat} = \mu X.1 + X$ mientras que en la F, G-diálgebra mediante $\mathsf{nat} = \mu X.(1, X)$. La segunda definición es más clara y mas fácil de manejar, con ella podemos tener los diagramas conmutativos por separado para cada uno de los constructores, es decir de los objetos básicos y las funciones constructoras, y en el caso de un tipo de dato coinductivo tendremos por separado a cada destructor. A ésta forma de separar las partes de los funtores y por consecuencia los constructores o destructores, la llamaremos uso de *múltiples constructores*.
Los múltiples constructores permiten un sistema más claro para definir tipos, del ejemplo anterior la definición de números naturales expuesto con diagramas conmutativos vista en

el capítulo anterior, queda de una mejor apariencia que ayuda a una mayor comprensión:



Ésta ventaja permite manejar los elementos de los pares y los copares separados, sin necesidad de ocupar funciones externas; así también el manejo de la monotonía se ve simplificada con el uso de múltiples constructores, el testigo de monotonía correspondiente a cada tipo lo llamaremos map_i, ejemplificaremos esta característica más adelante con los tipos de datos.

Hasta este momento podemos imaginar al sistema extendido, utilizando los pares y copares para codificar los morfismos, pero trabajar así requiere de funciones que manejen estas estructuras, es decir inyecciones, proyecciones, etc. Recordando a los tipos de datos inductivos, éstos tienen objetos básicos y funciones constructoras, categóricamente estos tipos de datos están modelados por las álgebras iniciales $\langle \mu XF, \mathsf{in}_F \rangle$ y los morfismos in_F están codificados en copares; ésta estructura no es cómoda ya que para poder trabajar con ella es necesario hacer análisis de casos e inyecciones, de ahí otra ventaja para usar diálgebras.

En resumen la extensión del sistema introduciendo los constructores para las diálgebras permite manejar a cada parte del tipo (co)inductivo por separado, cada uno tiene una parte del funtor F_i, su testigo de monotonía map_i y su correspondiente constructor in_i y/o destructor out_i.

3.2. Extensión del sistema F

Como vimos en la sección dedicada al cálculo lambda, el sistema F tiene un poder expresivo muy grande, definimos tipos recursivos, la iteración y la recursión; pero ahora hemos dicho que lo extenderemos para poder agregar como tipos a los operadores de punto fijo que definen los conjuntos inductivos y los coinductivos, es decir μXF y νXF y así poder construir objetos (co)inductivos.
Ésta extensión se hará en base a las diálgebras por las razones expuestas anteriormente, por lo tanto un tipo de dato inductivo estará determinado por los funtores que lo definen $\langle \mu X(F_1, \ldots, F_K), \mathsf{map}_i \rangle$ y por los morfismos $\mathsf{in}_{k,i}$ que determinan el constructor correspondiente al F_i del múltiple constructor del funtor. De igual manera un tipo de dato coinductivo está determinado por $\langle \nu X(F_1, \ldots, F_K), \mathsf{map}_i \rangle$ y por los morfismos $\mathsf{out}_{k,i}$. Para representar las funciones de (co)iteración y de (co)recursión será necesario conformarlas mediante los testigos de monotonía, las funciones de paso correspondientes a cada uno de los constructores o destructores según sea el caso, y el término al que se le aplicará la función, de ésta manera podremos tener en cada momento tanto los testigos como las funciones de paso para el tipo de dato con el que se está trabajando.

A partir de la definición del sistema F dada en la pagina 32, éste se extiende de la siguiente manera:

$$\mathsf{T} ::= \ldots \mid \mu X(F_1, \ldots F_k) \mid \nu X(F_1, \ldots F_k)$$

$$t ::= \ldots \mid \mathsf{It}(\vec{m}, \vec{s}, t) \mid \mathsf{Rec}(\vec{m}, \vec{s}, t) \mid \mathsf{in}_{k,i}\ t \mid \mathsf{CoIt}(\vec{m}, \vec{s}, t) \mid \mathsf{CoRec}(\vec{m}, \vec{s}, t) \mid \mathsf{out}_{k,i}\ t$$

La tipificación está dada por las siguientes reglas:

- Morfismo del álgebra inicial o doblez del mínimo punto fijo:

$$\frac{\Gamma \vdash t : F_i[X := \mu X(F_1, \ldots F_k)]}{\Gamma \vdash \mathsf{in}_{k,i}\, t : \mu X(F_1, \ldots F_k)}$$

- Iteración:

$$\frac{\begin{array}{l}\Gamma \vdash t : \mu X(F_1, \ldots F_k) \\ \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \;\; 1 \leq i \leq k \\ \Gamma \vdash s_i : F_i[X := B] \to B \;\; 1 \leq i \leq k\end{array}}{\Gamma \vdash \mathsf{It}_k(\vec{m}, \vec{s}, t) : B}$$

- Recursión primitiva:

$$\frac{\begin{array}{l}\Gamma \vdash t : \mu X(F_1, \ldots F_k) \\ \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \;\; 1 \leq i \leq k \\ \Gamma \vdash s_i : F_i[X := \mu X(F_1, \ldots F_k) \times B] \to B \;\; 1 \leq i \leq k\end{array}}{\Gamma \vdash \mathsf{Rec}_k(\vec{m}, \vec{s}, t) : B}$$

- Coiteración:

$$\frac{\begin{array}{l}\Gamma \vdash s_i : B \to F_i[X := B] \;\; 1 \leq i \leq k \\ \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \;\; 1 \leq i \leq k \\ \Gamma \vdash t : B\end{array}}{\Gamma \vdash \mathsf{CoIt}_k(\vec{m}, \vec{s}, t) : \nu X(F_1, \ldots F_k)}$$

- Correcursión primitiva:

$$\frac{\begin{array}{l}\Gamma \vdash s_i : B \to F_i[X := \nu X(F_1, \ldots F_k) + B] \;\; 1 \leq i \leq k \\ \Gamma \vdash m_i : F_i \,\mathsf{mon}\, X \;\; 1 \leq i \leq k \\ \Gamma \vdash t : B\end{array}}{\Gamma \vdash \mathsf{CoRec}_k(\vec{m}, \vec{s}, t) : \nu X(F_1, \ldots F_k)}$$

- Morfismo de la coálgebra final o desdoblez del máximo punto fijo:

$$\frac{\Gamma \vdash r : \nu X(F_1, \ldots F_k)}{\Gamma \vdash \mathsf{out}_{k,i}\, r : F_i[X := \nu X(F_1, \ldots F_k)]}$$

Y la semántica operacional con reducciones β es la siguiente:

$$\mathsf{It}_k(\vec{m}, \vec{s}, \mathsf{in}_{k,i}\ t) \quad \mapsto_\beta \quad s_i(m_i(\lambda x.\, \mathsf{It}_k(\vec{m}, \vec{s}, x))\, t)$$

$$\mathsf{Rec}_k(\vec{m}, \vec{s}, \mathsf{in}_{k,i}\ t) \quad \mapsto_\beta \quad s_i(m_i(\langle \mathsf{Id}, \lambda z.\, \mathsf{Rec}_k(\vec{m}, \vec{s}, x)\rangle)\, t)$$

$$\mathsf{out}_{k,i}\ \mathsf{CoIt}_k(\vec{m},\vec{s},t) \ \ \mapsto_\beta \ \ m_i(\lambda z.\,\mathsf{CoIt}_k(\vec{m},\vec{s},t))(s_i\,t)$$

$$\mathsf{out}_{k_i}\ \mathsf{CoRec}_k(\vec{m},\vec{s},t) \ \ \mapsto_\beta \ \ m_i([\mathsf{Id},\lambda z.\,\mathsf{CoRec}_k(\vec{m},\vec{s},z)](s_i\,t)$$

