Funciones de Orden Superior en Programación Funcional. Una Perspectiva Categórica

José Enrique Freire Brañas

Memoria presentada para optar al
Grado de Doctor en Informática

Director: José Luis Freire Nistal

AGRADECIMIENTOS

Huarte de San Juan, en su Examen de ingenios para las ciencias, afirma que *La autoconformidad con lo ya trabajado por nuestro propio entendimiento sin hacerle trabajar más también es un modo sobrevenido de infecundidad.* Quiero expresar mi gratitud a José Luis Freire por haberme mostrado, a lo largo de estos años, la veracidad de esta afirmación y haberme enseñado, en cierto modo, a paliar sus nocivos efectos. Me indicó, además, cómo aplicar la aserción del epigrama de Balmes: *El hombre de ciencia ha de ser capaz de admirarse de todo porque en todas las cosas hay una verdad más o menos escondida.*

Deseo mostrar mi gratitud a mis padres que me acompañaron, incondicionalmente, en todas mis aventuras.

Por último, mi enorme reconocimiento a Consuelo, Alejandro y Borja por soportar estoicamente mis alternancias de humor.

Tribunal :

PRESIDENTE:

Dr. D. Antonio Blanco Ferro

VOCALES:

Dr. D. Eladio Domínguez Murillo

Dr. D. Felipe Gago Couso

Dr. D. Laureano Lambán Pardo

SECRETARIA:

Dra. D^{a}. Felicidad Aguado Martín

José Luis Freire Nistal, como director del presente trabajo *Funciones de orden superior en programación funcional. Una perspectiva categórica* realizado por D. Enrique Freire Brañas, autorizo a éste a presentar dicho trabajo para optar al grado de Doctor en Informática.

La Coruña 7 de Julio de 2003

Fdo. José Luis Freire Nistal.

Índice

1 Introducción

Una tesis doctoral debe presentar y resolver un problema. El que suscita esta investigación es el cálculo y análisis computacional de las estructuras subyacentes a las álgebras de Eilenberg-Moore y de Kleisli ligadas a las mónadas que han ido surgiendo en computación los últimos años para simular ciertos aspectos imperativos de los que adolecen los lenguajes funcionales [Moggi 1989], [Wadler 1993].

Al profundizar en el tema van surgiendo *efectos colaterales*. Por una parte las mónadas describen tipos: el tipo de las listas, las continuaciones, las computaciones no deterministas, etc. Esto induce a pensar cómo describir los tipos de la forma más general posible. Como no todos los tipos proceden de funtores monádicos hemos de encajar su definición en una estructura más amplia. En segundo lugar, para poder operar con esos tipos deben generarse funciones de orden superior. Esto obliga a indagar cómo caracterizarlas y, en caso de ser posible, optimizarlas. Por último, para poder expresar determinadas construcciones es necesario disponer de lenguajes adaptados a ellas. Por ejemplo, las implementaciones estándar de Haskell y CAML no nos permiten definir tipos dependientes de parámetros [1] . Otro ejemplo: Haskell da grandes facilidades para trabajar con tipos existenciales [2] mientras que las versiones convencionales de CAML no facilitan su uso. Por ello, el planteamiento inicial se enriquece con las siguientes consideraciones:

1. cómo describir los tipos de la forma más *natural* posible y susceptible de generar, de forma casi automática, funciones sobre ellos. Grosso modo, cómo declarar tipos *abstractos*;

2. cómo construir las funciones mencionadas en el apartado anterior de forma que resulten eficientes;

3. cómo encajar esas funciones en problemas reales de programación;

4. qué estructuras subyacen a estas funciones que permitan su optimización;

5. a qué extensiones dan lugar todas las construcciones que se van determinando para resolver esos problemas.

Comprobamos que para analizar todas estas ideas la mejor herramienta de la que se podía disponer es la teoría de categorías [3]. La interrelación entre ésta y la programación ha sido y es enormemente provechosa. Mientras la primera aporta la fundamentación teórica para interpretar los lenguajes de computación, estos, al requerir estructuras ad-hoc para una semántica adecuada, exigen la generación de nuevas estructuras categóricas y su constante

[1] Salvo *de CAML*, la implementación experimental de CAML con tipado dependiente [Xi1999]

[2] A partir de las signaturas de rango 2

[3] De donde surgió el problema que originalmente nos habíamos planteado.

evolución para adaptarse a los avances obtenidos [Power 1998]. De hecho, *la programación con un conjunto fijo de funciones recursivas derivado de las definiciones del tipo conduce a una disciplina de programación estructurada que puede ser explotada para facilitar el cálculo con programas (sobre todo funcionales)* [Pardo 1997].

Contamos con una amplia colección de estudios generales de esta relación [Manes y Arbib 1986], [Asperti y Longo 1991], [Gunter 1992], [Crole 1993], [Pitts 1995], [Barr y Wells 1999].

El propósito de esta memoria es, pues, estudiar, desde una perspectiva categórica, las funciones de orden superior que surgen en programación funcional provenientes de la declaración de tipos (bien inductivos bien coinductivos) y, como caso particular, analizar las álgebras ligadas a las estructuras monádicas usadas en programación funcional con el fin de emular las características imperativas de las que carecen.

Para realizar este estudio decidimos abordar las cuestiones desde cuatro puntos de vista:

1. en primer lugar, analizando cómo los conceptos categóricos permiten generar tipos más próximos a las necesidades reales de la computación. Llegamos a la conclusión de que la mejor aproximación surgía de la noción de tipo categórico de datos introducida por Hagino [Hagino 1987]. En esta memoria, definimos lo que hemos denominado tipos $\mu\nu$-regulares con características de objeto inicial y final. Ello nos permite construir y optimizar código utilizando las propiedades universales;

2. construyendo, como aplicación de estas facilidades, funciones de orden superior cuya especialización o simplificación vengan garantizadas por la teoría implícita en sus definiciones. Utilizando ejemplos apropiados, pretendemos ver cómo desarrollar esas optimizaciones. Este aspecto, ampliamente utilizado por la comunidad Squiggol [Meijer y otros 1991], permite interesantes avances en la ingeniería del software;

3. estando interesados en el estudio del Cálculo de Construcciones Inductivas y, en particular, en el asistente de pruebas constructivas Coq [Dowek y otros 1993], analizamos cómo adecuar las definiciones de los tipos coinductivos para poder sacar mayor provecho de ellos tanto en su caracterización como objetos finales como en la posibilidad de simplificación de las funciones que a ellos llegan. Como una aplicación de todos estos conceptos, implementamos en este trabajo las funciones primitivo recursivas analizando, como caso colateral, por qué no lo es la función de Ackermann. Ésta es un catamorfismo; limitando el alcance de éstos ya no podemos expresarla. Dentro de este mismo contexto probamos el teorema de fusión cuando trabajamos con funciones sobre tipos dependientes;

4. indagando, por último, cuáles son las estructuras subyacentes a las mónadas utilizadas en los lenguajes funcionales para emular la programación imperativa.

1.1 *Estado de la cuestión*

En los últimos treinta años la contribución de la teoría de categorías a la computación ha ido en constante crecimiento. Si, inicialmente, la principal aportación era la interpretación semántica de los programas, aquélla ha ido extendiéndose hasta abarcar un espectro

de actuación más amplio. No vamos a mencionar todos estos aspectos sino únicamente aquellos que han ido llamando nuestra atención en la elaboración del presente trabajo.

Ya que nuestro interés se ha centrado en la teoría algebraica de tipos, en la computación monádica y en la optimización de funciones mencionaremos los progresos en estos ámbitos.

Iniciamos esta presentación indicando la relación establecida entre el lenguaje de categorías y el del λ-cálculo. Al ser el λ-cálculo tipado una teoría de las funciones computables y sabiendo que puede ser interpretado en una categoría cartesiana cerrada, esta imbricación conduce a realizaciones prácticas. En efecto, considerando los tipos del λ-cálculo tipado los siguientes (en notación BNF)

$$T ::= 1 | nat | T \times T | T \to T$$

y contando con los términos habituales de este cálculo, es posible construir una categoría cartesiana cerrada de la cual aquél sea el lenguaje interno ([Lambeck y Scott 1989], pág. 78). Es decir, la categoría $\lambda - \mathbf{Calc}$ con objetos los λ-cálculos tipados y morfismos las traslaciones entre ellos es equivalente a la categoría $\mathbf{Cart_N}$ de categorías cartesianas cerradas con objetos de números naturales débiles y funtores cartesianos cerrados preservando estos objetos de números naturales débiles [Lambeck y Scott 1989].

El polimorfismo abrió nuevas vías de investigación para intentar encajarlo dentro de una estructura categórica. Si hasta ese momento era posible trabajar en cualquier categoría cartesiana cerrada, como por ejemplo **Set**, la categoría de conjuntos, con la introducción de la cuantificación universal sobre tipos ($\forall \alpha . T(\alpha)$) la situación se transformó radicalmente. En efecto, para interpretar ese tipo en la categoría C debería utilizarse un objeto de la forma

$$\prod_{A \in \mathrm{C}} expr(A).$$

Ahora bien, es un teorema clásico de la teoría de categorías que si una categoría pequeña C tiene producto de familias del mismo tamaño que ella misma hay un único morfismo entre cada dos objetos, es decir, tal categoría es un preorden (para un esquema de la demostración ver [Gray 1991]). Hubo, entonces, que habilitar estructuras adecuadas para intepretar ese polimorfismo. Se necesitaba una categoría donde pudiese efectuarse tal producto. Pero esto sólo es posible usando la noción de productos internos en categorías muy especiales; un ejemplo de tal es la categoría **PER** de relaciones de equivalencia parciales sobre los números naturales [Gray 1991], [Gunter 1992].

Pero no terminan aquí las necesidades: la cuantificación existencial en los tipos y el polimorfismo sobre constructores presentan nuevos retos a la investigación. Como respuesta a estas cuestiones (productos y coproductos pequeños) surge la noción de categoría indexada. La interpretación del tipo existencial queda determinada por medio de los coproductos [Mitchell y Plotkin 1988]. Efectivamente, el tipo $\exists X . F(X)$ indica la existencia de un valor para X verificando la estructura F; podemos escribir

$$\exists X . F(X) = \coprod_{X} F(X).$$

Es, además, posible expresar la cuantificación existencial por medio de la universal, permitiendo, entonces, adoptar los modelos del λ- cálculo polimórfico para

interpretar los tipos existenciales que, por extensión, conforman los tipos abstractos [Mitchell y Plotkin 1988]. Así, la categoría donde interpretar estos tipos debe garantizar que el coproducto de una familia del tamaño de la propia categoría sea un elemento de ésta.

Cuando se trabaja con teorías de tipos incorporando tipos dependientes de términos (como, por ejemplo, el sistema de tipos de Martin-Löf o el lambda cálculo polimórfico de segundo orden de Girard-Reynolds) se precisan estructuras categóricas más específicas para establecer su semántica. Una aproximación que permite el desarrollo de esta semántica sin excesivas complicaciones es la de las categorías con atributo.

Categorías para interpretar correctamente el tipado no regular, el polimorfismo sobre constructores, los lenguajes lógico-funcionales, la distributividad o la concurrencia son objeto, en estos momentos, de profundo estudio [Hasegawa 1994], [Bellé 1996], [Bird y Paterson 1999 (2)], [Duggan 1999].

Un aspecto importante en esta relación es la posibilidad de usar las descripciones categóricas denotacionales en aspectos más operacionales de los lenguajes de programación. Ejemplos de estas *máquinas de entornos para los lenguajes funcionales* son las Categorical Abstract Machines de Cousineau et al., la máquina de Krivine o la de Asperti [Asperti 1992]. La construcción por Hagino [Hagino 1987] del lenguaje de programación categórico con la introducción de las declaraciones de tipos por medio de adjunciones ha sido un hito en este aspecto. Una implementación basada en estas ideas es el lenguaje Charity [Cockett y Fukushima 1992].

Una ventaja que aportan las construcciones categóricas es la posibilidad de aplicar los teoremas de la teoría subyacente a la optimización de funciones, tema activamente estudiado por la comunidad Squiggol [Meijer y otros 1991], [Bird y de Moor 1993], [Gibbons 1994]. Las contrucciones universales habilitan la definición de funciones unívocamente determinadas por su signatura, con la capacidad de reducción que ello implica [Meijer y otros 1991]. Además, esto exige la construcción de mecanismos de generalización, o sea, estructuras globales que permitan, por medio de instanciaciones, abarcar casos particulares. Ejemplos de esta clase de programación genérica los tenemos en las estructuras de clases provistas por Haskell [Jones 1995], la programación politipada [Jeuring y Jansson 1996] o los estudios acerca del ML funtorial [Jay y otros 1998].

Un aspecto relevante en los últimos años es el surgimiento de la computación monádica permitiendo simular la computación imperativa en lenguajes funcionales puros [Wadler 1993]. Con las estructuras monádicas podemos acercanos a las tareas de entrada-salida [Gordon 1994] o de cambios de estado [Wadler 1987] sin perder la transparencia referencial característica de la programación funcional. Una nueva vía de actuación [Fokkinga 1994], [Hu y Iwasaki 1995], [Meijer y Jeuring 1995] permite la estructuración de programas no ya dirigida a la sintaxis sino a la semántica.

Recientemente se ha empezado a estudiar la aplicación de la estructura dual a las mónadas, las comónadas, de cara a la utilización de parámetros implícitos [Kieburtz 1999], [Lewis y otros 2000].

1.2 *Objetivos del trabajo*

Al analizar los tipos recursivos comprobamos que había sustanciales diferencias entre ellos según fuesen tipos iniciales (menores puntos fijos o inductivos) o terminales (mayores puntos fijos o coinductivos). Observamos que, mientras algunos de estos tipos estaban absolutamente encuadrados dentro de uno de esos ámbitos, otros podían compartir aspectos de ambas estructuras. Los primeros casos que llamaron nuestra atención sobre esta segunda posibilidad fueron el de las listas y el de los árboles de aridad variable. Posteriormente vimos que otros muchos se acomodaban a estas especificaciones (de hecho, en condiciones adecuadas que referiremos, todos aquellos basados en funtores polinomiales). El porqué de esta dualidad se observa en la definición de las listas y en sus homólogas infinitas (usaremos la sintaxis de Coq en este ejemplo):

```
Inductive List [A:Set]:Set :=
Nil:(List A)
Cons:A->(List A)->(List A).

CoInductive ListI [A:Set]:Set :=
ConsI:A->(ListI A)->(ListI A).
```

Aparte de la caracterización como inductivas de las listas convencionales y como coinductivas las infinitas (proveniente del conocimiento de que disponemos acerca de que las listas corresponden a un tipo inicial y las listas infinitas a uno terminal), los constructores `Cons` y `ConsI` concuerdan absolutamente. La única diferencia entre ambos tipos estriba en que un elemento de tipo `List` puede tener final mientras que uno de tipo `ListI` indefectiblemente nunca termina. Por consiguiente, a efectos prácticos, pensamos que las listas convencionales, tal como las hemos definido, tienen ese doble carácter de inicial y final (naturalmente, hemos constatado la correción de esta idea en la bibliografía [Hasegawa 1996], [Erwig 1998], [Gibbons 2000]).

Estas estructuras de doble caracterización inicial-final permiten simplificaciones muy provechosas, facilitando la *construcción de un programa desde sus especificaciones*, en el sentido de la *programación transformacional* descrita por Darlington (citado por Gibbons en [Gibbons 1994]) como: ... *un programador no debe procurar producir directamente un programa correcto, comprensible y eficiente sino concentrarse en construir un programa tan claro y comprensible como sea posible ignorando cualquier cuestión relativa a la eficiencia* ...

Por lo tanto, indagamos en las posibilidades expresivas de estos tipos analizando las simplificaciones a que daban lugar. Vimos primeramente que la estructura que se adaptaba mejor a esta idea era la de diálgebra definida por Hagino [Hagino 1987]. La ventaja de conjugar tipos donde las funciones de entrada y las funciones de salida pueden determinarse de forma única es enorme. Éstas, compuestas, dan lugar a unas funciones no muy estudiadas hasta la fecha, los hilomorfismos [Meijer y otros 1991], [Hu, Iwasaki, Takeichi 1995], [Jürgensen 2000]. Es sabido que tanto la composición de catamorfismos como la de hilomorfismos no es, en general, ni un catamorfismo ni un hilomorfismo, respectivamente. Nos preguntamos bajo qué condiciones sí se verificaba este

resultado; comprobamos que, cuando los morfismos que determinan las estructuras de álgebras y coálgebras del tipo intermedio son isomorfos, la composición de hilomorfismos es un hilomorfismo. Introducimos, además, una condición suficiente para que la composición de catamorfismos también lo sea.

Los tipos anidados surgen naturalmente en computación pero, a causa de los problemas derivados de la incapacidad de definir funciones sobre ellos debido a las restricciones del sistema de tipos de Hindley-Milner, han sido poco utilizados. En las últimas versiones de algunos compiladores (Haskell, por ejemplo) usando signaturas de tipo de rango 2 ya es posible evitar esas limitaciones y utilizar estos tipos para trabajar en un contexto más acorde a las necesidades (en [Bird y Paterson 1999] se define la notación de de Bruijn basándose en su representación como un tipo de esta clase). El problema que todavía no está plenamente resuelto es cómo encajarlos dentro de una interpretación categórica. Analizamos algunas posibilidades comprobando que hay una clase de tipos anidados que pueden ser reducidos a tipos regulares pudiendo, en este caso, definir funciones sobre ellos de forma convencional (sin apelar a compiladores especializados). Para aquellos otros no reducibles a regulares encontramos, en Haskell, una solución que nos permite definir funciones generales sobre ellos (por ejemplo, los catamorfismos y anamorfismos correspondientes a sus condiciones de inicialidad y finalidad) con posibilidades reales de instanciación. En [Bird y Meertens 1998] se consideraron estos tipos como álgebras iniciales dentro de la categoría de funtores y transformaciones naturales. Pero, comprobaron (y nosotros hemos constatado) que, desde esta perspectiva, se pierde buena parte de la capacidad de instanciación que se tiene con los tipos convencionales (de hecho, al trasladar estos a aquella categoría, se pierde buena parte de la flexibilidad que los hace tan sumamente útiles). En un siguiente trabajo [Bird y Paterson 1999 (2)] corregían ese problema definiendo los denominados **folds generalizados** implicando un cambio en la técnica de programación para poder abarcar esta enorme generalización. Nuestra solución permite, sin esta ampliación y usando las técnicas válidas para los tipos regulares, utilizar los morfismos sobre y hacia los tipos anidados con sus respectivas instanciaciones. Para ello, se hace uso de las clases de Haskell, las cuales permiten acotar el rango de tipos sobre los cuales podemos definir las funciones. Por ejemplo, si dado el tipo

$$Llist\ A = () + A \times (Llist\ [A]))$$

se quiere definir la función *cataLlist* es necesario establecer que una función $f\ x$ da lugar a una función $f\ [x]$ y que otra función de tipo $((x, f[x]) \rightarrow f\ x$ genera otra de tipo $([x], f\ [[x]]) \rightarrow f\ [x])$. Esto se declara de la siguiente forma:

```
class Valid f where
    fValid1 :: (f x)->f [x]
    fValid2 :: ((x,f[x])->f x)->([x],f [[x]])->f[x]
```

Una extensión de los catamorfismos para actuar sobre procedimientos y no limitarse únicamente a funciones se determina automáticamente en Coq a partir de los argumentos de los constructores. Estos procedimientos, denominados *esquemas de eliminación dependiente* en [Paulin y Werner 1998], son generalizaciones de los catamorfismos convencionales: mientras en estos se actúa sobre la estructura funtorial que determina

el tipo incial, en aquellos esa actuación se extiende a los elementos de cada uno de los argumentos involucrados en el funtor. Nos preguntamos qué tipo de estructura algebraica subyace a esta generalización e intentamos ver por qué no cabe esta opción para los anamorfismos. Analizamos la posibilidad de extender los teoremas de simplificación válidos para los morfismos iniciales convencionales a estas generalizaciones. Además, usando la capacidad de Coq para usar tipos dependientes, definimos los tipos finales no por su resultado sino por su caracterización lo cual permite ampliar su uso. Así, por ejemplo, en lugar de definir el tipo de las *streams* como

```
CoInductive stream [C:Set] : Set :=
Inf : C -> (stream C) -> (stream C).
```

lo declaramos conforme a su condición de final:

```
CoInductive Stream [C:Set]:Set :=
Inf : (A,B:Set)
(B->A) -> (B->B) -> (A->C) -> B -> (Stream C).
```

Como casos singulares de álgebras, nos interesamos en las álgebras de Eilenberg-Moore y de Kleisli asociadas a las mónadas que nos permiten actuar de forma imperativa dentro del paradigma funcional sin perder la transparencia referencial. La importancia de éstas en el ámbito computacional queda reflejada en el capítulo 7.5 del libro *Practical Foundations of Mathematics* de Paul Taylor [Taylor 1999].

Analizamos también cómo podemos ligar las mónadas con las diálgebras y los requisitos necesarios para poder construir los catamorfismos monádicos y sus duales los anamorfismos comonádicos.

Por último, indagamos en las estructuras ligadas a la cuantificación existencial en tipos, estudiando la utilidad de que tal posibilidad provee a los lenguajes de programación que permiten su uso.

1.3 *Organización de la memoria*

Las nociones básicas para hacer esta memoria autocontenida aparecen, condensadas, en el segundo capítulo. Además de los conceptos clásicos relativos a las estructuras categóricas que serán necesarias (analizadas extensamente con ejemplos computacionales que nos aproximen a la orientación deseada) se hará una muy somera introducción a la sintaxis del λ-cálculo tipado polimórfico (desde la perspectiva de los lenguajes funcionales como construcciones basadas en ese paradigma parece procedente tal presentación). Además, nos permitirá disponer libremente de las λ-expresiones, usadas discrecionalmente a lo largo del trabajo. Cuando otras estructuras algebraicas sean usadas [filtros, dominios computacionales o retículos] serán introducidos en el capítulo correspondiente reseñando únicamente los resultados necesarios para su correcto entendimiento.

Estudios completos del λ-cálculo son [Barendregt 1984] y [Krivine 1990]; en cuanto a los conceptos categóricos inherentes al mundo computacional [Asperti y Longo 1991], [Crole 1993] y [Barr y Wells 1999] son claros, completos y profundos; en cuestiones

categóricas [Mac Lanes 1971] es sumamente útil. Para los conceptos relativos a dominios y retículos [Stoy 1977] y [Gunter 1992] son excelentes. En cuanto a las consideraciones de programación funcional remitimos a [Peyton Jones 1987], [Field y Harrison 1988], [Davie 1992], [Cousineau y Mauny 1995].

Los conocimientos necesarios para aproximarse a la sintaxis y a la semántica de Coq pueden obtenerse en [Dowek y otros 1993] siendo [Dowek 1998] una muy interesante introducción a la teoría de tipos. Estudios excelentes sobre la relación semántica entre los diferentes λ-cálculos y las adecuadas construcciones categóricas son [Lambeck y Scott 1989], [Asperti y Longo 1991], [Crole 1993]. Por último, una excelente aproximación a la implementación de las construcciones categóricas en un lenguaje funcional es [Rydeheard y Burstall 1988]. Recientemente, Huet ha emprendido esta implementación en Coq [Huet y Amokrane 2000] .

Los lenguajes que usaremos para expresar los ejemplos y para aclarar conceptos son CAML (indistintamente Objective CAML versión 2.02 para Windows y CAMLlight para DOS en su versión 0.5), Haskell (en su versión Hughs 98 de noviembre de 1999) y la versión 6.2.2 de Coq. Indicaremos cuando sea procedenete qué lenguaje utilizamos. Dependiendo de qué construcción deseemos implementar será más pertinente el uso de uno o de otro.

En el tercer capítulo, usando las nociones dialgebraicas introducidas por Hagino [Hagino 1987], introducimos el concepto de $\mu\nu$-tipo regular, caracterizada por la dualidad inicial-terminal. Tal definición da lugar a la preservación de las propiedades iniciales y/o finales en los morfismos (ley de fusión para catamorfismos y anamorfismos o deforestación), añadiendo la doble componente nuevas posibilidades de optimización. De hecho, las funciones composición de anamorfismos y catamorfismos (hilomorfismos) se aprovechan de ambas leyes para su simplificación (Sección 3.24). Se analiza la categoría $\mathbf{Dial}(\mathbf{FList}, \mathbf{\Pi})$ comprobando su completitud. Además, se analizan diferentes soluciones (algunas parciales y otras globales) para la definición de morfismos de orden superior tanto para los tipos anidados no regularizables como para aquellos regularizables. La solución que se obtiene en el primer caso hace uso (y sólo en este contexto tiene sentido) de las clases (ampliamente utilizadas en computación) y de las signaturas de tipo de rango 2, implementadas en los últimos compiladores de Haskell, permitiendo hacer cuantificación local, imprescindible para la expresión de algunos objetos.

