Universidad Nacional Autónoma
de México

Posgrado en Ciencia e Ingeniería de la Computación

Sistemas de Tipos Categóricos para Recursión por Curso de Valores

T E S I S

que para obtener el grado de:

Maestro en Ciencias e Ingeniería de la Computación

presenta:

Javier Enríquez Mendoza

TUTOR:

Dr. Favio Ezequiel Miranda Perea

Facultad de Ciencias UNAM

México, Cd. Mx. Febrero, 2023

Agradecimientos

Investigación realizada gracias al programa de Apoyo a Proyectos de Investigación e Innovación Tecnológica (PAPIIT) de la Universidad Nacional Autónoma de México, en el marco del proyecto:

Lógicas no clásicas: Aspectos deductivos de la Computación a la Filosofía (PAPIIT, IN119920)

Agradezco a la DGAPA-UNAM por la beca recibida.

También agradezco al Consejo Nacional de Ciencia y Tecnología (CONACyT) por la beca otorgada de 2020 a 2022.

Índice general

Introducción

Muchas áreas de la Matemática son muy similares a la programación funcional y esto no es casualidad ya que estas disciplinas están íntimamente conectadas. Muchas áreas de las Ciencias de la Computación son derivadas de fundamentos matemáticos. Gracias a esto se puede abstraer el comportamiento de los programas en estos lenguajes así como la forma de desarrollar estos programas de forma muy certera mediante mecanismos matemáticos. Esto permite razonar sobre estos de forma matemática y así tener una mayor comprensión sobre la forma en la que se programa, lo que es fundamental para poder mejorar los programas escritos en estos lenguajes.

Una de estas teorías que fundamenta la programación funcional es la Teoría de Categorías. Su impacto en las Ciencias de la Computación es muy claro y ampliamente estudiado en distintos aspectos, esto debido a que la Teoría de Categorías ofrece mecanismos de abstracción sumamente flexibles que permiten describir con gran detalle muchos aspectos de las Ciencias de la Computación, siendo uno de ellos la programación funcional. Las categorías dan un mecanismo de razonamiento sobre la programación en un sentido tanto semántico como sintáctico, siendo quizá una de las teorías que mejor describen la forma de estos lenguajes de programación.

Motivación

En el tratamiento categórico de la programación funcional, en donde los tipos son objetos y las flechas funciones de un tipo en otro, se pueden formalizar esquemas recursivos como combinadores categóricos con el mismo comportamiento. Los combinadores categóricos mas utilizados en programación funcional son los *catamorfismos* y *anamorfismos*, que se implementan como los operadores `fold` y `unfold` respectivamente. Sobre estos combinadores existen ya muchos trabajos que modelan sus funcionalidades y principalmente sus propiedades, con las que se vuelve mas sencillo razonar sobre programas que utilizan estos operadores. Como un ejemplo de ello existen las propiedades universales y de fusión que ofrecen una técnica de optimización sobre estos programas que muchos compiladores actuales tienen implementadas.

La definición de combinadores que abstraen esquemas recursivos es una tarea importante para obtener un razonamiento matemático sobre una mayor variedad de programas y esquemas y de esta forma poder mejorar el resultado obtenido por estos programas.

Un esquema recursivo muy frecuente en la programación funcional es la recursión por curso de valores, en el que el resultado de la evaluación en un punto de la función depende no sólo del resultado en el punto inmediato anterior sino también en distintos puntos anteriores de la evaluación. Este esquema queda fuera de los capturados por los anamorfismos y catamorfismos.

El objetivo principal de este trabajo es definir combinadores categóricos para abstraer el esquema recursivo de recursión por curso de valores e implementarlo como parte de un sistema de reescritura de términos tipados. Con el fin de obtener un mayor poder de razonamiento sobre las funciones que usan este esquema dentro de la programación funcional.

Acerca de este trabajo

La estructura general de este trabajo se divide en dos partes fundamentales. En primera instancia se definen los combinadores categóricos necesarios para la abstracción de la recursión por curso de valores y se agregan como operadores de un sistema de reescritura de términos tipados, definiendo su semántica operacional y sus reglas de tipado.

La semántica operacional de estos operadores está basada en el comportamiento de los diagramas categóricos de los combinadores en los que tienen su base, por lo que el razonamiento categórico sobre estos operadores es de suma importancia para su implementación. Por otro lado las reglas de tipado de los nuevos combinadores garantizan el cumplimiento de algunas propiedades importantes del sistema que se define.

La segunda parte del trabajo, consiste en una serie de casos de estudio reales de la programación funcional en donde el esquema de recursión por curso de valores es utilizado para solucionarlos, por lo que se presenta una explicación de la problemática, seguido de una solución tradicional utilizando recursión por curso de valores y posteriormente se muestra la misma solución dentro del sistema propuesto con el uso de los operadores agregados para el tratamiento de funciones con este esquema recursivo.

La estructura mas puntual de este trabajo es la siguiente: En el primer capítulo se presentan todos los conceptos fundamentales para el desarrollo del trabajo, tanto en el campo de la teoría de Categorías como en el de los lenguajes de programación. En el segundo capítulo se define un sistema de reescritura de términos tipados en el que se tienen como nativos operadores que corresponden a los catamorfismos y

anamorfismos que abstraen esquemas recursivos clásicos. Posteriormente, en el tercer capítulo se presenta definición del nuevo sistema propuesto en este trabajo que incluye los operadores para abstraer la recursión por curso de valores. En el último capítulo se estudian diferentes casos de estudio de problemas cuya solución emplea la recursión por curso de valores y se presentan estas soluciones dentro del sistema. Para finalizar se dan las conclusiones del trabajo y el potencial trabajo futuro para seguir desarrollando.

Capítulo 1

Preliminares

En este capitulo se presentan las bases teóricas necesarias para el desarrollo de este trabajo. Se presentan ejemplos y definiciones importantes de las áreas de teoría de categorías y lenguajes de programación que serán la base de los formalismos presentados en los capítulos siguientes.

1.1. (Co)Iteración

El término *recursión* se refiere al mecanismo de definición de una función en términos de si misma, es decir, una definición recursiva de una función se refiere al hecho de definir un punto de la función en términos de la misma función en otro punto, usualmente estos puntos son estructuralmente mas simples. La recursión también sirve como mecanismo de definición de conjuntos, permitiendo definir conjuntos infinitos con objetos finitos [1], estos conjuntos reciben el nombre de inductivos.

Quizá el ejemplo clásico de definiciones recursivas es el conjunto de los números naturales, definido a continuación.

Ejemplo 1.1 (Números naturales)**.** El conjunto de los números naturales se define recursivamente con las siguientes cláusulas:

- El cero es un número natural, $0 \in \mathbb{N}$.
- El sucesor de un natural, es también un natural, si $n \in \mathbb{N}$ entonces $s(n) \in \mathbb{N}$.
- Son todos.

En esta definición, la segunda cláusula es recursiva, pues para construir un natural con el constructor sucesor es necesario que exista otro natural (estructuralmente

mas simple) previamente. La tercera cláusula indica que $\mathbb{N}$ es el conjunto mas pequeño de elementos que cumplen la condiciones anteriores.

Ahora se presenta una función definida de forma recursiva.

Ejemplo 1.2 (Factorial)**.** Se define la función factorial, que es un clásico ejemplo de funciones recursivas como sigue:

$$\begin{aligned} fact\ 0 &= 1 \\ fact\ s(n) &= s(n) \times fact\ n \end{aligned}$$

ésta es una definición recursiva pues en el segundo caso de la función, para evaluar $fact$ en el punto $s(n)$ se usa la evaluación de la misma $fact$ en el punto n.

Existen dentro de la matemática muchos principios de recursión, siendo la iteración uno de ellos. Mas aún la iteración es la forma mas simple de la recursión y consiste en la aplicación repetitiva de una función, es decir, dada una función f iterarla $n+1$ veces consiste en obtener $f^{n+1}(x) = f(f^n(x))$.

Se ejemplifica el concepto de iteración para el caso especifico de los naturales.

Definición 1.1 (Principio de iteración para los naturales)**.** Dados un valor $a \in A$ y una función $g : A \to A$ existe una única función $f : \mathbb{N} \to A$ tal que:

$$\begin{aligned} f\ 0 &= a \\ f\ s(n) &= g(f\ n) \end{aligned}$$

Entonces se dice que f está definida de forma iterativa.

Se presenta un ejemplo de la función odd.

Ejemplo 1.3 (Función de paridad)**.** Se define la función $odd : \mathbb{N} \to Bool$ que decide si un número es par o no, definida de forma iterativa como sigue:

$$\begin{aligned} odd\ 0 &= True \\ odd\ s(n) &= not(odd\ n) \end{aligned}$$

en donde el valor a corresponde a la constante $True$ mientras que la función g es la negación booleana not.

Es importante notar como la definición de la función factorial presentada anteriormente no es iterativa ya que en el caso $fact\ s(n) = s(n) \times fact\ n$ se utiliza el punto específico en el cúal se está evaluando la función, en este caso $s(n)$ y eso no entra dentro del esquema del principio de iteración.

Es común encontrar en la matemática el concepto de dualidad, que se refiere, particularmente el dual de la inducción de la coinducción que se refiere a la definición de estructuras infinitas a partir de objetos finitos, infinitos o potencialmente infinitos [1].

Dualizando el concepto de inducción, para construir objetos a partir de un conjunto coinductivo es necesario destruir sus elementos para obtener otros que también pertenezcan al conjunto.

Por ejemplo, si dualizamos el concepto de lista, que se define inductivamente, obtenemos listas infinitas o *Streams* que se definen como sigue.

Ejemplo 1.4 (Listas infinitas o *Streams*)**.** El conjunto de listas infinitas sobre un conjunto A, denotado como Stream_A se define mediante las siguientes cláusulas:

- Si $s \in \mathsf{Stream}_A$ entonces $\texttt{head}\ s \in A$.
- Si $s \in \mathsf{Stream}_A$ entonces $\texttt{tail}\ s \in Stream_A$.
- Si X cumple las dos propiedades anteriores entonces $X \subseteq \mathsf{Stream}_A$.

La tercera condición enfatiza que Stream es el conjunto mas grande que cumple las otras dos cláusulas

También podemos definir funciones de forma coinductiva, dando los valores de la función para todos los destructores, es decir aplicar cada uno de los destructores a la función. De esta forma, podemos definir funciones de forma coiterativa que corresponde con el dual de la iteración. Por ejemplo, podemos definir la función `map` sobre la estructura coinductiva de listas infinitas.

Ejemplo 1.5 (Función `map` sobre Stream_A)**.** La función `map` aplica una función g a los elementos de la lista infinita y se puede definir coiterativamente como sigue:

$$\begin{array}{lcl} \texttt{head}\ (\texttt{map}\ g\ s) & = & g\ (\texttt{head}\ s) \\ \texttt{tail}\ (\texttt{map}\ g\ s) & = & \texttt{map}\ g\ (\texttt{tail}\ s) \end{array}$$

Para cualquier función que queramos definir coiterativamente para *Streams* tenemos el siguiente principio de coiteración.

Definición 1.2 (Principio de coiteración sobre Stream_A)**.** Dadas dos funciones $g_1 : D \to A$ y $g_2 : D \to D$ y una lista infinita s, existe una única función $f : D \to \mathsf{Stream}_A$ tal que:

$$\begin{array}{lcl} \texttt{head}\ (f\ s) & = & g_1\ s \\ \texttt{tail}\ (f\ s) & = & f\ (g_2\ s) \end{array}$$

Es importante observar la dualidad entre los principios de iteración y coiteración. En lo que resta de este trabajo se usarán estos conceptos y sobre todo se busca extenderlos para otro esquema recursivo, la recursión por curso de valores que se define en la sección siguiente.

1.2. Recursión por Curso de Valores

La recursión por curso de valores es un esquema que generaliza la recursión primitiva permitiéndonos definir el valor de una función en un argumento de un tipo de dato inductivo utilizando arbitrariamente todos los cómputos anteriores y no sólo el inmediato.

Por ejemplo en el caso de los números naturales, el valor de $n+1$ en una función f, definida por recursión por curso de valores, no solo depende de el valor de f en n sino que también de todos los valores anteriores $< n$.

Ejemplo 1.6 (Fibonacci)**.** Consideremos la siguiente definición de la función que calcula el n-ésimo número de la sucesión de Fibonacci.

$$\begin{aligned} \texttt{fib}(0) &= 1 \\ \texttt{fib}(1) &= 1 \\ \texttt{fib}(n+2) &= \texttt{fib}(n+1) + \texttt{fib}(n) \end{aligned}$$

La definición anterior no es recursión primitiva pues el valor de $\texttt{fib}(n+2)$ depende no sólo de $\texttt{fib}(n+1)$ sino también de $\texttt{fib}(n)$. Es decir, se trata de una definición por recursión por curso de valores.

A continuación se presenta un ejemplo en donde se lleva el concepto de la recursión por curso de valores al grado extremo de utilizar la evaluación en todos los puntos anteriores, este ejemplo se refiere a los números de Catalán, los cuales son una secuencia comúnmente encontrada en problemas de conteo. Esta sucesión recibe su nombre en honor al matemático Franco-Belga Eugène Charles Catalan.

Ejemplo 1.7 (Números de Catalán)**.** Los números de Catalán se definen de la siguiente forma:

$$\begin{aligned} C_0 &= 1 \\ C_{n+1} &= C_0C_n + C_1C_{n-1} + \cdots + C_{n-1}C_1 + C_nC_0 \end{aligned}$$

A partir de estos ejemplos se presenta la siguiente definición formal de la recursión por curso de valores sobre los naturales.

Definición 1.3 (Recursión por Curso de Valores)**.** Sean $a_0, \ldots, a_k \in A$ y $g_\ell : A^l \to A$ con $\ell \geqslant k$. La función $f : \mathbb{N} \to A$ definida por recursión por curso de

valores con casos base $a_0, \ldots, a_k$ y función de paso g se define como:

$$\begin{aligned} f(0) &= a_0 \\ &\vdots \\ f(k) &= a_k \\ f(n+1) &= g_\ell(f(n_1), \ldots, f(n_{\ell-1}), f(n_\ell)), \ell \geqslant k \end{aligned}$$

donde $n_1, \ldots, n_l < n + 1$

Sin embargo, la definición anterior tiene un problema en la función de paso g_ℓ, ya que ésta no es única, es decir, pueden existir muchas funciones de paso g_l en donde no sean necesarios todos los parámetros sino que solo requiera de algunos de ellos. Para dar una definición mas precisa se utiliza recursión con memoria, que se estudia en la siguiente sección.