3.2.1. Ejemplos en el sistema de tipos

Para definir las funciones $f : \mu X(F_1,\ldots,F_K) \to B$ ó $f : A \to \nu X(F_1,\ldots,F_K)$ se hará por medio de la (co)iteración y la (co)recursión. A partir del principio de iteración se puede asegurar que para la función f hay un programa que la defina si está determinada por las siguientes ecuaciones, es decir que incluya a las funciones de paso y a los testigos de monotonía:

$$\begin{array}{rcl} f(\mathsf{in}_{k,1}\,x) & = & s_1\,(m_1\,f\,x) \\ & \vdots & \\ f(\mathsf{in}_{k,k}\,x) & = & s_k\,(m_k\,f\,x) \end{array}$$

donde $s_i : F_i[X := B] \to B$ es la función de paso correspondiente al funtor F_i y $m_i : F_i\,\mathsf{mon}\,X, 1 \le i \le k$ es el testigo de monotonía. Con todas estas condiciones la función queda definida como: $f := \lambda z.\,\mathsf{It}_k(\vec{m},\vec{s},z)$ y la reducción de este programa es:

$$f(\mathsf{in}_{k,i}\,x) \to^+_\beta s_i\,(m_i\,f\,x)$$

Para definir una función utilizando el principio de recursión primitiva es necesario que la función a definir esté determinada por las siguientes ecuaciones:

$$\begin{array}{rcl} f(\mathsf{in}_{k,1}\,x) & = & s_1\ (m_1\ \langle \mathsf{Id}, f\rangle\ x) \\ & \vdots & \\ f(\mathsf{in}_{k,k}\,x) & = & s_k\ (m_k\ \langle \mathsf{Id}, f\rangle\ x) \end{array}$$

donde $s_i : F_i[X := \mu X(F_1,\ldots,F_k) \times B] \to B$ es la función de paso correspondiente al funtor F_i, la función queda definida como: $f := \lambda z.\,\mathsf{Rec}_k(\vec{m},\vec{s},z)$.
Dualizando estos principios y definiciones para definir una función $f : A \to \nu X(F_1,\ldots,F_K)$ es necesario que se satisfagan las ecuaciones:

$$\begin{array}{rcl} \mathsf{out}_{k,1}(fx) & = & (m_1\ f)(s_1\ x) \\ & \vdots & \\ \mathsf{out}_{k,k}(fx) & = & (m_k\ f)(s_k\ x) \end{array}$$

donde $s_i : A \to F_i[X := A]$ es la función de paso correspondiente a la función y $m_i : F_i\,\mathsf{mon}\,X, 1 \le i \le k$ es el testigo de monotonía. La función queda definida como: $f := \lambda z.\,\mathsf{CoIt}_k(\vec{m},\vec{s},z)$. Para definir una función utilizando el principio de correcursión primitiva es necesario que la función a definir esté determinada por las siguientes ecuaciones:

$$\begin{array}{rcl} \mathsf{out}_{k,1}(fx) & = & (m_1\ [\mathsf{Id}, f])(s_1\ x) \\ & \vdots & \\ \mathsf{out}_{k,k}(fx) & = & (m_k\ [\mathsf{Id}, f])(s_k\ x) \end{array}$$

donde $s_i : F_i[X := \nu X(F_1, \ldots, F_k) + A] \to A$ es la función de paso correspondiente a la función y por tanto la función queda definida como: $f := \lambda z.\, \mathsf{CoRec}_k(\vec{m}, \vec{s}, z)$.

A continuación se presentan algunos de los ejemplos vistos en el capítulo anterior y unos más programados bajo la extensión dada. Cada uno de ellos deja ver lo útil que es el manejo de cláusulas. La definición de los constructores y destructores es más clara gracias a que no se utilizan inyecciones o proyecciones, es decir, cada una de éstas son funciones: $\lambda x.\, \mathsf{in}_{k,i}\ x$ o $\lambda x.\, \mathsf{out}_{k,i}\ x$ facilitar la lectura y el manejo lo más posible de $\mathsf{in}_{k,i}$ y $\mathsf{out}_{k,i}$.
Así mismo se puede ver que las definiciones para el manejo de la monotonía están separadas, modularizadas, para que finalmente la definición de funciones (co)inductivas sea limpia e intuitiva. Dentro de estos ejemplos también se utilizarán las variables anónimas, estas variables se denominan así debido a que dentro de un término lambda no son relevantes, como por ejemplo $\lambda u.x$, en este caso se escribirá $\lambda\_.x$.

Ejemplo 1 Tipo Unitario y Tipo Vacío

Éstos dos tipos no han sido explicados durante el trabajo realizado, pero son importantes dado que son bastante utilizados, también son llamados tipos degenerados.
El tipo unitario solamente contempla a un habitante $\star$ y está definido dentro del sistema mediante $1 := \nu X()$, generalmente es usado para definir los objetos básicos de algún tipo o para manejar errores, $\mathsf{error} = \mathsf{inl}\,\star$.
El tipo vacío, como su nombre lo indica no contiene elementos, está definido mediante: $\mathsf{void} := \mu X()$. Es utilizado para indicar que una función diverge o para indicar algún otro tipo de error o excepción.

Ejemplo 2 Booleanos

El tipo de dato booleano sólo puede ser alguno de los siguientes: true o false, por lo tanto se define de la siguiente manera: $\mathsf{bool} := \mu X(1, 1)$.
Los testigos de monotonía son muy simples: $\mathsf{map}_1 := \mathsf{map}_2 := \lambda f \lambda x.x$.
Los únicos elementos del tipo son:

- $\mathsf{true} := \mathsf{in}_{2,1}\,\star$
- $\mathsf{false} := \mathsf{in}_{2,2}\,\star$

Este tipo de dato es muy usual, generalmente se usa cuando se quiere definir una función condicionada, es decir, el clásico if, esta función tiene el tipo $\mathsf{bool} \to A \to A \to A$, el tipo del caso then y el caso else son el mismo. Se define de la siguiente manera:

$$\mathsf{if} - \mathsf{then} - \mathsf{else} := \lambda z \lambda x \lambda y.\, \mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda\_.x, \lambda\_.y, z)$$

donde los argumentos que reciben las funciones de paso no deben ser iguales a x o y respectivamente.

Ejemplo 3 Números Naturales

Los números naturales han sido un ejemplo muy utilizado a lo largo de este trabajo, ahora aparecen definidos de la siguiente manera: $\mathsf{nat} := \mu X(1, X)$.
Los testigos de monotonía son: $\mathsf{map}_1 := \lambda f \lambda x.x$, $\mathsf{map}_2 := \lambda x.x$.
Este tipo de dato tiene como constructores al cero como objeto básico y la función sucesor:

- $0 : \mathsf{nat},\ 0 := \mathsf{in}_{2,1} \star$
- $\mathsf{succ} : \mathsf{nat} \to \mathsf{nat},\ \mathsf{succ} := \mathsf{in}_{2,2}$

Además de poder construir un número natural a partir de cualquiera previamente construido, podemos destruir cualquier número natural existente mediante la función predecesor, es decir a un número le quitamos la unidad:

$$\mathsf{pred} : \mathsf{nat} \to 1 + \mathsf{nat} \quad \mathsf{pred}\ 0 = \mathsf{error} \quad \mathsf{pred}\,(\mathsf{succ}\, n) = \mathsf{inr}\ \ n$$

por lo tanto definimos la función predecesor utilizando recursión:

$$\mathsf{pred} := \lambda n.\, \mathsf{Rec}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda_0, \lambda v.\, \mathsf{fst}\, v, n)$$