Nos interesamos, además, en la interpretación categórica de algunas funciones que han ido surgiendo a lo largo del trabajo, comprobando como los *pullbacks*, *pushouts*, igualadores y coigualadores permiten generar funciones eficientes. Finalizamos este capítulo analizando una función de semántica similar a la de los catamorfismos (o foldr) denominada en la literatura *foldl*. Nos planteamos cómo expresarla categóricamente y cómo aplicarla a cualquier tipo y no sólo a las listas que es donde habitualmente aparece. La idea de coigualador juega un rol importante en su construcción sobre las listas. Analizamos su posible uso en los otros tipos viendo su adecuación.

Las funciones de orden superior surgidas durante el tercer capítulo y las posibilidades de optimización para ellas derivadas son usadas en el cuarto para analizar e implementar varios problemas interesantes. El lenguaje usado aquí, al no ser requeridas estructuras complejas, es CAML. El primer problema tratado es el del cálculo del número cromático de un grafo (usando un atractivo algoritmo de tie-break en la elección de los nodos); una solución del conocido problema P-S de MacCarthy es el segundo caso tratado; con el cálculo

del spanning-tree de un grafo usando las dos estructuras especialmente detalladas en el capítulo anterior (listas y árboles de aridad variable) termina este capítulo eminentemente algorítmico.

Ya que nuestro interés radica en los tipos de datos y las funciones derivadas de su carácter de inicial o final, el sistema Coq nos permite afrontarlos desde una nueva perspectiva (pues en él los tipos inductivos generan automáticamente las funciones iniciales). Dedicamos, entonces, el quinto capítulo a este análisis. Ya hemos visto en el capítulo 3 cómo declarar los tipos terminales a partir de su caracterización con lo cual el problema que supone el hecho de que Coq no provea automáticamente las funciones terminales sobre ellos queda, en parte, solventado. Incorporamos una demostración computacional del teorema de Marie France Thibaut acerca de la representación de la función de Ackermann en estos sistemas: dada la construcción del tipo de los números naturales como álgebra inicial del funtor $F(A) = () + A$, es obvio que todas las funciones primitivo recursivas pueden expresarse a partir de la función inicial que lo caracteriza (esquema de eliminación dependiente en [Paulin y Werner 1998]). Pero queda la duda acerca de la capacidad de expresar otras funciones recursivas que no sean primitivas [Cockett y Fukushima 1992]. Comprobamos cómo la función de Ackermann (no primitivo recursiva) no sólo se puede expresar a partir de este esquema de eliminación dependiente sino que, incluso más, es un catamorfismo (esquema de eliminación no dependiente). Por contra, restringiendo la noción de catamorfismo, demostramos que esta función ya no satisface esa condición.

En el sexto capítulo estudiamos las mónadas, surgidas en el intento de hacer computación imperativa con los lenguajes funcionales sin realizar cambios de estados. Nos interesamos por las estructuras subyacentes en sus álgebras (Eilenberg-Moore y Kleisli); además, en dos casos (excepciones y lectores de estado), van acompañadas de las demostraciones en Coq de los asertos involucrados en su desarrollo.

El siguiente capítulo está dedicado a reunir las dos herramientas básicamente utilizadas en todo el trabajo: las mónadas y las funciones de orden superior construyendo, dentro del ámbito de las diálgebras, los ya citados catamorfismos monádicos [Fokkinga 1994], [Hu y Iwasaki 1995], [Meijer y Jeuring 1995]. Comprobamos que, a pesar de haber ampliado la perspectiva, la exigencia que hacemos es la misma que se requiere en el caso de endofuntores y álgebras iniciales. Analizamos, en el mismo contexto dialgebraico, la construcción dual que da lugar a los *anamorfismos comonádicos*. Utilizamos para ello la posibilidad de definir y manejar estructuras en Coq, obteniendo representaciones globales dentro del marco de las mónadas.

Para finalizar, hablamos de los tipos cuantificados existencialmente, sumamente útiles en varios aspectos: en el manejo de estructuras heterogéneas y, por ende, de funciones politipadas; en situaciones, como las que surgen en programación concurrente donde no se sabe inicialmente el tipo que corresponde a la respuesta del servidor; además, los tipos abstractos son tipos existencialmente cuantificados [Mitchell y Plotkin 1988].

2 Conceptos Básicos

2.1 *Introducción*

En este capítulo se presentan las nociones necesarias para hacer el trabajo autocontenido. En primer lugar se describe la sintaxis del λ-cálculo. Posteriormente se definen los conceptos categóricos necesarios para un claro entendimiento del resto del trabajo. Los ejemplos presentados de estas construcciones van encaminados a relacionar el λ-cálculo, la programación funcional y las categorías. Así, se analizan construcciones para describir la interpetración categórica del λ-cálculo y la semántica denotacional de un lenguaje funcional tipado. Los teoremas de esta sección son sólo enunciados y no van acompañados de demostración, citándose, cuando sea el caso, dónde pueden encontrarse.

2.2 *λ-cálculo tipado polimórfico*

En el siglo XX ha habido al menos dos líneas de desarrollo de la noción de tipo. Una de éstas intenta usar los tipos para resolver los problemas de los fundamentos de las matemáticas (problemas de paradojas e inconsistencias). Otra, que es la que aquí nos interesa, tiene como objetivo su uso en los lenguajes de programación: los tipos pueden ser usados para forzar restricciones sobre la estructura de los programas bien formados, detectando fallos en los programas en tiempo de compilación: *la programación es un proceso claramente propenso a errores y una gran cantidad de evidencias prácticas ha demostrado que el uso de un sistema de tipos en un lenguaje de programación puede de forma efectiva detectar errores de los programas en tiempo de compilación. Además, recientes estudios han indicado que el uso de tipos puede conducir a un significativo aumento del rendimiento de los programas en tiempo de ejecución* (sic, [Xi1999]). Por otra parte, pueden soportar abstracción de datos y modularidad. Pero, esto que por una parte es una ventaja tiene sus inconvenientes: puede ocurrir que algún programa aparentemete correcto sea rechazado aunque en tiempo de ejecución su comportamiento fuese razonable (veremos este caso con el estudio de los tipos anidados, rechazados en versiones sin cuantificación existencial de CAML o Haskell); además, el sistema de tipos puede demandar muchas anotaciones por parte del programador, consumiendo mucho tiempo a la hora de escribir o siendo tedioso de leer (por ejemplo, los programas escritos en deCAML, extensión de ML con tipos dependientes [Xi1999]). Ambas líneas se encardinan en el λ-cálculo de Alonzo Church.

Un λ-cálculo tipado polimórfico (denominado de Girard-Reynolds por haberlo descubierto ambos de forma independiente) es un lenguaje que consta de una colección de tipos y, para cada tipo, una colección de términos de ese tipo. Un tipo puede ser visto, entonces, como una propiedad de una expresión que describe cómo ésta puede ser usada;

por ejemplo, si decimos que la expresión (es decir, el programa) M tiene tipo $s \to t$ queremos decir que puede ser aplicada a una expresión de tipo s y da como resultado una expresión de tipo t. Que una expresión tenga un tipo polimórfico significa que puede tener muchos tipos. En una de sus formas más simples, el polimorfismo surge del uso de una variable que especifica un tipo indeterminado (o parametrizado) para una expresión: es lo que se denomina polimorfismo paramétrico. Seguimos en esta presentación el trabajo de Gray [Gray 1991].

Definición 2.1. *El conjunto* **Type** *de tipos se determina recursivamente como sigue:*

1. *hay un conjunto finito o numerable de tipos constantes básicos,* B;
2. *hay un conjunto numerable de variables de tipo* Tv. *Entonces,* $B + Tv$ *es el conjunto de tipos básicos;*
3. *siendo* σ *y* τ *tipos,* $[\sigma \to \tau]$ *es un tipo, denominado el* tipo espacio-función*;*
4. *siendo* σ *un tipo y* t *una variable de tipo,* $\forall t.\sigma$ *es un tipo, denominado el* tipo cuantificado universalmente. *Diremos que la variable* t *está* acotada *en* $\forall t.\sigma$; *en otro caso, la variable se dice* libre. *Al conjunto de variables libres en el tipo* τ *lo denotaremos con* $fv(\tau)$.

Construimos, a continuación, los términos del lenguaje. Para cada tipo τ hay

1. un conjunto numerable de variables de tipo τ, Var^τ;
2. un conjunto finito o numerable de constantes de tipo τ, $Const^\tau$. Entonces, $Atoms^\tau = Var^\tau + Const^\tau$ es el conjunto de átomos de tipo τ.

Indicaremos el hecho de que el término f tiene tipo τ escribiendo $f : \tau$.

Definición 2.2. *El conjunto de términos de tipo* τ, $Terms^\tau$, *se describe recursivamente como sigue:*

1. $Atoms^\tau \subset Terms^\tau$;
2. *si* $f : [\sigma \to \tau]$ *y* $g : \sigma$, $f\ g : \tau$. $f\ g$ *se denomina término* aplicación*;*
3. *si* $g : \tau$ *y* $x \in Var^\sigma$, $(\lambda x.g) : [\sigma \to \tau]$, *término denominado* abstracción*;*
4. *si* $t \in Tv$ *y* $g : \sigma$ *tiene la propiedad de que para cada* $x : \tau \in FV(g)$, $t \notin fv(\tau)$ *entonces* $\Lambda t.g : \forall t.\sigma$. $\Lambda t : g$ *es denominado un* término polimórfico *o la abstracción de tipo de un término (*$FV(g)$ *representa el conjunto de variables libres del término* g *que definimos posteriormente);*
5. *si* $f : \forall t.\sigma$ *y* $\tau \in$ **Type**, *entonces* $f[\tau] : [\tau/t]\sigma$, *se denomina* instanciación *del término* f *en el tipo* τ *(precisaremos más abajo el significado de la expresión* $[\tau/t]\sigma$*.)*

Al lenguaje del λ-cálculo así definido lo denotaremos por L. Un término de la forma $(\lambda x.t)u$ lo denominaremos *redex*. En lo que sigue prescindimos de indicar el tipo correspondiente a cada término.

Definición 2.3. *Las apariciones libres de una variable x en un término t se definen como sigue:*

1. *si $t = x$, x es libre en t;*
2. *si $t = u(v)$ las apariciones libres de x en t son las de x en u y v;*
3. *si $t = \lambda y.u$, las apariciones libres de x en t son las de x en u salvo si $x = y$ en cuyo caso caso x no ocurre libre en t;*
4. *si $t = \Lambda\alpha.u$, las apariciones libres de x en t son las de x en u;*
5. *si $t = u[\tau]$, las apariciones libres de x en t son las de x en u.*

A un término o a un tipo sin variables libres lo denominamos *cerrado*. Consideremos $u,\ t_1,\ \ldots, t_n$ términos del λ-cálculo y sean $x_1,\ \ldots, x_n$ variables distintas. Denotamos con $u < t_1/x_1, \ldots, t_n/x_n >$ el resultado de sustituir en u todas las apariciones libres de x_i por $t_i,\ 1 \leq i \leq n$.

Definimos una relación de equivalencia en L, denominada α-equivalencia e indicada por $u \equiv u'$ de la forma siguiente:

1. si u es una variable $u \equiv u'$ si y sólo si $u = u'$;
2. si $u = w(v)$, se tiene $u \equiv u'$ si y sólo si $u' = w'\ v'$ con $w \equiv w'$, $v \equiv v'$;
3. si $u = \lambda x.v$, $u \equiv u'$ si y sólo si $u' = \lambda x'.v'$ con $v < y/x > \equiv v' < y/x' >$ para casi todas las variables y.

Denotaremos con $\Lambda = \mathrm{L}/\equiv$.

Consideremos $u,\ t_1,\ \ldots, t_n$ términos del lenguaje y sean $x_1,\ \ldots, x_n$ variables distintas. Denotamos con $u[t_1/x_1, \ldots, t_n/x_n]$ el resultado de sustituir en u todas las apariciones libres de x_i por $t_i,\ 1 \leq i \leq n$ (obviamente, cada término con un tipo adecuado para efectuar la sustitución).

Vamos a definir una relación binaria β_0 sobre Λ. La indicaremos con $t\ \beta_0\ t'$ significando que t' se obtiene por β-reducción en t; de nuevo la definimos por inducción:

1. si t es una variable nunca se verifica $t\ \beta_0\ t'$;
2. si $t = \lambda x.u$ entonces $t\ \beta_0\ t'$ si $t' = \lambda x.u'$, donde $u\ \beta_0\ u'$;
3. si $t = u(v)$, $t\ \beta_0\ t'$ si:
   $t' = v(u')$, con $u\ \beta_0\ u'$ o
   $t' = v'(u)$, con $v\ \beta_0\ v'$ o
   $v = \lambda x.w$ y $t' = w[u/x]$.

La β-conversión es la menor relación binaria sobre Λ reflexiva transitiva y conteniendo β_0 . Cuando los conjuntos B y $Const^\tau$ son vacíos para cada tipo τ, el λ–cálculo se denomina λ–cálculo polimórfico tipado puro. Prescindiendo de los tipos cuantificados universalmente, de las abstracciones de tipo y de las instanciaciones de tipos de términos tenemos el λ-cálculo tipado. Cuando no hay siquiera términos constantes tenemos el λ-cálculo simplemente tipado.

2.3 *Prolegómenos categóricos*

2.3.1 Categorías, funtores, transformaciones naturales

Una **categoría C** consta de una colección de objetos $(A, B, \ldots)$ y morfismos $f : A \to B$, con dominio A y codominio B, junto con una ley de composición asociativa; es decir, si $f : A \to B$ y $g : B \to C$ son morfismos en $\mathbf{C}$, $f; g : A \to C$ es otro morfismo en la categoría, y se dice que f y g son combinables. Para cada objeto A hay un morfismo denominado *identidad*, $id_A : A \to A$. Éste verifica, para cualquier morfismo $f : A \to B$, que $f; id_B = f$ e $id_A; f = f$. Un morfismo $f : A \to B$ es un *isomorfismo* si existe $f^{-1} : B \to A$ en $\mathbf{C}$ tal que $f; f^{-1} = id_A$ y $f^{-1}; f = id_B$.

Denotaremos con $|\mathbf{C}|$ la colección de objetos y, dados $A,\ B \in |\mathbf{C}|$, $\mathbf{C}(A, B)$ representa la colección de morfismos de dominio A y codominio B. Cuando para cada par de objetos $A,\ B \in |\mathbf{C}|$, $\mathbf{C}(A, B)$ es un conjunto, la categoría $\mathbf{C}$ se llama *pequeña*.

Ejemplo 2.1. *[Gray 1991] Una función recursiva parcial es una función parcial entre nat y nat que es computada por alguna máquina de Turing. Consideramos PRF el conjunto de todas las funciones recursivas parciales; ya que PRF es numerable existe una función sobreyectiva $e : nat \to PRF$; indicamos con $n.m = e(n)(m)$ el valor de la n–ésima función recursiva parcial en el argumento m.*

Vamos a ver, entonces, que las relaciones de equivalencia parciales (denotadas por per) sobre los números naturales forman una categoría que denotaremos **PER**. *Si A es una per (relación simétrica y transitiva en $nat \times nat$), definimos $dom(A) = \{n | nAn\}$; denotamos con $[n]_A = \{m | nAm\}$ la clase de equivalencia de n mod A. Por último, el conjunto de las clases de equivalencia la representamos con $Q(A) = \{[n]_A | n \in dom(A)\}$.*

Siendo A y B pers, un morfismo f entre A y B es una función $f : Q(A) \to Q(B)$ tal que hay un $n_f \in nat$ (denominado testigo para f) con la propiedad de que para cada $d \in dom(A)$, $f([p]_A) = [n_f.p]_B$. La composición de morfismos de pers, definida como sigue, también es un morfismo de pers: si $f : A \to B$, $g : B \to C$ tienen como testigos, respectivamente,$n_f, ; n_g$, $g(f(p)) = [n_g.(n_f.p)]_C$.

Entonces, las pers junto con estos morfismos determinan la categoría **PER**.

Cada categoría $\mathbf{C}$ tiene su categoría dual (u opuesta), $\mathbf{C^{op}}$, cuyos objetos son los de $\mathbf{C}$ y $f : B \to A$ es un morfismo en $\mathbf{C^{op}}$ si y solo si $f : A \to B$ es un morfismo de $\mathbf{C}$. A cada concepto definido en $\mathbf{C}$ le corresponde un concepto dual (de nuevo en $\mathbf{C}$) que resulta de definir esa noción en $\mathbf{C^{op}}$. Por ejemplo, un objeto I es **inicial** en $\mathbf{C}$ si, para cada objeto A de $\mathbf{C}$ existe uno y sólo un morfismo en $\mathbf{C}$ entre I y A. Dualmente, un objeto T es **final** en $\mathbf{C}$ si es inicial en $\mathbf{C^{op}}$. Si existe un objeto inicial en una categoría, es único salvo isomorfismos.

Ejemplo 2.2. *Supongamos que $\mathbf{C}$ y $\mathbf{D}$ son dos categorías; la categoría producto $\mathbf{C} \times \mathbf{D}$ tiene como objetos pares (A, B), donde A es un objeto de la categoría $\mathbf{C}$ y B es un objeto de la categoría $\mathbf{D}$; un morfismo entre los objetos (A, B) y (A', B') es un par (f, g), con $f : A \to A'$ y $g : B \to B'$ morfismos, respectivamente, en $\mathbf{C}$ y en $\mathbf{D}$. La composición de los morfismos (f, g) y (h, k) es, como parece natural, $(f; h, g; k)$.*

Un *funtor* $F : \mathbf{C} \to \mathbf{D}$ entre categorías es una aplicación que asocia cada objeto A de $\mathbf{C}$ con un objeto $F(A)$ de $\mathbf{D}$ y asigna, a cada morfismo $f : A \to B$ de $\mathbf{C}$, un morfismo

$F(f) : F(A) \to F(B)$ en $\mathbf{D}$ (si el resultado es un morfismo $F(f) : F(B) \to F(A)$ el funtor será denominado contravariante) de forma que, para cualquier objeto A de $\mathbf{C}$, y cualesquiera morfismos en $\mathbf{C}$, f y g, combinables,

$$F(id_A) = id_{F(A)},\ F(f;g) = F(f);F(g).$$

El funtor $I : \mathbf{C} \to \mathbf{C}$, definido sobre los objetos de $\mathbf{C}$ por medio de $I(A) = A$ y, sobre cualquier morfismo en $\mathbf{C}$, $f : A \to B$, como $I(f) = f$ es denominado funtor identidad.

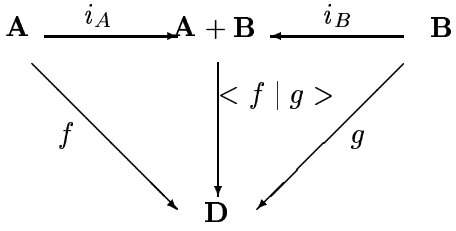
Sean $\mathbf{C}$ y $\mathbf{D}$ categorías y $D \in |\mathbf{D}|$; el funtor que asigna a cada objeto de $\mathbf{C}$ el objeto D y a cada flecha $f \in \mathbf{C}(C, C')$ el morfismo id_D se denomina *constante*.

Un funtor $S : \mathbf{B} \times \mathbf{C} \to \mathbf{D}$ es un *bifuntor* (sobre $\mathbf{B}$ y $\mathbf{C}$) si, fijando un argumento (bien de $\mathbf{B}$ bien de $\mathbf{C}$) el resultado es un funtor en el otro argumento.

Si $F : \mathbf{C} \to \mathbf{D}$ es un funtor y D es un objeto de $\mathbf{D}$, un **morfismo universal** de D a F es un par (R, u) formado por un objeto R de $\mathbf{C}$ y un morfismo $u : D \to F(R)$ en $\mathbf{D}$ tal que para cada par (C, f) con C objeto de $\mathbf{C}$ y $f : D \to F(C)$, existe un único morfismo $f' : R \to C$ que verifica $u; F(f') = f$. Es fácil ver un morfismo universal como un objeto inicial; para ello, basta con hallar la categoría adecuada. Así, en las condiciones anteriores, definimos la categoría $\mathbf{F}^D$ como sigue: sus objetos son pares (S, v) con S objeto de $\mathbf{C}$ y v morfismo entre D y $F(S)$; un morfismo entre los objetos (S, v) y (T, w) es un morfismo $f : S \to T$ en $\mathbf{C}$ que verifica $v; F(f) = w$. Entonces, un objeto inicial en esta categoría $\mathbf{F}^D$ es un morfismo universal de D a F.

Ejemplo 2.3. *Sean* $\mathbf{C}$ *una categoría y* $\Delta : \mathbf{C} \to \mathbf{C} \times \mathbf{C}$ *el funtor diagonal, definido sobre los objetos por* $\Delta(C) = <C, C>$ *y sobre los morfismos por* $\Delta(f) = <f, f>$. *Un morfismo universal desde un objeto* $<A, B>$ *de* $\mathbf{C} \times \mathbf{C}$ *al funtor* Δ *es denominado un diagrama coproducto. Consta de un objeto* $A + B$ *de* $\mathbf{C}$ *y un morfismo* $<A, B> \to <A + B, A + B>$ *en* $\mathbf{C} \times \mathbf{C}$, *i.e., un par de morfismos* $i_A : A \to A + B$, $i_B : B \to A + B$. *Para cualquier par de morfismos* $f : A \to D$, $g : B \to D$, *existe un único morfismo* $< f|g > : A + B \to D$ *con* $i_A; < f|g > = f$, $i_B; < f|g > = g$.

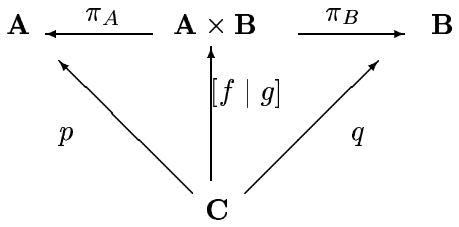
Podemos expresar esto por medio de un diagrama conmutativo:



Si cada par de objetos A y B in $\mathbf{C}$ tiene coproducto, el coproducto $+ : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ es un funtor en cada variable, con $p + q$ definido sobre los morfismos $p : A \to C$, $q : B \to D$ como el único morfismo $p + q : A + B \to C + D$ con $i_A; (p + q) = p; i_C$, $i_B; (p + q) = q; i_D$.

Dualmente, se tiene el concepto de producto del par de objetos A y B, $(A \times B, \pi_A, \pi_B)$, como el objeto final de la categoría cuyos objetos son tuplas (C, p, q) con C objeto de $\mathbf{C}$ y $p : C \to A$, $q : C \to B$ morfismo en $\mathbf{C}$ y teniendo como morfismo entre los objetos (C, p, q) y (D, r, s) un morfismo $k : C \to D$ que verifica $k; r = p$ y $k; s = q$.

Gráficamente estamos hablando de la conmutatividad del diagrama



Entonces, $\times : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ es el funtor producto.

Ejemplo 2.4. *Un lenguaje funcional basado en el λ-cálculo polimórfico puro presentado en la sección 2.2 es una categoría que, a lo largo del presente trabajo, denotaremos con* **T**. *En notación BNF describimos los tipos y los términos ya conocidos:*

$$\begin{array}{ll} \textit{Tipos} & A ::= X | A \to B | \forall X.A \\ \textit{Términos} & t ::= x | \lambda x.t | u(t) | \Lambda X.t | t(A) \end{array}$$

Un ejemplo de término definido en este lenguaje es la función (escrita en Coq*)*

```
Definition maplist:=[A,B:Set][f:A->B]
(list_rec A [_:(List  A)](List  B) (Nil B)
[y:A][_:(List  A)][l:(List  B)](Cons B (f y) l)).
```

que tiene tipo $\Lambda AB.(A \to B) \to (List\ A) \to (List\ B)$. *Entonces,*

```
Definition maplistnb:=(maplist nat bool).
```

que proviene de la instanciación de término $(\Lambda AB.(A \to B) \to (List\ A) \to (List\ B))(nat\ bool)$, *determina la función entre las listas de naturales y la lista de booleanos parametrizada sobre las funciones entre naturales y booleanos.*

Los tipos del lenguaje son los objetos de la categoría y las funciones entre tipos son los morfismos. El dominio y el codominio de un morfismo son, respectivamente, los tipos de entrada y salida de la correspondiente función. La composición en esta categoría es la composición de funciones en el lenguaje.