1.2.1. Recursión con Memoria

En recursión por curso de valores se pueden utilizar todos los cómputos anteriores en cada llamada recursiva. Esto hace que el mismo valor sea necesario en mas de una ocasión, por lo que el mismo valor se calcula tantas veces como es necesario y el tiempo de ejecución del programa crece considerablemente. De igual forma, el espacio en memoria se consume rápidamente, y esto causa que llamadas a la función con valores muy grandes agoten la memoria y no puedan resolverse provocando un impacto negativo sobre el programa.[2]

Para mejorar la complejidad en tiempo y espacio de las funciones definidas usando recursión por curso de valores se utiliza recursión con memoria. La idea es reemplazar el cómputo repetido de la función en un argumento dado por la búsqueda de dicho valor en una tabla. Esta tabla se va a ir construyendo con los valores de la función cuando se calculan por primera vez y es indexada con el tipo del dominio de la función a calcular, de tal forma que la celda x almacena el valor $f\, x$. Una función con memoria se puede ver como una composición de una función de construcción de la tabla y una función de búsqueda de un elemento en particular. Esta idea funciona directamente en lenguajes de programación en donde las funciones son puras, en el sentido matemático, debido a la ausencia de efectos laterales.

Con esta idea podemos dar una definición alterna a la recursión por curso de valores.

Definición 1.4 (Historia)**.** Sea $f : \mathbb{N} \to A, n \in \mathbb{N}$. La historia de f hasta n es una lista de elementos de A definida como:

$$\texttt{hist}_f(n) = [f(n), f(n-1), \cdots, f(0)]$$

A partir de la definición anterior se presenta una nueva definición para la recursión por curso de valores.

Definición 1.5 (Recursión por curso de valores)**.** Sean $a_0, \cdots, a_k \in A$, $g : [A] \to A$ la función $f : \mathbb{N} \to A$ definida por recursión por curso de valores con casos base $a_0, \cdots, a_k$ y función de paso g se define como:

$$\begin{aligned} f(0) &= a_0 \\ &\vdots \\ f(k) &= a_k \\ f(n+1) &= g(\texttt{hist}_f(n)), n \geqslant k \end{aligned}$$

1.3. Teoría de Categorías

En la actualidad la teoría de categorías juega un papel central en matemáticas así como en diversas áreas de teoría de las Ciencias de la Computación. De forma general la teoría de categorías es una teoría matemática general de estructuras y sistemas de estructuras. Mientras el área sigue madurando y evolucionando sus aplicaciones se desarrollan, multiplican y expanden. Se trata de un poderoso lenguaje que nos permite abstraer los componentes universales de una familia de estructuras de cierto tipo y también nos permite estudiar como estructuras de distintos tipos se relacionan entre si [3].

Actualmente se utilizan categorías en una gran cantidad de ramas de las Ciencias de la Computación, por lo que esta teoría resulta muy importante cuando se habla de computación teórica. Esto es tema de las llamadas *matemáticas formalizadas*.La influencia de esta rama de las matemáticas ha sido bastante evidente en diferentes campos de las Ciencias de la Computación, tales como diseño de lenguajes de programación, modelos semánticos para lenguajes de programación, modelos de concurrencia, teoría de tipos, polimorfismo, teoría de autómatas, diseño de algoritmos, entre muchas otras.

En la lista anterior se puede apreciar la generalidad de la teoría de categorías, pues se trata de un sistema conceptual y de notación básico en el mismo sentido que la teoría de conjuntos. La principal ventaja de esta generalidad es el alto nivel de abstracción de conceptos, lo que le permite ser versátil al modelar diferentes campos de la Computación.

En esta sección se presentan los conceptos de teoría de categorías que se utilizan en el desarrollo de este trabajo, primero dando sus definiciones y posteriormente algunos ejemplos ilustrativos.

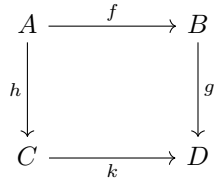
1.3.1. Conceptos Básicos

Intuitivamente la noción de categoría es muy simple, se trata de una colección de *objetos* y *flechas* que los relacionan, éstas ultimas reciben el nombre de morfismos. Los morfismos son funciones que van de un objeto a otro, siendo la composición un concepto fundamental en la Teoría de Categorías. La definición formal de categoría es la siguiente:

Definición 1.6 (Categoría)**.** Una categoría C es un par $\langle \mathcal{O}, \mathcal{M} \rangle$, en donde $\mathcal{O}$ es una colección de objetos y $\mathcal{M}$ una colección de morfismos $f : A \to B$ para cada par de objetos A, B y una operación $\circ$ de composición con las siguientes propiedades:

- Si se tienen los morfismos $f : A \to B$ y $g : B \to C$ entonces $g \circ f$ es un morfismo del objeto A al objeto C.
- Para cualesquiera morfismos $f : A \to B$, $g : B \to C$ y $h : C \to D$ con A, B, C y D objetos, se cumple que $h \circ (g \circ f) = (h \circ g) \circ f$
- Para cada objeto $X \in C$, existe un morfismo identidad denotado Id_X, tal que $\mathsf{Id}_X : X \to X$.
- Para cada morfismo $f : A \to B$ se tiene que $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$

Ejemplo 1.8. La categoría de los objetos $\{A, B, C, D\}$ y los morfismos $f : A \to B$, $g : B \to D$, $h : A \to C$ y $k : C \to D$ se puede representar gráficamente como se muestra a continuación.



Es posible encontrar categorías en todas las áreas de la matemática. Como ejemplo de esto podemos considerar las siguientes categorías.

Ejemplos 1.1 (Categorías conocidas)**.** El siguiente es un listado de categorías comúnmente mencionadas en trabajos sobre teoría de categorías.

- Set: es la categoría en donde los objetos es la clase de todos los conjuntos y los morfismos son las funciones definidas entre estos conjuntos siendo la composición de morfismos la composición usual de funciones. Es sabido que cada conjunto tiene una función identidad que se comporta de la forma esperada.

- Groups: los objetos son grupos y los morfismos son homomorfismos entre grupos.
- Poset: conjuntos con un orden parcial y funciones monótonas.
- Hask: es la categoría en donde los objetos son los tipos del lenguaje de programación Haskell y los morfismos son las funciones entre estos tipos, siendo la composición de morfismos la operación . que representa la composición de funciones en Haskell.

Otro concepto importante en Teoría de Categorías es el de funtor. Un funtor es un mapeo entre categorías, es decir, dadas dos categorías C y D, un funtor es una función de los objetos de C en los objetos de D y de las flechas de C en las flechas de D.

Definición 1.7 (Funtor)**.** Dadas dos categorías C y D, un funtor F es una función que toma cada objeto X en C y lo mapea a un objeto $F(X)$ en D, y preserva los morfismos descritos en C, es decir, para todo morfismo $f \in C$ si $f : X \to Y \in C$ entonces $F(f) : F(X) \to F(Y) \in D$. Tal que cumple las siguientes propiedades:

- Para todo objeto $X \in C$ sucede que $F(\mathsf{Id}_X) = \mathsf{Id}_{F(X)}$
- Para los morfismo en C, si $f : X \to Y$ y $g : Y \to Z$, entonces $F(g \circ f) = F(g) \circ F(f)$

Es decir, los funtores preservan la estructura de las categorías.

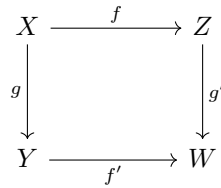
Las categorías pueden representarse gráficamente mediante diagramas como se mencionó con anterioridad. A continuación se definen formalmente conceptos importantes acerca de estos diagramas.

Definición 1.8 (Diagrama)**.** Para una categoría C un diagrama es una gráfica dirigida cuyos vértices están etiquetados con los objetos de C y sus aristas se etiquetan con los morfismos de C, de manera consistente, es decir, si una arista etiquetada con f va del nodo A al nodo B entonces en C se tiene que el morfismo $f : A \to B$.

Estos diagramas son utilizados para afirmar y demostrar propiedades de construcciones categóricas, a continuación presentamos una clase de diagramas que son de gran utilidad.

Definición 1.9 (Diagrama conmutativo)**.** Decimos que un diagrama conmuta si para cuales quiera par de vértices v, w en el diagrama, todos los caminos de v a w son iguales por composición.

Ejemplo 1.9 (Diagrama conmutativo)**.** Consideremos el siguiente diagrama.



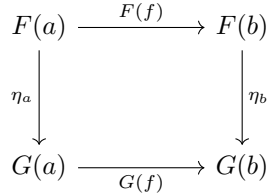
Decir que conmuta significa que $g' \circ f = f' \circ g$.

Así como los funtores son mapeos entre categorías, también existen mapeos entre funtores estos reciben el nombre de *transformaciones naturales*. Hasta ahora las flechas se han usado para denotar morfismos entre objetos y funtores entre categorías, para evitar confusiones se utiliza un nuevo símbolo para denotar las transformaciones naturales cuya notación es $F \Rightarrow G$ para referirnos a la transformación natural entre los funtores F y G.

Definición 1.10 (Transformación Natural)**.** Si $F, G : \mathcal{C} \to \mathcal{D}$ son dos funtores que van de la categoría $\mathcal{C}$ a la categoría $\mathcal{D}$, en una transformación natural $\eta : F \Rightarrow G$ por cada objeto $x \in \mathcal{C}$, existe un morfismo $\eta_x : F(x) \to G(x)$ tal que, si $f : a \to b$ es cualquier morfismo en $\mathcal{C}$, entonces:

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

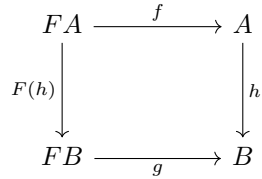
es decir, el siguiente diagrama conmuta:



Definición 1.11 (F-álgebra)**.** Dado un funtor $F : \mathcal{C} \to \mathcal{C}$ en una categoría $\mathcal{C}$ una F-álgebra es un par $\langle A, f \rangle$ en donde A es un objeto en $\mathcal{C}$ y f un morfismo en $\mathcal{C}$ tal que $f : FA \to A$

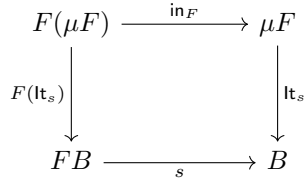
Los morfismos entre las álgebras se definen como sigue.

Definición 1.12 (Morfismos entre F-álgebras)**.** Dadas dos F-algebras $\langle A, f \rangle$ y $\langle B, g \rangle$ en una categoría $\mathcal{C}$ un morfismo de la primera en la segunda h es un morfismo en $\mathcal{C}$ tal que el siguiente diagrama conmuta:



Dentro la categoría de las F-álgebras hay objetos notorios como las álgebras iniciales que serán fundamentales para el desarrollo de este trabajo, las cuales se definen a continuación.

Definición 1.13 (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, entonces el álgebra inicial es única y se denota como $\langle \mu F, \mathsf{in}_F\rangle$. En otras palabras es la F-álgebra que hace que el siguiente diagrama conmute:

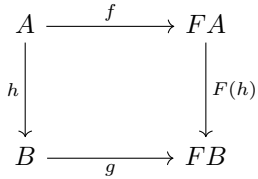


El concepto dual a una F-algebra es el de una F-coalgebra, que se puede definir como a continuación.

Definición 1.14 (F-coálgebra)**.** Dado un funtor $F : \mathcal{C} \to \mathcal{C}$ en una categoría $\mathcal{C}$ una F-álgebra es un par $\langle A, f\rangle$ en donde A es un objeto en $\mathcal{C}$ y f un morfismo en $\mathcal{C}$ tal que $f : A \to FA$

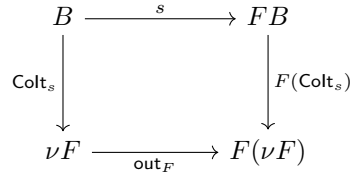
Los morfismos entre las F-coalgebras se definen a continuación.

Definición 1.15 (Morfismos entre F-coálgebras)**.** Dadas dos F-algebras $\langle A, f\rangle$ y $\langle B, g\rangle$ en una categoría $\mathcal{C}$ un morfismo de la primera en la segunda h es un morfismo en $\mathcal{C}$ tal que el siguiente diagrama conmuta:



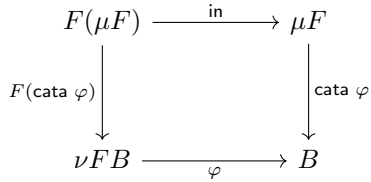
Así como con las F-coalgebras, también tenemos el dual para las álgebras iniciales, que corresponden a las coalgebras finales.

Definición 1.16 (Coálgebra final)**.** Un coálgebra final $\langle A, f \rangle$ es un objeto terminal en la categoría de las F-coálgebras, es decir, existe un único morfismo Colt_s de cualquier coálgebra $\langle B, s \rangle$ a la coálgebra final. Si existe, entonces la coálgebra final es única y se denota como $\langle \nu F, \mathsf{out}_F \rangle$. En otras palabras es la F-álgebra que hace que el siguiente diagrama conmute:



Dentro de la Teoría de Categorías existen combinadores importantes que son ampliamente estudiados y relacionados con la programación funcional. Los principales combinadores son el catamorfismo y el anamorfismo. Definidos a continuación.

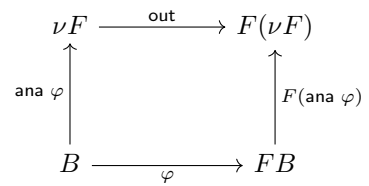
Definición 1.17 (Catamorfismo)**.** Sea $F : \mathcal{C} \to \mathcal{C}$ un funtor. Si existe el álgebra inicial de F denotada como $\langle \mu F, \mathsf{in} \rangle$ tal que para todo objeto B y morfismo $\varphi : FB \to B$ existe un único morfismo $\mathsf{cata}\ \varphi$ tal que el siguiente diagrama conmuta:



Esto es $(\mathsf{cata}\ \varphi) \circ \mathsf{in} = \varphi F(\mathsf{cata}\ \varphi)$. En tal caso el morfismo $\mathsf{cata}\ \varphi$ es llamado el catamorfismo y la ecuación anterior es el principio de iteración.

El dual de un catamorfismo es un anamorfismo que corresponde a la siguiente definición.

Definición 1.18 (Anamorfismo)**.** Sea $F : \mathcal{C} \to \mathcal{C}$ un funtor. Si existe la coálgebra final de F denotada como $\langle \nu F, \mathsf{out} \rangle$ tal que para todo objeto B y morfismo $\varphi : FB \to B$ existe un único morfismo $\mathsf{ana}\ \varphi$ tal que el siguiente diagrama conmuta:



Mas adelante en este trabajo se habla de la importancia de los combinadores anteriores en la programación funcional.

1.3.2. Tratamiento Categórico de la iteración por curso de valores

El esquema de recursión por curso de valores generaliza la iteración convencional, definida por los catamorfismos, al permitir que la evaluación de una función en un punto utilice los resultados de todos los puntos anteriores a este no sólo en el resultado del punto anterior. Este esquema se puede definir dentro de la Teoría de categorías mediante un combinador que abstraiga este comportamiento, como se muestra a continuación.

Para lograr esto primero es necesaria una álgebra que abstraiga este comportamiento, la cual recibe el nombre de cv-algebra y se define a continuación.

Definición 1.19 (cv-álgebra)**.** Sea $F : \mathcal{C} \to \mathcal{C}$ un functor con álgebra inicial $\langle \mu F, \mathsf{in} \rangle$, dado un objeto $C \in \mathcal{C}$ se define $F_C^{\times} X = C \times F\ X$. Asumiendo que para cada C existe una coálgebra final $\langle \nu F_C^{\times} X, \mathsf{out} \rangle$, se define un nuevo funtor F^{ν} tal que $F^{\nu} C = \nu F_C^{\times} X$. Una cv-álgebra es un par $\langle C, \varphi \rangle$ donde $\varphi : F(F^{\nu} C) \to C$.