Ésto es debido a que si se modelara con iteración, el término asociado sería un coproducto que a su vez involucría a un par, el cual no respetaría el uso de múltiples constructores.
A continuación veamos ejemplos de funciones que tienen por dominio a nat:

- $\mathsf{add} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat},\ \ \mathsf{add} := \lambda n \lambda m.\, \mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda\_.m, \mathsf{succ}, n).$
- $\mathsf{prod} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat},\ \ \mathsf{prod} := \lambda n \lambda m.\, \mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \mathsf{in}_{2,1}, \lambda y.\mathsf{sum}\, y\, m, n).$

Ejemplo 4 Listas finitas
Las listas finitas con objetos de tipo A se definen mediante: $\mathsf{list}(A) := \mu X(1, A \times X)$. Los testigos de monotonía están definidos con los siguientes términos: $\mathsf{map}_1 := \lambda f \lambda x.x,\ \ \mathsf{map}_2 := \lambda f \lambda x.\langle \mathsf{fst}\, x, f\ \ \mathsf{snd}\, x\rangle$. Para construir las listas finitas es necesario tener una lista básica que es la vacía y un constructor que agregue un elemento al inicio de una lista:

- $\mathsf{nil} : \mathsf{list}(A), \mathsf{nil} := \mathsf{in}_{2,1} \star$
- $\mathsf{cons} : A \times \mathsf{list}(A) \to \mathsf{list}(A), \mathsf{cons} := \mathsf{in}_{2,2}$

Las listas pueden ser destruidas en cabeza y cola:

$$\mathsf{head} : \mathsf{list}(A) \to 1 + A \quad \mathsf{tail} : \mathsf{list}(A) \to 1 + \mathsf{list}(A)$$

observemos que la lista vacía es una lista que no tiene ni cabeza ni cola: la cabeza devuelve un objeto del tipo 1 si tratamos de obtener la cabeza de la lista vacía, es decir $\mathsf{head}(\mathsf{nil}) = \mathsf{error}$, sino obtendremos un objeto de tipo A y la cola es la función que devuelve un objeto del tipo 1 si tratamos de obtener la cola de la lista vacía, es decir $\mathsf{tail}(\mathsf{nil}) = \mathsf{error}$, sino una lista finita. Del capítulo anterior tomaremos las funciones que definimos para las listas para definirlas bajo el sistema:

- $\mathsf{long} : \mathsf{list}(A) \to \mathsf{nat},$
  $\mathsf{long} := \lambda x.\, \mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda\_,0, \lambda y.\, \mathsf{succ}(\mathsf{snd}\, y), x)$
- $\mathsf{app} : \mathsf{list}(A) \to \mathsf{list}(A) \to \mathsf{list}(A),$
  $\mathsf{app} := \lambda x.\, \mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda y \lambda z.z, \lambda u \lambda v.\, \mathsf{cons}\langle \mathsf{fst}\, u, (\mathsf{snd}\, u)v\rangle, x)$
- $\mathsf{rev} : \mathsf{list}(A) \to \mathsf{list}(A),$
  $\mathsf{rev} := \lambda x.\, \mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \mathsf{nil}, \lambda z.\mathsf{app}(\mathsf{snd}\, z)(\mathsf{cons}\langle \mathsf{fst}\, z, \mathsf{nil}\rangle), x)$

- $\mathsf{maplist} : \forall X \forall Y.(X \to Y) \to \mathsf{list}(X) \to \mathsf{list}(Y)$,
  $\mathsf{maplist} := \lambda f \lambda x.\, \mathsf{It}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda\_.\, \mathsf{nil}, \lambda v.\, \mathsf{cons}\langle(f(\mathsf{fst}\, v)), (\mathsf{snd}\, v)\rangle, x)$

Ejemplo 5 Streams
Las listas infinitas de objetos de tipo A, que hemos llamado $\mathsf{stream}(A)$, están definidas mediante: $\mathsf{stream}(A) := \nu X(A, X)$ Los testigos de monotonía están determinados por los siguientes términos: $\mathsf{map}_1 := \lambda f \lambda x.x$, $\mathsf{map}_2 := \lambda f.f$ Las listas infinitas se destruyen en cabeza y cola para poder ser observadas:

- $\mathsf{head} : \mathsf{stream}(A) \to A$ $\mathsf{head} := \mathsf{out}_{2,1}$
- $\mathsf{tail} : \mathsf{stream}(A) \to \mathsf{stream}(A) \to \mathsf{stream}(A)$ $\mathsf{tail} := \mathsf{out}_{2,2}$

También es posible construir un stream a partir de un stream s dado, si se agrega un elemento al inicio de s obtendremos un nuevo stream: $\mathsf{cons} : A \times \mathsf{stream}(A) \to \mathsf{stream}(A)$
$\mathsf{cons} := \lambda x.\, \mathsf{CoRec}_2(\mathsf{map}_1, \mathsf{map}_2, \mathsf{fst}, \lambda z.\, \mathsf{inl}(\mathsf{snd}\, z), x)$.
Algunas funciones con streams son:

- Stream constante $\mathsf{cnt} : A \to \mathsf{stream}(A)$, $\mathsf{head}(\mathsf{cnt}\ a) = a$, $\mathsf{tail}(\mathsf{cnt}\ a) = \mathsf{cnt}\ a$
  $\mathsf{cnt} := \lambda x.\, \mathsf{CoIt}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda z.z, \lambda z.z, x)$
- Stream a partir de un número natural $\mathsf{from} : \mathsf{nat} \to \mathsf{stream}(\mathsf{nat})$,
  $\mathsf{head}(\mathsf{from}\ n) = n$, $\mathsf{tail}(\mathsf{from}\ n) = \mathsf{from}(\mathsf{succ}\, n)$
  $\mathsf{from} := \lambda x.\, \mathsf{CoIt}_2(\mathsf{map}_1, \mathsf{map}\, 2, \lambda z.z, \mathsf{succ}, x)$
- Map sobre un stream $\mathsf{mapstr} : \forall X \forall Y.(X \to Y) \to \mathsf{stream}(X) \to \mathsf{stream}(Y)$,
  $\mathsf{head}(\mathsf{mapstr} f\ x) = f(\mathsf{head}\, x)$, $\mathsf{tail}(\mathsf{mapstr} f\ x) = \mathsf{mapstr} f(\mathsf{tail}\, x)$
  $\mathsf{mapstr} f := \lambda z.\, \mathsf{CoIt}(\mathsf{map}_1, \mathsf{map}_2, \lambda x.f(\mathsf{head}\, x), \mathsf{tail}, z)$

Ejemplo 6 Árboles Binarios Infinitos con información en los nodos
Éstos árboles pueden ser definidos mediante: $\mathsf{InfBTree}A : \nu X(A, X \times X)$. La información de los nodos pertenece al conjunto A. Al ser una estructura infinita la destruiremos en dos partes, una para obtener la información del nodo y otra para obtener los dos descendientes de un nodo como un par:

- $\mathsf{rlabel} : \mathsf{InfBTree}A \to A$, $\mathsf{rlabel} := \mathsf{out}_{2,1}$
- $\mathsf{children} : \mathsf{InfBTree}A \to \mathsf{InfBTree}A \times \mathsf{InfBTree}$, $\mathsf{children} := \mathsf{out}_{2,2}$

Ejemplo 7 Árboles Binarios Finitos e Infinitos con información en los nodos
El ejemplo pasado destruye al árbol en la etiqueta del nodo raíz y un par que contiene a los dos hijos, para no trabajar con pares y tener un destructor para cada hijo se agrega esta estructura: $\mathsf{FinInfBTree}A := \nu X(A, 1 + X, 1 + X)$.
Para destruirlos tenemos los siguientes:

- $\mathsf{label} : \mathsf{FinInfBTree}A \to A$, $\mathsf{label} := \mathsf{out}_{3,1}$
- $\mathsf{lsubtree} : \mathsf{FinInfBTree}A \to 1 + \mathsf{FinInfBTree}$,
  $\mathsf{lsubtree} := \mathsf{out}_{3,2}$
- $\mathsf{rsubtree} : \mathsf{FinInfBTree}A \to 1 + \mathsf{FinInfBTree}$,
  $\mathsf{rsubtree} := \mathsf{out}_{3,3}$

Ejemplo 8 Árboles con ramas finitas y profundidad potencialmente infinita
Otro caso de árboles infinitos es restringiendo el número de hijos de un nodo a un número finito pero dejando libre la posibilidad de tener una profundidad infinita, están definidos mediante: $\mathsf{PInfTree}A := \nu X(A, \mathsf{list}(X))$.
Para facilitar la destrucción de éstos árboles se tiene una lista finita que contiene a los hijos de cada nodo, ya que no es posible determinar para cada nodo el número de hijos, la lista finita permite un manejo fácil y rápido. Los destructores son:

- $\mathsf{rlabel} : \mathsf{PInfTree}A \to A$
- $\mathsf{lsubtrees} : \mathsf{PInfTree}A \to \mathsf{list}(\mathsf{PInfTree}A)$

Los términos para la monotonía son: $\mathsf{map}_1 := \lambda f \lambda x.x$, $\mathsf{map}_2 := \mathsf{maplist}$, éste último es el correspondiente a los términos utilizados para las listas. Este ejemplo es muy importante debido a que hemos llegado a un punto en donde la inducción y la coinducción se mezclan, estamos utilizando la inducción para definir el testigo de monotonía de un tipo de dato coinductivo.

Un ejemplo de función sobre estos árboles es el cambio de las etiquetas de los nodos, dada una función que cambia los objetos de A en objetos de C $f : A \to C$ podemos cambiar las etiquetas:
$\mathsf{maptree} : (A \to C) \to \mathsf{PInfTree}A \to \mathsf{PInfTree}C$
$\mathsf{rlabel}(\mathsf{maptree}\ f\ t) = f(\mathsf{rlabel}\ t)$
$\mathsf{lsubtrees}(\mathsf{maptree}\ f\ t) = \mathsf{maplist}\ \mathsf{maptree}(\mathsf{lsubtrees}\ t)$
La función queda definida mediante el siguiente término:
$\mathsf{maptree}f := \lambda x.\, \mathsf{Colt}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda y.f(\mathsf{rlabel}\ y), \mathsf{lsubtrees}, x)$

Ejemplo 9 Árboles potencialmente infinitos con ramas etiquetadas
Este es un ejemplo que se describe en [1] en el ámbito de la teoría de categorías. Éstos árboles contienen información en las ramas, las ramas pueden variar en número o ser infinitas para cada nodo en el árbol por esto que se requiera de un destructor adecuado para manejar las ramas que nos llevan a los subárboles en cada nodo. El tipo de dato está definido por $\mathsf{BLTree}A := \nu X(\mathsf{list}(A \times X))$, el cual sólo tiene un destructor que obtiene una lista de los subárboles:

$$\mathsf{lsb} : \mathsf{BLTree}A \to \mathsf{list}(A \times \mathsf{BLTree}A) \quad \mathsf{lsb} := \mathsf{out}$$

Estos subárboles están conformados por una etiqueta para la rama de la que se desprende así como una lista de sus propios hijos. Las listas contienen a los subárboles ordenados de izquierda a derecha. El testigo para la monotonía es:

$$\mathsf{map} : \forall X \forall Y.(X \to Y) \to \mathsf{list}(A \times X) \to \mathsf{list}(A \times Y)$$
$$\mathsf{map}\ f\ \mathsf{nil} := \mathsf{nil} \quad \mathsf{map}\ f(\mathsf{cons}\langle\langle a, x\rangle, xs\ \rangle = \mathsf{cons}\langle\langle a, fx\rangle, (\mathsf{map}\ fxs)\rangle$$

Este término está utilizando la función maplist junto con la función que expresa la monotonía en las listas de las etiquetas $\forall X \forall Y.(X \to Y) \to (A \times X) \to (A \times Y)$. Se requiere del apoyo de las listas, es decir un objeto inductivo.

Definamos algunas funciones de búsqueda sobre éstos árboles, utilizaremos funciones auxiliares para el manejo de los subárboles:

- Búsqueda a lo ancho (BFS)
  La búsqueda a lo ancho está determinada por la siguiente función:

$$\mathsf{bfs} : \mathsf{BLTree}A \to A^{\infty} \quad \mathsf{bfs} := \mathsf{bfl} \circ \mathsf{lsb}$$
$$\mathsf{head}\,\mathsf{bfl}\,t = \mathsf{inr}(\mathsf{fst}(\mathsf{head}\,t))$$
$$\mathsf{tail}\,\mathsf{bfl}\,t = \mathsf{inr}\,\mathsf{bfl}(\mathsf{app}(\mathsf{tail}\,f)\;(\mathsf{lsb}(\mathsf{snd}(\mathsf{head}\,t))))$$

  Donde el tipo A^{∞} corresponde a las listas finitas e infinitas ya que los nodos pueden tener una infinidad de descendientes. La función bfl permite manejar las listas de árboles $\mathsf{bfl} : \mathsf{list}(A \times \mathsf{BLTree}A) \to A^{\infty}$ está definida mediante coiteración debido a que, compuesta con el destructor, permitirá el manejo de los subárboles que son una estructura coinductiva:

$$\mathsf{bfl} := \lambda x.\, \mathsf{Colt}_2(\mathsf{map}_1, \mathsf{map}_2, s_1, s_2, x)$$

  En donde:

$$\mathsf{map}_1 := \lambda f \lambda x.x \quad \mathsf{map}_2 := \lambda f \lambda x.\, \mathsf{case}(x, y.\, \mathsf{inl}\, y, z.\, \mathsf{inr}\, f z)$$
$$s_1 : \mathsf{list}(A \times \mathsf{BLTree}A) \to 1 + A \quad s_1 := \lambda w.\, \mathsf{inr}(\mathsf{fst}(\mathsf{head}\, w))$$
$$s_2 : \mathsf{list}(A \times \mathsf{BLTree}A) \to 1 + \mathsf{list}(A \times \mathsf{BLTree}A)$$
$$s_2 := \lambda w.\, \mathsf{inr}(\mathsf{app}\;(\mathsf{tail}\, w)(\mathsf{lsb}(\mathsf{snd}(headw))))$$

- Búsqueda a lo profundo (DFS)
  Ésta búsqueda es igual a la anterior solamente sufre una pequeña modificación que radica en el orden en que son agregados los siguientes nodos a visitar dentro del árbol, en BFS dada una lista de nodos éstos deben visitarse primero y posteriormente los subárboles de cada uno de ellos como se ve en la segunda función de paso de bfl. En DFS sólo es necesario agregar los subárboles de cada nodo inmediatamente después de haber visitado al padre para realizar una búsqueda siguiendo una rama, por lo tanto sólo se modificará s_2:

$$s_2 : \mathsf{list}(A \times \mathsf{BLTree}A) \to 1 + \mathsf{list}(A \times \mathsf{BLTree}A)$$
$$s_2 := \lambda w.\, \mathsf{inr}(\mathsf{app}\;(\mathsf{lsb}(\mathsf{snd}(\mathsf{head}\, w)))(\mathsf{tail}\, w))$$

Ejemplo 10 Autómatas determinísticos

Este ejemplo utiliza las ideas de [9]. Una máquina finita de estados determinista es muy común dentro de la teoría de autómatas; en ellas, dada una entrada y un estado existe una única transición al estado siguiente.
Un autómata tiene un alfabeto de entrada Σ y uno de salida B que lo definen: $\mathsf{daut}(\Sigma, B) := \nu X.(\Sigma \to X, B)$. Dado que es un tipo de dato coinductivo sus destructores son:

- $\mathsf{next} : \mathsf{daut}(\Sigma, B) \to (\Sigma \to \mathsf{daut}(\Sigma, B))$
- $\mathsf{obs} : \mathsf{daut}(\Sigma, B) \to B$

Estos destructores permiten obtener el siguiente estado a partir de un estado y una entrada y permiten observar el comportamiento del autómata, es decir su salida.