En **T** *hay un objeto inicial,* $\bot = \forall X.X$, *y un objeto final,* $\Upsilon = \forall X.X \to X$ *[Plotkin y Abadi 1993]. Además, posee tanto productos como coproductos binarios, expresados como sigue:*

$$A \times B = \forall X.(A \to B \to X) \to X;$$

$$A + B = \forall X.(A \to X) \to (B \to X) \to X.$$

Aunque este lenguaje carece de tipos constantes básicos (enteros, booleanos) podemos obtenerlos a partir de las construcciones permitidas [Gray 1991], [Pierce 1993]:

$$bool = \forall X.(X \to (X \to X));$$

$$nat = \forall X.(X \to (X \to X) \to X).$$

Tipos más complejos como el de las listas de elementos del tipo T también pueden describirse [Pierce 1993]

$$List\ A = \forall X.((A \to X \to X) \to X \to X).$$

Los elementos de este tipo quedan expresados como sigue:

$$nil = \lambda\ (X : Set).(h : A \to X \to X).(x : X).x;$$

$cons = \lambda\ (a : A).(ls : (List\ A).(X : Set).(h : A \to X \to X).(x : X).(ls\ X\ h\ (h\ a\ x)).$

Podemos, incluso, expresar la cuantificación existencial sobre los tipos [Plotkin y Abadi 1993], [Hasegawa 1994]:

$$\exists X.F(X) = \forall Y.(\forall X.(F(X) \to Y) \to Y).$$

Usando la `formulae-as-types analogy` *[Asperti y Longo 1991] se puede comprobar la exactitud de todas estas expresiones [recordemos que, en cálculo de proposiciones, bastan los símbolos $\neg$ y $\vee$ para expresar cualquier fórmula contándose, además de las leyes de De Morgan, con las siguientes equivalencias:*

$$A \to B \equiv \neg A \vee B;$$

$$\neg(A \vee B) \equiv (\neg A) \wedge (\neg B);$$

$$\neg(A \wedge B) \equiv (\neg A) \vee (\neg B)$$

Como caso particular, comprobamos el caso de la cuantificación existencial:

$$\forall Y.(\forall X.(F(X) \to Y) \to Y) \equiv \forall Y.(\forall X.\neg(F(X) \to Y) \vee Y) \equiv$$

$$\forall Y.(\forall X.\neg(\neg(F(X)) \vee Y) \vee Y) \equiv \forall Y.(\forall X.((F(X)) \wedge \neg Y) \vee Y) \equiv$$

$$\forall Y.(\forall X.((F(X)) \vee Y) \wedge (\neg Y \vee Y) \equiv \forall Y.(\forall X.((F(X)) \vee Y);$$

ahora bien, si no existiera algún X verificando $F(X)$ resultaría $\forall Y.(\forall X.Y)$. Pero, $\forall Y.Y = \bot$, de donde la veracidad de ese predicado implica la existencia de un valor, X_0, que cunpla $F(X_0)$.

Definición 2.4. *[Cockett y Fukushima 1992] Una categoría predistributiva es una categoría con productos finitos y coproductos binarios verificando que la aplicación*

$$< in_A \times id_X | in_B \times id_X >: (A \times X) + (B \times X) \to (A + B) \times X$$

es invertible.

Definición 2.5. *Una categoría* $\mathbf{C}$ *es* cartesiana *si:*

1. *tiene un objeto terminal, Υ;*
2. *cualquier par $A,\ B \in |\mathbf{C}|$ tiene producto $(A \times B, \pi_1 : A \times B \to A, \pi_2 : A \times B \to B)$.*

Definición 2.6. *Sea* $\mathbf{C}$ *una categoría cartesiana y sean* $A,\ B \in |\mathbf{C}|$*; el* exponente *de* A *y* B *es un objeto* B^A *con un morfismo* $eval_{A,B} : A \times B^A \to B$ *que verifica, para todos los morfismos* $f : A \times C \to B$ *existe un único morfismo* $f^\sharp : C \to B^A$ *que verifica*

$$(id_A \times f^\sharp); eval_{A,B} = f.$$

Fijando un objeto $A \in \mathbf{C}$, construimos el funtor $(-)^A : \mathbf{C} \to \mathbf{C}$: dado un objeto $B \in \mathbf{Set}$, B^A es el exponente de A y B; y dado un morfismo $f : B \to C$, f^A es el morfismo (único por definición) $(eval_{A,B}; f)^\sharp : B^A \to C^A$.
En el caso de $\mathbf{Set}$, $f^A = \lambda h : B^A.\lambda a : A.f(h(a))$.

Definición 2.7. *Una categoría* $\mathbf{C}$ *es* cartesiana cerrada *si:*

1. $\mathbf{C}$ *es cartesiana;*
2. *para cualquier par* $A,\ B \in |\mathbf{C}|$ *existe su exponencial.*

Ejemplo 2.5. *[Gray 1991] La categoría PER descrita en el ejemplo 2.1 es cartesiana cerrada. Si* A *y* B *son pers, decimos que* n $A \times B$ m *si y sólo si* $(p_1.n)$ A $(p_1.m)$ *y* $(p_2.n)$ B $(p_2.m)$*, siendo* $p_1,\ p_2 : nat \times nat \to nat$ *las proyecciones primera y segunda.*

El objeto espacio de funciones se define como sigue: si A *y* B *son pers,* n $[A \to B]$ m *si y sólo si* $\forall p,\ q,\ p\ A\ q \Rightarrow (n.p)\ B\ (n.q)$.

Para finalizar, el morfismo aplicación es $app_{A,B} : [A \to B] \times A \to B$ *viene dado por* $app_{A,B}([n]_{A\to B}, m_A) = [n.m]_B$.

Ejemplo 2.6. *[Gray 1991] El* λ*–cálculo polimórfico de la sección 2.2 puede ser interpretado en la categoría* $||\mathbf{Per}|^{Tv} \to \mathbf{Per}|$*, donde* $|\mathbf{PER}|^{Tv}$ *es la categoría [cartesiana cerrada] de funtores entre la categoría discreta* Tv *y* $\mathbf{Per}$ *que podemos interpretar como la categoría [discreta] de entornos de tipo.*

Vamos a definir la interpreción de los tipos. Para ello, definimos $D_{(-)} : Type \to ||\mathbf{Per}|^{Tv} \to \mathbf{Per}|$ *por medio de las siguientes cláusulas:*

1. $D_t = \lambda S.S(t)$;
2. $D_{\sigma\to\tau} = [D_\sigma \to D_\tau]$;
3. $D_{\forall t.\sigma} = \lambda S.\prod_{A\in|\mathbf{PER}|} D_\sigma(S[A/t])$.

De modo análogo definimos una interpretación de los términos.

1. *Dado* $g \in Terms^\sigma$ *consideramos*

$$Env(g)(S) = (\prod_{x\in FV(g)} D_{type(x)})(S) = \prod_{x\in FV(g)} (D_{type(x)})(S))$$

de donde $[[g]]_S : Env(g)(S) \to D_\sigma(S)$*, es decir,*

$$[[g]]_S : \prod_{x\in FV(g)} (D_{type(x)})(S)) \to D_\sigma(S).$$

2. $[[\Lambda t.g]]_S : \prod_{x \in FV(\Lambda t.g)}(D_{type(x)})(S)) \to \prod_{A \in |\mathbf{PER}|}(D_\sigma(S[A/t])$.

3. *Si* $f \in Terms^{\forall t.\sigma}$,

$$[[f]]_S : \prod_{x \in FV(f)} (D_{type(x)})(S)) \to \prod_{A \in |\mathbf{PER}|} (D_\sigma(S[A/t]).$$

Definición 2.8. *Sea* $\mathbf{C}$ *una categoría con productos y coproductos finitos. Un funtor* **polinomial** $F : \mathbf{C} \to \mathbf{C}$ *es cualquier funtor construido a partir de los funtores constantes y el funtor identidad por medio de productos, coproductos y composiciones de estos.*

Definición 2.9. *Consideremos* $\mathbf{Cat}$ *la categoría cuyos objetos son categorías y cuyos morfismos son funtores y sea* $\mathbf{C}$ *una categoría arbitraria; una categoría* indexada *por* $\mathbf{C}$ *es un funtor* $T : \mathbf{C^{op}} \to \mathbf{Cat}$. *Toda categoría* $\mathbf{C}$*-indexada* $T : \mathbf{C^{op}} \to \mathbf{Cat}$ *define una categoría fibrada* $\mathbf{C}(T)$ *con objetos pares* (i, X), i *objeto de* $\mathbf{C}$ *y* X *objeto de* $T(i)$ *y morfismos* $(\phi, f) : (i, X) \to (j, Y)$ *donde* $\phi : i \to j$ *es un* $\mathbf{C}$*-morfismo y* $f : X \to T_\phi(Y)$ *es un morfismo en* $T(i)$. *A* $T(i)$ *la denominamos fibra sobre* i.

Las categorías fibradas dan un marco perfecto para interpretar, en otro contexto, el lambda cálculo polimórfico del ejemplo 2.2 [Crole 1993].

Si $F, G : \mathbf{C} \to \mathbf{D}$ son funtores, una colección $\sigma \equiv \sigma_A : F(A) \to G(A)$ de morfismos de $\mathbf{D}$, indicada por los objetos A of $\mathbf{C}$ se denomina **transformación natural** si, para todo morfismo $f : A \to B$ en $\mathbf{C}$, es conmutativo el diagrama

$$\begin{array}{ccc} \mathbf{F(A)} & \xrightarrow{F(f)} & \mathbf{F(B)} \\ {\scriptstyle \sigma_A}\downarrow & & \downarrow{\scriptstyle \sigma_B} \\ \mathbf{G(A)} & \xrightarrow[G(f)]{} & \mathbf{F(B)} \end{array}$$

es decir,

$$\sigma_A ; G(f) = F(f) ; \sigma_B.$$

Se representarán las transformaciones naturales de F a G por medio de $\sigma : F \Rightarrow G$.

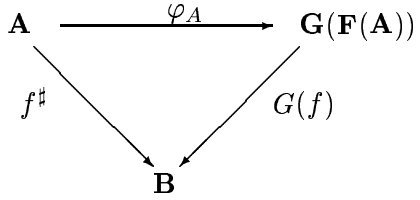
Ejemplo 2.7. *Para cada tipo* A, *sea* B^A *el exponente de* A *y* B, *representando el tipo función* $A \to B$. *Esta exponenciación determina el endofuntor sobre* $\mathbf{T}$, $(-)^A$, *en el caso general. Como vimos en la Definición 2.6 para el caso de conjuntos, dado un morfismo* $h : B \to C$, $h^A(f) = f ; h$, *para cada* $f : A \to B$. *En tales condiciones,* $eval_{A,B} : A \times B^A \to B$, *definida por* $eval_{A,B}(x, f) = f(x)$, *determina una transformación natural* $eval_{A,-} : (A \times (-)^A) \Rightarrow I$. *En efecto,* $eval_{A,B} ; h = (id_A \times h^A) ; eval_{A,C}$ *ya que ésta es, precisamente, la condición que nos permitió construir* h^A *[Definición 2.6].*

2.3.2 Adjunciones. Pullbacks

Una *adjunción* entre las categorías $\mathbf{A}$ y $\mathbf{B}$ es una terna (F, G, φ), con $F : \mathbf{A} \to \mathbf{B}$ y $G : \mathbf{B} \to \mathbf{A}$ funtores, $\varphi : 1_\mathbf{A} \Rightarrow (F ; G)$ una transformación natural (denominada la *unidad*

de la adjunción) que verifica, para cada par de objetos $A \in \mathbf{A}$, $B \in \mathbf{B}$ y cada morfismo $f \in \mathbf{B}(F(A), B)$ existe un único morfismo $f^\sharp \in \mathbf{A}(A, G(B))$ tal que $\varphi_A; G(f) = f^\sharp$.

Es decir, $f^\sharp$ es el único morfismo haciendo conmutativo el diagrama



Ejemplo 2.8. *([Rydeheard 1985]) Sean* **Set** *la categoría de conjuntos y* **Monoids** *la categoría de los monoides. Consideremos, además, List el funtor que transforma un conjunto S en el monoide List S, cuyos elementos son las listas finitas sobre S y que tiene como operación asociativa la concatenación, ::, y como elemento neutro la lista vacía, [ ]. Por último, sea U el funtor de olvido, que lleva a cualquier monoide en su conjunto soporte. Es bien conocido que el funtor List es adjunto a la izquierda del funtor U. Por consiguiente, dada una aplicación (en* **Set***) $f : S \to nat$, existe un único homomorfismo de monoides $f^\sharp : (List\ S, ::, [\ ]) \to (nat, +, 0)$. Este homomorfismo satisface las dos condiciones siguientes:*
$f^\sharp\ [\ ] = 0$;
$f^\sharp\ x :: xs = (f x) + (f^\sharp\ xs)$.
Cuando$f$ es la función que a cualquier elemento de S le asigna 1, *$f^\sharp$ determina la longitud de las listas sobre S.*

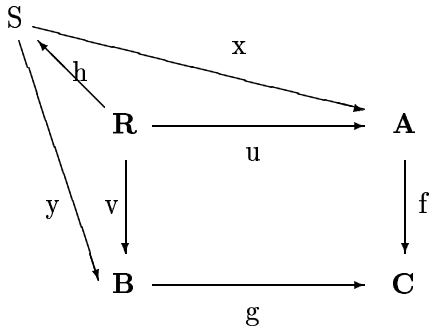
Ejemplo 2.9. *Siendo* **C** *una categoría cartesiana cerrada y A un objeto fijo en* **C** *, el endofuntor $(-)^A$ es adjunto a la derecha del endofuntor $A \times (-)$.*

Para cada morfismo $f : A \times B \to C$ existe un único morfismo $f^\sharp : B \to C^A$ verificando $(id_A \times f^\sharp); eval_{A,C} = f$ [Definición 2.6]. Podemos definir la unidad, $\varphi_B : B \to (A \times B)^A$, como ${id_{A\times B}}^\sharp$. Entonces, $\varphi_B; f^A = f^\sharp$.
Veamos por qué:

$$(\varphi_B; f^A); eval_{A,C} = {id_{A\times B}}^\sharp; ((eval_{A,B}; f)^\sharp; eval_{A,C}) = {id_{A\times B}}^\sharp; (eval_{A,B}; f) =$$

$({id_{A\times B}}^\sharp; eval_{A,B}); f = id_{A\times B}; f = f$.
Dado que $f^\sharp$ es el único morfismo que compuesto con $eval_{A,C}$ retorna f, $f^\sharp$ y $\varphi_B; f^A$ deben ser iguales.

Otra construcción susceptible de ser analizada por sus propiedades de finalidad en una categoría adecuada es la de *pullback*. Sean $f : A \to C$, $g : B \to C$ dos morfismos en la categoría **C**; consideramos la categoría cuyos objetos son ternas (R, u, v), con R objeto de la categoría y $u : R \to A$, $v : R \to B$ verificando la comnmutatividad de



es decir, $u; f = v; g$. Un morfismo entre (R, u, v) y (S, x, y) es $h : R \to S$ con la condición de que $h; x = u$, $h; y = v$. Un *pullback* es un objeto terminal en esta categoría; la noción dual es la de *pushout*. Cuando $A = B$ y $u = v$ al *pullback* se le denomina igualador y al *pushout* coigualador. A continuación vemos un sencillo ejemplo de *pushout*.

Ejemplo 2.10. *Sean*

```
let f(n,m)=n*m;;

let g(n,m)=(n*n,m*m);;
```

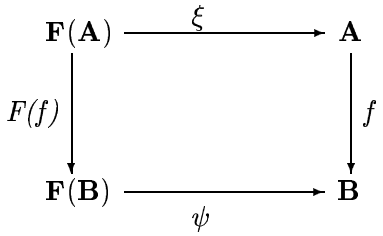
dos funciones. A partir de la unión disjunta `nat+nat*nat` *podemos construir el coigualador de* `f;in1, g;in2`*, que denotamos con* `(CoIg(f;in1,g;in2),H)`*. Si existe una terna* `(A,u,v)` *tal que* `f;u=g;v`*, por la propiedad universal de inicialidad del coproducto, existe un único morfismo* `[u,v]:nat+nat*nat -> A` *que verifica* `in1;[u,v]=u,` *y* `in2;[u,v]=v`*. Como el coigualador se obtiene a partir de* `nat+nat*nat` *por medio de un cociente [Lambeck y Scott 1989] podemos definir un único morfismo* `h:CoIg(f;in1,g;in2) -> A` *satisfaciendo la conmutatividad requerida. Es evidente que, en este caso,* `CoIg(f;in1,g;in2)=nat` *y* `H=[H1,H2]`*, donde* `H1 n=n*n, H2 n=n.`

Más adelante tendremos ocasión de ver que muchas funciones interesantes pueden ser interpretadas como coigualadores y como *pullbacks*; es natural, entonces, pretender construir funciones, partiendo de sus especificaciones, como *pullbacks* y coigualadores.

2.3.3 Álgebras y puntos fijos. Límites y colímites

Definición 2.10. *([Manes y Arbib 1986]) Sean* **C** *una categoría y F un endofuntor de* **C**, *es decir, un funtor de* **C** *en sí misma; una F-***álgebra** *o un álgebra sobre F es un par (A, ξ), con A un objeto de* **C** *y ξ un morfismo entre FA y A. Dualmente, una F-***coálgebra** *o una coálgebra sobre F será un par (A, ξ) con A un objeto de* **C** *y ξ un morfismo entre A y FA.*

Además, si (A, ξ) y (B, ψ) son F-álgebras, un morfismo de F-álgebras es un morfismo $f : A \to B$ tal que $\xi; f = F(f); \psi$ es decir, conmuta el diagrama



Dualmente, dadas (A, ξ) *y* (B, ψ) *F-coálgebras,* $f : A \to B$ *es un morfismo de F-coálgebras si* $\xi; Ff = f; \psi$.

Definición 2.11. *Un* **punto fijo** *de F es un par* (A, ξ) *con* ξ *un isomorfismo,* $(F(A) \simeq A)$.

La funtorialidad de F da lugar a las categorías $\mathbf{F} - \mathbf{Alg}$ de las F-álgebras y $\mathbf{F} - \mathbf{CoAlg}$ de las F-coálgebras.

Si (L, μ) es una F-álgebra, aplicando F a μ resulta que $(FL, F\mu)$ es una F-álgebra y μ es un morfismo de F-álgebras entre $(FL, F\mu)$ y (L, μ). Si (L, μ) es inicial, existe un único morfismo de F-álgebras $f : (L, \mu) \to (FL, F\mu)$; pero id_L es el único morfismo de (L, μ) en sí mismo, de modo que $f; \mu = id_L$. Por consiguiente, $\mu; f = (Ff); (F\mu) = F(f; \mu) = F(id_L) = id_{FL}$. Esto muestra que μ es un isomorfismo con inversa f y por lo tanto:

Teorema 2.1. *([Manes y Arbib 1986]) Un objeto inicial de* $\mathbf{F} - \mathbf{Alg}$ *(un objeto final de* $\mathbf{F} - \mathbf{CoAlg}$*) es un punto fijo de F.*

Definición 2.12. *([Manes y Arbib 1986]) Sea* $F : \mathbf{C} \to \mathbf{C}$ *un endofuntor; un* **menor punto fijo** *de F es un objeto inicial de* $\mathbf{F} - \mathbf{Alg}$*; un* **mayor punto fijo** *de F es un objeto final de* $\mathbf{F} - \mathbf{CoAlg}$*. Nótese que ambos objetos, si existen, son únicos salvo isomorfismos de la categoría correspondiente.*

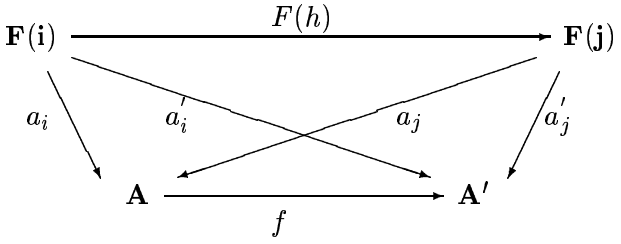
Ejemplo 2.11. *Retomando la categoría del Ejemplo 2.4 determinada por el* λ*-cálculo polimórfico , los tipos derivados como álgebras iniciales o como coálgebras finales se definen, respectivamente,*

$$\mu X.F(X) = \forall X.(F(X) \to X) \to X),$$

$$\nu X.F(X) = \exists X.((X \to F(X)) \times X)$$

como puede verse en [Plotkin y Abadi 1993], [Hasegawa 1994]

Sea $\mathbf{I}$ una categoría pequeña; un funtor $F : \mathbf{I} \to \mathbf{B}$ determina un diagrama en la categoría $\mathbf{B}$ indicado por $\mathbf{I}$. Un *cocono* de este diagrama es un objeto de $\mathbf{B}$, A, junto con morfismos $a_i : F(i) \to A$, $i \in \mathbf{I}$, de forma que, siendo $h : i \to j$ un morfismo en $\mathbf{I}$, conmuta



o sea, $F(h); a_j = a_i$. Un morfismo entre los coconos A y A' del diagrama anterior, es una flecha $f : A \to A'$ que verifica $a_i; f = a'_i$, para cada $i \in \mathbf{I}$. Los coconos de un diagrama y sus morfismos forman una categoría, cuyo objeto inicial se denomina *colímite* del diagrama. Dualmente obtenemos los conceptos de *cono* y *límite*.

Un ejemplo de colímite lo tenemos en el coproducto anteriormente definido donde la categoría $\mathbf{I}$ es la categoría *discreta* (los únicos morfismos son las identidades) con dos objetos. Dualmente, el límite del diagrama sobre esa categoría es el producto.

Ejemplo 2.12. *Consideremos* $\mathbf{I}$ *la categoría cuyos objetos son los números naturales y existe un morfismo entre* n, $m \in \mathbf{I}$ *si* $n \leq m$*; en este caso, el diagrama* $C : \mathbf{I} \to \mathbf{C}$ *se llama* cadena a la derecha *o* w-diagrama. *Es, entonces, una sucesión* $\{c_n : C_n \to C_{n+1} | n \in \mathbf{I}\}$ *donde* $C_i \in \mathbf{C}$, $i \in \mathbf{I}$. *Una* cota superior *de la cadena a la derecha* $\{c_n : C_n \to C_{n+1} | n \in \mathbf{I}\}$ *es un cocono. El objeto inicial de la categoría de los coconos (si existe) es el colímite de la cadena. Análogamente, si* $C : \mathbf{I} \to \mathbf{C}$ *es un funtor contravariante, el diagrama se llama* cadena a la izquierda *o* w^{OP}-diagrama*; esto es, una sucesión* $\{c_n : C_{n+1} \to C_n | n \in \mathbf{I}\}$. *Sus conos se denominan* cotas inferiores *y el límite de estos, si existe, es el objeto final de la categoría que determinan.*

Definición 2.13. *Un funtor* $F : \mathbf{C} \to \mathbf{D}$ *es* **cocontinuo** *si para cualquier cadena a la derecha* $\{c_n : C_n \to C_{n+1} | n \in \mathbf{I}\}$ *en* $\mathbf{C}$ *teniendo a* (U, α) *como colímite,* $(FU, F(\alpha))$ *es el colímite de la cadena*
$\{F(c_n) : F(C_n) \to F(C_{n+1}) | n \in \mathbf{I}\}$ *en* $\mathbf{D}$.

Un funtor $F : \mathbf{C} \to \mathbf{D}$ *es* **continuo** *si para cualquier cadena a la izquierda* $c_n : C_{n+1} \to C_n | n \in N$ *en* $\mathbf{C}$ *teniendo a* (L, β) *como límite,* $(FL, F(\beta))$ *es el límite de la cadena* $F(c_n) : F(C_{n+1}) \to F(C_n) | n \in N$ *en* $\mathbf{D}$.

Un funtor a la par continuo y cocontinuo se denomina bicontinuo.

Definición 2.14. *Una categoría* $\mathbf{C}$ *es* w*-cocompleta si existe el colímite de cualquier* w*-diagrama en* $\mathbf{C}$.

Una categoría $\mathbf{C}$ *es* w*-completa si existe el límite de cualquier* w^{OP}*-diagrama en* $\mathbf{C}$.