El combinador categórico que abstrae la recursión con memoria es el histomorfismo.

Definición 1.20 (Histomorfismo)**.** Sea $\langle \mu F, \mathsf{in} \rangle$ el álgebra inicial del funtor F, dada una cv-álgebra $\langle C, \varphi \rangle$, el histomorfismo $\mathsf{histo}\ \varphi : \mu F \to A$ es el único morfismo que hace que siguiente diagrama conmute:

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{\ \mathsf{in}\ } & \mu F \\ {\scriptstyle F(\mathsf{ana}\ \langle \mathsf{histo}\ \varphi, \mathsf{in}^{-1} \rangle)}\downarrow & & \downarrow{\scriptstyle \mathsf{histo}\ \varphi} \\ F(F^{\nu} C) & \xrightarrow[\ \varphi\ ]{} & C \end{array}$$

donde $\mathsf{ana}\ \langle \mathsf{histo}\ \varphi, \mathsf{in}^{-1} \rangle$ denota el anamorfismo del funtor $F_C^{\times}$ con un morfismo de paso $< \mathsf{histo}\ \varphi, \mathsf{in}^{-1} \rangle$. Con lo que la ecuación:

$$(\mathsf{histo}\ \varphi) \circ \mathsf{in} = \varphi \circ F(\mathsf{ana}\ \langle \mathsf{histo}\ \varphi, \mathsf{in}^{-1} \rangle)$$

es el principio de iteración por curso de valores

La idea intuitiva sobre el tratamiento categórico para la recursión por curso de valores presentado en esta sección es abstraer este patrón recursivo mediante la recursión

con memoria, es decir, la función ene el punto n tiene una historia, que se representa como una Colista, en la que se almacena la evaluaciones de la función en todos los puntos menores a n.

1.4. El sistema F

El Cálculo Lambda polimorfico de segundo orden, también conocido como sistema F, es un sistema de tipos inventado por Girard (1972) y Reynolds (1974), que introduce un mecanismo de cuantificación universal sobre tipos al Cálculo Lambda con tipos simples, formalizando el polimorfismo paramétrico en los lenguajes de programación[4][5][6][7].

El *polimorfismo paramétrico* es una técnica que permite la definición de procedimientos genéricos en un lenguaje de programación, evitando así conversiones innecesarias entre tipos o errores de tipos. En otras palabras, el polimorfismo paramétrico es cuando un procedimiento o función está definida sobre una variedad de tipos en lugar de uno sólo. Hoy en día es muy común la presencia de esta característica en los lenguajes de programación, ya que aumenta el poder de expresividad del lenguaje así como su flexibilidad para la reutilización del código.

El uso de este lenguaje ayuda a evitar la necesidad de *down casting* y le permite al compilador encontrar una mayor cantidad de errores gracias a su habilidad de verificar el tipado en tiempo de compilación (verificación estática de tipos).

El Sistema F es un lenguaje fuertemente normalizable, esto quiere decir que cualquier programa en este lenguaje define un cómputo que termina. Ésta es una propiedad importante que indica que el lenguaje carece de recursión general y solo puede expresar funciones totales.

Una de las principales limitantes del lenguaje es la eficiencia de las funciones definidas sobre tipos coinductivos. Esto debido a que los operadores correcursivos nativos abarcan únicamente coiteración. Es decir los operadores sobre tipos coinductivos solo abarcan coiteración y no correcursión. Por ejemplo, para la implementación de la función predecesor sobre los números naturales es necesario transversar todo el natural para quitar un constructor (sucesor) lo que tiene una complejidad temporal de $O(n)$ en lugar de la complejidad $O(1)$ clásica para esta operación implementada es recursión primitiva.

La problemática anterior ha llevado a la investigación sobre extensiones del Sistema F con otros operadores que soporten diferentes esquemas recursivos.

1.4.1. Definición del lenguaje

En esta sección se describe el sistema F de Girard y Reynolds, a la definición original del sistema se le agregan como primitivos los tipos suma, producto y unitario. Esto por conveniencia para simplificar el desarrollo de este trabajo.

Definición 1.21 (Sistema F)**.** Se define el lenguaje como sigue:
Tipos Se considera la existencia de un conjunto infinito de variables de tipos denotadas con la metavariable X.

$$A, B, C, F, G ::= X \mid A \to B \mid \forall X.A \mid A + B \mid A \times B \mid 1$$

Términos Se considera que existe un conjunto infinito de variables de términos denotadas con la metavariable x.

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

Contextos Se definen los contextos como conjuntos finitos de parejas de la forma $x : A$ en donde x es una variable de término y A es un tipo, y se lee como x tiene el tipo A, siempre se asume que hay una única indicación de tipo por cada variable dentro del contexto. Los contextos se denotan con Γ. La notación $\Gamma, x : A$ representa $\Gamma \cup \{x : A\}$.
Semántica Estática Se definen las reglas de tipado con juicios de la forma $\Gamma \vdash t : A$ que se lee como: de gamma se deriva que el término t tiene tipo A.

$$\frac{}{\Gamma, x : A \vdash x : A}(Var) \qquad \frac{}{\Gamma \vdash \star : 1}(Unit)$$

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x.r : A \to B}(\to_I) \qquad \frac{\Gamma \vdash r : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B}(\to_E)$$

$$\frac{\Gamma \vdash t : A \quad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X.A}(\forall_I) \qquad \frac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t : A[X := F]}(\forall_E)$$

$$\frac{\Gamma \vdash r : A}{\Gamma \vdash \mathsf{inl}\ r : A + B}(+_{Il}) \qquad \frac{\Gamma \vdash r : B}{\Gamma \vdash \mathsf{inr}\ r : A + B}(+_{Ir})$$

$$\frac{\Gamma \vdash r : A + B \quad \Gamma.x : A \vdash s : C \quad \Gamma, y : B \vdash t : C}{\Gamma \vdash \mathsf{case}(r, x.s, y.t) : C}(+_E)$$

$$\frac{\Gamma \vdash r : A \quad \Gamma \vdash s : B}{\Gamma \vdash \langle r, s\rangle : A \times B}(\times_I)$$

$$\frac{\Gamma \vdash r : A \times B}{\Gamma \vdash \mathsf{fst}\ r : A}(\times_{El}) \qquad \frac{\Gamma \vdash r : A \times B}{\Gamma \vdash \mathsf{snd}\ r : B}(\times_{Er})$$

Semántica Operacional Se define la semántica dinámica del lenguaje con la regla de reducción $t \to t'$

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

Para el desarrollo de este trabajo se usarán las bases teóricas estudiadas en este capítulo para definir extensiones sobre el sistema F agregando operadores para definir esquemas recursivos, estos basados en combinadores categóricos.

Capítulo 2

Sistemas de Tipos Categóricos

El objetivo es presentar extensiones del Sistema F de tal forma que se pueda abstraer el esquema recursivo por curso de valores mediante un sistema de reescritura de términos tipados. Para esto se modelan combinadores cuyo funcionamiento se basa en las definiciones categóricas de los mismos. Esta técnica de modelado de combinadores ha sido ampliamente utilizada para agregar esquemas recursivos del cálculo lambda con tipos, recordando que a diferencia del cálculo lambda puro, el cálculo lambda con tipos carece de recursión general ya que los combinadores de punto fijo, como el combinador Y no son tipables. [8][9][10][11][12][13]

2.1. De Categorías a Tipos

Para poder implementar combinadores categoricos como constructores del cálculo lambda se tiene que modelar el sistema como una categoría que llamaremos $\mathcal{T}$. Esta categoría es similar a Hask descrita en el capítulo 1: los objetos son tipos del Sistema F y los morfismos son las funciones entre estos tipos, por lo que la composición de morfismos es la composición de funciones usual, y el morfismo identidad es la función identidad de cada tipo[1][14].

Dentro de $\mathcal{T}$ un funtor $F : \mathcal{T} \to \mathcal{T}$ es una transformación entre tipos, que dentro de la programación funcional pueden verse como constructores de tipos. Para el desarrollo de este trabajo se usaran en particular transformaciones de tipos que dependen de una variable de tipo X que transforma un tipo B al tipo FB, para este tipo de transformaciones se puede usar la notación $\lambda X.F$ que involucra la abstracción de la variable de tipo X, esto no es mas que notación y no pertenece a la sintaxis formal

del sistema debido a que el sistema usado en este trabajo no es de orden superior por lo que no tienen este tipo de abstracciones. Sobre esta notación una aplicación de la forma $(\lambda X.F)B$ se entiende como la sustitución $F[X := B]$.

Es importante notar que las transformaciones como se definieron no pueden ser consideradas por si solas como funtores, ya que solo se define su comportamiento sobre los objetos de $\mathcal{T}$ mas no sobre sus morfismos, por lo que estas abstracciones reciben el nombre de prefuntores.

El efecto de las abstracciones $\lambda X.F$ en los morfismos de $\mathcal{T}$ se representa de forma interna por medio de un término $\mathsf{map} : (\lambda X.F)\ \mathsf{mon}$ para un contexto dado. El tipo $(\lambda X.F)\ \mathsf{mon}$ se define como $\forall X.\forall Y.(X \to Y) \to (\lambda X.F)\ X \to (\lambda X.F)\ Y$ y representa el hecho de que el funtor $\lambda X.F$ es monótono (covariante) respecto a su argumento X. El término map recibe el nombre de *testigo de monotonicidad*[15][16].

Entonces un functor dentro del sistema es una pareja $\langle \lambda X.F, \mathsf{map} \rangle$ en donde $\lambda X.F$ es un prefuntor y el término map es el testigo de monotonicidad en el contexto necesario. Esta representación de funtores es la misma que se utiliza para dar instancias de la clase `Funtor` en Haskell.

Ejemplo 2.1 (Funtor)**.** Se considera la transformación $\lambda X.F$ en donde $F = 1 + X$, para que este prefuntor sea un funtor es necesario definir el testigo de monotonicidad para el tipo $(\lambda X.\, 1 + X)\ \mathsf{mon}$, que es un término $\mathsf{map} : \forall X.\forall Y.(X \to Y) \to (1 + X) \to (1 + Y)$.
Dada una función $f : X \to Y$ y un término $t : (1 + X)$ se construye un habitante del tipo $1 + Y$ mediante un análisis de casos sobre t:

$$\mathsf{map}\ f\ t = \begin{cases} t & \text{si } t = \mathsf{inl}\star \\ f\ m & \text{si } t = \mathsf{inr}\ m \end{cases}$$

En el Sistema F se define $\mathsf{map} = \lambda f.\lambda x.\mathsf{case}(x, y.\mathsf{inl}\ y, z.\mathsf{inr}\ (f\ z))$.

Para lograr definir un sistema de reescritura de términos tipados los tipos (co)inductivos deben modelarse como álgebras iniciales o coálgebras finales de funtores. Dado un funtor $\langle \lambda X.F, \mathsf{map} \rangle$ se modela el álgebra inicial con el tipo inductivo $\mu(\lambda X.F)$ y su coálgebra final con el tipo coinductivo $\nu(\lambda X.F)$. La existencia del testigo de monotonicidad map está forzada por el sistema de tipos.

2.2. El Sistema It

Se presenta una extensión del Sistema F con constructores nativos para modelar iteración y coiteración, estos constructores están definidos con base en los principios de iteración y coiteración presentados en el capítulo anterior para los combinadores categoricos de catamorifismo y anamorfismo respectivamente. Por lo que la semánti-

ca, tanto estática como operacional, de los constructores se define en términos de la conmutatividad de los diagramas categóricos para cada uno de los combinadores. Este sistema es llamado Sistema It.

Álgebra Inicial: Se modela el álgebra inicial $\langle \mu F, \mathsf{in} \rangle$ de un funtor $\langle F, m \rangle$ como un nuevo tipo μF, llamado un tipo inductivo, y un constructor de tipo in definido con la siguiente regla de tipado:

$$\frac{\Gamma \vdash t : F(\mu F)}{\Gamma \vdash \mathsf{in}\ t : \mu F}(\mu I)$$

Es importante observar que esta regla no involucra al testigo de monotonicidad m por lo que es posible construir un tipo inductivo μF para cada prefuntor dentro del sistema. En caso de no existir el testigo de monotonicidad, el μF no es un álgebra inicial. Sin embargo en el sistema el término m es indispensable para el uso del principio de iteración.

Catamorfismo: Se introduce un nuevo combinador ternario cata para modelar catamorfismos dentro el sistema, con la siguiente regla de tipado:

$$\frac{\begin{array}{c}\Gamma \vdash t : F(\mu F) \\ \Gamma \vdash m : \mathsf{mon}\ F \\ \Gamma \vdash \varphi : F\,B \to B\end{array}}{\Gamma \vdash \mathsf{cata}\ m\ \varphi\ t : B}(\mu E)$$

La semántica operacional del combinador se define con la siguiente regla de reducción que representa el principio de iteración modelado por el catamorfismo:

$$\mathsf{cata}\ m\ \varphi\ (\mathsf{in}\ t) \to \varphi\ (m\ (\mathsf{cata}\ m\ \varphi)\ t)$$

en donde $\mathsf{cata}\ \ m\ \ \varphi =_{def} \lambda x.\mathsf{cata}\ \ m\ \ \varphi\ \ x$, a partir de este punto se usara esta notación para simplificar la lectura de las reglas de reducción. Esta regla está basada en la conmutatividad del diagrama del combinador categórico cata.

Coálgebra final: Dado un prefuntor $F = \lambda X.G$ se modela la coálgebra final $\langle \nu F, \mathsf{out} \rangle$ del funtor $\langle F, m \rangle$ por medio de un nuevo tipo νF, que recibe el nombre de tipo coinductivo, y un constructor de tipo out con la siguiente regla de tipado:

$$\frac{\Gamma \vdash t : \nu F}{\Gamma \vdash \mathsf{out} : F(\nu F)}(\nu E)$$

Anamorfismo: De igual forma se agrega el operador ternario ana que modela el combinador categórico de anamorfismo. La regla de tipado para este combinador es la siguiente:

$$\frac{\begin{array}{c}\Gamma \vdash t : B \\ \Gamma \vdash m : \mathsf{mon}\ F \\ \Gamma \vdash \varphi : B \to F(B)\end{array}}{\Gamma \vdash \mathsf{ana}\ m\ \varphi\ t : \nu B}(\nu I)$$

Su semántica operacional con base en el diagrama conmutativo para el principio de coiteración, se define con la siguiente regla de reducción:

$$\mathsf{out}\ (\mathsf{ana}\ m\ \varphi\ t) \rightarrow m\ (\mathsf{ana}\ m\ \varphi)\ (\varphi\ t)$$

Los operadores definidos en este capítulo agregan esquemas recursivos básicos al sistema F, con los que se agrega iteración (cata) y coiteración (ana) como operaciones primitivas al lenguaje, usando como base combinadores categóricos. El propósito principal de este trabajo es agregar nuevos operadores que agreguen iteración por curso de valores como primitiva del lenguaje. Para esto, en el siguiente capítulo se definen nuevas extensiones del sistema It.