Los testigos de monotonía están determinados por los siguientes términos: $\mathsf{map}_1 := \lambda f \lambda g \lambda x.f(gx)$, $\mathsf{map}_2 := \lambda f \lambda x.x$. Podemos decir que éstos autómatas son los llamados autómatas de Moore, se definen por un par $M = \langle \delta, o \rangle$ donde $\delta : Q \to \Sigma \to Q$ es una función de transición y $o : Q \to B$ una función para las observaciones. Los elementos de este tipo de dato coinductivo son funciones que permiten ver el comportamiento del autómata a partir de algún estado: $\mathsf{beh}(q) : \Sigma^* \to B$. Por lo tanto para codificar un autómata será por medio de una función: $\mathsf{caut} := \lambda z.\, \mathsf{Colt}_2(\mathsf{map}_1, \mathsf{map}_2, \delta, o, z)$ que puede ser destruida de la misma manera, podemos obtener el siguiente estado o podemos observar qué sucede con el autómata en cierto momento:

- $\mathsf{next\ caut}\ q = \lambda x.\mathsf{caut}((\delta\ q)x)$
- $\mathsf{obs\ caut}\ q = o\, q$

Para codificar el comportamiento del autómata, dada la función de comportamiento $\mathsf{beh}(q) : \Sigma^* \to B$ tenemos la función cbeh definida mediante coiteración, ésta operación se apoya en las listas: $\mathsf{cbeh} := \lambda z.\, \mathsf{Colt}_2(\mathsf{map}_1, \mathsf{map}_2, \lambda f \lambda a \lambda w.f(a.w), \lambda g.g(\epsilon), z)$ donde $\epsilon,\ a.w$ son las operaciones nil y cons correspondientes a Σ^*. Los destructores para esta función son:

- $\mathsf{next}\ (\mathsf{cbeh\ beh}\ (q)) = \lambda a.\mathsf{cbeh}(\lambda w.\mathsf{beh}(q)(a.w))$
- $\mathsf{obs}\ (\mathsf{cbeh\ beh}\ (q)) = \mathsf{beh}(q)\ \epsilon$

Veamos algunos ejemplos de funciones con autómatas, tomaremos a los conjuntos Q, Σ como finitos y el conjunto de salidas como bool:

- Complemento de un autómata $\mathsf{comp} : \mathsf{daut}(Z, \mathsf{bool}) \to \mathsf{daut}(Z, \mathsf{bool})$
  Simplemente estamos cambiando los estados finales por no finales y viceversa:

$$\begin{aligned} \mathsf{next}\ (\mathsf{comp}\ M) &= \mathsf{next}\ M \\ \mathsf{obs}\ (\mathsf{comp}\ M) &= \not\prime(\mathsf{obs}\ M) \end{aligned}$$

- Producto de dos autómatas $\mathsf{prod} : \mathsf{daut}(Z, \mathsf{bool}) \to \mathsf{daut}(Z, \mathsf{bool}) \to \mathsf{daut}(Z, \mathsf{bool})$
  El producto de dos autómatas puede ser hecho con tres distintas funciones, dependiendo del lenguaje deseado, es decir podemos intersectar, unir o restar lenguajes:

$$\begin{aligned} \mathsf{next}\ (\mathsf{prod}\ M_1\ M_2) &= \mathsf{prod}(\mathsf{next}\ M_1)(\mathsf{next}\ M_2) \\ \mathsf{obs}\ (\mathsf{prod}\ M_1\ M_2) &= \mathsf{obs}\ M_1)\ (\mathsf{obs}\ M_2) \\ \mathsf{obs}\ (\mathsf{prod}\ M_1\ M_2) &= \mathsf{obs}\ M_1)\mathsf{or}\ (\mathsf{obs}\ M_2) \\ \mathsf{obs}\ (\mathsf{prod}\ M_1\ M_2) &= \mathsf{obs}\ M_1)\ \ \not\prime(\mathsf{obs}\ M_2) \end{aligned}$$

3.2.2. Propiedades del sistema de tipos

La extensión del sistema F expuesta permite programar con tipos (co)inductivos, pero ¿cualquier programa desarrollado bajo él termina?, es decir ¿este sistema es seguro?.

Existen sistemas en los cuales se puede programar a pesar de no ser seguros, pero para el sistema propuesto comentaremos dos características que lo hacen seguro: la normalización fuerte y la preservación de tipos.
El sistema presentado en la sección pasada está basado en el trabajo hecho en [6], en él hay dos sistemas el primero MICT *Monotone Inductive and Coinductive Types* que es una extensión del sistema F que incluye a las álgebras iniciales y las coálgebras finales para obtener constructores de los tipos (co)inductivos; y el segundo MCICT *Monotone Clausular Inductive and Coinductive Types* en el cual se incluyen a las diálgebras en lugar de las (co)álgebras para permitir el uso de los constructores múltiples de ahí que un fragmento de éste último sea la base del sistema con la que culmina este trabajo. En [6] se encuentran las demostraciones correspondientes para los dos sistemas expuestos, para el que se dio en este trabajo se comentarán las demostraciones, a continuación se nombran las definiciones de éstas dos características:

- Normalización fuerte
  Todo término t correctamente tipado es fuertemente normalizable.
  Explicando esta definición podemos decir que no existe una secuencia de reducción infinita que comienza por t o que toda secuencia de reducción que comienza por t debe terminar, en otras palabras, la semántica operacional del sistema garantiza que un programa termine.

- Preservación de tipos
  Si bajo un contexto Γ se obtiene que el término r es del tipo A y además se sabe que bajo la reducción $\rightarrow_\beta$ se cumple que $r \rightarrow_\beta r'$ entonces del mismo contexto se puede decir que $\Gamma \vdash r' : A$.

La demostración de normalización fuerte para cualquier programa del sistema se basa en la semántica operacional definida. Por otro lado la prueba de preservación de tipos en los programas presenta un detalle que hace difícil una demostración de ello: la extensión del sistema F se hizo bajo la tipificación *à la Curry*, es decir, se maneja un polimorfismo implícito en donde no hay anotaciones de tipos. Si nuestro sistema tuviera una tipificación *à la Church* sería trivial la demostración, pero no es el caso; para consultar una demostración de preservación de tipos, como se dijo anteriormente se puede ver [6] ya que ésta va más allá de los límites de este trabajo.