Teorema 2.2. *([Manes y Arbib 1986], pág. 270) Sea* $\mathbf{C}$ *una categoría con objeto inicial* $\perp$ *de forma que cada cadena a la derecha tenga un colímite; entonce cada funtor cocontinuo* $F : \mathbf{C} \to \mathbf{C}$ *tiene como menor punto fijo el colímite de la cadena*

$$\perp \to F(\perp) \to F^2(\perp) \to F^3(\perp) \to \ldots$$

siendo, para cada $n \in N$, $F^n(!) : F^n(\perp) \to F^{n+1}(\perp)$ *la única función existente entre ambos.*

Teorema 2.3. *([Manes y Arbib 1986], pág. 276) Sea* $\mathbf{C}$ *una categoría con objeto final* Υ *de forma que cada cadena a la izquierda tenga un límite; entonce cada funtor continuo* $F : \mathbf{C} \to \mathbf{C}$ *tiene como mayor punto fijo el límite de la cadena*

$$\ldots \to F^3(\Upsilon) \to F^2(\Upsilon) \to F(\Upsilon) \to \Upsilon$$

siendo, para cada $n \in N$, $F^n(¡):F^{n+1}(\Upsilon) \to F^n(\Upsilon)$ *la única función existente entre ambos.*

Como los funtores polinómicos son continuos y cocontinuos (por lo tanto bicontinuos ([Manes y Arbib 1986], pág. 276)) todos los funtores polinomicos tienen menor y mayor punto fijo. En [Erwig 1998], [Erwig 2000] se afirma que para estos funtores podemos suponer que los soportes de ambos puntos fijos (menor y mayor) son coincidentes. Generalmente, esto no es así. El contraejemplo más obvio es el que concierne a las *streams*, provenientes del endofuntor $F_A(B) = A \times B$. El mayor punto fijo de este funtor es el conjunto de las listas infinitas de elementos de tipo A mientras que el menor punto fijo es el tipo vacío [Paulin y Werner 1998]. Otro ejemplo, diferente del anterior en el sentido de que los soportes sí son isomorfos, lo tenemos cuando se considera el funtor $F(A) = \Upsilon + A$. Su menor punto fijo es el conjunto de los números naturales mientras que su mayor punto fijo es ese mismo conjunto extendido con un objeto ∞. Ahora bien, el uso de semánticas no estrictas (como ocurre en el caso de Haskell) tiene la ventaja de que el tipo recursivo generado por el endofuntor correspondiente es, a la vez, su menor y mayor punto fijo [Gibbons 2000].

Ejemplo 2.13. *Notemos, pues, con anaF θ el único morfismo entre la F-coálgebra (B, θ) y el mayor punto fijo de F y sea cataF ξ el morfismo, también único, entre el punto fijo inicial y la F-álgebra (A, ξ). Entonces, si el morfismo de comparación, $f : (L, in_L) \to (G, out_G)$, es un isomorfismo, entre la coálgebra y el álgebra siempre tenemos un morfismo,*

$$f_{\theta\xi} = (anaF\ \theta); f^{-1}; (cataF\ \xi)$$

que verifica la condición

$$f_{\theta\xi} = \theta; F(f_{\theta\xi}); \xi.$$

Sí sucede, como mencionamos en la Introducción, que, en algunos casos relevantes en computación (listas, árboles) podemos contar con el mismo soporte para las estructuras finales e iniciales. Esto genera, como se verá en el siguiente capítulo, interesantes posibilidades de extensión de esos morfismos.

Analizamos a continuación con detalle otro funtor polinómico cuyos puntos fijos final e inicial no son (en principio) iguales. Lo único que podemos afirmar es que existe un único morfismo (denominado de comparación) entre el menor y el mayor punto fijo verificando las ecuaciones de álgebra y coálgebra ([Manes y Arbib 1986], pág. 247).

Ejemplo 2.14. *Tomemos como base una categoría cartesiana (con objeto final Υ), predistributiva, w-cocompleta con objeto inicial $\perp$. Fijando el objeto A, el funtor que determina las listas (de tipo A) es $F_A(B) = \Upsilon + A \times B$ que es polinómico y, por ende, bicontinuo. Entonces, el menor punto fijo de F_A se obtiene como el colímite de la siguiente cadena:*

$$\perp \to \Upsilon + A \times \perp \to \Upsilon + A \times (\Upsilon + A \times \perp) \to \Upsilon + A \times (\Upsilon + A \times (\Upsilon + A \times \perp)) \to$$

$$\to \Upsilon + A \times (\Upsilon + A \times (\Upsilon + A \times (\Upsilon + A \times \perp))) \to \ldots$$

Tomando en consideración que $A \times \perp \simeq \perp$, $A + \perp \simeq A$, $A \times \Upsilon \simeq A$ y la distributividad tenemos

$$\perp \to \Upsilon \to \Upsilon + A \to \Upsilon + A + A^2 \to \Upsilon + A + A^2 + A^3 \ldots$$

Es decir, tenemos como resultado

$$\coprod_{n\in N} A^n,$$

siendo $A^0 = \Upsilon$. *El mayor punto fijo de* F_A *es el límite de la cadena:*

$$\Upsilon \leftarrow \Upsilon + A \times \Upsilon \leftarrow \Upsilon + A \times (\Upsilon + A \times \Upsilon) \leftarrow \Upsilon + A \times (\Upsilon + A \times (\Upsilon + A \times \Upsilon)) \leftarrow$$

$$\leftarrow \Upsilon + A \times \Upsilon + A \times (\Upsilon + A \times (\Upsilon + A \times \Upsilon)) \leftarrow \ldots$$

es decir,

$$\Upsilon \leftarrow \Upsilon + A \leftarrow \Upsilon + A + A^2 \leftarrow \Upsilon + A + A^2 + A^3 \leftarrow \Upsilon + A + A^2 + A^3 + A^4 \ldots$$

En este caso, el resultado es

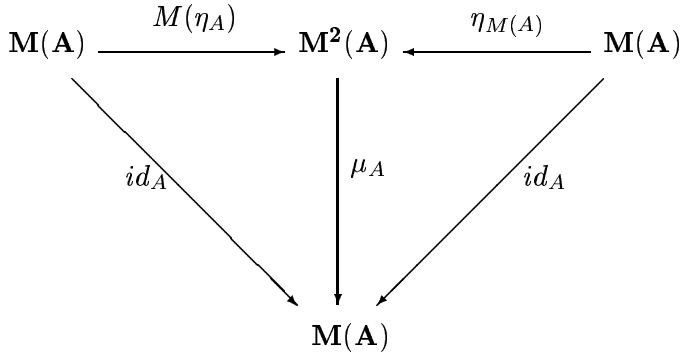
$$\coprod_{n\in N} A^n + A^\infty,$$

siendo A^∞ *la sucesión no finita de elementos de* A *(comprobando, de paso, la sustancial diferencia que hay entre el límite y el colímite de una cadena). Pero, tal como ya hemos mencionado anteriormente, relajando el contexto y adoptando una semántica no estricta, podemos considerar los dos soportes idénticos.*

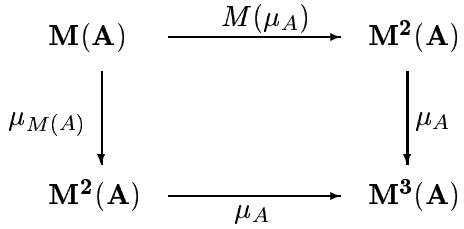
2.3.4 Mónadas

Concluimos este capítulo de definiciones con una noción que, en los últimos tiempos, ha alcanzado gran notoriedad en el ámbito computacional por sus múltiples aplicaciones en la programación funcional.

Definición 2.15. *Una* mónada *sobre un categoría* $\mathbf{C}$ *consta de un endofuntor* $M : \mathbf{C} \to \mathbf{C}$ *y dos transformaciones naturales (la unidad)* $\eta : 1_{\mathbf{C}} \Rightarrow M$, *(la asociatividad)* $\mu : M^2 \Rightarrow M$, *verificando las conmutatividades de los siguientes diagramas:*



o sea, $M(\eta_A); \mu_A = 1_{MA}$; *y* $\eta_{MA}; \mu_A = 1_{MA}$;



es decir, $M(\mu_A);\mu_A = \mu_{MA};\mu_A$.

Ejemplo 2.15. *Un uso de las mónadas y sus teorías en los lenguajes funcionales es la introducción de conceptos imperativos [Moggi 1989], [Wadler 1987], [Wadler 1993], [Wadler 1994], [Kieburtz y Lewis 1995]. Incluso han sido usadas como modelos computacionales en semántica declarativa [Moggi 1989].*

Así, la computación monádica es útil en la caracterización de los estados. Si S es un tipo fijo, la mónada de los estados de tipo S viene determinada por el funtor St definido sobre los objetos por $St(A) = S \to A \times S$ *y, siendo* $f : A \to B$ *un morfismo,* $St(f)(K) = \lambda s.(f(a), s')$, *donde* $(a, s') = K(s)$. *La unidad es la función* $\eta_A(x) = \lambda s.(x, s)$; *la asociatividad viene dada por* $\mu_A(\bar{o})=\lambda s.(x, s'')$, *siendo* $(x, s'') =\underline{o}(s')$ *y* $(\underline{o},s')=\bar{o}(s)$.

Un concepto más próximo al lenguaje de la computación y que deviene en el anterior es el de triple de Kleisli.

Definición 2.16. *Un* triple de Kleisli *es una terna* $(T, \eta, -^*)$, *donde T es un endomorfismo sobre los objetos de* $|\mathbf{C}|$, η_A *es un morfismo entre A y* $T(A)$ *y, para cada* $f : A \to T(B)$, $f^* : T(A) \to T(B)$ *verificando las igualdades:*

$$(\eta_A)^* = id_{T(A)};$$

$$\eta_A; f^* = f;$$

$$f^*; g^* = (f; g^*)^*.$$

$A\ -^*$ *se le denomina el* operador de Kleisli *asociado al triple.*

Entonces, cada triple de Kleisli, $(T, \eta, -^*)$, se corresponde con una mónada (T, η, μ) donde $T(f : A \to B) = (f; \eta_B)^*$ y $\mu_A : T^2(A) \to T(A)$ es $(id_{T(A)})^*$.

Ejemplo 2.16. *Sea* $P^0 : |\mathbf{Set}| \to |\mathbf{Set}|$ *la función que a cada conjunto A le asocia el conjunto formado por sus subconjuntos. Definimos* $\eta_A : A \to P^0(A)$ *como* $\eta_A(x) =\{x\}$, $x \in A$. *Dado un morfismo* $f : A \to P^0(B)$ *y un subconjunto* $X \subset A$, $f^*(X) =\{f(x)|x \in X\}$. $(P^0, \eta, -^*)$ *es un triple de Kleisli y, si* $XX \in (P^0)^2(A)$, *definimos* $\mu_A(XX) = (id_{T(A)})^*(XX) =\{id_{T(A)}(X)|X \in XX\} =\{X|X \in XX\}$, (P^0, η, μ) *es una mónada.*

Definición 2.17. *Dada la mónada* (M, η, μ) *en* $\mathbf{C}$, *definimos la categoría de Kleisli,* $\mathbf{C_M}$, *como aquélla que tiene por objetos los de* $\mathbf{C}$ *y un morfismo* $f : A \hookrightarrow B$ *en* $\mathbf{C_M}$ *es el morfismo* $f : A \to M\ B$ *en* $\mathbf{C}$. *La composición en* $\mathbf{C_M}$ *de los morfismos* $f : A \hookrightarrow B$ *y* $g : B \hookrightarrow C$, $f \circ g$, *viene dada por* $f; Mg; \mu_C$ *en* $\mathbf{C}$. *Además, para cada objeto* $A \in \mathbf{C}$, *el morfismo de* $\mathbf{C}$ η_A *es la identidad en* $\mathbf{C_M}$.

Ejemplo 2.17. *Sea (P^0, η, μ) la mónada del ejemplo 2.2.11. Un morfismo en la categoría de Kleisli* $\mathbf{Set}_{P^0}$, $f : A \hookrightarrow B$, *es una función $f : A \to (P^0(B))$ en* **Set** *que asigna a cada elemento de A un subconjunto de B; la composición de dos morfismos $f : A \hookrightarrow B$, $g : B \hookrightarrow C$ viene dada por:*

$$(f; g)(a) = \cup_{b \in f(a)} g(b).$$

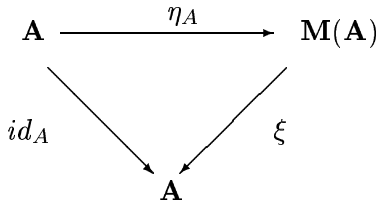
Entonces, $\mathbf{Set}_{P^0}$ *es la categoría* **ND** *de conjuntos y aplicaciones no deterministas [Moggi 1989].*

Definición 2.18. *Dado el triple $\mathbf{M} = (M, \eta, \mu)$, una* **M**-*álgebra es un par (A, ξ), con $A \in |\mathbf{C}|$ y $\xi : M(A) \to A$ satisfaciendo las igualdades*

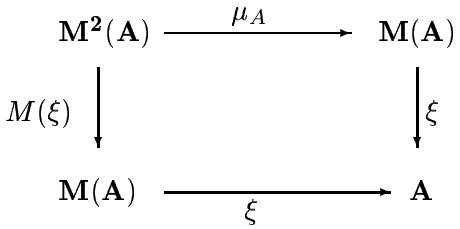
$$\eta_A; \xi = id_A \text{ y}$$

$$\mu_A; \xi = M(\xi); \xi$$

correspondientes a la conmutatividad de los diagramas



y



Las **M**-*álgebra forman una categoría denominada de Eilenberg-Moore, denotada por* $\mathbf{C}^M$.

Ejemplo 2.18. *Consideremos el funtor List que asigna a cada tipo $A \in |\mathbf{T}|$ el objeto cuyos elementos son las listas de tipo A; entonces, considerando como la unidad la función $\eta_A(x) = [x]$, $x \in A$, la lista cuyo único elemento es x, y como asociatividad $\mu_A(\bar{o})$=$flatten(\bar{o})$, donde $flatten$ es la función que aplana listas, es decir, dada una lista de listas, la convierte en una única lista formada por los elementos de todas las sublistas en el orden original,* $\mathbf{List} = (List, \eta, \mu)$ *es una mónada. Si (A, ξ) es una* **List**-*álgebra, debe verificarse que*

$$\xi[x] = x, \ x \in A;$$

además, dada la lista $[[x; y]; [z]]$, debe ocurrir que

$$\xi(flatten[[x; y]; [z]]) = \xi([\xi[x; y]; \xi[z]]);$$

aplicando el valor de $flatten$

$$\xi([x;y;z]]) = \xi([\xi[x;y];z]).$$

Pero, como también ocurre que $(flatten[x;[y;z]]) = [x;y;z]$, *tiene que verificarse que*

$$\xi([x;y;z]]) = \xi([x;\xi[y;z]]),$$

y, por consiguiente,

$$\xi([\xi[x;y];z]) = \xi([x;\xi[y;z]]),$$

lo cual queire indicar que ξ *es un operador asociativo. Por otra parte, se verifican las dos siguientes igualdades:*

$$\xi(flatten[[x];[\,]]) = \xi[x];$$

$$\xi(flatten[[x];[\,]]) = \xi[\xi[x];\xi[\,]].$$

Reuniéndolas, tenemos que $\xi[\xi[x];\xi[\,]] = \xi[x]$, *de donde concluimos que* $\xi[\,]$ *es el elemento neutro para la operación* ξ. *Así pues, la categoría de Eilenberg-Moore ligada al funtor List es una subcategoría de la de los monoides.*

Dualmente se define el concepto de *comónada.*

Como se menciona en [Lewis y otros 2000], mientras las mónadas permiten modelar el efecto de la ejecución de una computación, estando relacionadas, en tal caso, con las salidas, las comónadas modelan la estructura de los entornos y, entonces, están relacionadas con las entradas.

Estudiaremos en un capítulo posterior las álgebras ligadas a las mónadas definidas a lo largo del texto.

2.3.5 Una breve descripción de Coq

El Cálculo de Construcciones Inductivas (CIC) es una teoría de tipos que resulta de la combinación de la teoría intuicionista de tipos de Martin-Löf y el λ–cálculo polimórfico de Girard, F_ω. Los teoremas a probar se representan como tipos, y las pruebas son términos con ese tipo. Esta forma de ver las cosas es consecuencia del isomorfismo de Curry-Howard [Howard 1980]. Se fundamenta éste en la analogía "fórmulas como tipos"; esta corresponencia ha sido la principal herramienta para la correcta interpretación de la relación entre la lógica intuicionista y el λ-cálculo tipado. La idea básica del isomorfismo de Curry-Howard consiste en pensar que las fórmulas lógicas pueden ser interpretadas como tipos en una teoría de tipos adecuada; entonces, una prueba de una fórmula está asociada con un λ-término del tipo correspondiente y la reducción de un prueba por corte-eliminación se corresponde con la normalización del λ-término asociado. Así, si una fórmula es derivable en un cierto sistema lógico el tipo correspondiente contiene términos en la teoría de tipos asociada.

Hay dos tipos básicos en Coq: "Set"para la definición de objetos, y "Prop"que permite la definición de predicados y relaciones concernientes a estos objetos; igualmente, posibilita establecer proposiciones susceptibles de demostración acerca de esos objetos.

La notación `a:A` será leída como `a` es de tipo `A` e interpretada como sigue: `a` es una prueba de .ª", cuando `A` es de tipo "Prop.º `a` es un elemento de `A`, cuando `A` es de tipo "Set".

Las construcciones permitidas son `x|(M N)|[x:T]M|(x:T)P` .
En la primera `x` denota tanto a las variables como a las constantes; la segunda es la aplicación. La tercera representa el programa (λ–expresión) de parámetro `x` y cuerpo `M` (la sustitución de la variable `x` de tipo `T` en `M`). Finalmente, la cuarta es el tipo de los programas que admiten una entrada de tipo `T` y devuelven un resultado de tipo `P`. Este tipo se denomina *tipo producto* y, en teoría de tipos, se representa $\Pi x : T.P$ o también $\forall x : T.P$. Si `x` no figura libre en `P` entonces escribiremos simplemente `T->P`, el tipo de las funciones entre esos dos tipos.

A la hora de probar resultados en Coq debemos tener en cuenta si el tipo sobre el que hacemos alguna afirmación es inductivo o coinductivo. Dependiendo de la naturaleza del problema se utilizan diferentes estilos de razonamiento, tales como el razonamiento ecuacional, la inducción o los lemas de aproximación [Gibbons 2000] para los tipos inductivos o la bisimilaridad para los coinductivos.

Dos expresiones son *bisimilares* si sus árboles de transición son idénticos cuando ignoramos las expresiones en los nodos y sólo consideramos las observaciones que etiquetan las aristas [Gordon 1994]. Dada una relación de transición etiquetada, una bisimulación es una relación R sobre las expresiones tal que si $a\ R\ b$ entonces:

$$cuando\ a \xrightarrow{o} a'\ existe\ b'\ con\ b \xrightarrow{o} b'\ y\ a'\ R\ b'\ y$$

$$cuando\ b \xrightarrow{o} b'\ existe\ a'\ con\ a \xrightarrow{o} a'\ y\ a'\ R\ b'.$$

La mayor bisimulación bajo el orden de inclusión sobre las relaciones es lo que denominamos bisimilaridad, denotada por $\sim$.

Entonces, el método básico para probar que dos expresiones son bisimilares es la coinducción, dada por la siguiente equivalencia:

$$a \sim b \Leftrightarrow \exists R.R\ es\ una\ bisimulacion\ y\ a\ R\ b.$$

3 Programación Algebraica

3.1 *Introducción*

Hagino en [Hagino 1987] presenta los tipos de datos categóricos a partir de adjunciones. Así, los tipos producto o exponencial son ejemplos de tipos adjuntos a la derecha mientras que el coproducto o las listas son tipos adjuntos a la izquierda. Esta caracterización permite determinar tanto los tipos como los morfismos que de ellos dependen.

Ejemplo 3.1. *Supongamos que hemos definidos los tipos (finales)* `1` *y* `prod` *representando, respectivamente, el objeto terminal de la categoría y el producto en ella. Que el tipo exponencial sea* `final` *implica (ver Definición 2.6) la existencia de* $curry(f) : C \to B^A$ *para cada* $f : A \times C \to B$*. Además, para su correcta especificación necesitamos la función* $eval : A \times B^A \to B$*. La definición en el lenguaje categórico de Hagino es ([Hagino 1987], pág. 81)*

```
right object exp(X,Y) with curry is
eval:prod(exp,X)->Y
end object
```

Los argumentos de esta declaración son claramente interpretables:

1. *un nuevo tipo,* `exp(X,Y)`*, el cual, al ser declarado* `right` *es terminal;*
2. *un morfismo que determina esa condición final,* `curry`*;*
3. *un morfismo, eval, que relaciona el tipo recién declarado con sus constituyentes.*

Análogamente se define un objeto inicial como, por ejemplo, las listas:

```
left object list(X) with prl is
nil:1->list
cons:prod(X,list)->list
end object
```

La función *prl* desempeña el rol de la recursión primitiva sobre las listas: es decir, dadas $f : 1 \to Y$, $g : X \times Y \to Y$, $prl(f,g) : list\ X \to Y$ satisface las dos leyes siguientes:
$prl(f,g)\ nil\ =\ f$;
$prl(f,g)\ cons(x,xs)\ =\ g(x, prl(f,g)\ xs)$.

Mientras *eval* se denomina *destructor* (informa de cómo descomponer un elemento del objeto adjunto a la derecha), *nil* y *cons* se denominan *constructores* del tipo pues indican cómo se generan los elementos del objeto adjunto a la izquierda.

Mas, incluso dentro de cada una de estas clases de tipos (left, right) hay diferencias sustanciales: por ejemplo, el tipo producto (tal como hasta ahora está definido) satisface una relación final inequívoca sin posibilidad alguna de establecer una condición de inicialidad. En cambio, las listas, que satisfacen condiciones iniciales (como hemos visto en el Ejemplo 2.14) permiten, dada su construcción, condiciones de finalidad [Erwig 1998], [Gibbons 2000] (son, en este contexto, bien conocidos los anamorfismos de listas [Meijer y otros 1991]).

Para analizar estas construcciones Hagino introduce la noción de **diálgebra**. Durante los últimos años se ha introducido ligeramente el uso de esta estructura [Fokkinga 1996], [Erwig 1998], [Erwig 2000], [Erwig 2000(2)] y es ahora cuando comienza a estudiársela por sus particulares características como generalización de las álgebras y de las coálgebras [Backhouse y Hoogendijk 1999], [Poll y Zwanenburg 2001]
aunque todavía de una forma poco sistemática. Nuestra intención es avanzar en ambas direcciones: estudiándola como concepto que permite enriquecer la representación de los tipos y viendo, como objeto con identidad propia, qué optimizaciones se pueden llevar a cabo con las funciones que de ella dependen. Presentamos, pues, la aproximación dialgebraica que nos permitirá trabajar con la citada dualidad y obtener tipos más expresivos y con mayor capacidad operativa con su entorno. Desde esta perspectiva se puede, incluso, rebasar la restricción mencionada acerca del carácter exclusivamente final del producto (o el inicial del coproducto).

Como casos paradigmáticos analizamos con detalle el tipo de las listas y el tipo de los árboles de aridad variable, comprobando sobre ellos las ventajas que proveen las funciones de orden superior. Además, vemos cómo la versatilidad de Coq permite definir las condiciones de finalidad de los cotipos. Analizamos los casos de los naturales extendidos con el infinito, las *streams* y el tipo exponencial.

Por otra parte, hay otros tipos que sólo en los últimos tiempos han concitado la atención de los investigadores, denominados no regulares o anidados (*nested datatypes* en la bibliografía inglesa) [Bird y Meertens 1998], [Hinze 2000], [Bird y Paterson 1999], [Bird y Paterson 1999 (2)], [Martin y Gibbons 2001], [Martin, Gibbons y Bayley 2001]. La capacidad expresiva de las diálgebras permite obtener respuestas a problemas suscitados acerca de la expresividad de los catamorfismos, anamorfismos e hilomorfismos derivados de ellos.

Por último, analizamos el operador tail-recursive foldl: al contrario que el funcional foldr, sistematizado como el único morfismo proveniente del tipo inicial, la función foldl no está definida de forma genérica sobre todos los tipos. De hecho, en los últimos tiempos se intenta en varios proyectos de investigación obtener la forma global de este morfismo. Analizamos una posible forma de afrontar esa definición.

3.2 *Tipos y cotipos*

En esta sección se revisan algunos conceptos relativos a los (co)tipos listas, exponencial, *streams* y árboles de aridad variable en cuyo estudio dialgebraico profundizaremos más adelante.

3.3 *Tipos iniciales: listas y listas snoc*

3.3.1 Listas

Consideramos en primer lugar las listas observando tres características:

1. su carácter de punto fijo de un endofuntor;
2. la actuación sobre los morfismos del funtor que genera el tipo de las listas;
3. el funcional que caracteriza su condición de inicial dentro de una determinada álgebra (el catamorfismo de las listas).