Capítulo 3

Sistemas con Iteración por Curso de Valores

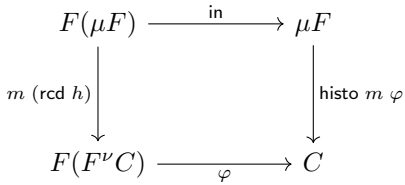
En este capítulo se agregan dos operadores al lenguaje It para modelar la recursión por curso de valores, en primera instancia de define el operador histo que abstrae el comportamiento de los *histomorfismos* con los que se modela la recursión por curso de valores mediante la definición de recursión con memoria estudiada en el capítulo 1, para lo que se agrega en cada punto de la función su historia[17][18][19][20][21][22][23][24].

Posteriormente se agrega un combinador mas expresivo, hylo , este operador modela la recursión por curso de valores mediante un *hilomorfismo* en el que se introduce una estructura intermedia en la que se almacenan las evaluaciones de la función en los puntos anteriores y la definición de ésta queda dentro de la misma definición del combinador, dando de esta forma mayor flexibilidad a la forma y contenido de la historia de la función, con lo que el patrón recursivo que modela es también mas flexible.

El nuevo sistema en el que se tienen como nativos estos dos operadores recibe el nombre de CvIt.

3.1. Histomorfismos

Se agrega al lenguaje It el constructor de término histo $m\ \varphi\ t$ con el uso de una función rcd n cuyo funcionamiento es calcular la historia de la función hasta el punto n. El comportamiento del nuevo término histo $m\ \varphi\ t$ se da por la conmuntatividad del siguiente diagrama:



En donde, dado un prefuntor $F =_{def} \lambda X.G$, se definen los prefuntores asociados $F_C^\times =_{def} \lambda X.C \times F\ X$ y $F^\nu =_{def} \lambda Z.\nu(F_Z^\times) =_{def} \lambda Z.\nu(\lambda X.Z \times F\ X)$, y a partir del testigo de monotonicidad m tal que $\langle F, m\rangle$ es un funtor, se definen los testigos de monotonicidad $m^\times =_{def} \lambda f.\lambda p.\langle \mathsf{fst}\ p, m\ f\ (\mathsf{snd}\ p)\rangle$ y $m^\nu =_{def} \lambda f.\mathsf{ana}\ m^\times\langle f \circ \mathsf{cur}, \mathsf{prev}\rangle$ de tal forma que $\langle F_C^\times, m^\times\rangle$ y $\langle F^\nu, m^\nu\rangle$ son también funtores.

En este razonamiento se utilizan los nombre cur y prev en lugar de los clásicos head y tail para los destructores de colistas.

La semántica estática del operador histo se define con la siguiente regla de tipado:

$$\frac{\begin{array}{c}\Gamma \vdash t : \mu F \\ \Gamma \vdash m : F\mathsf{mon} \\ \Gamma \vdash \varphi : F(F^\nu C) \to C\end{array}}{\Gamma \vdash \mathsf{histo}\ m\ \varphi\ t : C}\,\mu E^{cv}$$

Mientras que la semántica operacional se define con la siguiente regla de reducción:

$$\mathsf{histo}\ m\ \varphi\ (\mathsf{in}\ t) \to \varphi\ (m\ (\mathsf{rcd}\ (\mathsf{histo}\ m\ \varphi))\ t)$$

Para la definción de la regla anterior es necesario definir la semántica operacional de la función rcd por lo que la regla de reducción del histomorfismo queda de la siguiente forma:

$$\mathsf{histo}\ m\ \varphi\ (\mathsf{in}\ t) \to \varphi\ (m\ (\mathsf{cata}\ m\ (\mathsf{ana}\ m^\times\langle\varphi, m\ \mathsf{prev}\rangle))\ t)$$

ya que la función $\mathsf{rcd}\ h =_{def} \mathsf{cata}\ m\ (\mathsf{ana}\ m^\times\langle\varphi, m\ \mathsf{prev}\rangle)$

Por lo que la regla de reducción anterior corresponde con el principio de recursión por curso de valores.

Ejemplo 3.1 (Fibonacci)**.** La función `fibo` : $\mathbb{N} \to \mathbb{N}$ se define como un histomorfismo sobre el funtor $\langle F, \mathsf{mapN}\rangle$ con :

$$\begin{array}{rcl} F & = & \lambda X.\,1 + X \\ \mathsf{mapN} & = & \lambda f.\lambda x.\mathsf{case}(x, y.\mathsf{inl}\ y, w.\mathsf{inr}(f\ w)) \end{array}$$

con lo que la función `fibo` se define como sigue:

$$\texttt{fibo} = \mathsf{histo}\ \mathsf{mapN}\ [z, s]$$

en donde

$$\begin{aligned} z &= \lambda\_.\, 1 \\ s &= \lambda x.\mathsf{case}(\mathsf{prev}\ x, y.\, 1, z.(\mathsf{cur}\ x) + (\mathsf{cur}\ z)) \end{aligned}$$

Los histomorfismos capturan el esquema de iteración por curso de valores, con base en el concepto de recursión con memoria. Sin embargo son muy restrictivos de los puntos anteriores que pueden usarse en la evaluación de una función, ya que los resultados recursivos disponibles son únicamente aquellos estructuralmente mas simples pero que se obtienen a partir de los destructores del tipo inductivo. Esto en naturales quizá no se vea como una restricción fuerte, pues todos los puntos anteriores pueden construirse mediante el predecesor, sin embargo en estructuras como listas si representa una limitante ya que los puntos estructuralmente mas pequeños solo pueden generarse a partir de las funciones que regresan la cabeza y la cola de una lista, dejando fuera todas las sublistas que comparten elementos pero no en el mismo orden.

Por ejemplo, si se tiene la lista $[1, 2, 3, 4, 5]$ una sublista de ésta es $[1, 3, 5]$, sin embargo ésta no puede obtenerse con los destructores cabeza y cola por lo que el histomorfismo no permite llamar recursivamente sobre ella, por otro lado la lista $[3, 4, 5]$ también es sublista y si se puede generar con los destructores por lo que sobre ésta si se puede llamar recursivamente con los histomorfismos. En el caso de algoritmos sobre listas ésta es una restricción muy fuerte.

Por la razón anterior se propone un combinador con mayor flexibilidad para la definición de funciones por iteración por curso de valores, este combinado es el hilomorfismo que se define en la sección siguiente.

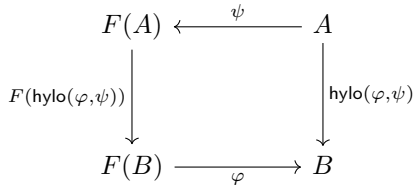
3.2. Hilomorfismos

A pesar de que los histomorfismos son suficientes para modelar recursión por curso de valores, su poder de expresividad es limitado pues se requiere que el patrón recursivo de la función sea el mismo que el del tipo inductivo, en otras palabras está limitado al uso de los destructores del tipo inductivo. Por esta razón muchos algoritmos de programación dinámica no pueden ser definidos en términos de este esquema recursivo, como es el caso de algoritmos de ordenamiento sobre listas. En esta sección se define un nuevo combinador que quita esta restricción al ser mas flexible en la forma de almacenar los puntos anteriores de la función y de esta forma capturar todos los algoritmos recursivos de programación dinámica, este operador es el *hilomorfismo*.

Aristoteles definió al hilomorfismo como la concepción de que toda entidad física está compuesta de su forma material y su forma inmaterial al mismo tiempo, en donde ambas formas son opuestas y complementarias, pero coexistiendo para construir en conjunto al ente. Siguiendo esta filosofía se define categóricamente el combinador

hilomorfismo como una composición de un anamorfismo y un catamorfismo, al estos definir comportamientos opuestos, uno de construcción y otro de destrucción.

Definición 3.1 (Hilomorfismo)**.** Sean $\langle A, \psi \rangle$ una F-co-álgebra y $\langle B, \varphi \rangle$ una F-álgebra sobre el mismo funtor F. Se define $\mathsf{hylo}(\varphi, \psi)$ como el morfismo f tal que el siguiente diagrama conmuta:

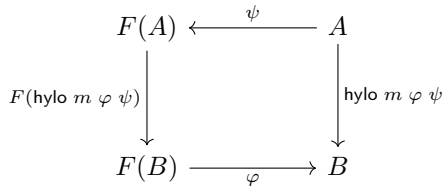


Y se cumple la ecuación: $\mathsf{hylo}(\varphi, \psi) = \mathsf{cata}\ \varphi \circ \mathsf{ana}\ \psi$ En tal caso el morfismo $\mathsf{hylo}(\varphi, \psi)$ es el hilomorfismo.

Los hilomorfismos capturan el esquema de recursión general al construir una estructura intermedia y posteriormente colapsándola para obtener un resultado. Esta estructura intermedia en la mayoría de los casos corresponde con el árbol de llamadas recursivas de una función[1]. De esta forma un hilomorfismo puede verse como la composición de un anamorfismo para construir la estructura intermedia, seguido de un catamorfismo para colapsarla en un valor, ambos sobre el mismo funtor F.

La principal ventaja de los hilomorfismos es la flexibilidad que nos da el poder elegir la estructura intermedia que se utilizará, de esta forma se pueden capturar una mayor cantidad de esquemas recursivos que con los histomorfismos, ya que el esquema recursivo de la función no debe corresponder estrictamente con el del tipo inductivo.

A continuación se define una extensión del Sistema It para agregar como nativo el combinador hylo que modela hilomorfismos. Esto con base en la conmutatividad del siguiente diagrama categórico:



De esta forma se definen la regla de tipado del combinador como sigue:

[1]Se le llama árbol de llamadas recursivas, a la estructura en forma de árbol que describe gráficamente las llamadas recursivas generadas en el proceso de evaluación de una función.

$$\frac{\begin{array}{c}\Gamma \vdash t : A \\ \Gamma \vdash m : F\mathsf{mon} \\ \Gamma \vdash \varphi : F(B) \to B \\ \Gamma \vdash \psi : A \to F(A)\end{array}}{\Gamma \vdash \mathsf{hylo}\ m\ \varphi\ \psi\ t : B}\,\mu E^{cv}$$

Es importante notar como el tipo del término t no debe ser inductivo, sino que puede ser cualquier tipo de dato y que tanto el anamorfismo, el catamorfismo y el testigo de monotonicidad son definidos sobre el funtor que modela la estructura intermedia, esto aumenta el poder de expresividad de las funciones recursivas al no limitarlas a un tipo de dato inductivo.

La siguiente regla de reducción modela la semántica operacional del combinador hylo esto con base en la idea de ver a los hilomorfismos como una composición de un anamorfismo y un catamorfismo.

$$\mathsf{hylo}\ m\ \varphi\ \psi\ t \to \mathsf{cata}\ m\ \varphi\ (\mathsf{ana}\ m\ \psi\ t)$$

Se desarrolla el mismo ejemplo que en la sección anterior (los números de fibonacci) para notar la diferencia en el uso de los operadores.

Ejemplo 3.2 (Fibonacci)**.** Para modelar la función fibonacci como un hilomorfismo:

$$\texttt{fibo} = \mathsf{hylo}\ \langle \varphi, \psi \rangle$$

Se propone el siguiente prefuntor como estructura intermedia:

$$G = 1 + X^2$$

el cúal define la estructura del árbol de llamadas recursivas de fibonacci, en donde el 1 corresponde a la hoja en donde se almacena el resultado de los casos base, mientras que la X^2 es cada nodo interno en donde se hacen dos llamadas recursivas a la función.
por lo que las funciones de paso $\varphi : G(\mathbb{N}) \to \mathbb{N}$ y $\psi : \mathbb{N} \to G(\mathbb{N})$ se definen como:

$$\begin{array}{rcl}\psi & = & \lambda n.\mathsf{case}(n, z.\mathsf{inl}\star, m.\mathsf{case}(m, x.\mathsf{inl}\star, y.\mathsf{inr}\langle m, y\rangle)) \\ \varphi & = & \lambda t.\mathsf{case}(t, x.\,1, y.(\mathsf{fst}\ y + \mathsf{snd}\ y))\end{array}$$

Es importante notar como el functor sobre el que se opera con los combinadores cata y ana no es el correspondiente a los números naturales sino el que abstrae la estructura del árbol de llamadas recursivas de la función fibo

En general la representación intermedia usada en el hilomorfismo corresponde con el árbol de llamadas recursivas, en el ejemplo anterior por ejemplo el árbol de llamadas recursivas de `fibo` 4 se ve como sigue:

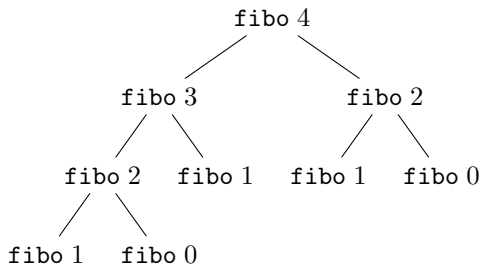


Figura 3.1: Árbol de llamadas recursivas de `fibo` 4

Se puede ver que corresponde con la estructura descrita por el prefuntor usado $1+X^2$.

En el siguiente capítulo se presentan una serie de casos de uso para estos combinadores que sirven de ejemplos para su uso.

3.3. Propiedades del sistema

En esta sección se estudian las propiedades del sistema CvIt que se define como It agregando como nativos los operadores de histo e hylo definidos en la sección anterior.

3.3.1. Seguridad del lenguaje

La seguridad de un lenguaje es una propiedad muy importante para un sistema de reescritura de términos tipados como el que se define en este trabajo, pues esta propiedad implica que el sistema es apropiado para implementarse como (parte de) un lenguaje de programación. Se demuestra esta propiedad para CvIt al probar la preservación y el progreso del sistema respecto a su semántica operacional y estática.

Primero se enuncia la propiedad de preservación que garantiza que la semántica operacional no cambia el tipo de los términos del lenguaje en el proceso de evaluación.

Proposición 3.1 (Preservación)**.** Sea e un término en CvIt tal que $\Gamma \vdash e : \mathsf{T}$ y $e \to e'$ entonces $\Gamma \vdash e' : \mathsf{T}$.

Demostración. Es importante notar que la demostración no es trivial ya que las reglas de tipado polimórfico no son dirigidas por sintaxis, por lo que la prueba queda fuera del alcance de este trabajo. Sin embargo la persona que lee debe convencerse de que se puede probar esta proposición de forma similar a la demos-

tración presentada para el sistema ItRec en [20] que a su vez está basada en la prueba de la misma proposición para el sistema F en [7]. □

Ahora definimos la propiedad de progreso que garantiza que un término bien tipado del lenguaje no será bloqueado en la semántica operacional.

Proposición 3.2 (Progreso)**.** Sea e una expresión de CvIt tal que $\Gamma \vdash e : \mathsf{T}$, entonces se cumple una y sólo una de las siguientes:

- e es un valor.
- Existe e' tal que $e \to e'$

Demostración. Directa por inducción sobre $\vdash$ □

Las propiedades de preservación y progreso son sumamente importantes en el estudio de lenguajes de programación ya que modelan la interacción de los dos niveles semánticos del lenguaje.