3.2.3. Prototipo de lenguaje de programación funcional

La extensión del sistema F contiene términos muy complejos, a pesar de haber reducido el trabajo que implica el uso de la teoría de categorías para programar con tipos de datos (co)inductivos, el manejo de los términos del cálculo lambda es un poco tedioso, el sólo hecho de verificar una definición de función de los ejemplos anteriores se vuelve más compleja dependiendo del tipo, por ejemplo realizar alguna operación con alguno de los árboles infinitos o con algún autómata resulta en una serie de términos que no permiten tener una idea general de lo que se está programando. Para simplificar el uso de éstos términos haremos unas simplificaciones a la sintaxis del sistema, agregando lo que se conoce como *azúcar sintáctica* que es una forma de hacer más fácil el trabajo de un

programador, es decir hacer más práctica la forma de escribir programas. Este término fue introducido por Peter J. Landin para permitir más practicidad y sencillez en el momento de programar con el cálculo lambda, en pocas palabras la azúcar sintáctica hace más "dulce" la vida de un programador.

A continuación se presenta una propuesta de azúcar sintáctica para el sistema mostrado, trata de ser lo más intuitiva y fácil de recordar que se pueda, para que el programar con ella resulte sencillo y más practico, así mismo se trata de reflejar lo que se explicó en las primeras secciones del primer capítulo, es decir, encapsular las ideas iniciales e intuitivas de la inducción y la coinducción.

Para representar un tipo de dato inductivo $\mathcal{I}$, es necesario recordar que tiene constructores, es decir objetos básicos y funciones constructoras $c_i := \mathsf{in}_{k,i}$ y que con esto basta para definirlo, por lo tanto un tipo de dato inductivo $\mathcal{I} := \mu X(F_1, \ldots, F_n)$ se declara de la siguiente manera:

$$\begin{aligned} < \textit{typename} > \quad = \quad & \texttt{inductive}\ X\ \texttt{with constructors} \\ & c_1 : F_1 \to X \\ & \vdots \\ & c_n : F_n \to X \end{aligned}$$

Para definir funciones mediante iteración o recursión, $\mathsf{fun} : \mathcal{I} \to B$:

$$\mathsf{fun} := \lambda x.\, \mathsf{It}_k(\vec{m}, \vec{s}, x) \quad \mathsf{fun} := \lambda x.\, \mathsf{Rec}_k(\vec{m}, \vec{s}, x)$$

se hará por medio de la siguiente representación intercambiando `iterator` por `recursor` según sea el caso:

$$\begin{aligned} \mathsf{fun} \quad = \quad & \texttt{iterator of}\ \mathcal{I}\ \texttt{to}\ B\ \texttt{with steps} \\ & s_1 \\ & \vdots \\ & s_k \\ & \texttt{where}\ \mathsf{map}_1 = m_1, \ldots \mathsf{map}_k = m_k \end{aligned}$$

Para representar un tipo de dato coinductivo $\mathcal{C}$, es necesario recordar que solamente consta de destructores, es decir observaciones a los objetos dados $d_i := \mathsf{out}_{k,i}$, por lo tanto un tipo de dato coinductivo $\mathcal{C} := \nu X(F_1, \ldots, F_n)$ se declara de la siguiente manera:

$$\begin{aligned} < \textit{typename} > \quad = \quad & \texttt{coinductive}\ X\ \texttt{with destructors} \\ & d_1 : X \to F_1 \\ & \vdots \\ & d_n : X \to F_n \end{aligned}$$

Para definir funciones mediante coiteración o correcursión, $\mathsf{fun} : B \to \mathcal{C}$:

$$\mathsf{fun} := \lambda x.\, \mathsf{CoIt}_k(\vec{m}, \vec{s}, x) \quad \mathsf{fun} := \lambda x.\, \mathsf{CoRec}_k(\vec{m}, \vec{s}, x)$$

se hará por medio de la siguiente representación y de la misma manera que en los tipos inductivos sólo se cambiará `coiterator` por `corecursor`:

$$\begin{aligned} \mathsf{fun} \;=\; & \texttt{coiterator of } \mathcal{C} \texttt{ from } B \texttt{ with steps} \\ & \quad s_1 \\ & \quad \vdots \\ & \quad s_k \\ & \texttt{where } \mathsf{map}_1 = m_1, \ldots \mathsf{map}_k = m_k \end{aligned}$$

La semántica operacional está definida mediante:

Iteración: $\mathsf{fun}(c_i\, x) \to s_i(\mathsf{map}_i\ \mathsf{fun}\, x)$

Recursión: $\mathsf{fun}(c_i\, x) \to s_i(\mathsf{map}_i\langle \mathsf{Id}, \mathsf{fun}\rangle\, x)$

Coiteración: $d_i(\mathsf{fun}\, x) \to \mathsf{map}_i\ \mathsf{fun}(s_i\, x)$

Correcursión: $d_i(\mathsf{fun}\, x) \to \mathsf{map}_i[\mathsf{Id}, \mathsf{fun}](s_i\, x)$

Además del azúcar sintáctico agregado, los términos lambda cambiarán su sintaxis, se utilizará la usada por el lenguaje de programación funcional Haskell. En él se pueden expresar las funciones por medio de términos lambda, por ejemplo la función $\lambda x.x + 1$ se traduce en `\x -> x+1`.
Este último cambio en la sintaxis se debe a que el trabajo desarrollado está pensado para llevarlo a la práctica, extender Haskell mediante el sistema expuesto ayudaría a manejar los tipos (co)inductivos de una manera más clara. Por ejemplo, en Haskell, los tipos de datos se exponen con constructores y la definición está hecha por medio de disyunciones; si queremos definir el tipo de dato correspondiente a los árboles binarios con información en las hojas solamente damos los constructores correspondientes:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

En esta definición `Tree a` corresponde a X, `Leaf a` a a y `Branch (Tree a) (Tree a)` a $X\ X$ en la definición del tipo mediante $\mu X = A \times (X, X)$.
Al comparar esta definición con la que se puede dar por medio de álgebras iniciales: $\mathsf{tree}A = \mu X.(A + X \times X)$ y con la dada con diálgebras $\mathsf{tree}A = \mu X.(A, X \times X)$ podemos ver que se asemeja más a la que se puede dar con la primera.

Por otro lado, si deseamos obtener la información del tipo dato en el intérprete obtenemos lo siguiente:

```
Main> :info Tree
-- type constructor
data Tree a

-- constructors:
Leaf :: a -> Tree a
Branch :: Tree a -> Tree a -> Tree a
```

La información es presentada como en la definición de tipos que se usa en el sistema con azúcar sintáctica propuesto, se dan los constructores del tipo y se indica cuál es el tipo de cada uno. El agregar el azúcar sintáctico sugerido a Haskell haría que la definición con múltiples constructores sea más fácil de leer y además no haya cambio entre una definición de tipo hecha en un programa y la que el mismo intérprete pueda inferir a partir de ese programa, además de que es más fácil trabajar, teóricamente, con la definición que utiliza múltiples constructores.