Utilizando el lenguaje funcional CAML, declaramos el tipo de las listas polimórficas (aunque, al ser un tipo interno del sistema, ya tiene determinados sus constructores, `[]` para la lista vacía y `::` para pegar (por la izquierda) un elemento a una lista):

```
type 'a list =
Nil
| Cons of 'a*('a list);;
```

Una versión más adaptada al contexto categórico que propugnamos es la siguiente:

```
type ('a,'b) Flist =
FNil
| FCons of 'a*'b;;
type 'a list =
In of ('a,'a list) Flist;;
```

donde `Flist` representa un bifuntor (en este caso, el que asigna a los objetos A, B de una categoría C el objeto $1 + A \times B$) e `In` es el isomorfismo entre `'a list` y `('a,'a list) Flist` indicando su carácter de punto fijo. Las funciones que nos permiten `intercambiar` elementos entre estos dos tipos son

```
let Input = function
FNil->(In FNil)
|(FCons(a,l))->(In (FCons(a,l)));;

let Output = function
(In FNil)->FNil
|(In (FCons(a,l)))->(FCons(a,l));;
```

obviamente isomorfa una de la otra.

Completar el carácter funtorial de `Flist` implica conocer cómo actúa sobre los morfismos:

```
let map_Flist g h = function FNil->FNil
|(FCons(x,y))->FCons(g x,h y);;
```

A partir de esta función podemos construir la que caracteriza a `list` como funtor:

```
let rec map_list g = function x ->
(Input ((map_Flist g (map_list g)) (Output x)));;
```

Veamos una cierta característica estructural común a muchas funciones definidas *sobre* las listas. Observemos, por ejemplo, lo similares que son las construcciones de la función que calcula la longitud de una lista y las que efectán la suma y el producto de sus elementos (cuando estos son numéricos, por supuesto):

```
let rec length = function
        (In FNil) -> 0
                |(In (FCons(a,l))) -> 1+(length l);;

let rec sum = function
        (In FNil) -> 0
                |(In (FCons(a,l))) -> a+(sum l);;

let rec producto = function
        (In FNil) -> 1
                |(In (FCons(a,l))) -> a*(producto l);;
```

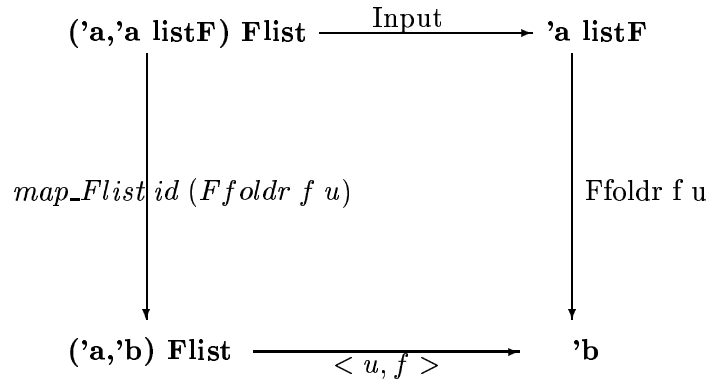
Esa similitud queda reflejada en la función

```
let rec foldr g u = function
        (In FNil) -> u
             | (In (FCons(a,l))) -> g a (foldr g u l);;
```

de forma que `length = foldr` λ `xy.1+y 0, sum = foldr` λ `xy.x+y 0`,
`producto = foldr` λ `xy.x*y 1`.

Esta función `foldr` se corresponde con *prl*, la función declarada en el lenguaje categórico de Hagino relacionada con la recursión primitiva sobre las listas. Representa el carácter inicial de éstas y, como ya quedó indicado anteriormente, se denomina *catamorfismo*.

El isomorfismo `Input` es la función correspondiente a un álgebra sobre el funtor `'a Flist`; entonces, dada otra `'a Flist`-álgebra, `('b,<u,f>)` (con `u` elemento de `'b` y `f : 'a -> 'b -> 'b`) existe un único morfismo entre `'a list` y `'b`, `Ffoldr f u`, verificando
`Input;(Ffoldr f u)=(map_Flist id (Ffoldr f u));<u,f>:`



Esta igualdad queda constatada en la siguiente definición:

```
let rec Ffoldr f u = function x ->
(app f u (map_Flist id (foldr2 f u) (Output x)))
where
 app f u = function
FNil->u
|FCons(a,b)->(f a b);;
```

donde `app f u` representa el par `<u,f>`.

Un caso particular surge cuando `a`=nat, `b`=nat^n y `u` y `f` son funciones primitivo recursivas (`u`, en este caso, siempre lo es al ser constante). Entonces, `foldr f u` es una función primitivo recursiva sobre las listas de naturales. Por ejemplo, si

```
let rec exponential x = function
0->1
|n->x*(exponential x (n-1));;

let exp_list = (foldr exp 1)
where
exp = function x -> function y ->(exponential x y);;
```

resulta que `exp_list`$[a;b;c;d] = a^{b^{c^d}}$.

3.3.2 Listas snoc

Cuando en la definición del tipo `'a list` cambiamos la estructura del constructor `Cons`, obtenemos las denominadas en la bibliografía listas `snoc`:

```
 type 'a snoc_list=
nil
|snoc of ('a snoc_list* 'a);;
```

Así, al igual que en las listas convencionales añadimos elementos del tipo correspondiente por la izquierda, en las listas snoc ese pegado se hace por la derecha.

Podemos convertir listas convencionales en listas snoc y viceversa:

```
 let rec conv1 = function []->nil
|(x::xs)->(snoc(conv1 l,x));;

 let rec conv2=function  nil->[]
|(snoc(x,xs))->x::(conv2 xs);;
```

La definición de la función `conv2` carece de interés en cuanto a un análisis más profundo; la de `conv1`, en cambio, es más interesante y, en una próxima sección veremos su significación categórica. La traslación de las funciones sobre listas a funciones sobre listas snoc es inmediata, de forma que, a cada morfismo sobre listas f, le asociaremos un morfismo sobre listas snoc, `traslacion f`.

```
let traslacion f s = f(conv2 s);;
```

Así, por ejemplo, la función `map_snoc_list` puede definirse como

```
let map_snoc_list f = conv1(traslacion (map f));;
```

Estas funciones trasladadas adolecen de eficiencia debido a la generación de listas intermedias; por ello, surgen dos necesidades:

1. proveer estas funciones de forma automática ligándolas a la definición del tipo;
2. obtener mecanismos de optimización para las funciones que permitan minorar el efecto de esas construcciones intermedias.

Así, es deseable obtener una función `snoc_map` basada en sus constructores:

```
 let rec snoc_map f=function nil->nil
|(snoc(xs,x))->snoc(snoc_map f xs,f x);;
```

Podemos, también, definir el funcional equivalente a `foldr`:

```
 let rec snoc_foldr f u=function nil->u
|(snoc(xs,x))->f(snoc_foldr f u xs,x);;
```

3.4 *Tipos finales: exponencial,* streams, *naturales con infinito, árboles de aridad variable*

3.4.1 Tipo exponencial

Hemos mencionado al principio del capítulo cómo define Hagino el tipo exponencial a partir de la adjunción con el producto. Podemos establecer esa condición en Coq por medio de la siguiente declaración:

```
CoInductive exp [A,B:Set]:Set :=
curry : (C:Set) (A*C->B) -> C -> (exp A B).

Definition eval:= [A,B:Set] [x:A*(exp A B)]
        Cases (Snd x) of
                (curry C f c) => (f(Fst x,c))
        end.

Definition uncurry := [A,B,C:Set] [f:C->(exp A B)] [x:A*C]
(eval A B (Fst x,f(Snd x))).
```

Se verifican las dos propiedades siguientes concernientes a la relación categórica entre las funciones `curry` y `uncurry` (donde hacemos explícito el *surjective pairing*):

```
(* Axioma del surjective pairing *)

Axiom previo: (A,B:Set) (x:A*B) x=(Fst x,Snd x).
Hint previo.

Lemma exp_uno : (A,B,C:Set) (f:A*C->B)
{f_1:C->(exp A B)|(x:A*C)(eval A B (Fst x,f_1(Snd x)))=(f x)}.
Intros.
Exists (curry A B C f).
Unfold eval.
Simpl.
Replace ((Fst x),(Snd x)) with x.
Trivial.
Trivial.
Qed.

Lemma exp_dos : (A,B,C:Set) (f:C->(exp A B))
{f_1:A*C->B|(x:A*C)(eval A B (Fst x,f(Snd x)))=(f_1 x)}.
Intros.
Exists (uncurry A B C f).
Intros.
Trivial.
Qed.
```

La inclusión del conjunto de las funciones entre `A` y `B` en `exp A B` es sencilla:

```
Inductive Unit: Set := unit:Unit.

Definition inclusion := [A,B:Set] [f:A->B]
(curry A B Unit [x:(A*Unit)](f (Fst x)) unit).
```

siendo `unit` el único elemento del tipo `Unit`.

Notemos, además, que cada valor de `exp A B` determina una función:

```
Definition instanciacion := [A,B:Set] [x:(exp A B)] [a:A]
        Cases x of
                (curry C f c) => (f(a,c))
        end.
```

Obviamente, la representación de una función como un elemento de `exp A B` no tiene (y de hecho no lo es) por qué ser única. Para que sí lo fuera debería definirse una relación indicando cuando dos valores de `exp A B` representan a una misma función:

```
Hypothesis relacion : (A,B:Set) (x,y:(exp A B)) (a:A)
((eval A B (a,x))=(eval A B (a,y)))->(x=y).
Hint relacion.

Lemma eval_uno : (A,B:Set) (f:A->B) (a:A)
(instanciacion A B (inclusion A B f) a)=(f a).
Intros.
Trivial.
Qed.

Lemma eval_dos : (A,B:Set) (x:(exp A B)) (a:A)
(eval A B (a,(inclusion A B (instanciacion A B x))))=
(eval A B (a,x)).
Intros.
Trivial.
Qed.

Lemma eval_tres : (A,B:Set) (x:(exp A B))
(inclusion A B (instanciacion A B x))= x.
EAuto.
Qed.
```

La función denotando el carácter funtorial del tipo es la siguiente donde vemos que el funtor es contravariante en la primera variable:

```
Definition map_exp := [A,B,C,D:Set] [f:C->A] [g:B->D]
[x:(exp A B)]
   Cases x of
        (curry X h x0) =>
        (curry C D X [y:C*X](g(h(f(Fst y),Snd y))) x0)
   end.
```

3.4.2 *Streams*

Las *streams* proveen de un ejemplo singular dentro de este espectro de tipos iniciales y finales. Su definición proviene del endofuntor continuo $FST(A,B) = A \times B$, del cual es una coálgebra final. Al ser el endofuntor continuo, es posible hallar el mayor punto fijo de la cadena que termina en el objeto final de la categoría, () :

$$() \leftarrow A \times () \leftarrow A \times (A \times ()) \leftarrow A \times (A \times (A \times ())) \leftarrow \ldots$$

Dada la equivalencia $A \simeq A \times ()$, tenemos que

$$Stream\ A = \prod_{n \in N} A.$$

El isomorfismo es

$$A \times \prod_{n \in N} A \simeq \prod_{n \in N} A.$$

A partir de él es posible definir el tipo (escrito en OCaml)

```
type 'a stream = Inf of 'a*('a stream);;
```

Viene acompañado de dos destructores: dado un elemento de tipo `'a stream`, `Inf(x,xs)`, `hd (Inf(x,xs))=x`$\in$`'a`, `tl (Inf(x,xs))=xs` $\in$`'a stream`.
Definimos las funciones homólogas a las construidas sobre los otros tipos:

```
let rec map_stream f =
function Inf(x,xs) -> Inf(f x,map_stream f xs);;

let rec unfold theta b=

let (a,b')=(theta b) in Inf(a,unfold theta b');;
```

Mientras que el cotipo exponencial ha sido definido por medio de su adjunción con el producto, es útil desde el punto de vista de la abstracción definir el cotipo de las *streams* por medio de su caracterización como coálgebra colibre. Como se menciona al principio de la sección, las *streams* son la coálgebra final del endofuntor $FST(A,B) = A \times B$. Esto, trasladado al lenguaje de las construcciones colibres, quiere decir que, dada una coálgebra cualquiera (A, B, θ), con $\theta : B \to A \times B$, y dado un morfismo $f : A \to C$ existe un único morfismo, $ana\ \theta\ f : B \to stream\ C$, verificando que, para cada $b \in B$,

$$hd\ (ana\ \theta\ f\ b)\ = f\ a;$$

$$tl\ (ana\ \theta\ f\ b)\ = ana\ \theta\ f\ b'$$

donde $(a, b') = \theta\ b$. Se puede expresar esta condición, usando de nuevo Coq, de la siguiente forma:

```
Section stream.

CoInductive Stream [C:Set]:Set :=
S:(A,B:Set)(B->A)->(B->B)->(A->C)->B->(Stream C).

Variable A,B,C:Set.

Variable h1:B->A.
```

```
Variable h2:B->B.

Variable f:A->C.

Variable b:B.

Definition hd := [x:(Stream C)]
        Cases x of
                (S A B h1 h2 f b) => (f (h1 b))
        end.

Definition tl := [x:(Stream C)]
        Cases x of
         (S A B h1 h2 f b) => (S C A B h1 h2 f (h2 b))
        end.

End stream.
```

Esto significa que, para definir un elemento de tipo `Stream C` precisamos dos conjuntos `A, B` y funciones $h1 : B \to A$, $h2 : B \to B$ que conforman la anterior función θ, un morfismo $f : A \to C$ y, por supuesto, el elemento catalizador $b \in B$. Hemos definido en este ámbito las funciones `hd, tl`, los destructores del tipo. Puesto que estamos habituados a trabajar con constructores más que con destructores nos interesa saber quién es `Inf:C*(Stream C) -> (Stream C)`. Dada la definición de los objetos del tipo, debe ser la función que al par `(x,xs)` le asigna el objeto `S C C*(Stream C) id_C (y,ls)->(hd ls,tl ls)`.
En efecto, así definida se verifica la condición natural de los morfismos `hd, tl` como vemos en el siguiente enunciado (introducimos como axioma, nuevamente, el surjective–pairing):

```
Definition aux1 := [C:Set] [x:C*(Stream C)](Fst x).

Definition aux2 := [C:Set] [x:C*(Stream C)]
(hd C (snd C (Stream C) x),tl C (snd C (Stream C) x)).

Definition aux3 := [C:Set] [x:C]x.

Definition Inf := [C:Set] [y:C*(Stream C)]
(S C C C*(Stream C) (aux1 C) (aux2 C) (aux3 C) y).

Axiom surjective_pairing :(C:Set)(x:(Stream C))
(Inf C (hd C x,tl C x))=x.

Lemma inf1 : (C:Set)(y:C*(Stream C))
(hd C (Inf C y))=(fst C (Stream C) y).
Intros.
```

```
Simpl.
Trivial.
Qed.

Lemma inf2 : (C:Set)(y:C*(Stream C))
(tl C (Inf C y))=(snd C (Stream C) y).
Intros.
Simpl.
Replace (aux2 C y) with
(hd C (snd C (Stream C) y),tl C (snd C (Stream C) y)).
Replace (S C C C*(Stream C) (aux1 C) (aux2 C) (aux3 C)
  ((hd C (Snd y)),(tl C (Snd y)))) with
  (Inf C ((hd C (Snd y)),(tl C (Snd y)))).
Apply surjective_pairing.
Trivial.
Trivial.
Qed.
```

De hecho, podemos definir las listas, finitas e infinitas, de este modo:

```
Inductive ListI [C:Set]:Set :=
S:(A,B:Set)(B->bool)->(B->A)->(B->B)->(A->C)->B->(ListI C).

Variable A,B,C:Set.

Variable h1:B->A.

Variable h2:B->B.

Variable f:A->C.

Variable b:B.

Definition is_nil :=[x:(ListI C)]
Cases x of
        (S A B p h1 h2 f b) =>(p b)
end.
```

En contextos con semánticas no estrictas ambos tipos, *ListI* y *List*, son isomorfos [Gibbons 2000].

3.4.3 Naturales con infinito

Esta posibilidad de definir los cotipos a partir de su condición de coálgebras finales nos permite definir el cotipo de los naturales extendidos con un símbolo de infinitud de la siguiente forma:

```
Section natI.

CoInductive natI : Set :=
L:(B:Set)(B->bool)->(B->B)->B->natI.

Variable B:Set.

Variable p:B->bool.
Variable h:B->B.
Variable b:B.

Definition tPb := [p:Prop]
Cases p of
        True => true
        |_   => false
end.

Definition iszero := [x:natI]
Cases x of
        (L B p f b)=>(p b)
end.

Definition pred := [x:natI]
Cases x of
        (L B p f b) =>(L B p f (f b))
end.

Definition is_infty := [x:natI]
(trPb ((pred x)=x)).

 End natI.
```

Entonces, aunque en Coq no se derivan automáticamente las funciones que determinan la finalidad de un tipo coinductivo (cotipo) éstas pueden establecerse al definir el cotipo en función de su condición final.

3.4.4 Árboles

Se introducen a continuación árboles con nodos de aridad variable (0, 1 o más ramas) [Cousineau y Mauny 1995] (notemos que no admitimos árboles vacíos). Cada una de las ramas de un nodo dado se representa por medio de una lista de nodos. Una hoja es representada como un nodo con una lista vacía de ramas. Para ello, se usa el constructor `Nodo`.

```
CoInductive arbol [A:Set] : Set :=
Nodo : A -> (Lista (arbol A)) -> (arbol A).
```

Se puede, tal como en los casos precedentes, recurrir a su condición de coálgebra terminal para definirlo. Ya que el tipo de los árboles depende del de las listas, es necesario contar con una estructura abstracta de este tipo, *coeps* : $B \to (\ ) + A \times B$; [4] además, se precisa un morfismo *codelta* : $A \to C \times B$ que represente, en abstracto, la *destrucción* de un elemento de tipo `arbol A` en sus componentes de tipo `A` y `Lista (arbol A)`. Entonces, a partir de un morfismo $f : C \to D$ y un elemento $a \in A$ se construye un objeto del tipo `arbol A`: [5]

```
Section Arbol.

CoInductive Lista [A:Set] : Set :=
Nil : (Lista A)
|Cons: A -> (Lista A) -> (Lista A).

Inductive ProdUnit [A,B:Set] : Set :=
unit:(ProdUnit A B)
|P:A*B->(ProdUnit A B).

Variable A,B,C,D:Set.
Variable coeps:B->(ProdUnit A B).
Variable f:A->C.

CoFixpoint analist : B -> (Lista C) :=
[b:B]Cases (coeps b) of
unit => (Nil C)
|(P(a,b')) => (Cons C (f a) (analist b'))
end.
End Arbol.

CoInductive arbol [D:Set] : Set :=
ND: (A,B,C:Set)(B->(ProdUnit A B))->(A->C*B)->(C->D)->
A->(arbol D).

Definition hd := [D:Set] [x:(arbolAV D)]
Cases x of
(ND A B C coeps codelta f a) => (f(Fst(codelta a)))
end.

Definition tl := [D:Set] [x:(arbolAV D)]
Cases x of
        (ND A B C coeps codelta f a) =>
        (analist A B (arbolAV D) coeps
        (ND D A B C coeps codelta f)
```

[4] Donde ya avanzamos parte de la fundamentación de este capítulo pues introducimos el tipo de las listas no en su condición de objeto inicial sino como objeto final

[5] Usamos un tipo auxiliar `ProdUnit A B` para representar la estructura $(\ ) + A \times B$

```
        (Snd(codelta a)))
end.
```

El morfismo `NodoInv` es el destructor concreto del cotipo `arbol`. Al igual que hicimos con el tipo de las *streams* podemos definir el constructor asociado a este tipo:

```
Definition aux1 := [D:Set] [x:(Lista (D*(Lista (arbol D))))]
Cases x of
        Nil =>          (unit D*(Lista (arbol D))
                        (Lista (D*(Lista (arbol D)))))
        |(Cons x l) =>  (P D*(Lista (arbol D))
                        (Lista (D*(Lista (arbol D))))
                        ((fst D (Lista (arbol D)) x,
                        snd D (Lista (arbol D)) x),l))
end.

Definition aux2 := [D:Set] [x:D*(Lista (arbol D))]
(fst D (Lista (arbol D)) x,
Cons D*(Lista (arbol D)) x (Nil D*(Lista (arbol D)))).

Definition aux3 := [D:Set] [x:D]x.

Definition Node := [D:Set] [x:D*(Lista (arbol D))]
(ND D D*(Lista (arbol D)) (Lista (D*(Lista (arbol D))))
D (aux1 D) (aux2 D) (aux3 D) x).


Lemma uno : (D:Set) (x:D*(Lista (arbol D)))
(hd D (Node D x))=(fst D (Lista (arbol D)) x).

Intros.
Simpl.
Trivial.
Qed.
```

3.5 *Descripción de las diálgebras*

Según la terminología de Malcolm, dado un bifuntor $F : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ un tipo regular es un funtor $D : \mathbf{T} \to \mathbf{T}$ que verifica, para cualquier objeto $A \in |\mathbf{T}|$, $D(A) \simeq F(A, D(A))$. El tipo de datos, D, consta, para cada objeto $A \in |\mathbf{T}|$, de un morfismo, $in_A : F(A, D(A)) \to D(A)$, denominado *constructor* y otro morfismo, $out_A : D(A) \to F(A, D(A))$, llamado *destructor*, inverso uno del otro. La dicotomía fundamental entre ambos mecanismos de relación con el tipo puede describirse [Jacobs y Rutten 1997] como *construcción* versus *observación*. Mientras el constructor informa de cómo generar elementos del tipo de datos, el destructor indica qué puede saberse acerca de los elementos del tipo.

Nuestro interés, aparte de en estas construcciones, se centra en la posibilidad de definir y trabajar con tipos abstractos. Para eso precisamos la extensión de algunos conceptos.

Definición 3.1. *([Hagino 1987],pág. 68) Sean F, $G : \mathbf{C} \to \mathbf{D}$ dos funtores; una F G-diálgebra es un objeto A de $\mathbf{C}$ junto a un morfismo $\xi_A : F(A) \to G(A)$ en $\mathbf{D}$. Dadas dos F G-diálgebras (A, ξ_A), (B, ξ_B), un morfismo de F G-diálgebras es una función $p : A \to B$ tal que*

$$\xi_A; G(p) = F(p); \xi_B.$$

Las F G-diálgebras y sus morfismos forman una categoría que denotaremos $\mathbf{DAlg}(F, G)$.

Como casos particulares, cuando G es el funtor identidad, las F G-diálgebras son las F-álgebras y cuando es F el funtor identidad las F G-diálgebras se corresponden con las G-coálgebras.

Otro caso especial de diálgebras se tiene cuando entre los funtores F y G hay con una transformación natural. La conmutatividad requerida para los morfismos de diálgebras es, precisamente, la condición de naturalidad de la transformación.

Ejemplo 3.2. *Dada la categoría $\mathbf{C}$, consideremos los endofuntores $List$ y N, siendo este último el funtor constante que lleva cualquier objeto en al conjunto de los naturales y cualquier morfismo en la identidad. Entonces, una $List$ N-diálgebra es un par (A, ξ_A), donde $\xi_A : List(A) \to nat$; si $p : A \to B$ es un morfismo en la categoría subyacente, será morfismo de diálgebras si verifica la identidad*

$$\xi_A = (maplist\ f); \xi_B.$$

Es decir, dada una lista de elementos de A, $[x_1, \ldots, x_n]$,

$$\xi_A[x_1, \ldots, x_n] = \xi_B[p(x_1), \ldots, p(x_n)].$$

Tomando $A = nat$ y $B = nat$ podemos considerar $p = -^2$, $\xi_A = (\times)_{List}; (-)^2$ y $\xi_B = \times_{List}$.

Los morfismos ξ_A y ξ_B no tienen por qué formar parte de una estructura universal, de forma que las diálgebras permiten definir funciones con menos restricciones que las impuestas por la condición de transformación natural.

Definición 3.2. *Sean (A, ξ_A) una FG-diálgebra y (B, θ_B) una GF-diálgebra; un morfismo $p : A \to B$ es* compatible *con ambas estructuras si se verifica que*

$$\xi_A; G(p); \theta_B = F(p).$$

Ejemplo 3.3. *Cuando el morfismo de comparación entre el menor y el mayor punto fijo de un endofuntor G es un isomorfismo, vimos en el Ejemplo 2.13 que dadas cualquier coálgebra y cualquier álgebra, existía entre la primera y la segunda un morfismo compatible con esa estructura (siendo $F = id$).*

Cuando $A = B$ y $p = id_A$, tenemos una estructura *FG-bialgebraica.* En este caso, $\xi_A; \theta_A = id_{FA}$.