3.3.2. Terminación

Por último se probara la propiedad de terminación o normalización fuerte del sistema, la cual garantiza que la semántica operacional siempre termina para todos los términos bien tipados del sistema, es decir, siempre se llega a un valor en el proceso de evaluación.

Proposición 3.3 (Terminación)**.** El lenguaje CvIt es fuertemente normalizable.

Demostración. se demuestra al probar que ambos combinadores histo e hylo son azúcar sintáctica de los operadores definidos en It, y como se demuestra en [20] It cumple la propiedad de terminación.

- histo se puede desendulzar como $\mathsf{cur}(\mathsf{rcd}\ h\ (\mathsf{in}\ t)) \to^{+} h\ (\mathsf{in}\ t)$ y la función rcd está definida directamente en It.
- El caso de hylo es mas evidente ya que su semántica operacional está definida en términos de los operadores cata y ana nativos de It

Por lo tanto, CvIt es fuertemente normalizable. □

En este capítulo se agregaron los combinadores categóricos que modelan el esquema de iteración por curso de valores como operadores del sistema It. En el siguiente capítulo se presentan ejemplos concretos del uso de estos operadores para exhibir la importancia de este esquema recursivo en distintos casos de estudio.

Capítulo 4

Casos de Estudio

En este capítulo se desarrollan una serie de casos de estudio en los que se usa el patrón de recursión por curso de valores para definir funciones que den solución a estos problemas. Se eligieron estos ejemplos porque exhiben casos interesantes dentro del patrón recursivo estudiado.

Algunos de los ejemplos siguientes se pueden resolver tanto con histomorfismos como con hilomorfismos, sin embargo hay otros en los que no es posible presentar una solución usando el combinador histomorfismo, en cada sección se especifica el por que entra o no dentro del patrón que definen estos combinadores.

4.1. Tríangulo de Pascal

Como primer caso de estudio se presenta la definición del triángulo de Pascal, el cuál es una representación gráfica de los coeficientes binomiales ordenados en forma de triángulo. Este recibe su nombre en honor al filósofo y matemático Blaise Pascal quien introdujo esta notación en 1654, en su Tratado del triángulo aritmético [25].

El triángulo de Pascal se construye siguiendo el siguiente patrón: Se comienza desde la punta del triángulo con el número 1 hacia abajo, a modo de árbol; se clasifica en filas, empezando por la fila cero que solo incluye al 1 de la punta. Este árbol tiene nodos, que son cada número que compone el triángulo. Si sumamos dos nodos nos dará de resultado el nodo situado debajo de estos dos, y así sucesivamente.

Visualmente puede verse como se muestra en la siguiente figura, en donde se muestran los primeros 8 niveles del triángulo.

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
 & 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1 \\
1 & & 7 & & 21 & & 35 & & 35 & & 21 & & 7 & & 1
\end{array}
$$

El triángulo de Pascal tiene distintos usos en la matemática siendo el principal el desarrollo de las potencias de los binomios dadas por a fórmula $(a + b)^n$ en donde n es el nivel que corresponde dentro del triángulo. También se puede usar para el estudio de combinatoria y fractales entre otras áreas.

La función que calcula el valor del triángulo en una columna y fila especificas se define en el lenguaje de programación Haskell como sigue:

```
pas :: Nat -> Nat -> Nat
pas 0 _ = 1
pas _ 0 = 1
pas r c = pas r (c - 1) + pas (r - 1) c
```

Ésta es una función recursiva que sigue el esquema de recursión por curso de valores ya que la evaluación de `pas` en un punto especifico, requiere no solo el resultado el el punto anterior sino el resultado en el punto de la fila anterior y el resultado en la columna anterior. Por lo que se puede definir con los combinadores de histomorfismo e hilomorfismo.

4.1.1. Como un histomorfismo

Para definir `pas` como un histomorfismo se buscan las expresiones z y s que satisfagan la siguiente ecuación. En donde `mapN` es el testigo de monotonicidad para los naturales como se definió en el capítulo 3.

$$\texttt{pas} = \texttt{hist mapN}\ [z, s]$$

en donde las funciones de paso $z : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$ y $s : \mathsf{CoList}_{\mathsf{Nat}} \times \mathsf{CoList}_{\mathsf{Nat}} \to \mathsf{Nat}$ se definen como:

$$
\begin{aligned}
z &= \lambda\_.\, 1 \\
s &= \lambda p.(\mathsf{prev}\ (\mathsf{fst}\ p)) + (\mathsf{prev}\ (\mathsf{snd}\ p))
\end{aligned}
$$

Se reescriben las funciones usando *pattern matching*[1] de manera similar a la sintaxis que se usa en el lenguaje de programación Haskell, esta sintaxis no está definida en el sistema CvIt pero es una buena técnica para simplificar la lectura de las ecuaciones anteriores.

$$\begin{aligned} z\ \langle 0, c\rangle &= 1 \\ z\ \langle r, 0\rangle &= 1 \end{aligned}$$

$$s\ \langle r, c\rangle = (\texttt{prev}\ r) + (\texttt{prev}\ c)$$

Es importante notar que la sintaxis de *pattern matching* es simplemente otra notación, pero las definiciones de las funciones son equivalentes.

4.1.2. Como un hilomorfismo

Para definir la función `pas` mediante el operador `hylo` dentro del sistema presentado en este trabajo, se propone el funtor

$$F(X) = 1 + X^2$$

Que abstrae la estructura intermedia con la que se representa el árbol de llamadas recursivas de la función, siendo el 1 el caso base para cuando la fila o la columna buscadas con 1 mientras que la X^2 modela el caso recursivo en donde se tienen 2 llamadas a la función en distintos puntos.

De esta forma se define `pas` como:

$$\texttt{pas} = \texttt{hylo}\ \varphi\ \psi$$

En donde las funciones de paso $\varphi : F(\mathsf{Nat}) \to \mathsf{Nat}$ y $\psi : \mathsf{Nat} \times \mathsf{Nat} \to F(\mathsf{Nat}^2)$ se definen de la siguiente forma:

$$\psi = \lambda p.\mathsf{case}(\mathsf{fst}\, p, x.\mathsf{inl}\star, r.\mathsf{case}(\mathsf{snd}\ p, z.\mathsf{inl}\star, c.\mathsf{inr}\langle\langle r, c-1\rangle.\langle r-1, c\rangle\rangle))$$

$$\varphi = \lambda p.\mathsf{case}(p, x.\,1, y.\mathsf{fst}\ y + \mathsf{snd}\ y)$$

De igual forma se reescriben con *pattern matching*.

$$\begin{aligned} \psi\ \langle 0, c\rangle &= \mathsf{inl}\ \star \\ \psi\ \langle r, 0\rangle &= \mathsf{inl}\ \star \\ \psi\ \langle r, c\rangle &= \mathsf{inr}\ \langle r, c-1\rangle, \langle r-1, c\rangle \end{aligned}$$

[1] La traducción a español mas cercana es coincidencia de patrones.

$$\begin{aligned} \varphi\ (\mathsf{inl}\ \star) &= 1 \\ \varphi\ (\mathsf{inr}\ \langle p_1, p_2 \rangle) &= p_1 + p_2 \end{aligned}$$

De esta forma se puede ver que la función que encuentra los elementos en el triángulo de pascal puede definirse como un histomorfismo y un hilomorfismo.

4.2. Partición binaria

El número de particiones binarias de un natural n es la cantidad de formas en la que n puede descomponerse como una suma de potencias de 2, sin considerar el orden [26]. Por ejemplo el número 4 tiene 4 particiones binarias, las cuales son: $\langle 1,1,1,1 \rangle, \langle 2,1,1 \rangle, \langle 2,2 \rangle$ y $\langle 4 \rangle$.

En Haskell se puede definir la función `bp` que calcula el número de particiones binarias de un número n de la siguiente forma:

```
bp :: Nat -> Nat
bp 0         = 1
bp n
 | odd n     = bp (n - 1)
 | otherwise = bp (n - 1) + bp (n `div` 2)
```

Esta es una función recursiva, que sigue el esquema de recursión por curso de valores por lo que podemos definirla con los combinadores de histomorfismo e hilomorfismo.

4.2.1. Como un histomorfismo

La definición de `bp` como un histomorfismo debe satisfacer la igualdad siguiente.

$$\texttt{bp}\ n = \texttt{hist mapN}\ [z, s]$$

en donde las funciones de paso $z : \mathsf{Nat} \to \mathsf{Nat}$ y $s : \mathsf{CoList}_{\mathsf{Nat}} \to \mathsf{Nat}$ se definen como:

$$z = \lambda\_.\, 1$$

$$s = \lambda \ell.\texttt{if (odd n) then (cur } \ell\texttt{) else (cur } \ell\texttt{)} + \ell!!(\texttt{n}/2 - 1)$$

4.2.2. Como un hilomorfismo

Para definir `bp` mediante el operador `hylo` se propone el funtor

$$F(X) = 1 + X^2$$

Para modelar el árbol de llamadas recursivas. El 1 es el caso base, que corresponde a la función evaluada en el punto 0, mientras que la X^2 modela las dos llamadas recursivas necesarias para evaluar la función en el punto n, la primera correspondiente al punto anterior $n-1$ mientras que la segunda es la función evaluada en el punto $n/2$.

De esta forma se define bp como:

$$\texttt{bp} = \texttt{hylo}\ \varphi\ \psi$$

En donde las funciones de paso $\varphi : F(\mathsf{Nat}) \to \mathsf{Nat}$ y $\psi : \mathsf{Nat} \to F(\mathsf{Nat})$ se definen de la siguiente forma:

$$\psi \quad = \quad \lambda n.\mathsf{case}(n, z.\mathsf{in}_1\ \star, \texttt{m.if (odd n) then } \mathsf{in}_2\ \texttt{m else } \mathsf{in}_3\ (\texttt{m}, \texttt{n}/2))$$

$$\varphi \quad = \quad \lambda p.\mathsf{case}_3(p.x, 1, y.y, z.\mathsf{fst}\ z + \mathsf{snd}\ z)$$

En donde case_3 es azúcar sintáctica para referirnos a un operador case ternario, que se construye al anidar dos case. En general se usa case_i para referirnos a la generalización del operador case con i posibles casos.

Como otra alternativa para simplificar la lectura, al igual que en la sección anterior se usa *pattern matching*.

$$\begin{array}{rcl} \psi\ 0 & = & \mathsf{in}_1\ \star \\ \psi\ n & = & \begin{cases} \mathsf{in}_2\ n-1 & \texttt{odd } n \\ \mathsf{in}_3\ (n-1, n/2) & \text{en otro caso} \end{cases} \end{array}$$

$$\begin{array}{rcl} \varphi\ (\mathsf{in}_1\ \star) & = & 1 \\ \varphi\ (\mathsf{in}_2\ n) & = & n \\ \varphi\ (\mathsf{in}_3\ (n, m)) & = & n + m \end{array}$$

Así se puede observar que la función que indica la cantidad de particiones binarias de un número se puede definir con los esquemas de histomorfismos e hilomorfismos.

4.3. La subsecuencia común mas larga

El problema de la subsecuencia común mas larga es un ejemplo clásico de programación dinámica. En este capítulo se presentan distintas soluciones a este problema, desde un punto de vista meramente funcional y también mediante el uso de los combinadores para recursión por curso de valores definidos en este trabajo [27].

El problema de la subsecuencia común mas larga consiste en lo siguiente: dadas dos secuencias de valores, se busca encontrar una subsecuencia presente en ambas de tal forma que no existe una de longitud mayor. En donde una subsecuencia se define como una secuencia de valores que aparecen en el mismo orden, aunque no necesatiamente de forma contigua.

Pro ejemplo, para la cadena de texto `"abcdefg"` algunas subsecuencias son: `"abc"`, `"abg"`, `"bdf"`, `"aeg"`, `"acefg"`, etc.

Por simplicidad en el desarrollo de este caso de estudio se usarán secuencias de caracteres, en donde las secuencias se definen como listas, es decir, trabajaremos con cadenas de texto, bajo el entendido de que esto es generalizable a listas de valores de cualquier tipo de dato.

4.3.1. Una solución ingenua

La solución ingenua a este problema es evidente, primero se buscan todas las posibles subsecuencias de las listas, se eligen aquellas en común y seleccionamos aquella de longitud mayor. Esta solución se define con el programa declarativo siguiente:

```
lcs :: [a] -> [a] -> [a]
lcs xs ys = longest $ common (subsqs xs) (subsqs ys)
```

Primero la función `subsqs` se encarga de calcular todas las subsecuencias de una lista.

```
subsqs :: [a] -> [[a]]
subsqs []     = []
subsqs (x:xs) = map (x:) (subsqs xs) ++ subsqs xs
```

Después de obtener las subsecuencias de las listas `xs` y `ys` se filtran para conservar unicamente aquellas que sean subsecuencias de ambas listas. Definido en la función `common` como sigue:

```
common :: [a] -> [a] -> [a]
common xs ys = filter elm xs
 where elm x = elem x ys
```

Por último se busca la subsecuencia común de mayor longitud mediante la función `longest` definida de la siguiente forma.

```
longest :: [[a]] -> [a]
longest = sortBy f
 where f x y = compare (length x) (length y)
```

Ésta es una simple y elegante solución en donde la especificación del problema queda plasmado en la definición de la solución, es decir, es una solución declarativa. Sin embargo presenta un problema, su eficiencia, específicamente la complejidad en tiempo de esta solución.

El algoritmo ingenuo para encontrar la subsecuencia mas larga entre dos listas tiene una complejidad $O(n^2)$ ya que tiene que calcular todas las posibles subsecuencias de ambas cadenas y posteriormente operar con ellas para encontrar la mas larga. Esta complejidad se puede mejorar al evitar trabajar con las listas de subsecuencias explícitamente. En la sección siguiente se propone una solución con complejidad en tiempo $O(n)$.

4.3.2. Una solución lineal

La idea intuitiva detrás de la solución lineal es ir iterando sobre las listas encontrando aquellos elementos que tengan en común y agregándolos a la subsecuencia mas larga. El proceso termina cuando alguna de las dos listas es la vacía, ya que no hay mas elementos que comparar. De esta forma se evita el calcular subsecuencias innecesarias y solo se calculan aquellas que de principio son comunes en ambas listas. Esta solución se define en Haskell como sigue:

```
lcs :: [a] -> [a] -> [a]
lcs [] _          = []
lcs _ []          = []
lcs (x:xs) (y:ys)
 | x == y = x : lcs xs ys
 | x /= y = longest (lcs (x:xs) ys) (lcs xs (y:ys))
```

La función `longest` regresa la lista con longitud mayor entre las dos listas que recibe y se define como sigue:

```
longest :: [a] -> [a] -> [a]
longes xs ys
 | lx > ly   = xs
 | otherwise = ys
where lx = length xs
      ly = length ys
```

Se puede observar que esta solución recorre a lo más una vez cada lista para ya que al mismo tiempo que va construyendo las subsecuencias verifica que sean con los elementos en común. Y se detiene cuando termina de iterar sobre la lista de longitud menor. Por estas razones la complejidad en tiempo de esta solución es $O(n)$.