En [15] y [3] se proponen extensiones de Haskell para tipos coinductivos mediante la definición `codata`, esta idea es semejante a la que se da en el azúcar sintáctico propuesto en este trabajo, tener un mecanismo que incluye propiamente a los tipos coinductivos mostrando los destructores.
A continuación veamos unos ejemplos bajo esta nueva sintaxis:

- Listas
  Para redefinir este tipo de dato tomemos su definición bajo el sistema: $\mathsf{list}(A) := \mu X(1, A \times X)$ agregando los constructores nil y cons obtenemos la definición con el azúcar sintáctica:

$$\begin{aligned} \mathsf{list}(A) \;=\; & \texttt{inductive}\ X\ \texttt{with constructors} \\ & \mathsf{nil} : 1 \to X \\ & \mathsf{cons} : A \times X \to X \end{aligned}$$

  Si deseamos definir una función bajo iteración, como por ejemplo maplist tenemos:

$$\begin{aligned} \mathsf{maplist}\ f \;=\; & \texttt{iterator of}\ \mathsf{list}(A)\ \texttt{to}\ \mathsf{list}(B)\ \texttt{with steps} \\ & s_1 = \texttt{\textbackslash\_ -> nil} \\ & s_2 = \texttt{\textbackslash v ->(f(fst v)):(snd v)} \\ & \texttt{where}\ \mathsf{map}_1 = \texttt{\textbackslash f,x -> x}\ \ \mathsf{map}_2 = \texttt{\textbackslash f,x -> (fst x,f(snd x))} \end{aligned}$$

  Finalmente veamos como se puede aplicar la siguiente función: maplist f l donde $f : A \to B$. Tomando $A = B = \mathsf{nat}$ escogemos una función sencilla $f =$ `\x -> x+1`

y una lista $l = \mathsf{cons}\langle 1, \mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\rangle\rangle\rangle$:

$$
\begin{aligned}
\mathsf{maplist}\ f\ l \rightarrow\ & s_2\Big((\texttt{\textbackslash f,x -> (fst x,f(snd x))})\ \mathsf{maplist}\ f\ \langle 1, \mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\rangle\rangle\rangle\Big)\\
\rightarrow\ & s_2\Big((1, \mathsf{maplist}\ f\ \ \mathsf{cons}\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\rangle\rangle)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((\texttt{\textbackslash f,x -> (fst x,f(snd x))})\ \mathsf{maplist}\ f\langle 2, \mathsf{cons}\langle 3, \mathsf{nil}\rangle\rangle\big)\big)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((2, \mathsf{maplist}\ f\ \ \mathsf{cons}\langle 3, \mathsf{nil}\rangle\big)\big)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((2, s_2\big((\texttt{\textbackslash f,x -> (fst x,f(snd x))})\ \mathsf{maplist}\ f\ \mathsf{cons}\langle 3, \mathsf{nil}\rangle\big)\big)\big)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((2, s_2\big((3, \mathsf{maplist}\ f\ \ \mathsf{nil}\big)\big)\big)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((2, s_2\big((3, (\texttt{\textbackslash\_ ->}\,\mathsf{nil})\ ((\texttt{\textbackslash f,x ->x})\ \mathsf{maplist}\ f\ *)\big)\big)\big)\big)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((2, s_2\big((3, \mathsf{nil}\big)\big)\big)\big)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((2, (\texttt{\textbackslash v ->(\textbackslash x -> x+1 y(fst v)):(snd v)})(3, \mathsf{nil})\big)\big)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((2, ((\texttt{\textbackslash x -> x+1})\ 3) : \mathsf{nil})\big)\Big)\\
\rightarrow\ & s_2\Big((1, s_2\big((2, 4 : \mathsf{nil})\big)\Big)\\
\rightarrow\ & s_2\Big((1, ((\texttt{\textbackslash x -> x+1})\ 2) : (4 : \mathsf{nil}))\Big)\\
\rightarrow\ & s_2\Big((1, (3 : (4 : \mathsf{nil})))\Big)\\
\rightarrow\ & ((\texttt{\textbackslash x -> x+1})\ 1 : (3 : (4 : \mathsf{nil})))\\
\rightarrow\ & (2 : (3 : (4 : \mathsf{nil})))
\end{aligned}
$$

- Árboles potencialmente infinitos con ramas etiquetadas
  Éstos árboles tienen la siguiente definición en el sistema, $\mathsf{BLTree}(A) := \nu X(\mathsf{list}(A \times X))$ y usa sólo un destructor para obtener la lista de subárboles lsb por lo tanto bajo el azúcar sintáctica tenemos:

$$
\begin{aligned}
\mathsf{BLTree}(A) \ =\ & \texttt{coinductive}\ X\ \texttt{with destructors}\\
& \mathsf{lsb} : \mathsf{BLTree}A \rightarrow \mathsf{list}(A \times \mathsf{BLTree}A)
\end{aligned}
$$

  La función BFS está definida mediante la composición: $\mathsf{bfs} := \mathsf{bfl} \circ \mathsf{lsb}$. La función a definir mediante coiteración es bfl que es la que maneja la lista de subárboles de cada nodo:

$$
\begin{aligned}
\mathsf{bfl} \ =\ & \texttt{coiterator of}\ A^\infty\ \texttt{from}\ \mathsf{list}(A \times \mathsf{BLTree}(A))\ \texttt{with steps}\\
& \quad s_1 = \texttt{\textbackslash w -> if (isnil w) then error else inr (fst(head w))}\\
& \quad s_2 = \texttt{\textbackslash w -> if (isnil w) then error else}\\
& \qquad\quad \texttt{(inr tail w ++ lsb(snd(head w)))}\\
& \texttt{where}\ \mathsf{map}_1 = \ \texttt{\textbackslash f,x -> x}\\
& \qquad\quad \mathsf{map}_2 = \ \texttt{\textbackslash f,x -> case(x, y.inl y, z.inr f z)}
\end{aligned}
$$

  Partimos del tipo de los árboles $\mathsf{list}(A \times \mathsf{BLTree}(A))$ y llegamos al tipo correspondiente a las listas infinitas o finitas, A^∞, ya que los hijos que pueda tener un nodo pueden ser o no infinitos y por tanto hay dos funciones de paso.

Veamos como se puede aplicar la función, recordemos que sólo se puede observar el objeto coinductivo que se obtiene después de aplicar la función en cuestión:

$$\begin{aligned}
\mathsf{head}(\mathsf{bfs}\ t) &\rightarrow (\mathsf{head}\ \mathsf{bfl} \circ \mathsf{lsb}\ t) \\
&\rightarrow \mathsf{head}(\mathsf{bfl}\ [(a_1,t_1),(a_2,t_2),\dots]) \\
&\rightarrow (\texttt{\textbackslash f,x -> x})\ \mathsf{bfl}\ (s_1[(a_1,t_1),(a_2,t_2),\dots]) \\
&\rightarrow (\texttt{\textbackslash f,x -> x})\ \mathsf{bfl}\ (\mathsf{inr}(\mathsf{fst}(a_1,t_1))) \\
&\rightarrow (\mathsf{inr}(\mathsf{fst}(a_1,t_1))) \\
&\rightarrow (\mathsf{inr}\ a_1)
\end{aligned}$$

$$\begin{aligned}
\mathsf{tail}(\mathsf{bfs}\ t) &\rightarrow (\mathsf{tail}\ \mathsf{bfl} \circ \mathsf{lsb}\ t) \\
&\rightarrow \mathsf{tail}(\mathsf{bfl}\ [(a_1,t_1),(a_2,t_2),\dots]) \\
&\rightarrow \mathsf{map}_2\ \mathsf{bfs}\big(\texttt{\textbackslash w -> if (isnil w) then error else} \\
&\qquad \texttt{(inr tail w ++ lsb(snd(head w)))}\ [(a_1,t_1),(a_2,t_2),\dots]\big) \\
&\rightarrow \mathsf{map}_2\ \mathsf{bfs}\big(\ \texttt{inr}\ ([(a_2,t_2),\dots] + +\texttt{lsb}t_1)\big) \\
&\rightarrow (\texttt{\textbackslash f,x -> case(x,y.inl y,z.inr f z)})\mathsf{bfs}\big(\texttt{inr}\ ([(a_2,t_2),\dots,\texttt{lsb}\ t_1])\big) \\
&\rightarrow \mathsf{bfs}\big(\texttt{inr}\ ([(a_2,t_2),\dots,\texttt{lsb}\ t_1])\big)
\end{aligned}$$

En estos ejemplos, puede apreciarse el esfuerzo y trabajo que implica trabajar con términos lambda, el azúcar sintáctico permite subir un nivel y hacer un trabajo limpio. La idea de llevar esta extensión al propio lenguaje permitiría trabajar en ese nivel, las derivaciones se dejarían a Haskell.