Lema 3.1. *Sean* (A, ξ_A, θ_A), (B, ξ_B, θ_B) *FG-biálgebras; sea* $p : A \to B$ *un morfismo de FG-diálgebras. Entonces, p es un morfismo compatible con ambas estructuras.*

Demostración. Al ser p morfismo de FG-diálgebras, $\xi_A; G(p) = F(p); \xi_B$; entonces,

$$(\xi_A; G(p)); \theta_B = (F(p); \xi_B); \theta_B.$$

Ya que en una categoría la composición es asociativa,

$$(\xi_A; G(p)); \theta_B = F(p); (\xi_B; \theta_B) = F(p); id_{FA} = F(p).$$

□

Corolario 3.1. *Siendo* (A, ξ_A, θ_A), (B, ξ_B, θ_B) *FG-biálgebras; y* $p : A \to B$ *un morfismo de GF-diálgebras, p es un morfismo compatible con ambas estructuras.*

A partir de la definición de diálgebra construye Hagino los tipos categóricos bien como adjuntos a la izquierda bien como adjuntos a la derecha.

Definición 3.3. *([Hagino 1987], pág. 70) Sean* $\mathbf{C}, \mathbf{D}, \mathbf{E}$ *categorías y sean* $F : \mathbf{C} \times \mathbf{D} \to \mathbf{E}$, $G : \mathbf{C} \times \mathbf{D}^{op} \to \mathbf{E}$ *funtores. Dado un objeto* A *en* $\mathbf{D}$ *se define* $< Left[F, G], \eta_A >$ *como el objeto inicial de la categoría* $\mathbf{DAlg}(F(., A), G(., A))$. *Dualmente, se define* $< Right[F, G], \epsilon_A >$ *como el objeto final de esa categoría.*

Definición 3.4. *Sean* $F, G : \mathbf{C} \times \mathbf{D} \to \mathbf{E}$ *funtores; sea* A *un objeto de* $\mathbf{C}$*; una* F G-**diálgebra libre** *sobre* A *es una terna* $((A, \mu_A), \xi_A)$, *con* $\xi_A : F(A, \mu_A) \to G(A, \mu_A)$, *de forma que, si* $((B, C), \psi)$ *es otra* F G*-diálgebra y* $f : A \to B$ *es un morfismo en* $\mathbf{C}$, *existe un único morfismo* ${f_{AC}}^{\psi} : \mu_A \to C$ *que verifica* $(f, {f_{AC}}^{\psi})$ *es un morfismo de* F G*-diálgebras, es decir*

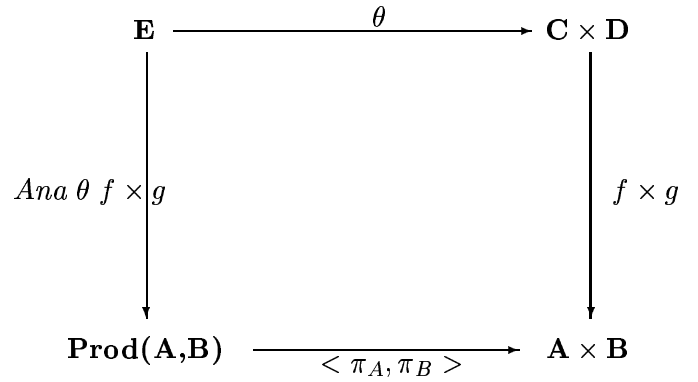
$$F(f, {f_{AC}}^{\psi}); \psi = \xi_A; G(f, {f_{AC}}^{\psi}).$$

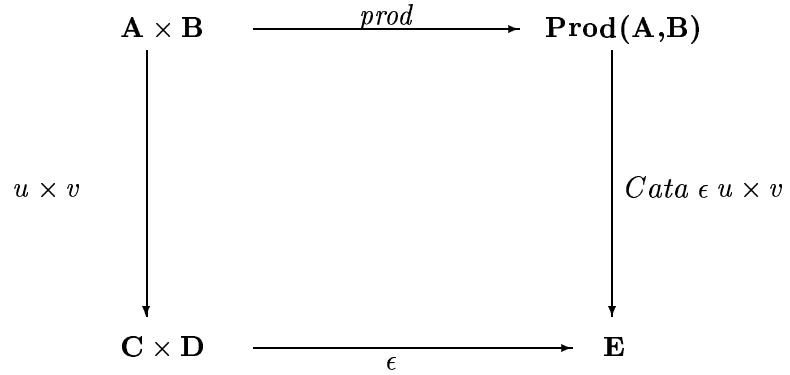
Dualmente se define el concepto de F G-**diálgebra colibre**.

Introducimos a continuación la noción de tipo $\mu\nu$-regular, basada en la anterior, que permite, por una parte, aprovechar las ventajas estructurales aportadas por las diálgebras y, por otra, enlazar con las ideas expresadas en [Erwig 1998].

Definición 3.5. *Sean* $F, G : \mathbf{C} \times \mathbf{D} \to \mathbf{E}$ *funtores; un* $\mu\nu$*-tipo regular es un funtor* $L : \mathbf{C} \to \mathbf{D}$ *que verifica, para cada* $A \in |\mathbf{C}|$ *existe una FG-diálgebra libre* $((A, LA), in_A)$ *y una GF-diálgebra colibre con el mismo soporte,* $((A, LA), out_A)$, *de forma que* $in_A : F(A, LA) \to G(A, LA)$, $out_A : G(A, LA) \to F(A, LA)$ *son isomorfismos en* $\mathbf{E}$ *que verifica* $in_A; out_A = 1_{F(A,LA)}$. *Es decir,* $((A, LA), in_A, out_A)$ *es una FG-biálgebra. El único morfismo desde la diálgebra libre a cualquier otra diálgebra será denominado catamorfismo. El único morfismo hacia la diálgebra colibre proveniente de cualquier otra diálgebra será el anamorfismo . La composición de un anamorfismo y un catamorfismo se llamará hilomorfismo.*

Ejemplo 3.4. *Vamos a ver, desde esta óptica, un tipo isomorfo al producto pero con mayor capacidad expresiva que él. Consideramos los funtores* $F,\ G : (\mathbf{Set} \times \mathbf{Set}) \times \mathbf{Set} \to \mathbf{Set}$ *definidos por:* $F((A,B),C) = A \times B,\ G((A,B),C) = C$. *Fijando el par* (A,B), *existe un conjunto, que denotamos* $Prod(A,B)$, *acompañado de un constructor,* $prod : A \times B \to Prod(A,B)$, *un destructor,* $< \pi_A, \pi_B >: Prod(A,B) \to A \times B$, *y, para cada pareja de morfismos* $(f : C \to A,\ g : D \to B)$, $(u : A \to C, v : B \to D)$, *dos funciones* $Ana\ \theta\ f \times g$, $Cata\ \epsilon\ u \times v$ *verificando la conmutatividad de los siguientes diagramas:*





Es decir,

$$\theta; f \times g \ = \ Ana\ \theta\ f \times g;$$

$$L; Cata\ \epsilon\ u \times v \ = \ u \times v\ \epsilon.$$

donde las definiciones de de Ana, Cata e Hilo se deducen de las restricciones establecidas:
$Ana\ \theta\ f\ \times\ g\ e\ =\ prod(f(a), g(b))$ *siendo* $(a,b) = \theta(e)$;
$Cata\ \epsilon\ u\ \times\ u\ prod(a,b)\ =\ \epsilon(f(a), g(b))$;
$Hilo\ \theta\ \epsilon\ f_1\ f_2\ g_1\ g_2\ e\ =\ \epsilon(g_1(f_1(c)), g_2(f_2(d)))$ *siendo* $(c,d) = \theta(e)$.

El tipo $Prod(A,B)$ *es isomorfo a* $A \times B$ *y tiene una caracterización de inicialidad de la que éste carece.*

No todos los tipos son $\mu\nu$-regulares pero, como ya se mencionó anteriormente, en el contexto de una semántica no estricta, sí se puede considerar esta dualidad para todos los tipos basados en funtores polinómicos. De ahí que un tipo como el producto, paradigma del objeto terminal, tenga su carácter de inicialidad dentro del contexto dialgebraico.

A continuación se analizan los tres aspectos considerados fundamentales en cuanto a la aportación de los tipos $\mu\nu$-regulares.

3.6 *Diálgebras y listas*

Se estudia, en primer lugar, el tipo de las listas. Fijado el tipo A , $list\ A$ es el menor punto fijo del endofuntor $F(B) = () + A \times B$. Para su modelización como diálgebra se definen los funtores $FList$, $\Pi : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ cuya acción sobre los objetos es: $FList(A, B) = () + A \times B$, $\Pi(A, B) = B$. Además, dados dos morfismo $f : A \to A'$, $g : B \to B'$, $FList(f, g) = ! + f \times g$, siendo ! el único morfismo de () en sí mismo y $\Pi(f, g) = g$. Siguiendo la definición, una $FList$ Π-diálgebra es un par $((A, B), \epsilon)$ con $\epsilon : () + A \times B \to B$. Dadas las $FList$ Π-diálgebras $((A, B), \epsilon)$ y $((A', B'), \epsilon')$, un homomorfismo entre ellas es un par de funciones $(f : A \to A', g : B \to B')$ verificando

$$\epsilon; g = ! + (f \times g); \epsilon'.$$

Denotamos esta categoría por $\mathbf{DAlg(FList, \Pi)}$. Analizamos a continuación algunas construcciones relevantes en esta categoría.

Teorema 3.1. *La categoría* $\mathbf{DAlg(FList, \Pi)}$ *es cartesiana y tiene objeto terminal.*

Demostración. En efecto, sean $M1 = (A, B, \xi_{AB}, m_{AB})$, $M2 = (C, D, \xi_{CD}, m_{CD})$ modelos de listas. Consideramos $M1 \times M2 = (A \times C, B \times D, \rho_{ABCD}, (m_{AB}, m_{CD})$ donde $\rho_{ABCD} = \phi_{ABCD}; (\xi_{AB} \times \xi_{CD})$ siendo $\phi_{ABCD} : (A \times C) \times (B \times D) \to (A \times B) \times (C \times D)$ el isomorfismo definido como $\phi_{ABCD}((a, c), (b, d)) = ((a, b), (c, d))$. Las proyecciones son $proy_1 : M1 \times M2 \to M1$, $proy_1 = (\pi_1^{AC}, \pi_1^{BD})$ y $proy_2 : M1 \times M2 \to M2$, $proy_2 = (\pi_2^{AC}, \pi_2^{BD})$. Entonces, efectivamente, $M1 \times M2$ es un producto ya que, si $M = (X, Y, \xi, m)$ es un objeto de esta categoría y $(p1, q1) : M \to M1$, $(p2, q2) : M \to M2$ son morfismos en ella, existe un único morfismo $((p1, p2), (q1, q2)) : M \to M1 \times M2$ que verifica

$$(q1, q2)(m) = (m_{AB}, m_{CD}),$$

$$(q1, q2)(\xi(x, y)) = (\phi_{ABCD}; (\xi_{AB}, \xi_{CD}))(((p1, p2), (q1, q2))(x, y)).$$

La primera igualdad es obvia al ser $(p1, q1)$ y $(p2, q2)$ morfismos en esta categoría. La segunda se sigue inmediatamente de

$$\begin{gathered}(\phi_{ABCD}; (\xi_{AB}, \xi_{CD}))(((p1, p2), (q1, q2))(x, y)) = \\ (\xi_{AB}, \xi_{CD})((p1(x), q1(y)), (p2(x), q2(y))) = \\ (\xi_{AB}, \xi_{CD})((p1(x), q1(y)), (p2(x), q2(y))) = \\ (\xi_{AB}(p1(x), q1(y), \xi_{CD}(p2(x), q2(y))) = \\ (q1(\xi(x, y)), q2(\xi(x, y))).\end{gathered}$$

El objeto terminal es $\Upsilon = ((), (), !, unit)$ siendo () el objeto terminal de la categoría subyacente y $unit$ su único elemento. □

Teorema 3.2. *Cada par de flechas en* $\mathbf{Dial}(\mathbf{FList}, \mathbf{\Pi})$ *tiene un igualador.*

Demostración. Sean $M1 = (A, B, \xi_{AB}, m_{AB})$, $M2 = (C, D, \xi_{CD}, m_{CD})$ dos modelos de listas. Sean, además, $(p, q), (r, s) : M1 \to M2$ dos pares de morfismos de diálgebras, es decir, $p, r : A \to C$, $q, s : B \to D$ verificando

$$q(m_{AB}) = m_{CD};$$

$$q(\xi_{AB}(a, b)) = \xi_{CD}(p(a), q(b));$$

$$s(m_{AB}) = m_{CD};$$

$$s(\xi_{AB}(a, b)) = \xi_{CD}(r(a), s(b)).$$

Es obvio que en $\mathbf{T}$ hay igualadores: dadas dos funciones α, β entre los tipos $T1$, $T2$, $eq(\alpha, \beta) = \{x : T1 | \alpha(x) = \beta(x)\}$. Entonces, existe (X, f) igualador de p y r y existe (Y, g) igualador de q y s (siendo, además, f y g mónicas). Por consiguiente, se verifican las siguientes relaciones:

$$\forall x \in X,\ p(f(x)) = r(f(x));$$

$$\forall y \in Y,\ q(g(y)) = s(g(y)).$$

Por lo tanto, $\forall x \in X$, $\forall y \in Y$,

$$q(\xi_{AB}(f(x), g(y))) = \xi_{CD}(p(f(x)), q(g(y))) = \xi_{CD}(r(f(x)), s(g(y))) =$$

$$s(\xi_{AB}(f(x), g(y))).$$

Es decir, el par $\xi_{AB}(f(X), g(Y)), inc_B)$ iguala las flechas q y s: por la definición de igualador, existe un único morfismo $h : \xi_{AB}(f(X), g(Y)) \to Y$ que verifica $h; g = inc_B$. Así, definimos $\epsilon : X \times Y \to Y$, $\epsilon(x, y) = h(\xi_{AB}(f(x), g(y)))$, $x \in X$, $y \in Y$. Entonces, ya que $q(m) = s(m)$ y el igualador es maximal, $(X, Y, \epsilon, g^{-1}(m))$ es un modelo de listas que iguala los morfismos (p, q), (r, s). Resta ver su carácter terminal. Pero, si existiera $((U, V, \delta, o), (k, l))$ tal que $p(k(u)) = r(k(u))$, $q(l(v)) = s(l(v))$, $u \in U$, $v \in V$, al ser (X, f) igualador de (p, r), existirá un único morfismo $k^* : U \to X$ verificando $k^*; f = k$; análogamente, al ser (Y, g) igualador de (q, s), existirá un único morfismo $l^* : V \to Y$ verificando $l^*; g = l$. Además,

$$l^*(o) = g^{-1}(m);$$

$$g(l^*(\delta(u, v))) = l(\delta(u, v)) = \xi_{AB}(k(u)), l(v))) =$$

$$\xi_{AB}(f(k^*(u)), g(l^*(v))) = g(\epsilon(k^*(u), l^*(v))).$$

Entonces, al ser g mónica,

$$l^*(\delta(u, v)) = \epsilon(k^*(u), l^*(v)),$$

por lo cual (k^*, l^*) es un morfismo en $\mathbf{Dial}(\mathbf{FList}, \mathbf{\Pi})$. □

Corolario 3.2. *([Rydeheard y Burstall 1988], pág. 95) La categoría* $\mathbf{Dial}(\mathbf{FList}, \mathbf{\Pi})$ *tiene todos los límites finitos (es decir, es completa).*

Teorema 3.3. *La categoría* $\mathbf{Dial}(\mathbf{\Pi}, \mathbf{FList})$ *tiene coproductos binarios.*

Es fácil probar que $(A + C, B + D, \psi)$ es el coproducto de los modelos de listas (A, B, θ_{AB}), (C, D, θ_{CD}), siendo $\psi : B + D \to () + (A + C) \times (B + D)$ definida por: $()$ para $b \in B$ que verifica $\theta_{AB}(b) = ()$ y para $d \in D$ con $\theta_{CD}(d) = ()$.

$$\psi(in_B^{BD}(b)) = (in_A^{AC}(a), in_B^{BD}(b')),\ (a, b') = \theta_{AB}(b);$$
$$\psi(in_D^{BD}(d)) = (in_C^{AC}(c), in_D^{BD}(d')),\ (c, d') = \theta_{CD}(d).$$

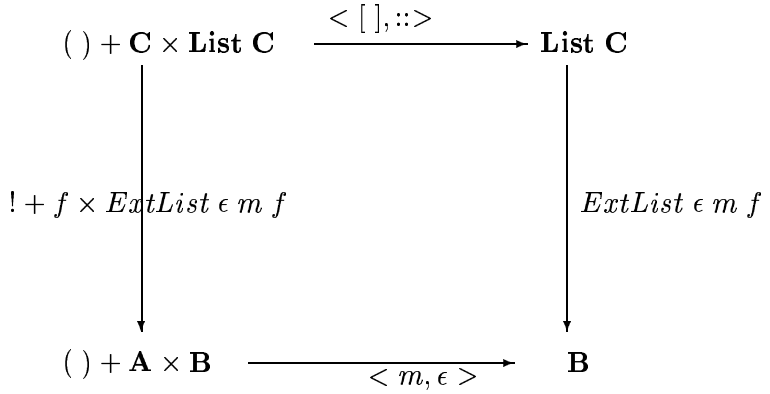
Sea (X, Y, θ_{XY}) otra $(\Pi_2, FList)$-diálgebra y $(f1, g1) : (A, B, \theta_{AB}) \to (X, Y, \theta_{XY})$, $(f2, g2) : (C, D, \theta_{CD}) \to; (X, Y, \theta_{XY})$ morfismos en la categoría: entonces, ya que en **T** hay coproductos binarios podemos definir $[f1, f2] : A + C \to X$, $[g1, g2] : B + D \to Y$. Comprobar que $([f1, f2], [g1, g2])$ es un morfismo en $\mathbf{Dial}(\Pi_2, FList)$ es una tarea inmediata.

Teorema 3.4. *La categoría* $\mathbf{Dial}(\mathbf{\Pi}, \mathbf{FList})$ *tiene coigualador para cada par de flechas.*

La demostración es dual a la del Teorema 3.3.2 teniendo en cuenta que el coigualador de dos flechas $f, g : A \to B$ en **T** viene dado por (C, e) siendo $C = B/\equiv$, con $\equiv$ la menor relación de equivalencia sobre B tal que $f(b) \equiv g(b)$ para todo $b \in B$ y e la sobreyección correspondiente.

Proposición 3.1. *Sea* C *un tipo y consideremos* (A, B, ϵ, m) *un objeto en* $\mathbf{Dial}(FList, \Pi)$. *Entonces,* $((C, List\ C, ::, [\,]))$ *es la* $\mathbf{Dial}(FList, \Pi)$*-diálgebra libre sobre* C.

Demostración. En efecto, dado un morfismo $f : C \to A$, existe un único morfismo que denotaremos por $ExtList\ \epsilon\ m\ f : List\ C \to B$ verificando la conmutatividad del siguiente diagrama:



es decir,

$$< [\,], ::>; (ExtList\ \epsilon\ m\ f) = (! + f \times (ExtList\ \epsilon\ m\ f)); \epsilon.$$

Podemos definir esta función en CAML como sigue:

```
let ExtList  epsilon m f=Frec where
       rec Frec=function []->m
                  | (x::l)->epsilon ( f(x), Frec l);;
```

La unicidad de tal función es evidente: si (f, g) es otro morfismo de diálgebras entre $((C, List\ C, ::, [\,]))$ y (A, B, ϵ, m) deben verificarse las dos siguientes ecuaciones:

$$g\ [\,] = n;$$
$$g\ x :: xs = \epsilon(f\ x, g\ xs).$$

Pero, ésta es, precisamente, la definición de $ExtList\ \epsilon\ n\ f$. Por consiguiente, $g = ExtList\ \epsilon\ n\ f$. □

Este funcional nos va a resultar muy útil: primeramente, porque satisface varias propiedades relevantes de cara a la optimización de programas que analizaremos más adelante; en segundo lugar, el hecho de utilizar una función extra como argumento posibilita su utilización en contextos más amplios de lo usual. Como un sencillo ejemplo de su uso, podemos considerar la función que, dada una lista de cualquier tipo, la convierte en la suma (entera) de sus transformados por medio de un filtro. Para esta transformación contamos con dos funciones, dependientes, en su aplicación a cada elemento del tipo original, de la verificación o no de una determinada proposición:

```
  let H p f g=ExtList  (prefix +) 0
(function x->if (p x) then (f x) else (g x));;
```

Aplicando `H` a las funciones y lista siguientes

```
let p x=x mod 7=3;;
let f x=x*x-2;;
let g x=x-4;;
let lista=[10;20;30;40;50;60;70];;
```

obtenemos como resultado `344`.

Dada la importancia del lenguaje de adjunciones, es interesante poder analizar este funcional desde esa perspectiva.

Proposición 3.2. *Consideremos los funtores* $F : \mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi}) \to \mathbf{T}$, $G : \mathbf{T} \to \mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi})$ *definidos por:*

$$F(A, B, \epsilon, m) = A,\ F((f, g) : (A, B, \epsilon, m) \to (C, D, \sigma, n) = f,$$
$$G(A) = (A, List\ A, ::, [\,]),\ G(f : A \to C) = (f, mapList\ f).$$

Entonces, G es adjunto a la izquierda de F.

Demostración. En primer lugar, la transformación natural $\eta : 1_{\mathbf{T}} \Rightarrow (G; F)$ es la identidad. Además, $\mu : (F; G) \Rightarrow 1_{\mathbf{DAlg}(\mathbf{FList},\mathbf{\Pi})}$ asigna a cada diálgebra (A, B, ϵ, m) la flecha $\mu_{(A,B,\epsilon,m)} : (A, List\ A, ::, [\,]) \to (A, B, \epsilon, m)$ definida por $\mu_{(A,B,\epsilon,m)} = (f, ExtList\ \epsilon\ m\ id_A$. Por lo tanto, hemos de comprobar que, dados (A, B, ϵ, m) un modelo de listas y C un tipo, existe un funcional, φ, entre $\mathbf{T}(C, F(A, B, \epsilon, m))$ y $\mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi})(G(C), (A, B, \epsilon, m))$ verificando, para cada morfismo $f : C \to F(A, B, \epsilon, m)$, la igualdad $\eta_C; F(\varphi(f)) = f$. Definimos $\varphi(f) : G(C) \to (A, B, \epsilon, m)$ como $\varphi(f) = (f, ExtList\ \epsilon\ m\ f)$. Además, dado un morfismo $((g1, g2) : (C, List\ C, ::, [\,]) \to (A, B, \epsilon, m)$, si definimos $\psi(g1, g2) = g1 : C \to A)$, se verifica, por la unicidad de los homomorfismos de diálgebras desde el modelo canónico, que $\varphi(\psi(g1, g2)) = (g1, g2)$. □

Ahora bien ([Rydeheard y Burstall 1988]), esta adjunción nos permite dar un paso más: sabemos que un funtor que es adjunto por la izquierda preserva colímites; entonces, ya que la categoría **T** tiene coproductos finitos, al conservar el funtor G estos coproductos, si A es una colección finita de tipos,

$$(\coprod_{\alpha \in A} \alpha, list\ (\coprod_{\alpha \in A} \alpha), ::, [\,])$$

es el el modelo de listas coproducto; notemos que, este coproducto da lugar a listas **heterogéneas**, es decir, listas con elementos de diferentes tipos. Además, si consideramos **T** una categoría cocompleta, no precisamos limitarnos a coproductos finitos, de donde la heterogeneidad abarcará a todos los tipos (en cantidad finita o no) que se puedan construir en la categoría.