4.3.3. Hilomorfismos

En esta sección se presenta una implementación de hilomorfismos en Haskell, en la cual se define como estructura intermedia el árbol de llamadas recursivas. La idea central es traducir la definición del hilomorfismo como una composición de un operador `fold` y un `unfold`. En donde `unfold` construye el árbol mientras que `fold` lo consume. En secciones siguientes se usa esta implementación para solucionar el problema de la cifra exacta.

La estructura arbórea intermedia generada por el operador `unfold` se define con el siguiente tipo de dato algebraico:

```
data Tree a = Leaf a | Node [Tree a]
```

Definimos ahora los operadores de plegado par esta estructura que serán los componentes del hilomorfismo.

```
fold :: (a -> b) ->  ([b] -> b) -> Tree a -> b
fold l _ (Leaf x)  = l x
fold l n (Node ts) = n $ map (fold l n) ts

unfold :: (b -> Bool) -> (b -> a) -> (b -> [b]) -> b -> Tree a
unfold p f g x
| p x = Leaf (f x)
| otherwise = Node $ map (unfold p f g) (g x)
```

De esta forma el hilomorfismo general sobre `Tree a` se define como sigue:

```
hylo :: (a->b)->([b]->b)->(c->Bool)->(c->a)->(c->[c])->c->b
hylo f g p v h = fold f g . unfold p v h
```

En la definición anterior se puede evitar la construcción del árbol como representación intermedia, a este proceso de evitar estructuras innecesarias en un programa se le conoce como *deforestación*. Esta idea se implementa con la siguiente versión del hilomorfismo:

```
hylo' :: (a->b)->([b]->b)->(c->Bool)->(c->a)->(c->[c])->c->b
hylo' f g p v h
 | p x = f x
 | otherwise = g $ map hylo' (h x)
```

Sin embargo para en muchos casos es conveniente contar explícitamente con el árbol de llamadas recursivas, pues ofrece información adicional sobre la ejecución del programa. El problema de la cifra exacta es uno de esos casos, en donde el árbol generado es importante para encontrar la solución, el por qué será evidente mas adelante.

Opuesto al proceso de deforestación existe la idea de *anotar*, en donde se conserva la estructura arbórea hasta el final del computo, pero se agregan etiquetas a cada nodo que acumula el valor del hilomorfismo para el subárbol definido por el nodo. Para implementar arboles etiquetados se define el siguiente tipo de dato:

```
data LTree a = LLeaf a | LNode a [LTree a]
```

Estructuralmente es igual a `Tree` salvo por que los nodos internos del árbol también guardan información.

La idea es a partir de un `Tree` construir un `LTree` al anotar sus nodos internos con el resultado pero preservando la estructura, en donde la etiqueta es el resultado de aplicar el operador `fold` al subárbol y de esta forma la raíz almacena el resultado del hilomorfismo. Este comportamiento se abstrae en la función `fill` definida como sigue:

```
fill :: (a -> b) -> ([b] -> b) -> Tree a -> LTree b
fill f g = fold (lleaf f) (lnode g)
```

En donde se utilizan constructores inteligentes para el tipo `LTree`, definidos como:

```
lleaf :: (a -> b) -> a -> LTree b
lleaf f = LLeaf . f

lnode :: ([a] -> a) -> [LTree a] -> LTree a
lnode f ts = LNode (f $ map label ts) ts
```

La función `label` recupera el valor almacenado en la raíz de un árbol etiquetado:

```
label :: LTree a -> a
label (LLeaf x)   = x
label (LNode x _) = x
```

De esta forma se puede implementar el operador `hylo` construyendo explícitamente el árbol etiquetado:

```
hylo::(a->b)->([b]->b)->(c->Bool)->(c->a)->(c->[c])->c->b
hylo f g p v h = label . fill f g . unfold p v h
```

En la siguiente sección se usa esta definición de hilomorfismo para computar todas las posibles subsecuencias de una lista de elementos.

4.3.4. Las subsecuencias

Las subsecuancias de una lista son todas aquellas sub-listas que se pueden construir a partir de los elementos de la lista original. Las subsecuencias se pueden calcular mediante un proceso iterativo en donde en cada iteración quitamos un elemento de la lista, esta iteración se captura en la función `minors` definida como sigue:

```
minors :: [a] -> [[a]]
minors [x,y]  = [[x],[y]]
minors (x:xs) = map (x:) (minors xs) ++ [xs]
```

Por ejemplo

```
minors "abcde" = ["abcd", "abce", "abde", "acde", "bcde"]
```

Para construir un árbol etiquetado con todas las posible subsecuencias de una lista, se aplica la función `minors` para encontrar los hijos de cada nodo. En la figura 4.1 puede verse un ejemplo del árbol para la cadena `"abcd"`.

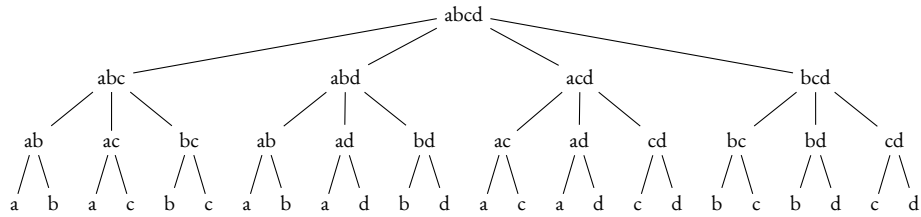


Figura 4.1: Árbol etiquetado de subsecuencias para la cadena `"abcd"`

De la definición del hilomorfismo de la sección anterior se recupera la construcción del árbol etiquetado y se define como una función independiente `mkTree`:

```
mkTree :: (b -> Bool) -> (b -> a) -> (b -> [b]) -> b -> Tree a
mkTree f g h = unfold f g h
```

Se puede observar que en la construcción del árbol de subsecuencias las hojas almacenan los valores iniciales de la lista, por lo que la función `g :: b -> a` es la función `id`. Igualmente, la construcción del árbol se va a detener cuando se llega a un nivel en el que solo hay un elemento, la raíz del árbol, en la figura 4.1 corresponde al nodo con la cadena completa `"abcd"`, este predicado puede definirse mediante una función `single`:

```
single :: [a] -> Bool
single (_:[]) = True
single xs     = False
```

Por otro lado la función de paso es la función `minors` que calcula las subsecuencias inmediatas de una lista. Por lo que la función `mkTree` para el árbol de subsecuencias puede definirse como:

```
mkTree :: [a] -> Tree [a]
mkTree = unfold single id minors
```

Esta función construye un árbol sin etiquetas, por lo que en realidad el árbol generado es el que se puede ver en la figura 4.2. Para anotar el árbol y llegar al que se muestra en la figura 4.1 se define la siguiente función:

```
mkLTree :: Tree [a] -> LTree [a]
mkLTree = fill id recover
```

En donde `recover` esta implementado como sigue:

```
recover :: [[a]] -> [a]
recover xss = head (head xss) : last xss
```

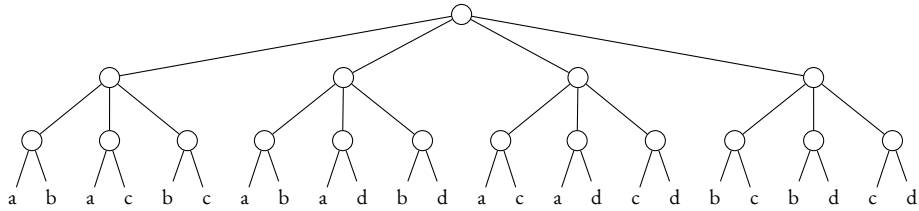


Figura 4.2: Árbol de subsecuencias para la cadena `"abcd"`

4.3.5. Soluciones basadas en Hilomorfismos

En esta sección se presentan las tres soluciones vistas al principio del capitulo mediante el uso de los combinadores de hilomorfismos.

Solución ingenua

A partir del árbol de subsecuencias que se definió al final de la sección anterior se puede definir la solución ingenua del problema de la subsecuancia mas larga al calcular las subsecuencias de cada lista usando el hilomorfismo que construye este árbol. Esto se logra al cambiar la definición de `subsqs` para que construya este árbol.

```
subsqs :: [a] -> [[a]]
subsqs xs = nub $ flat sqsTree
 where sqsTree = mkTree xs
```

En donde `flat` es la función que aplana el árbol de subsecuencias en una lista que la contenga todas. Definida como sigue:

```
flat :: Tree [a] -> [[a]]
flat (Leaf a)    = [a]
flat (Node a ts) = a: map flat ts
```

Esta función cabe dentro del patrón definido con el operador `fold` que define el catamorfismo de la estructura `Tree`, por lo que se puede redefinir de la siguiente forma:

```
flat :: Tree [a] -> [[a]]
flat = fold id (:)
```

De esta forma, es claro que la función `subsqs` que encuentra todas las subsecuencias de una lista es un hilomorfismo, en donde la estructura intermedia definida es el árbol de subsecuencias construido mediante un anamorfismo y posteriormente reducido con un catamorfismo a una lista con la función `flat`.

Sin embargo, este cambio no afecta a la complejidad en tiempo del algoritmo ingenuo ya que de igual forma se calculan todas las subsecuencias de ambas listas y el resto del proceso para encontrar la de mayor tamaño se conserva. Adicional a esto se tiene que construir el árbol de subsecuencias lo que aumenta el tiempo de ejecución, pero se conserva la cota de $O(n^2)$

Solución lineal

El objetivo de la solución lineal es evitar el cálculo de todas las subsecuencias de las listas, por esta razón usar el árbol de subsecuencias como estructura intermedia no haría mas que afectar negativamente la complejidad en tiempo de la solución. Por esto se propone una nueva estructura intermedia en donde se almacenen únicamente los valores necesarios para obtener la solución buscada con base en la función `lcs`.

```
lcs :: [a] -> [a] -> [a]
lcs [] _          = []
lcs _ []          = []
lcs (x:xs) (y:ys)
 | x == y = x : lcs xs ys
 | x /= y = longest (lcs (x:xs) ys) (lcs xs (y:ys))
```

En la definición lineal de `lcs` se puede observar que las cabezas de ambas listas son necesarias en la llamada recursiva para compararlas, en caso de ser iguales el resultado depende de la llamada sobre las colas de ambas listas `xs` y `ys`, en el otro caso se regresa

la subsecuencia mas larga entre las dos llamadas recursivas conservando la cabeza en la primera lista o en la segunda respectivamente.

Con este análisis sobre la función `lcs` se puede ver que los valores necesarios para encontrar la solución son las cabezas de las listas de entrada y las tres posibles llamadas recursivas, por lo que la estructura intermedia debe almacenar estos datos.

Una vez construida esta estructura intermedia, la subsecuencia mas larga se va a encontrar al doblar esta estructura comparando las cabezas y usando el resultado de la llamada recursiva correspondiente.

Dentro del sistema

Para definir la solución lineal al problema de la subsecuencia mas larga mediante el operador `hylo` dentro del sistema presentado en este trabajo, se propone el álgebra

$$F(X) = 1 + C^2 \times X^3$$

La cual abstrae la estructura en donde se almacenan los primeros elementos de la secuencia (C^2) junto con el resultado de las tres posibles llamadas recursivas (X^3) para poder así construir el resultado.

De esta forma se define `lcs` como:

$$\texttt{lcs} = \texttt{hylo}\ \varphi\ \psi$$

En donde las funciones de paso $\varphi : \mathsf{List}_C \times \mathsf{List}_C \to F(\mathsf{List}_C^2)$ y $\psi : F(\mathsf{List}_C) \to \mathsf{List}_C \times \mathsf{List}_C^2$ se definen de la siguiente forma:

$$\begin{aligned}
\psi \quad = \quad & \lambda p.\mathsf{case}(\mathsf{fst}\ p, x.\mathsf{inl}\star, as.\mathsf{case}(\mathsf{snd}\ p, z.\mathsf{inl}\star, \\
& bs.\mathsf{inr}\langle \texttt{head}\ as, \texttt{head}\ bs\rangle, \langle \texttt{tail}\ as, bs\rangle, \langle as, \texttt{tail}\ bs\rangle, \langle \texttt{tail}\ as, \texttt{tail}\ bs\rangle))
\end{aligned}$$

$$\begin{aligned}
\varphi \quad = \quad & \lambda t.\mathsf{case}(t, x.[\,], \\
& y.\texttt{if}\ \mathsf{fst}\ (\mathsf{py}_1\ \mathsf{y}) = \mathsf{snd}\ (\mathsf{py}_1\ \mathsf{y})\ \mathsf{fst}\ (\mathsf{py}_1\ \mathsf{y}) : (\mathsf{py}_4\ \mathsf{y})\ \texttt{else} \\
& \texttt{if}\ \#(\mathsf{py}_2\ y) > \#(\mathsf{py}_3\ y)\ \texttt{then}\ \mathsf{py}_2\ y\ \texttt{else}\ \mathsf{py}_3\ y)
\end{aligned}$$

En este caso se utilizan como constructores nativos del sistema las proyecciones de la forma py_i que son una generalización de los operadores fst y snd para tipos suma de anidados, con el fin de simplificar la notación en las expresiones. Estos operadores se pueden representar como azúcar sintáctica dentro del sistema mediante el uso de las proyecciones ya definidas, por lo que su presencia en las expresiones no modifica la semántica de ésta. Las definiciones de estas proyecciones generalizadas sigue la siguiente lógica:

$$
\begin{aligned}
\mathsf{py}_1 &= \mathsf{fst} \\
\mathsf{py}_2 &= \mathsf{fst} \circ \mathsf{snd} \\
\mathsf{py}_3 &= \mathsf{fst} \circ \mathsf{snd} \circ \mathsf{snd} \\
&\vdots \\
\mathsf{py}_i &= \mathsf{fst} \circ \underbrace{\mathsf{snd} \circ \cdots \circ \mathsf{snd}}_{i-1 \text{ veces}}
\end{aligned}
$$

Para este ejemplo es mucho mas evidente la necesidad de *pattern matching* para simplificar la escritura y lectura de las ecuaciones anteriores, esto debido a que las expresiones son en si complejas y la sintaxis presentada para el sistema CvIt hace mas compleja la estructura de las expresiones.

Por este motivo de aquí en adelante sólo se presentan las funciones usando ésta notación sin aviso para la persona que lee.

$$
\begin{aligned}
\psi\ ([\,], bs) &= \mathsf{inl}\ \star \\
\psi\ (as, [\,]) &= \mathsf{inl}\ \star \\
\psi\ ((a : as), (b : bs)) &= \mathsf{inr}\ ((a, b), (as, (b : bs)), ((a : as), bs), (as, bs))
\end{aligned}
$$

$$
\begin{aligned}
\varphi\ (\mathsf{inl}\ \star) &= [\,] \\
\varphi\ (\mathsf{inlr}\ ((a, b), x, y, z)) &= \texttt{if } a = b \texttt{ then } a : z \texttt{ else if } \#x > \#y \texttt{ then } x \texttt{ else } y
\end{aligned}
$$

Todas estas soluciones basadas en hilomorfismos ilustran que el problema de la subsecuencia mas larga cabe en el esquema de recursió por curso de valores que abstrae el combinador hilomorfismo. Sin embargo no es posible definirlo con histomorfimos, esto se debe a la variedad de sublistas sobre las que se hacen las llamadas recursivas del algoritmo pues el esquema del histomorfismo se restringe unicamente a las listas estructuralmente inmediatas, es decir, a las generadas a partir de `tail`.