Bajo estas definiciones es claro ver el sentido dual de la inducción y la coinducción, se puede ver reflejada la idea principal de cada uno de ellos, permitiendo conjuntar la teoría que los define a través de una máscara dulce e intuitiva. Las propuestas para extender Haskell, además de conservar el alma de las definiciones (co)inductivas, permiten un desarrollo ideal sobre el lenguaje de programación además de un mejor aprovechamiento de sus cualidades.

Conclusiones y Trabajo Futuro

Este trabajo inició con una recopilación de las teorías que estudian los sistemas de tipos desde el punto de vista matemático, en el capítulo uno se reunieron las bases, se dieron definiciones y los términos más usados en este trabajo, esto con la idea de aportar una base que ayude a los científicos de la computación en su reafirmación de conocimientos de inducción para dar paso al dual de la inducción, la coinducción, así como una pequeña introducción a la teoría de puntos fijos y a la teoría de las categorías.
La teoría de categorías ha sido un soporte importante para comprender muchos de los trabajos realizados en este campo, es una de las herramientas matemáticas más sobresalientes que ayudan a ver de una manera más clara el comportamiento de objetos matemáticos, en particular para este trabajo permite estudiar a los tipos de datos. Al hacer énfasis en esta teoría se busca que más personas se interesen en ella y así poder aspirar a un mejor aprovechamiento de nuestra base matemática.

Recordando que en la introducción se plantearon algunas preguntas, éstas fueron respondidas, se discutió ampliamente la inducción, se explicó que el uso de la palabra inducción a veces es como un calificativo que reciben algunos términos al incluir la palabra inducción. Para explicar la coinducción se tomó como punto de partida a la inducción dejando claramente que cuando se hace referencia al dual de la inducción se está hablando de la coinducción; de la misma manera cuando se habla del dual de los constructores de un tipo inductivo, es inmediata la asociación de los tipos coinductivos los cuales son observados mediante las funciones destructoras.
Finalmente la extensión del sistema, dada en el último capítulo, permite responder a la pregunta que tiene más importancia: la de poder manejar objetos infinitos dentro de la computadora; esto se realiza mediante la evaluación perezosa. Además la extensión dada tiene una notable aportación: es un sistema que incluye tanto a la inducción como a la coinducción en todos sentidos, permite definir tipos de datos (co)inductivos así como las definiciones de los principios de estos tipos, es decir, la (co)iteración y la (co)recursión. Pocos son los lenguajes de programación que permiten el uso de la inducción y la coinducción, Haskell es uno de ellos, proporciona un ambiente en el que conviven ambos lo que lo hace apto para la implementación del sistema de tipos expuesto. Además la característica importante que liga al cálculo lambda con Haskell hace que sea inmediato el paso entre la teoría del sistema propuesto hacia la extensión en el propio lenguaje de programación. Las definiciones de términos dadas en el sistema expuesto tienen semejanzas con la sintaxis utilizada por Haskell por ejemplo, el uso de los constructores múltiples. El uso de esta sintaxis ha sido con la idea de implementar el sistema propuesto como fue realizado en [22].

Haskell es un lenguaje que sobresale de entre otros, ya que solo evalúa lo que el programa requiere para responder alguna pregunta, es decir su "pereza", esta característica es la que permite el manejo de objetos coinductivos o infinitos ya que no se requiere de una evaluación inmediata. Dado que la computadora es una herramienta en nuestra vida diaria, se concluye que el sistema expuesto disminuye el esfuerzo en la programación de tipos de datos (co)inductivos, no sólo al programar usando las herramientas matemáticas en las que se basa el desarrollo del sistema, como lo es la teoría de categorías, sino también en la programación directa en una computadora y así mejorar la compresión y la inclusión de la teoría en la práctica. El siguiente paso, después de este trabajo, es llevar el azúcar sintáctico propuesto a Haskell y así aportar una manera más fácil e intuitiva de programar.

Bibliografía

[1] Bart Jacobs, Jan Rutten, A Tutorial on (Co)Algebras and (Co)Induction, Bulletin of the European Association for Theoretical Computer Science, Vol. 62, 222–259, 1997.

[2] Benjamin C. Pierce, Types and Programming Languages, MIT Press, 2002.

[3] D. A. Turner, Total Functional Programming, Journal of Universal Computer Science, 2004.

[4] Erik Meijer, Graham Hutton, Bananas in Space: Extending Fold and Unfold to Exponential Types, Record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Int. Conf. on Functional Programming Languages and Computer Architecture, ACM Press, Nueva York, 324–333, 1995.

[5] Erik Meijer, Maarten Fokkinga, Ross Paterson, Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, bookProceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, Springer-Verlag, Berlin, 124–144, 1991.

[6] Favio Ezequiel Miranda Perea, On Extensions of AF2 with Monotone and Clausular (Co)inductive Definitions, Ludwig-Maximilians-Universität München, Alemania, 2004.

[7] Gordon, Andrew, A tutorial on Co-induction and Functional Programming, University of Cambridge Computer Laboratory, Glasgow, 1994.

[8] Hendrik Pieter Barendregt, The lambda calculus: Its syntax and semantics, Studies in logic and the foundations of mathematics v.103, Holanda, 1984.

[9] J. J. M. M. Rutten, Automata and coinduction (an exercise in coalgebra), Centrum voor Wiskunde en Informatica (CWI), 1998.

[10] John Greiner, Programming with Inductive and Co-Inductive Types, CMU-CS-92-109, Carnegie Mellon University, 1992.

[11] Lawrence C. Paulson, A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions, Computer Laboratory, University of Cambridge, Inglaterra, 1998.

[12] Michael A. Arbib and Ernest G. Manes, Arrows, Structures, and Functors: The Categorical Imperative, Academic Press, Nueva York, 1975.

[13] Ralph Matthes, Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types, Universität München, Alemania, 1998.

[14] Ralph Matthes, Tarski's fixed-point theorem and higher-order term rewrite systems, Universität München, Alemania, 1999.

[15] Richard Kieburtz, Codata and Comonads in Haskell, 1999.

[16] Roy L.Crole, Categories for types, Cambridge Mathematical Textbooks, Cambridge University Press, 1993.

[17] Sabine Glesner, An Introduction to (Co)Algebras and (Co)Induction and their Application to the Semantics of Programming Languagues, 2005-22, University of Karlsruhe, 2005.

[18] Tarmo Uustalu, Varmo Vene, Least and Greatest Fixedpoints in Intuitionistic Natural Deduction, Theoretical Computer Science, 2002.

[19] Tarmo Uustalu, Varmo Vene, Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically, Informatica Lithuanian Academy of Sciences, Vol. 10, No. 1, 1999.

[20] Tarski, Alfred, A lattice-theoretical fixpoint theorem and its applications, 1955.

[21] Tatsuya Hagino, A Typed Lambda Calculus with Categorical Type Constructors, In D.H. Pitt, A. Poigné, D.E. Rydeheard, Category Theory and Computer Science, Springer Verlag, 1987.

[22] Varmo Vene, Categorical programming with Inductive and Coinductive types, University of Tartu, 2000.

[23] Splawski Zdzizlaw, Urzyczyn Pawel, Type Fixpoints: Iteration vs. Recursion, In Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Vol. 34 SIGPLAN Notices ACM, Paris, France, 102–113, 1999.