Como avanzamos en la introduccion, podemos analizar el tipo de las listas desde otra perspectiva. Si consideramos las G F-diálgebras, resulta que $(C, List\ C, split)$, siendo $split$`[]=[]` y $split\ (x :: xs) = (x, xs)$, es la G F-diálgebra colibre sobre C. Es decir, si (A, B, θ) es una G F-diálgebra, con $\theta : B \to () + A \times B$, dado el morfismo $g : A \to C$, existe un único morfismo $(AnaList\ \theta\ g) : B \to List\ C$ con $(g, AnaList\ \theta\ g)$ morfismo de G F-diálgebras. Su definición es como sigue (por una cuestión de comodidad notacional introducimos un tipo auxiliar `analist`):

```
  type ('a,'b) analist=
Nil
| Prod of 'a*'b;;

  let rec AnaList  theta g=
function  b->match (theta(b)) with
Nil->[]
|(Prod(a,b'))->(g a)::(AnaList  theta g b');;
```

Podríamos albergar la duda de si esta función realmente existe: no se ha impuesto ninguna restricción acerca de la finalización del proceso `AnaList theta g b'`) lo cual podría hacer suponer la necesidad de extender las listas al caso infinito (como hemos hecho al calcular el mayor punto fijo del funtor $F(B) = () + A \times B$). Ahora bien, si se define el módulo Haskell `infinito.hs` con

```
Module naturales where
 from n = n:(from (n+1))
 naturales = from 0
```

y se quiere conocer el contenido de la posición 100 escribiendo

```
infinito> naturales !! 100
100
infinito>
```

vemos que, esa posibilidad de no terminación no condiciona, cuando hacemos una evaluación lazy, la definibilidad. Siempre podemos acceder a una posición determinada para conocer su contenido. También, como otra posibilidad para atestiguar la validez de esta construcción, usamos el sistema `Coq` obteniendo lo siguiente:

```
Definition igual :=[P:Prop]Cases P of
True => true
|False => false end.

CoInductive anaprod [A,B:Set]:Set:=I:(anaprod A B)
|Pair:A->B->(anaprod A B).

CoInductive lista [A:Set]: Set :=Nil:(lista A)
|Cons :A->(lista A)->(lista A).

Definition AnaList :=[A,B,C:Set][t:B->(anaprod A B)][f:A->C]
CoFix F {F:B->(lista C):=[b:B]
Cases (t b) of
        I            => (Nil C)
        |(Pair a b') => (Cons C (f a) (F b'))
end}.
```

La respuesta que se obtiene, **AnaList is defined**, nos garantiza que la construcción es correcta.

La función `zip` ([Meijer y otros 1991]) permite transformar un par de listas (una de tipo `string` y otra de tipo `int`) en una lista de pares indicando la primera coordenada la longitud de cada una de las cadenas y la segunda si el número es múltiplo de 5 o no; es un ejemplo del uso del funcional `AnaList` :

```
let zip=AnaList  Theta
(function (x,y)->(string_length x,y mod 5=0))
where
Theta=function ([],[])->Nil
|(x::xs,(y::ys))->Prod((x,y),(xs,ys));;
```

Vemos, pues, que hemos definido un anamorfismo generalizado sobre las listas. Esta posibilidad de revertir las diálgebras la veremos en otras construcciones con lo cual enriquecemos la capacidad estructural de los tipos, ya que estos morfismos, independientemente de la condición de inicial o terminal, tendrán opciones de simplificación. Vemos, a continuación, otra perspectiva que nos ofrecen, en este caso, los colímites. Fijemos un tipo A. Consideramos el diagrama formado por todos los modelos de listas cuyo soporte es A con sus morfismos de modelos de listas; es decir, los objetos son cuádruplas (A, B, ϵ_B, m_B), con $\epsilon_B : A \times B \to B$ y $m_B \in B$ y un morfismo entre (A, B, ϵ_B, m_B) y (A, C, ϵ_C, m_C) es un par (id_A, f_{BC}), con $f_{BC} : B \to C$ verificando $f_{BC}(m_B) = m_C$ y $f_{BC}(\epsilon_B(a, b)) = \epsilon_C(a, f_{BC}(b))$.

Teorema 3.5. $(A, List\ A, ::, [\,])$ *es el límite del diagrama anterior, siendo el morfismo de modelos de listas entre él y* (A, B, ϵ_B, m_B) *el par* $(id_A, ExtList\ \epsilon_B\ m_B\ id_A)$.

Demostración. Comprobamos primeramente que $(A, List\ A, ::, [\,])$ determina un cono. Para ello, debemos ver que

$$(ExtList\ \epsilon_B\ m_B\ id_A); f_{BC} = (ExtList\ \epsilon_C\ m_C\ id_A).$$

Lo haremos inductivamente sobre la longitud de la lista.

$$f(ExtList\ \epsilon_B\ m_B\ id_A\ [\,]) = f(m_B) = m_C = (ExtList\ \epsilon_C\ m_C\ id_A\ [\,]).$$

Supongamos que, para cualqier lista xs de longitud n

$$f(ExtList\ \epsilon_B\ m_B\ id_A\ xs) = (ExtList\ \epsilon_C\ m_C\ id_A\ xs).$$

Entonces, si $x \in C$,

$$f(ExtList\ \epsilon_B\ m_B\ id_A\ (x :: xs)) = f(\epsilon_B(x, ExtList\ \epsilon_B\ m_B\ id_A\ xs) =$$

$$\epsilon_C(x, f(ExtList\ \epsilon_B\ m_B\ id_A\ xs)) = \epsilon_C(x, ExtList\ \epsilon_C\ m_C\ id_A\ xs)) =$$

$$ExtList\ \epsilon_C\ m_C\ id_A\ (x :: xs).$$

La condición de inicial es la que ha quedado caracterizada previamente en la sección 3.6. □

Dualmente podemos enunciar lo siguiente:

Teorema 3.6. *$((A, list\ A, split), analist)$ es el colímite del diagrama cuyos vértices son los objetos (A, B, θ_B) con $\theta : B \to () + A \times B$ y con aristas $f_{BC} : B \to C$ verificando, para cada $b \in B$,*

$$(! + id_A \times f_{BC})(\theta_B(b)) = \theta_C(f_{BC}(b)).$$

Demostración. Veamos que $((A, list\ A, split), AnaList)$ es un cocono. Hemos de demostrar que

$$f_{BC}; (AnaList\ \theta_C\ id_A) = (AnaList\ \theta_B\ id_A).$$

Sea $b \in B$ y supongamos que $\theta_B(b) = ();$, entonces,

$$(AnaList\ \theta_B\ id_A)(b) = [\,].$$

Como

$$(! + id_A \times f_{BC})(\theta_B(b)) = \theta_C(f_{BC}(b)),$$

$$\theta_C(f_{BC}(b)) = (),$$

de donde

$$(AnaList\ \theta_C\ id_A)(f_{BC}(b)) = [\,].$$

Supongamos, pues, que $\theta_B(b) = (a, b')$.

$$(AnaList\ \theta_B\ id_A)(b) = a :: (AnaList\ \theta_B\ id_A)(b').$$

Además,

$$\theta_C(f_{BC}(b)) = (a, f_{BC}(b')),$$

de donde

$$(AnaList\ \theta_C\ id_A)(f_{BC}(b)) = a :: (AnaList\ \theta_C\ id_A\ f_{BC}(b')).$$

□

3.6.1 Algunas propiedades de las construcciones sobre las listas

El funcional $ExtList\ \epsilon\ m\ f$ es una síntesis de la expresión map f;$foldr\ \epsilon\ m$. Utilizando esta formalización hemos llegado a lo que, en lenguaje de catamorfismos, es el conocido *teorema de fusión* para las listas. Tenemos, pues, una nueva perspectiva de este útil teorema de optimización.

Por otra parte, $ExtList\ \epsilon\ m\ f$ es el funcional `list_extend` que aparece en [Rydeheard y Burstall 1988]. Veremos en la próxima sección, como una aplicación del funcional `ExtList` , que las listas forman un monoide libre (también un semigrupo libre). Esta estructura monoidal es la de la concatenación de listas con la lista vacía como elemento neutro. Si se considera $\eta : A \to List\ A$ la función que transforma cada elemento en una lista con ese único elemento, para cada monoide $B = (B, *, e)$ y cada función $f : A \to B$, hay un único homomorfismo de monoides, $f^\sharp$, del monoide de las listas sobre A en B de forma que

$$\eta; f_\sharp = f.$$

Nótese que $foldr\ \epsilon\ m$ coincide con $ExtList\ \epsilon\ m\ id$, de forma que, ligando este punto de vista con el del BMF, se ve que, en este nivel inicial, nuestra aproximación es ligeramente más general reemplazando la función identidad $id : A \to A$ por una función arbitraria $f : A \to B$.
Es fácil verificar, por ejemplo, las siguientes igualdades:

```
composition=ExtList (prefix o) I I
map         =ExtList cons []
length      =ExtList (prefix +) 0 (function x->1)
any_true    =ExtList (or) false I
```

Consideremos la siguiente función también expresable como una instanciación de *ExtList*:

```
let rec it_list f x l =
 match l with []->x
    |(a::l)->it_list f (f a x) l;;
```

La semántica de `it_list` es la como sigue: `it_list` f x [a_1, ... a_n] = f a_n (...(f a_2 (f a_1 x))...)

Por inducción sobre el tamaño de la lista y razonando ecuacionalmente, se puede demostrar que

`it_list` f x l=`Ext` g x (reverse l), siendo g(x,y)=f x y.

La determinación categórica de esta función se puede ver en el teorema que viene a continuación:

Teorema 3.7. *Sean* $(A, List\ A, [], cons)$ *el modelo canónico de listas y* $((A, B), \epsilon)$ *un modelo de listas arbitrario. Consideremos las funciones* $h_1, h_2 : () + B \times A \times A\ list \to B \times List\ A$ *definidas por:*

$$h_1(b, x, xs) = (b, (x :: xs));$$

$$h_2(b, x, xs) = (\epsilon(x, b), xs);$$

$$h_1() = h_2() = (\epsilon(), [\,]).$$

Entonces, $(B, \xi(\epsilon))$ *es el coigualador de* h_1, h_2*, viniendo* $\xi(\epsilon)$ *definida como:*

$$\xi(\epsilon)(b, [\,]) = b;$$

$$\xi(\epsilon)(b, (x :: xs)) = \xi(\epsilon)(\epsilon(x, b), xs).$$

Demostración. En efecto: probemos, en primer lugar, que coiguala las dos funciones:

$$\xi(\epsilon)(h_1(b, x, xs)) = \xi(\epsilon)(b, (x :: xs));$$

$$\xi(\epsilon)(h_2(b, x, xs)) = \xi(\epsilon)(\epsilon(x, b), xs).$$

En segundo lugar, sea (C, h) tal que $h : B \times List\ A \to C$ y $h_1; h = h_2; h$; podemos definir, por supuesto, la función $f : B \to C$ como $f(b) = h(b, [\,])$; ésta es la función que buscamos. En efecto, es sencillo, por inducción, ver que $\xi(\epsilon); f = h$.

$$f(\xi(\epsilon)(b, [\,]) = f(b) = h(b, [\,]);$$

$$f(\xi(\epsilon)(b, x :: [y]) = f(\xi(\epsilon)(\epsilon(x, b), [y])) =$$

$$f(\xi(\epsilon)(\epsilon(y, \epsilon(x, b)), [\,])) = f(\epsilon(y, \epsilon(x, b))) = h(\epsilon(y, \epsilon(x, b)), [\,]).$$

Pero, al ser h el coigualador de h_1 y h_2 se verifica la siguiente cadena:

$$h(\epsilon(y, \epsilon(x, b)), [\,]) = h(h_2(\epsilon(x, b), y, [\,])) = h(h_1(\epsilon(x, b), y, [\,])) =$$

$$h((\epsilon(x, b), [y])) = h(h_2(b, x, [y])) = h(h_1(b, x, [y])) = h(b, x :: [y])$$

que es lo que deseábamos ver. La unicidad es, también, simple de mostrar: si existiera otra función $g : B \to C$ que verifica $\xi(\epsilon); g = h$ se verificaría que, para cada $b \in B$ $g(\xi(\epsilon)(b, [\,])) = h(b, [\,])$. Ahora bien, como $\xi(\epsilon)(b, [\,]) = b$ esto implica que $g(b) = h(b, [\,])$ es decir, $f = g$. Es obvio ver que `it_list f` $x\ l = \xi(f)(x, l)$. □

Nótese que es muy sencillo extender la función `it_list` tal como se ha hecho con `foldr`:

```
 let rec Ext_it_List epsilon m f=function []->m
|(x::xs)->
Ext_it_List epsilon (epsilon(f x,m)) f xs;;
```

Obtenemos una versión eficiente del funcional `ExtList` al ser una función tail recursive; así, permite efectuar en tiempo lineal operaciones que podrían, de otra forma, ser cuadráticas o exponenciales y, además, reduce significativamente las memorias intermedias. Estas dos funciones verifican la ecuación `Ext_it_List` ϵ m f l=`Ext` ϵ f m (reverse l).

La interpretación categórica de `Ext_it_List` es análoga al caso de la función `it_list`:

Teorema 3.8. *Dados* $((B,C),\epsilon)$ *una* $FList$ Π*-diálgebra y* $f : A \to B$ *un morfismo,* `Ext_it_List` $\epsilon_{B\times C}$ $\epsilon_{()}$ f l *es* $(\xi\ \epsilon_{B\times C}\ \epsilon_{()}\ f)(\epsilon_{()}, l)$*, siendo* ξ *la representación del coigualador de los morfismos* $h_1, h_2 : () + C \times A \times List\ A \to C \times List\ A$ *definidos por:*

$$h_1(c,x,xs) = (c,(x :: xs));$$
$$h_2(c,x,xs) = (\epsilon(fx,c),xs);$$
$$h_1() = h_2() = (\epsilon(),[\,]).$$

Se verifica la siguiente propiedad relativa a `ExtList` y `Ext_it_List`:

Teorema 3.9. $(List\ A, reverse, I)$ *es un* pullback *de las funciones* `Ext_it_List` ϵ m *f:List* $A \to B$ *y* `Ext` ϵ m *f:List* $A \to B$*.*

Como otro ejemplo interesante, la función `list_hom`, definida en [Cousineau y Mauny 1995] por

```
 let rec list_hom m f=fun
[]->m
|(x::l)->f x (list_hom m f l);;
```

es otro caso particular de `Ext` debido a que
`list_hom m f=ExtList eval m f`.

Hablamos anteriormente de las listas snoc; desde una perspectiva categórica, una lista snoc es una $FList$ Π-diálgebra libre sobre su segunda coordenada, siendo π el funtor definido sobre los objetos por $\pi(A,B) = A$. Indicamos allí que la interpretación categórica de la transformación de listas convencionales en listas snoc es interesante. En efecto, podemos ver el siguiente:

Teorema 3.10. *Consideremos los morfismos* $h_1, h_2 : (() + A \times List\ A) \times A$ `snoc_list` $\to List\ A \times A$ `snoc_list` *definidos por:*

$$h_1\ ((),s) = h_2\ ((),s) = ([\,],s);$$
$$h_1\ (x,xs,s) = ((x :: xs),s);$$
$$h_2\ (x,xs,s) = (xs, snoc(s,x)).$$

Entonces, $(A$ `snoc_list`$,\xi)$*, con* $\xi : List\ A \times A$`snoc_list`$\to A$`snoc_list` *verificando*

$$\xi([\,],s) = s$$

y

$$\xi((x :: xs),s) = \xi(xs, snoc(s,x))$$

es el coigualador de h_1 *y* h_2*.*

Cabe pensar, ya que en ello trabajamos, cómo podemos definir los funcionales `snoc_Ext` y `snoc_it_list`:

```
 let rec snoc_ExtList epsilon m f=function nil->m
|(snoc(xs,x))->
epsilon(f x,snoc_ExtList epsilon m f xs);;

 let rec snoc_it_list epsilon m f=function nil->m
|(snoc(xs,x))->
snoc_it_list epsilon (epsilon(f x,m)) f xs;;
```

Observemos que sus semánticas respectivas son:
`snoc_Ext` ϵ m f (snoc(...(snoc(nil,a_1), ... ,a_n)=
ϵ(f a_n, ϵ(f $a_{n-1}, \ldots, \epsilon$(f a_1, m),...)), y
`snoc_it_list` ϵ m f (snoc(...(snoc(nil,a_1), ... ,a_n)
=ϵ(f a_1, ϵ(f $a_2, \ldots, \epsilon$(f a_n, m),...)).

Entonces, `snoc_it_list` es una función iterativa *tail recursiva* mientras que `snoc_Ext` es recursiva.

Una curiosa relación surge entre las listas convencionales y las listas snoc.

Teorema 3.11. *Consideremos el modelo de listas* (A, B, ϵ, m), *h:*$C \to A$ *y las funciones* `Ext` ϵ *m h:*C *list* $\to B$ *y* `snoc_Ext` ϵ *m h:*C `snoc_list` $\to B$*; entonces,* (C *list*, *conv*1 : C *list* $\to$ C `snoc_list`, *reverse* : C *list* $\to$ C *list*) *es un* pullback *para las funciones anteriores. Además, también es un* pullback *para las funciones* `it_list` ϵ *m h y* `snoc_it_list` ϵ *m h.*

Demostración. La conmutatividad del diagrama correspondiente se obtiene por aplicación inductiva sobre listas convencionales y listas snoc; además, si f:$D \to C$ `snoc_list` y g:$D \to C$ *list* son funciones tales que g;`Ext` ϵ m h=f;`snoc_Ext` ϵ m h, cuando consideramos x$\in D$, si g x=[$a_1; \ldots; a_n$], f x=snoc(...(nil,a_n),...,),a_1); entonces, existe una única función k:$D \to C$ *list*, k x=conv2(f x)=reverse(g x), que hace conmutativos los diagramas correspondientes. □

La siguiente propiedad que relaciona anamorfismos y catamorfismos será generalizada más adelante en el caso de diálgebras:

Teorema 3.12. *Siendo* A *y* C *tipos, y considerando* (::) *la operación de concatenación de listas y split su inversa (split* [ ] = ( ) *y split* $(x :: xs) = (x, xs)$*), se verifica que, siendo* $p : A \to C$ *un morfismo,* `AnaList` *split* p = *ExtList* (::) [ ] p.

3.7 *Diálgebras y* streams

Podemos definir las *streams* como provenientes de los funtores $FST(A, B) = A \times B$, $GST(A, B) = B$. Entonces, dada una FST GST-diálgebra arbitraria $((B, C), \theta : C \to B \times C)$ y un morfismo $f : B \to A$ existe un único morfismo entre C y *Stream* A, *anaStream* θ f de forma que (f, *anaStream* θ f) es un morfismo de FST GST-diálgebras. La definición de este morfismo es inmediatamente expresable a partir de esta premisa:

```
 let rec anaStream theta f c = let (b,c')=(theta c)
in Inf(f b,anaStream theta f x');;
```

La función `unfold` puede definirse a partir de ésta:

```
let unfold theta = anaStream theta id;;
```

El carácter inicial de este tipo deviene del hecho de ser vacío cuando se lo considera como inductivo [Paulin y Werner 1998].

3.8 *Diálgebras y árboles*

Para comprobar las ventajas en cuanto a la abstracción consideramos el tipo de los árboles de aridad variable. La declaración habitual que encontramos para este tipo ([Wraith 1989], [Jeuring y Jansson 1996]) es la siguiente:

```
type 'a tree=
Nodo of 'a*('a tree) list;;
```

Fijando el tipo A podemos definir $F(B) = A \times List\ B$. La coálgebra final de este endofuntor es el tipo $Tree\ A$. El morfismo $[L, S] : Tree\ A \to A \times Tree\ A$ definido como $L(Nodo(x, xs)) = x,\ S(Nodo(x, xs)) = xs$ da la estructura de F-coálgebra. Dada otra F-coálgebra, (B, θ), con $\theta : B \to A \times List\ B$, existe un único morfismo de F-coálgebras $AnaTree\ \theta : B \to Tree\ A$ definido como sigue (con la adecuación sintáctica pertinente):

```
  type ('a,'b) anatree =
Prod of 'a*('b list);;

  let rec AnaTree theta = function  b->
match (theta(b)) with (Prod(a,xs))
->Nodo(a,map_list (AnaTree theta) xs);;
```

Como comprobaremos a continuación, utilizando este funtor se prescinde de la capacidad de generar funciones de orden superior más genéricas. En efecto, retomemos nuevamente la categoría $\mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi})$ Construimos los funtores $F,\ G : \mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi}) \times \mathbf{T} \to \mathbf{T}$ definidos sobre los objetos como $F((A, B, \epsilon, m), C) = C \times B$ y $G((A, B, \epsilon, m), C) = A$. Una $F\ G$-diálgebra es una terna $((A, B, \epsilon, m), C, \delta)$ donde $\delta : C \times B \to A$; un morfismo de $F\ G$-diálgebras entre $((A, B, \epsilon, m), C, \delta)$ y $((A', B', \epsilon', m'), C', \delta')$ es una terna $(f : A \to A', g : B \to B', h : C \to C')$ verificando que (f, g) es un morfismo en $\mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi})$ y $\delta; f = (h \times g); \delta'$. Entonces, la $F\ G$-diálgebra libre sobre D es la terna $((D\ tree, D\ tree\ list, ::, []), D, Nodo)$. Dado un morfismo $h : D \to C$, existe un único morfismo $ExtTree\ \epsilon\ \delta\ f : D\ tree \to A$ que verifica $Nodo; (ExtTree\ \epsilon\ \delta\ f) = (h \times ExtList\ \epsilon\ (ExtTree\ \epsilon\ \delta\ f)); \delta$. Su definición en CAML es como sigue:

```
let ExtTree epsilon m delta g=Trec where
 rec Trec=function
  (Nodo(x,l))->delta( g x, (ExtList epsilon m Trec) l);;
```

Al ser la función `Ext_it_List` más eficiente que `ExtList`, podemos optimizar la función `ExtTree` definiendo:

```
let Ext_it_Tree epsilon m delta g=Trec where
 rec Trec=function
  (Nodo(x,l))->delta( g x, (Ext_it_List epsilon m Trec) l);;
```

Teorema 3.13. *La categoría de los modelos de árbol de aridad variable es cartesiana y tiene objeto final.*

Demostración. Sean $M_1 = ((A, B, \xi_{AB}, m_{AB}), X, \delta_{ABX}$,
$M_2 = ((C, D, \xi_{CD}, m_{CD}), Y, \delta_{CDY}$ dos objetos en esta categoría (siendo (A, B, ξ_{AB}, m_{AB}), (C, D, ξ_{CD}, m_{CD}) objetos de $\mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi})$. Hemos visto que esta categoría es cartesiana: consideramos, pues, $M_1 \times M_2 = (A \times C, B \times D, \rho ABCD, (m_{AB}, m_{CD})), X \times Y, \delta_{ABXCDY}$ donde $\rho ABCD$ es el morfismo definido en el Teorema 3.3.1 y $\delta_{ABXCDY} : (X \times Y) \times (B \times D) \to A \times C$ es $\delta_{ABXCDY} = \phi_{XBYD}; (\delta_{ABX} \times \delta_{CDY}$. Las proyecciones son las obvias y la prueba de su carácter terminal es obvia siguiendo el patrón del teorema antes mencionado.

Además, $\Theta = (\Upsilon, (\ ), !)$ es el objeto terminal de esta categoría, donde Υ es el modelo de listas definido en el Teorema 3.3.1. □

Naturalmente, no hemos perdido la condición terminal que antes teníamos. Construimos, para ello, los funtores $F,\ G : \mathbf{DAlg}(\mathbf{\Pi_2}, \mathbf{FList}) \times \mathbf{T} \to \mathbf{T}$ definidos sobre los objetos como $F(((A, B), \rho), E) = E \times B$ y
$G(((A, B), \rho), E) = A$. Una G F-diálgebra es una terna $((((A, B), \rho), E), \theta)$, con $\theta : A \to E * B$. Dado, entonces, el morfismo $g : E \to C$, existe un morfismo $AnaTree\ \rho\ \theta\ g : A \to C\ tree$ definido por:

```
  type ('a,'b) anatree=
Prod of 'a*'b;;

  let LS (Nodo(a,xs))=Prod(a,xs);;

  let rec AnaTree rho theta g c=
match (theta(c)) with
(Prod(a,b))
->Nodo(g a,AnaList  rho (AnaTree rho theta g) b);;
```

de forma que $(g, AnaTree\ \rho\ \theta\ g)$ es el único morfismo de diálgebras entre ambas.