4.4. Merge Sort

Merge Sort es un algoritmo de ordenamiento basado en la técnica de *divide y vencerás*, la idea es dividir la lista por la mitad y recursivamente ordenar cada mitad con el mismo algoritmo, hasta llegar a las listas unitarias que por definición siempre se encuentran ordenadas. Una vez ordenadas las dos mitades se combinan de nuevo en una sola mediante una función `merge` la cuál se encarga de combinar ambas listas ordenadas preservando el orden, es decir, toma dos listas ordenadas y regresa una lista con los elementos de ambas que también se encuentra ordenada.

Este algoritmo es interesante de analizar ya que se define mediante un esquema recursivo sobre listas en donde las sublistas sobre las cuales se hace recursión no son

resultado de los destructores del tipo de dato, es decir, las sublistas no se definen con el uso de las funciones `head` y `tail`. Pero aun así son sublistas generadas a partir de los elementos de la lista original. Es por esta razón que este algoritmo no puede implementarse dentro del esquema de los histomorfismos, pero si dentro del hilomorfismo al tener mayor flexibilidad en la estructura intermedia generada.

A continuación se presenta la implementación del algoritmo *Merge Sort* con la función `msr`

```
msr :: (Ord a) => [a] -> [a]
msr [] = []
msr [x] = [x]
msr xs = merge f s
 where h       = div (length xs) 2
       (f',s') = splitAt h xs
       f       = msr f'
       s       = msr s'
```

En donde la función `merge` que combina los elementos de las listas se define como sigue:

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys               = ys
merge xs []               = xs
merge l1@(x:xs) l2@(y:ys)
 | x < y                  = x : merge xs l2
 | otherwise              = y : merge l1 ys
```

En la sección siguiente se da una definición de este algoritmo usando hilomorfismos.

4.4.1. Dentro del sistema

Para definir *Merge Sort* como un hilomorfismo se propone el siguiente funtor para modelar la estructura intermedia necesaria:

$$F(X) = 1 + C + X^2$$

El 1 corresponde al caso de la lista vacía dentro de nuestra función en donde no es necesario almacenar ningún valor en la estructura intermedia, el C corresponde a las listas unitarias en donde lo único que se almacena es el elemento de ésta. Y por último el X^2 abstrae la llamada recursiva sobre las dos mitades de la lista original. De esta forma se define `ms` como:

$$\texttt{ms} = \texttt{hylo}\ \varphi\ \psi$$

En donde las funciones de paso $\varphi : F(\mathsf{List}_C) \to \mathsf{List}_C$ y $\psi : \mathsf{List}_C \to F(\mathsf{List}_C)$, es interesante observar como los tipos de las funciones de paso φ y ψ son inversos entre sí, esto se debe a que ψ construye una estructura intermedia a partir de una lista y a su vez φ destruye esta estructura para volver a obtener una lista. Y se definen de la siguiente forma:

$$\begin{array}{lcl} \psi\ [\,] & = & \mathsf{in}_1\ \star \\ \psi\ [x] & = & \mathsf{in}_2\ x \\ \psi\ xs & = & \mathsf{in}_3\ (l, r) \end{array}$$

l y r son la primera y segunda mitad de la lista respectivamente.

$$(l, r) \quad = \quad \texttt{splitAt}\ (\#xs/2)\ xs$$

En donde $\#$ calcula la longitud de la lista xs.

$$\begin{array}{lcl} \varphi\ (\mathsf{in}_1\ \star) & = & [\,] \\ \varphi\ (\mathsf{in}_2\ x) & = & [x] \\ \varphi\ (\mathsf{in}_3\ (l, r) & = & \texttt{merge}\ l\ r \end{array}$$

En donde `merge` es la función definida en la sección anterior que combina dos listas ordenadas. Sin embargo, se puede observar que la misma función `merge` es también una función recursiva que se puede definir usando el operador `hylo`, como se muestra a continuación.

4.4.2. Merge como un hilomorfismo

Para definir `merge` mediante un hilomorfismo se propone el siguiente funtor:

$$G(X) = \mathsf{List}_C + C^2 \times X^2$$

Así se define `merge` como:

$$\texttt{merge} = \texttt{hylo}\ \varphi\ \psi$$

Con las funciones de paso $\varphi : G(\mathsf{List}_C) \to \mathsf{List}_C$ y $\psi : \mathsf{List}_C^2 \to G(\mathsf{List}_C^2)$ para generar y posteriormente doblar la estructura intermedia se definen como sigue:

$$\begin{array}{lcl} \psi\ ([\,], ys) & = & \mathsf{inl}\ ys \\ \psi\ (xs, [\,]) & = & \mathsf{inl}\ xs \\ \psi\ ((x : xs), (y : ys)) & = & \mathsf{inr}\ (x, y, (xs, (y : ys)), ((x : xs), ys)) \end{array}$$

$$
\begin{array}{lcl}
\varphi\ (\mathsf{inl}\ xs) & = & xs \\
\varphi\ (\mathsf{inr}\ (a, b, x, y) & = & \texttt{if}\ a < b\ \texttt{then}\ a : x\ \texttt{else}\ b : y
\end{array}
$$

En este ejemplo es mas evidente la flexibilidad que ofrecen los hilomorfismos sobre los histomorfimos en la estructura de las listas sobre las que se hacen las llamadas recursivas, al ser estas sublistas que no fueron generadas por los destructores de `head` y `tail`, pero aún así entran dentro del esquema de la recursión por curso de valores.

4.5. Quick Sort

Quick Sort es un algoritmo de ordenamiento propuesto por el científico de la computación Charles Antony Richard Hoare en el año 1961. Es un algoritmo que se sigue utilizando en la actualidad ya que una buena implementación de este llega a ser mas rápido, en la mayoría de los casos, que otros algoritmos con la misma complejidad como lo es *Merge Sort* ambos con complejidad en tiempo de $O(n \log n)$.

Este algoritmo está basado en la idea de seleccionar un pivote y ordenar el resto de los elementos acorde a este. De esta forma, se divide la lista original en dos partes, aquellos elementos menores al pivote y los que son mayores. Y recursivamente se ordenan ambas listas, hasta llegar a la lista vacía que por definición está ordenada.

En la primera versión de este algoritmo se tomaba la cabeza de la lista como el pivote, sin embargo en algunos casos la elección de este elemento como pivote resultaba en un tiempo de ejecución de $O(n^2)$ por ejemplo con listas ya ordenadas en donde una lista contiene 0 elementos mientras que la otra tendría $n - 1$. Por esta razón se buscaron distintas eucarísticas para la elección del pivote de tal forma que la distribución de los elementos entre las dos listas no sea tan dispareja.

Para este trabajo se presenta la versión original de *Quick Sort*, usando la cabeza como pivote, ya que la eficiencia del algoritmo no es el objeto de estudio en este ejemplo sino la estructura recursiva del mismo la cuál no cambia sin importar el pivote.

Al igual que en *Merge Sort* este algoritmo obtiene dos sublistas sin el uso de los destructores `head` y `tail`, por lo que no cabe en el esquema recursivo de un histomorfismo pero si en el de un hilomorfismo.

Primero se presenta la definición de la función `qsr` que define el algoritmo *Quick Sort* con recursión general.

```
qsr :: (Ord a) => [a] -> [a]
qsr []     = []
qsr (x:xs) = l ++ x:h
 where h = qsr [e | e <- xs, e > x]
       l = qsr [e | e <- xs, e < x]
```

4.5.1. Dentro del sistema

Para definir el algoritmo `Quick Sort` mediante un hilomorfismo se propone el siguiente funtor para modelar la estructura intermedia:

$$F(X) = 1 + C \times X^2$$

el 1 corresponde al caso de la lista vacía en donde no es necesario almacenar nada en la estructura, mientras que X^2 define el caso recursivo sobre las dos sublistas. De esta forma se define `qs` como:

$$\texttt{qs} = \texttt{hylo}\ \varphi\ \psi$$

En donde las funciones de paso $\varphi : F(\mathsf{List}_C) \to \mathsf{List}_C$ y $\psi : \mathsf{List}_C \to F(\mathsf{List}_C)$ se definen de la siguiente forma:

$$\begin{array}{lcl} \psi\ [\,] & = & \mathsf{inl}\ \star \\ \psi\ (x : xs) & = & \mathsf{inr}\ (x, l, h) \end{array}$$

l es la lista de todos los elementos de xs menores a x mientras que h es la lista que contiene todos los elementos de xs mayores a x.

$$\begin{array}{lcl} l & = & \mathsf{filter}\ (< x)\ xs \\ h & = & \mathsf{filter}\ (> x)\ xs \end{array}$$

$$\begin{array}{lcl} \varphi\ (\mathsf{inl}\ \star) & = & [\,] \\ \varphi\ (\mathsf{inr}\ (x, l, h) & = & l +\!\!+\ x : h \end{array}$$

Al igual que en el ejemplo anterior se puede concluir la necesidad de los hilomorfismos para definir esta función que sigue el patrón de recursión por curso de valores.

4.6. Distancia de Leveshtein

La distancia de Leveshtein mide la diferencia entre dos cadenas de texto. Representa el mínimo número de alteraciones necesarias para igualar ambas cadenas, las alteraciones permitidas son: insertar un caracter nuevo, eliminar un caracter de la cadena y sustituir un caracter por otro.

Las cadenas tiene una distancia de Leveshtein de 0 respecto a si mismas. Y por ejemplo las cadenas `"pavos"` y `"patos"` tienen una distancia de Leveshtein de 1 ya que se pueden igualar al sustituir el caracter v por t.

La función `levr` calcula la distancia de Leveshtein entre dos cadenas y se define como sigue:

```
levr :: (Eq a) => ([a],[a]) -> Int
levr ([],bs)                = length bs
levr (as,[])                = length as
levr (l1@(a:as),l2@(b:bs)) = minimum
                              [levr (as,l2) + 1,
                              levr (l1,bs) + 1,
                              levr (as,bs) + if a==b then 0 else 1]
```

4.6.1. Dentro del sistema

En esta sección se define la distancia de Leveshtein como un hilomorfismo, para lo que se propone el funtor siguiente como estructura intermedia:

$$F(X) = \mathsf{List}_C + C^2 \times X^3$$

El List_C corresponde a los casos base de la función en donde una de las listas es vacía y el resultado depende completamente de la otra lista, la cuál se almacena en la estructura intermedia. Mientras que $C^2 \times X^3$ representa al caso recursivo en el que son necesarias las cabezas de ambas listas (C^2) y los resultados de tres llamadas diferentes sobre la función (X^3).

De esta forma se define `lev` como:

$$\texttt{lev} = \texttt{hylo}\ \varphi\ \psi$$

En donde las funciones de paso $\varphi : F(\mathsf{Nat}) \to \mathsf{Nat}$ y $\psi : \mathsf{List}_C \times \mathsf{List}_C \to F(\mathsf{List}_C^2)$ se definen de la siguiente forma:

$$\begin{array}{lcl}
\psi\ \langle as, [\,]\rangle & = & \mathsf{inl}\ as \\
\psi\ \langle [\,], bs\rangle & = & \mathsf{inl}\ bs \\
\psi\ \langle (a : as), (b : bs)\rangle & = & \mathsf{inr}(\langle a, b\rangle, \langle as, (b : bs)\rangle, \langle (a : as), bs\rangle, \langle as, bs\rangle)
\end{array}$$

$$\begin{array}{lcl}
\varphi\ (\mathsf{inl}\ xs) & = & \#xs \\
\varphi\ (\mathsf{inr}\ (\langle a, b\rangle, x, y, z) & = & \mathsf{min}(x + 1, y + 1, z + (\mathsf{if}\ a = b\ \mathsf{then}\ 0\ \mathsf{else}\ 1))
\end{array}$$

Todos los casos de estudio presentados en este capítulo sirven como ejemplos de uso del esquema recursivo que se presenta en este trabajo, la recursión por curso de valores, en todos los ejemplos se exhiben casos importantes del uso de este esquema. De la misma forma este capítulo sirve para ejemplificar las principales diferencias entre los dos operadores propuestos (histo e hylo) ya que en algunos casos los histomorfismos no son suficientemente flexibles para abstraer el esquema recursivo presente en

la definición de la función. También se puede encontrar como parte del apendice las implementaciones de las funciones vistas en este capítulo en el lenguaje de programación Haskell en donde se hace uso de los combinadores estudiados en este trabajo.

Conclusiones y Trabajo Futuro

En este trabajo se discutieron e implementaron dos operadores para la abstracción del esquema de iteración por curso de valores. Se definió un sistema polimórfico de tipos monótonos (co)inductivos. El enfoque utilizado se fundamenta en la Teoría de Categorías para modelar el razonamiento de combinadores categóricos como operadores recursivos del sistema propuesto. Los combinadores que se implementaron fueron los *histomorfismos* mediante el operador histo y el *hilomorfismo* con el operador hylo con los que se puede abstraer el esquema recursivo de iteración por curso de valores.

De igual forma se estudio la seguridad del sistema propuesto en este trabajo, por lo que el sistema puede implementarse como un prototipo de un lenguaje de programación real,de esta forma se pueden llevar los resultados obtenidos en este trabajo a la práctica y así lograr operadores recursivos mas expresivos que garanticen la correctez de un programa.

Por último se presentaron una serie de casos prácticos en los que la recursión por curso de valores juega un papel fundamental para la solución de estas problemáticas. Y se mostró como todas estas soluciones caben dentro del sistema Cvlt, propuesto en este trabajo, mediante el uso de los operadores de iteración por curso de valores. Estos casos de estudio sirvieron también para ejemplificar las diferencias entre los *histomofirmos* e *hilomorfismos* así como las restricciones y ventajas del uso de cada uno en ciertos casos.

Conclusiones

Como parte de las conclusiones de este trabajo, podemos mencionar las siguientes.

Los operadores que abstraen esquemas recursivos son de suma importancia en la programación funcional, por lo que su definición e implementación puede ayudar a mejorar el razonamiento sobre los programas que escribimos y asegurar ciertas propiedades sobre ellos. Los esquemas recursivos son ampliamente utilizados en el desarrollo de programas funcionales, debido a esto contar con operadores que los abstraigan en lenguajes funcionales simplifica no sólo el razonamiento sino también la implementación de estos programas.

La flexibilidad de los operadores que se definidos en este trabajo es fundamental para su uso en la resolución de problemas, como se vio con el uso de *histomorfismos* en donde el esquema que modela es muy restrictivo y no permite que algunas funciones sean definidas a partir de ellos,y en contra parte los *hilomorfismos* si permitían las implementaciones con su uso al gozar de una mayor flexibilidad proporcionada por la estructura intermedia que se modela a necesidad.

La mayoría de las funciones en programación funcional siguen fielmente algunos esquemas recursivos evidentes, al lograr abstraerlos e implementarlos en sistemas como el que se presentó en este trabajo se puede definir un estilo de programación basado en combinadores, siempre que los operadores sean suficientemente expresivos.

Trabajo futuro

La vertiente central de este trabajo está dirigida únicamente a los esquemas iterativos por curso de valores, sin embargo hay muchos otros esquemas que vale la pena explorar para hacer mas rico y expresivo el sistema propuesto.

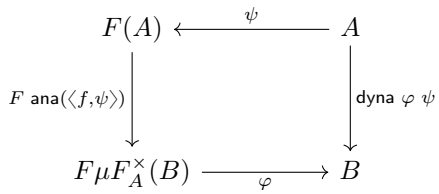
Dinamorfismos

Los *hilomorfismos* presentan cierta flexibilidad ante el esquema recursivo que modelan, sin embargo son ineficientes pues requieren de la construcción completa de la estructura intermedia para su evaluación. Por lo que para su implementación en un lenguaje real es necesaria una optimización. Mediante el proceso de deforestación, que ya está presente en distintos compiladores para lenguajes de programación, se puede eliminar la presencia de la estructura intermedia que necesitan los *hilomorfismos*.

Como alternativa, también se han propuesto nuevos combinadores que modelen la recursión por curso de valores de la misma forma en la que lo hacen los *hilomorfismos* pero de forma eficiente al mejorar su complejidad tanto espacial como temporal. El combinador mas popular con esta filosofía es el *dinamorfismos*, que recibe este nombre por su uso en problemas de programación dinámica [17].

Los *dinamorfismos* modelan un esquema capaz de capturar todos los algoritmos recursivos de programación dinámica.

Definición[Dinamorfismos] Sean $\psi : A \to F\ A$ una F-coalgebra y $\varphi : F\mu F_A^{\times}(B) \to B$ sea una cv-algebra. Un *dinamorfismo* denotado como $\mathsf{dyna}\ \varphi\ \psi$ es la única flecha $f : A \to B$ que hace que el siguiente diagrama conmute.



Los *dinamorfismos* son estructuralmente muy similares a los *hilomorfismos* la diferencia principal es que en lugar de usar un *catamorfismo* utilizan un *histomorfismo*, por lo que su semántica operacional se puede definir con la regla de reducción siguiente:

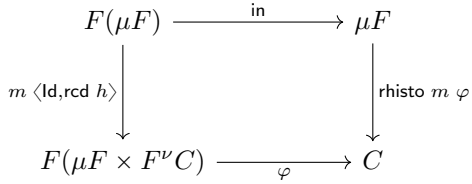
$$\mathsf{dyna}\ \varphi\ \psi \to \mathsf{histo}\varphi \circ \mathsf{ana}\psi$$

Como trabajo futuro se considera agregar este operador al sistema y así aprovechar sus propiedades en la definición de nuevos casos de estudio del área de programación dinámica.

(Co)Recursión

Como se menciono al principio de esta sección, en este trabajo se presentaron principalmente combinadores que modelan la iteración por curso de valores, como trabajo futuro se considera también modelar la recursión por curso de valores mediante un operador en el sistema. La cuál se puede modelar como un *histomorfismo recursivo*.

Definición[Histomorfismos recursivos] Sea $\langle \mu F, \mathsf{in} \rangle$ el álgebra inicial del funtor F, dada una cv-álgebra $\langle C, \varphi \rangle$, el histomorfismo recursivo $\mathsf{rhisto}\ \varphi : \mu F \to C$ es el único morfismo que hace que siguiente diagrama conmute



Con la definición anterior se puede agregar un operador rhisto con semántica operacional dada por la siguiente regla de recucción:

$$\mathsf{rhisto}\ m\ \varphi\ (\mathsf{in}\ t) \to \varphi\ (m\ \langle \mathsf{Id}, \mathsf{rcd\ h} \rangle\ \mathsf{t})$$

Adicionalmente también se puede agregar un operador que modele la correcursión por curso de valores dualizando la definición de los *histomorfismos recursivos*.

Con esto concluye la parte de conclusiones y trabajo futuro en donde se presentaron los puntos primordiales del trabajo así como las posibles extensiones al sistema que se pueden definir a partir de otros combinadores categóricos.

Apéndices

Implementaciones en Haskell

En este punto es claro que existen muchas funciones que comparten el mismo esquema recursivo, es decir, hay familias de funciones que tienen en común la estructura recursiva con la que son definidas. Esto es relevante en la teoría para el estudio de estas funciones y lograr generalizar los resultados a todas aquellas que cumplan con los patrones, pero también es interesante en la práctica en donde estos esquemas pueden ser modelados mediante operadores nativos del lenguaje con el fin de abstraer la codificación de las funciones en cuestión, como se ha visto en el desarrollo de este trabajo.

Estos esquemas recursivos son muy presentes en la definición de las funciones en programación funcional, por esta razón los lenguajes de programación han adoptado los operadores recursivos como parte de la sintaxis del lenguaje. El caso mas evidente de esto son las funciones `foldr` y `unfoldr`, los cuales modelan catamorfismos y anamorfismos respectivamente, y que están presentes en la mayoría de lenguajes de programación de la actualidad, no solo lenguajes funcionales.

Adicionalmente, lenguajes como Haskell han agregado más operadores que modelan esquemas recursivos menos comunes como lo son los hilomorfismos estudiados en este trabajo, si bien este operador no es nativo del lenguaje se puede usar con un paquete externo llamado `recursion-schemes` que no sólo cuenta con hilomorfismos sino muchos otros operadores basados en combinadores categóricos para modelar esquemas recursivos.

En este apéndice se presentan los casos de estudio vistos en el último capítulo, escritos directamente en Haskell con el uso del paquete `recursion-schemes` representándolos como hilomorfismos.

Ejemplo .1 (Triángulo de Pascal)**.** .

```
data Tree r =
      Leaf
```

```
    | Node r r
    deriving Functor

psi :: (Int,Int) -> Tree (Int,Int)
psi (1,_) = Leaf
psi (_,1) = Leaf
psi (r,c) = Node (r,c-1) (r-1,c)

phi :: Tree Int -> Int
phi Leaf       = 1
phi (Node p q) = p + q

pas :: (Int,Int) -> Int
pas = hylo phi psi
```

En el ejemplos anterior se puede ver como el esquema recursivo del operador `hylo` requiere de la representación de la estructura intermedia generada, la cual se modela con el tipo de dato algebraico `Tree`, que debe ser una instancia de la clase `Functor`, a partir de esta instancia se reconoce el testigo de monotonicidad como la función `fmap` que hace al tipo un funtor, también son necesarias las funciones intermedias del anamorfismo (`psi`) y el catamorfismo (`phi`) con las que se construye y dobla la estructura intermedia respectivamente.

Este mismo esquema se seguirá en los siguiente ejemplos. Usando los mismos nombres para la estructura intermedia, así como para las funciones de paso del anamorfismo y catamorfismo.

Ejemplo .2 (Particiones binarias)**.** .

```
data Tree r =
      Leaf
    | Branch r
    | Node r r
    deriving Functor

psi :: Int -> Tree Int
psi 0 = Leaf
psi n
 | odd n     = Branch (n-1)
 | otherwise = Node (n-1) (n `div` 2)

phi :: Tree Int -> Int
phi Leaf       = 1
```

```
phi (Branch n) = n
phi (Node n m) = n + m

bp :: Int -> Int
bp = hylo phi psi
```

Para el caso de estudio de la subsecuencia mas larga sólo se define la solución lineal presentada en el capítulo 4.

Ejemplo .3 (La subsecuencia mas larga)**.** .

```
data Tree a r =
      Empty
    | Node a a r r r
    deriving Functor

psi :: ([a], [a]) -> Tree a ([a], [a])
psi ([],_)                = Empty
psi (_,[])                = Empty
psi (l1@(a:as),l2@(b:bs)) = Node a b (as,l2) (l1,bs) (as,bs)

phi :: (Eq a) => Tree a [a] -> [a]
phi Empty              = []
phi (Node a b x y z)
 | a == b              = a : z
 | length x > length y = x
 | otherwise           = y

lcsh ::(Eq a) => ([a],[a]) -> [a]
lcsh = hylo phi psi
```

Para el algoritmo *MergeSort* se presenta la definición de la función que ordena los elementos de una lista así como la función `merge`, que como se vio en el trabajo también entra en el esquema de recursión por curso de valores, por lo que puede definirse mediante un hilomorfismo.

Ejemplo .4 (MergeSort)**.** .

```
data Tree a r =
    Empty
  | Leaf a
  | Node r r
  deriving (Show,Eq,Functor)
```

```haskell
psi :: [a] -> Tree a [a]
psi []  = Empty
psi [x] = Leaf x
psi xs  = Node l r
 where (l,r) = splitAt (length xs `div` 2) xs

phi ::(Ord a) => Tree a [a] -> [a]
phi Empty      = []
phi (Leaf x)   = [x]
phi (Node l r) = m l r

msh :: (Ord a) => [a] -> [a]
msh = hylo phi psi

data TreeM a r =
     LeafM [a]
    | NodeM a a r r
    deriving(Functor)

psiM :: ([a],[a]) -> TreeM a ([a],[a])
psiM ([],ys)               = LeafM ys
psiM (xs,[])               = LeafM xs
psiM (l1@(x:xs),l2@(y:ys)) = NodeM x y (xs,l2) (l1,ys)

phiM :: (Ord a) => TreeM a [a] -> [a]
phiM (LeafM xs) = xs
phiM (NodeM a b x y)
 | a < b     = a:x
 | otherwise = b:y

m :: (Ord a) => ([a],[a]) -> [a]
m = hylo phiM psiM
```

El siguiente ejemplo es el algoritmo de ordenamiento *Quicksort*.

Ejemplo .5 (QuickSort)**.** .

```haskell
data Tree a r =
    Void
  | Branch a r r
  deriving (Show,Eq,Functor)
```

```haskell
psi :: (Ord a) => [a] -> Tree a [a]
psi []     = Void
psi (x:xs) = Branch x l h
 where h = [e | e <- xs, e > x]
       l = [e | e <- xs, e < x]

phi :: Tree a [a] -> [a]
phi Void         = []
phi (Branch x l h) = l ++ x:h

qsh :: (Ord a) => [a] -> [a]
qsh = hylo phi psi
```

Como último ejemplo se presenta la distancia de Leveshtein.

Ejemplo .6 (Distancia de Leveshtein)**.** .

```haskell
data Tree a r =
      Leaf [a]
    | Node a a r r r
    deriving Functor

psi :: ([a],[a]) -> Tree a ([a],[a])
psi ([],bs)               = Leaf bs
psi (as,[])               = Leaf as
psi (l1@(a:as),l2@(b:bs)) = Node a b (as,l2) (l1,bs) (as,bs)

phi :: (Eq a) => Tree a Int -> Int
phi (Leaf xs)        = length xs
phi (Node a b x y z) = minimum [ x+1, y+1, z + if a==b then 0 else 1]

levh :: (Eq a) => ([a],[a]) -> Int
levh = hylo phi psi
```

Con estos programas se puede ejemplificar el uso de la biblioteca `recursion-schemes` para la definición de funciones que siguen los esquemas recursidos definidos por los combinadores categóricos.

Bibliografía

[1] Lourdes Del Carmen González Huesca. *Coinducción: de la Teoría de Categorías a la Programación Funcional*. Universidad Nacional Autónoma de México, 2007. 1.1, 1.1, 2.1

[2] R. Hinze. Memo functions, polytipically! 01 2000. 1.2.1

[3] Jan-Willem Buurlage. *Categories and Haskell An introduction to the mathematics behind modern functional programming*. Notes on Category Theory and Haskell, 2018. 1.3

[4] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Tesis doctoral, 1972. 1.4

[5] John Reynolds. *Towards a Theory of Type Structure*. 1974. 1.4

[6] Brian T. Howard. *Fixed Points and Extensionality in Typed Functional Programming Languages*. PhD thesis, Stanford, CA, USA, 1992. UMI Order No. GAX93-02219. 1.4

[7] J.L. Krivine. *Lambda-calculus, Types and Models*. Computers and their applications. Ellis Horwood, 1993. 1.4, 3.3.1

[8] Andreas Abel. A polymorphic lambda-calculus with sized higher-order types. 01 2006. 2

[9] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1):3–66, 2005. Foundations of Software Science and Computation Structures. 2

[10] R.L. Crole. *Categories for Types*. Cambridge Mathematical Textbooks, Cambridge Uni- versity Press., 1993. 2

[11] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In David H. Pitt, Axel Poigné, and David E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–157, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. 2

[12] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1, 06 2005. 2

[13] Favio Miranda-Perea and Lourdes del Carmen González Huesca. Mendler-style iso-(co)inductive predicates: a strongly normalizing approach. *Electronic Proceedings in Theoretical Computer Science*, 81, 03 2012. 2

[14] Benjamin C. Pierce. *Basic category theory for computer scientists*. MIT Press, 1991. 2.1

[15] Ralph Matthes. Monotone (co)inductive types and positive fixed-point types. *RAIRO Theor. Informatics Appl.*, 33:309–328, 1999. 2.1

[16] Ralph Matthes. Monotone fixed-point types and strong normalization. In *Proceedings of the 12th International Workshop on Computer Science Logic*, page 298–312, Berlin, Heidelberg, 1998. Springer-Verlag. 2.1

[17] Jevgeni Kabanov and Varmo Vene. Recursion schemes for dynamic programming. In Tarmo Uustalu, editor, *Mathematics of Program Construction*, pages 235–252, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 3, 4.6.1

[18] Varmo Kabanov, Jevgeni y Vene. Recursion schemes for dynamic programming. *J. Comb. Theory, Ser. A*, 98:33–45, 2002. 3

[19] Favio Ezequiel Miranda-Perea. Some remarks on type systems for course-of-value recursion. *Electronic Notes in Theoretical Computer Science*, 247:103–121, 2009. Proceedings of the Third Workshop on Logical and Semantic Frameworks with Applications (LSFA 2008). 3

[20] Favio Miranda-Perea. Two extensions of system f with (co)iteration and primitive (co)recursion principles. *ITA*, 43:703–766, 10 2009. 3, 3.3.1, 3.3.2

[21] D. Turner. Total functional programming. *J. UCS*, 10:751–768, 01 2004. 3

[22] Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10, 10 1999. 3

[23] Ralf Hinze and Nicolas Wu. Histo- and dynamorphisms revisited. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, page 1–12, New York, NY, USA, 2013. Association for Computing Machinery. 3

[24] Manuel Cunha. Recursion patterns as hylomorphisms. II 2003. 3

[25] B. Pascal. *Traité du triangle arithmetique, avec quelques autres petits traitez sur la mesme matiere. Par monsieur Pascal*. Guil. Desprez, 1665. 4.1

[26] Øystein J. Rødseth and James A. Sellers. Binary partitions revisited. *J. Comb. Theory, Ser. A*, 98:33–45, 2002. 4.2

[27] Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010. 4.3