Si deseamos englobar ambos casos dentro de una misma estructura dialgebraica, podemos construir la categoría, **List** que reúne las características de $\mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi})$ y de $\mathbf{DAlg}(\mathbf{\Pi}, \mathbf{FList})$; es decir, sus objetos son cuaternas (A, B, ϵ, θ) de forma que $(A, B, \epsilon) \in \mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi})$ y $(A, B, \theta) \in \mathbf{DAlg}(\mathbf{\Pi}, \mathbf{FList})$; un morfismo entre (A, B, ϵ, θ) y $(A', B', \epsilon', \theta')$ es un par de funciones (p, q) tales que $\epsilon; q = (! + p * q); \epsilon'$ y
$q; \theta' = \theta; (! + p * q)$. Entonces, se verifica

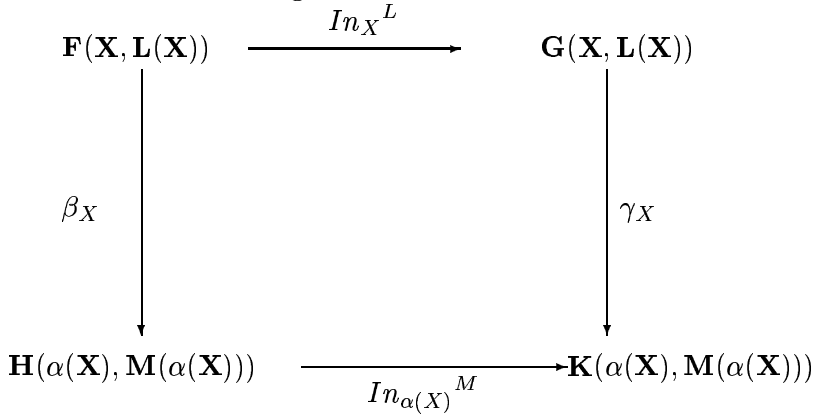
Teorema 3.14. *Para cualquier función $p : A \to B$,*

$$ExtTree\ (::)\ [\,]\ Nodo\ p = AnaTree\ split\ LS\ p.$$

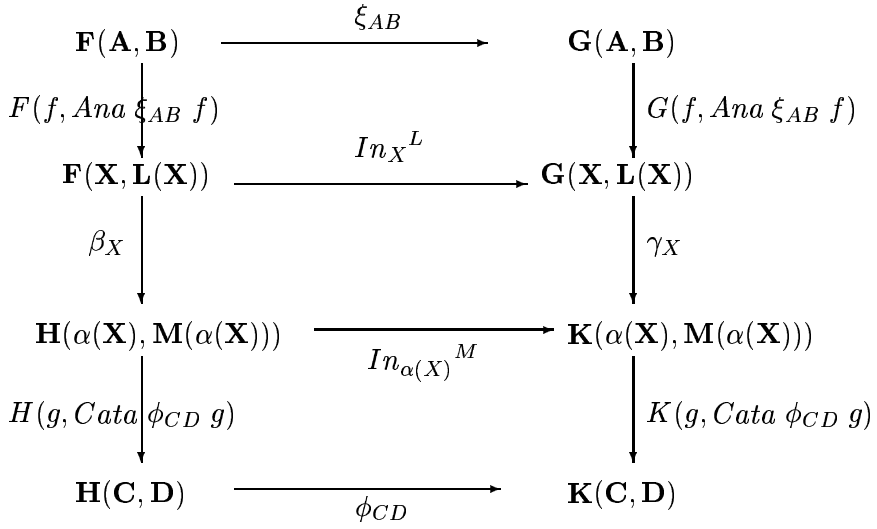
3.9 *Diálgebras y morfismos entre estructuras tipadas*

Es interesante estudiar bajo qué condiciones es posible definir morfismos `estructurales` entre dos tipos determinados por diálgebras. Consideremos para ello $F,\ G : \mathbf{X} \times \mathbf{T} \to \mathbf{T}$, $H,\ K : \mathbf{Y} \times \mathbf{T} \to \mathbf{T}$ funtores y $L : \mathbf{X} \to \mathbf{T}$, $M : \mathbf{Y} \to \mathbf{T}$ los tipos $\mu\nu$-regulares ligados a las FG y HK diálgebras, respectivamente. Supongamos definido un funtor $\alpha : \mathbf{X} \to \mathbf{Y}$; podemos considerar, entonces, para $X \in \mathbf{X}$, las dos estructuras $(X, L(X), {In_X}^L)$, $(\alpha(X), M(\alpha(X)), {In_{\alpha(X)}}^M)$.

En las condiciones anteriores, un morfismo entre las estructuras de las FG diálgebras y las HK diálgebras es, para cada $X \in \mathbf{Y}$, un par de morfismos $\beta_X : F(X, L(X)) \to H(\alpha(X), M(\alpha(X)))$, $\gamma_X : G(X, L(X)) \to K(\alpha(X), M(\alpha(X)))$ verificando la conmutatividad del diagrama



Dadas una FG-diálgebra (A, B, ξ_{AB}) y una HK-diálgebra (C, D, ϕ_{CD}) podemos determinar, dado un morfismo entre las dos estructuras tipadas, un morfismo entre ellas. En efecto, dados $f : A \to X$, $g : \alpha(X) \to C$ tenemos homomorfismos (únicos) $Ana\ \xi_{AB}\ f : B \to L(X)$, $Cata\ \phi_{CD}\ g : M(\alpha(X)) \to D$. Entonces, se verifica la conmutatividad del diagrama



es decir,

$$F(f, Ana\ \xi_{AB}\ f); \beta_X; H(g, Cata\ \phi_{CD}\ g); \phi_{CD} =$$
$$\xi_{AB}; G(f, Ana\ \xi_{AB}\ f); \gamma_X; K(g, Cata\ \phi_{CD}\ g).$$

3.10 *Tipos de datos no regulares*

Los dos tipos definidos en la sección anterior son ejemplos de tipos recursivos regulares; existen otros tipos que no se ajustan al isomorfismo descrito anteriormente. Se llaman tipos anidados [Bird y Meertens 1998] o no regulares.

Definición 3.6. *Dados dos funtores* $F,\ G : \mathrm{T} \times \mathrm{T} \to \mathrm{T}$ *y un endofuntor* $H : \mathrm{T} \to \mathrm{T}$ *(diferente del identidad, en cuyo caso tenemos un tipo regular) un funtor* K *determina un tipo anidado si, para cualquier tipo* A *existe un isomorfismo* $F(A, K(H(A))) \simeq G(A, K(A))$.

Distinguimos dentro del conjunto de los tipos no regulares dos clases diferentes:

1. aquellos para los cuales, en la definición recursiva del tipo con diferentes argumentos, obtenemos el tipo recursivo con los argumentos originales tras un número finito de iteraciones. Es fácil, en este caso, expresar estos tipos por medio de tipos inductivos equivalentes;

2. aquellos en los que esa repetición no existe; es decir, no podemos identificarlos con ningún tipo inductivo.

3.11 *Tipos anidados equivalentes a tipos regulares*

Consideremos el siguiente isomorfismo correspondiente a la adjunción entre producto y exponenciación: $\mathbf{C}(A \times B, C) \simeq \mathbf{C}(A, C^B)$; aplicándolo al caso particular en que $C = B^A$, deducimos que $\mathbf{C}(A \times B^A, B) \simeq \mathbf{C}(A, B^{B^A})$. Aquí surge un anidamiento ya que, mientras en la parte izquierda de la ecuación el exponente es A en la derecha éste es B^A. Los

primeros tipos anidados que vamos a considerar verifican la condición de que esta segunda parte con argumentos diferentes a los originales pueden representarse por medio de tipos regulares.

El segundo caso de esta clase queda descrito con el tipo `Rope` ([Meijer y Jeuring 1995]), donde hay una alternancia de los argumentos en la declaración del tipo:

```
  type ('a,'b) Rope=
Nil
|Tor of 'a*('b,'a) Rope;;
```

Su análisis dialgebraico surge de los funtores $F,\ G : (\mathrm{T} \times \mathrm{T}) \times \mathrm{T} \to \mathrm{T}$ definidos como sigue:

$$F((A,B),C) = () + A \times C,\ G((A,B),C) = C;$$

además, necesitamos el endofuntor $H : \mathrm{T} \times \mathrm{T} \to \mathrm{T} \times \mathrm{T}$ definido como $H(A,B) = (B,A)$; entonces, el tipo es un funtor $Rope : \mathrm{T} \times \mathrm{T} \to \mathrm{T}$ verificando, para cada par$(A,B) \in \mathrm{T} \times \mathrm{T}$ el isomorfismo

$$F((A,B), Rope(H(A,B))) \simeq G((A,B), Rope(A,B)).$$

En este caso particular, podemos construir un tipo regular equivalente al tipo `Rope`:

```
  type ('a,'b) Rope2=
Nil2
|El of 'a
|Tor2 of 'a*'b*('a,'b) Rope2;;
```

verifica esta condición. Obviamente, es regular ya que proviene de los funtores $F,\ G : \mathbf{T} \times \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ definidos sobre los objetos como sigue (la definición sobre los morfismos se colige de ésta):
$F(A,B,C) = () + A + A * B * C$, $G(A,B,C) = C$.

El isomorfismo entre los tipos `Rope` y `Rope2` viene dado por las siguientes funciones:

```
let rec conv12=fun
Nil->Nil2
|(Tor(x,Nil))->El(x)
|(Tor(x,Tor(y,z)))->Tor2(x,y,conv12 z);;

let rec conv21=fun
Nil2->Nil
|(El(x))->Tor(x,Nil)
|(Tor2(x,y,z))->Tor(x,Tor(y,conv21 z));;
```

Un modelo abstracto del tipo `Rope2` es, por consiguiente, un par $((A,B,C),\rho)$, con $\rho : () + A + A * B * C \to C$, de forma que, ahora sí, tenemos un catamorfismo generalizado con el siguiente código (donde el funcional ρ se corresponde con la suma $\epsilon + \theta$):

```
 let rec ExtRope2 epsilon theta f g b=
function Nil2->b
|(El x)->epsilon(f x)
|(Tor2(x,y,z))
->theta(f x,g y,ExtRope2 epsilon theta f g b z);;
```

El valor `b` de tipo `'b` es un elemento distinguido necesario para poder identificar el valor `Nil2`.

A partir de estos datos no resulta complicado construir un catamorfismo generalizado sobre el tipo no regular `Rope`:

```
 let rec ExtRope epsilon theta f g b=
function Nil->b
|(Tor(x,Nil)->epsilon(f x)
|(Tor(x,Tor(y,z)))
->theta(f x,g y,ExtRope epsilon theta f g b z);;
```

Como en los casos anteriores, podemos caracterizar los anamorfismos:

```
  type ('a,'b,'c) anarope=
AnaNil
|AnaEl of 'a
|AnaProd of 'a*'b*'c;;


  let rec AnaRope2 theta f g c=
match (theta c) with
AnaNil->Nil2
|(AnaEl a)->El(f x)
|(AnaProd(a,b,c'))
->Tor2(f a,g b,AnaRope theta f g c'));;
```

Al igual que en el caso del catamorfismo para el tipo `Rope`, se puede definir un anamorfismo generalizado:

```
  let rec AnaRope theta f g c=
match (theta c) with
AnaNil->Nil
|(AnaEl a)->Tor(f a,Nil)
|(AnaProd(a,b,c'))
->Tor(f a,Tor(g b,AnaRope theta f g c'));;
```

Podemos expresar estos morfismos en **Hugs98** comprobando que, con esta clase particular de tipos, no se requieren estructuras computacionales adicionales:

```
module Rope where
data Rope a b = N | T a (Rope b a)

extRope :: b->(a->b)->((a,c,b)->b)->(d->a)->(e->c)->
```

```
Rope d e -> b
extRope b0 ep th f g N = b0
extRope b0 ep th f g (T x xs) = ep (f x)
extRope b0 ep th f g (T x (T y xs)) =
th(f x,g y, extRope b0 ep th f g xs)

anaRope :: (b->Bool)->(b->Bool)->(b->(a,c,b))->(a->d)->
(c->e)->b->Rope d e
anaRope p q th f g z = if p z then N else if q z
then T (f x) N
else
T (f x) (T (g y) (anaRope p q th f g z'))
where (x,y,z')=th z

f1 :: c -> Rope c d
f1 x = T x N

f2 :: (c,d,Rope c d) -> Rope c d
f2 (x,y,N) = T x (T y N)
f2 (x,y,T z zs) = T x (T y (T z zs))

mapRope1 :: (a->c)->(b->d)->Rope a b -> Rope c d
mapRope1 f g = extRope N f1 f2 f g

pRope :: Rope a b -> Bool
pRope N = True
pRope _ = False

qRope :: Rope a b -> Bool
qRope (T x N) = True
qRope _ = False

proyRope :: Rope a b-> (a,b,Rope a b)
proyRope (T x (T y zs)) = (x,y,zs)

mapRope2 :: (a->c)->(b->d)->Rope a b -> Rope c d
mapRope2 f g = anaRope pRope qRope proyRope f g
```

Notemos que, para no diversificar los tipos (como hemos hecho en **CAML**) debemos introducir varias funciones auxiliares (`pRope, qRope, proyRope`) para poder expresar los `maps` correctamente.

Teorema 3.15. *Para cualesquiera morfismos* `f, g`*, se verifica que* `mapRope1 f g=mapRope2 f g`.

La demostración es trivial.

Podemos construir, como composición del anamorfismo y del catamorfismo, el hilomorfismo correspondiente seguido del resultado de su optimización:

```
hilRope :: (b->Bool)->(b->Bool)->(b->a)->(b->(a,c,b))->
(a->d)->(c->e)->y->(x->y)->((x,z,y)->y)->(d->x)->(e->z)->
b->y
hilRope p q ep1 th1 f g y0 ep2 th2 h k =
extRope y0 ep2 th2 h k.anaRope p q ep1 th1 f g


hilRopeOpt :: (b->Bool)->(b->Bool)->(b->a)->(b->(a,c,b))->
(a->d)->(c->e)->y->(x->y)->((x,z,y)->y)->(d->x)->(e->z)->b->y
hilRopeOpt p q ep1 th1 f g y0 ep2 th2 h k y =
if p y then y0 else if q y then
ep2(h(f(ep1 y))) else
th2(h(f x),k(g z),hilRopeOpt p q ep1 th1 f g y0 ep2 th2 h k y')
where (x,z,y')=th1 y
```

Por consiguiente, los tipos recursivos no regulares definidos de forma tal que en un número finito de etapas es posible obtener el tipo inicialmente declarado, son caracterizables por tipos inductivos equivalentes a ellos a partir de los cuales se obtienen los catamorfismos y anamorfismos correspondientes. Notemos que, con esta caracterización queda demostrada la existencia del tipo `Rope`. Además, respondemos a la cuestión planteada por Erik Meijer y Johan Jeuring en [Meijer y Jeuring 1995] acerca de cómo describir, de una forma sencilla, un catamorfismo para `Rope`: sugieren una posibilidad (según ellos, poco elegante) consistente en definir un catamorfismo para cada una de las aplicaciones recursivas de `Rope`, es decir, sustituir el tipo `('a,'b) Rope` por los tipos `('a,'b) Zig` y `('a,'b) Zag`.

3.12 *Tipos anidados en general*

Veamos cómo afrontar el problema que surge cuando tenemos tipos anidados que no pueden reducirse a regulares. Contamos con dos cuestiones a responder: la primera, si el tipo anidado realmente existe; la segunda, supuesta su existencia, cómo derivar las funciones correspondientes.

Analizaremos dos casos particulares y observaremos qué características tienen en tales circunstancias los catamorfismos y los anamorfismos. Al no poder definir en CAML las funciones anidadas que requiere nuestra aproximación, utilizamos el compilador de Haskell **Hugs98**.

Ejemplo 3.5. *Consideramos, para empezar, el tipo*

```
data Llist a= NilL
| ConsL(a,Llist [a])
```

Los valores de este tipo son listas cuya aridad va incrementándose a medida que aplicamos el constructor `ConsL`. *Consideremos, para su interpretación, los funtores*

$F(A,B) = () + A \times B$, $G(A,B) = B$, $H = List$. Entonces, Llist debería verificar $() + A \times Llist(ListA) \simeq Llist\ A$. A partir de la definición, y denotando $List^n(A)$ la aplicación iterada n veces del funtor List sobre el tipo A, tenemos lo siguiente (considerando la categoría subyacente adecuada para las simplificaciones que vamos a establecer como hemos hecho en el Ejemplo 2.14 establecemos):

$$Llist(A) \simeq () + A \times Llist(List(A)) \simeq$$

$$() + A \times (() + List(A) \times Llist(List(List(A)))) \simeq$$

$$() + A \times (() + List(A) \times (() + List(list(A)) \times Llist(List(List(List(A)))))) \simeq$$

$$() + A + A \times List(A) + A \times List(A) \times List(List(A)) + \ldots$$

$$() + A + A \times List(A) + A \times List(A) \times List^2(A)) + \ldots \simeq \coprod_{n \in nat} \prod_{n \in nat} List^n(A),$$

con $H^0(A) = ()$.

Entonces, si la categoría subyacente es predistributiva, ω-completa y ω-cocompleta, contiene el tipo $Llist(A)$. Además, cuando $G(A,B) = B$ y $F(A,B) = A + B$ o $F(A,B) = A \times B$, la existencia del funtor K está garantizada. En el primer caso, al ser la categoría ω-cocompleta, se verifica la siguiente cadena de isomorfismos:

$$K(A) \simeq A + K(H(A)) \simeq A + H(A) + K(H^2(A))$$

$$\simeq A + H(A) + H^2(A) + K(H^3(A)) \simeq \ldots,$$

para cualquier tipo A, K es el coproducto

$$\coprod_{n \in nat} H^n(A),$$

siendo $H^0 = I$ el funtor identidad. Análogamente, en el segundo caso, tenemos la siguiente cadena de isomorfismos:

$$K(A) \simeq A \times K(H(A)) \simeq A \times H(A) \times K(H^2(A))$$

$$\simeq A \times H(A) \times H^2(A) \times K(H^3(A)) \simeq \ldots;$$

a causa de ser la categoría subyacente ω-completa definimos

$$K(A) = \coprod_{n \in nat} H^n(A).$$

El isomorfismo requerido es inmediatamente demostrable en cualquiera de los dos casos:

$$F(A, K(H(A))) = A + K(H(A)) = A + \coprod_{n \in nat} H^{n+1}(A) =$$

$$\coprod_{n \in nat} H^n(A) = K(A);$$

en el segundo caso tenemos

$$F(A, K(H(A))) = A \times K(H(A)) = A \times \prod_{n \in nat} H^{n+1}(A) =$$

$$\prod_{n \in nat} H^n(A) = K(A);$$

Cuando el funtor F es constante, el tipo K es, obviamente, esa constante. Bajo estas condiciones, el funtor K existe independientemente de cuál sea el funtor H. En cambio, si F es la segunda proyección es necesario tener en cuenta quién es el funtor H; en efecto, cuando $F(A, B) = A$ debería ocurrir que $K(A) \simeq A$, ecuación que verifica el funtor identidad; en cambio, si $F(A, B) = B$, debería verificarse que $K(H(A)) \simeq K(A)$, de lo cual no podemos, a priori, concluir nada.

Por ejemplo, si definimos el tipo

```
data Same a = SA Same [a]
```

carecemos de la posibilidad de generar elementos de este tipo, ya que estos formarían una cadena ilimitada $SA(SA(SA(SA.....$ Así, podemos garantizar que, cuando $G(A, B) = B$ y F es la suma, el producto, una constante, la primera proyección o cualquier combinación de estos funtores, el tipo recursivo no regular K existe.

Ejemplo 3.6. *([Bird y Meertens 1998], pág. 52) Se define el tipo* `Nest` *como sigue:*

```
data Nest a = NilN
|ConsN(a,(Nest(a,a))
```

Considando el funtor $H(a) = (a, a)$, el tipo Nest debe satisfacer el isomorfismo $() + A \times Nest(H(A)) \simeq Nest(A)$. Se verifica, tal como hemos visto en el Ejemplo 3.2.5 (cambiando List por H)

$$Nest(A) = \coprod_{n \in nat} \prod_{n \in nat} H^n(A)$$

donde, nuevamente, $H^0(A) = ()$.

Como ya hemos indicado, aparte de la existencia del tipo anidado, nos interesan las posibilidades de generación [automática] de funciones. Es decir, cuando podemos garantizar la existencia del tipo anidado, ¿qué podemos afirmar acerca de los catamorfismos o de los anamorfismos? ¿Podemos definirlos de forma similar al caso de los tipos regulares o es preciso realizar alguna transformación? Comenzamos analizando los dos casos precedentes e intentaremos ver qué dificultades surgen en el decurso del desarrollo.

Ejemplo 3.7. *Probemos a definir el catamorfismo generalizado sobre el tipo Llist. Los elementos de este tipo son como el siguiente:*
$ConsL(a, ConsL([b; c], ConsL([[d; e; f]; [h]], NilL)))$.
Observamos la presencia de cuatro elementos diferentes a tener en cuenta:

1. *el constructor $ConsL$ que lo transformaremos en la función ξ (veremos más adelante quiénes son el dominio y el codominio);*

2. *el constructor (::) de las listas (lo cambiaremos por ϵ);*

3. *el constructor ([]) de las listas (lo cambiaremos por b_0);*

4. *el constructor (NilL), transformado en d_0.*

Así, pues, el contexto en que estamos es el siguiente:

1. *debemos disponer de un morfismo $\xi : B \times C \to C$;*

2. *es necesario un morfismo $\epsilon : B \times B \to B$;*

3. *precisamos un elemento $b_0 \in B$;*

4. *por último, debemos contar con un elemento $d_0 \in C$.*

Entonces, dado un morfismo $p : A \to B$, existe un único morfismo $extLlist\ \xi\ d_0\ \epsilon\ b_0\ p : Llist\ A \to C$ verificando:

$$extLlist\ \xi\ d_0\ \epsilon\ b_0\ p\ NilL = d_0;$$

$$extLlist\ \xi\ d_0\ \epsilon\ b_0\ p\ ConsL(x, xs) =$$

$$\xi(p(x), extLlist\ \xi\ d_0\ \epsilon\ b_0\ (extList\ \epsilon\ b_0\ p)\ xs).$$

Su definición en **Hugs98** *es como sigue (usando la definición en este mismo lenguaje del funcional* `extList`*):*

```
extList :: (a->b->b)->b->(c->a)->[c]->b
extList e m f []=m
extList e m f (x:xs) = e (f x) (extList e m f xs)

extLlist :: (b->c->c)->c->(b->b->b)->b->(a->b)->(Llist a->c)
extLlist e c0 h b0 f NilL = c0
extLlist e c0 h b0 f (ConsL(x,xs)) =
e (f x) (extLlist e c0 h b0 (extList h b0 f) xs)
```

Notemos, entonces, que la función mapLlist no puede ser definida por medio de este funcional: para ésta precisamos cambiar en cada iteración el tipo al cual la aplicamos, hecho que no hemos previsto en el catamorfismo. Hemos, por lo tanto, de derivarla aparte:

```
mapLlist :: (a->b)->(Llist a -> Llist b)
mapLlist f NilL = NilL
mapLlist f (ConsL(x,xs)) = ConsL(f x,mapLlist (map f) xs)
```

Siguiendo una estructura análoga a la descrita podemos definir la función `anaLlist`*:*

```
incL :: Llist a->Llist [a]
incL NilL = NilL
incL (ConsL(x,xs)) = ConsL([x],(incL xs))

anaLlist :: (b->(a,b))->(b->Bool)->(a->c)->b->Llist c
anaLlist theta p f y = if p y then NilL else
ConsL(f x,incL (anaLlist theta p f y'))
where (x,y')=theta y
```

Ejemplo 3.8. *Analicemos ahora el tipo $Nest$. Un elemento de este tipo tiene la forma $ConsN(a, ConsN((b,c), ConsN((d,e),(f,g), NilN)))$. Como antes, debemos establecer un morfismo que sustituya a $ConsN$, un elemento fijo que represente a $NilN$ y un morfismo ϵ que transforme los pares. Entonces, los requisitos son los siguientes:*

1. *un morfismo $\xi : B \times C \to C$;*
2. *es necesario un morfismo $\epsilon : B \times B \to B$;*
3. *$d_0 \in D$.*

Entonces,

$$extNest\ \xi\ d_0\ h\ \epsilon\ p\ NilN = d_0;$$

$$extNest\ \xi\ d_0\ h\ \epsilon\ p\ ConsN(x, ConsN((y,z), xs) =$$

$$\xi(p(x), \xi(\epsilon(fy, fz), extNest\ \xi\ d_0\ h\ \epsilon\ b_0\ (extList\ \epsilon\ b_0\ p)\ xs).$$

En **Hugs98***:*

```
extNest :: (b->c->c)->c->(b->b->b)->(a->b)->(Nest a ->c)
extNest e c0 h f NilN = c0
extNest e c0 h f (ConsN(x,xs)) =
e (f x) (extNest e c0 h (theta h f) xs)

theta  :: (b->b->b)->(a->b)->(a,a)->b
theta h f (x,x')=h (f x) (f x')
```

Por el mismo motivo que en el ejemplo anterior, esta definición nos limita la obtención de mapNest por instanciación. Se define de forma normal:

```
par :: (a->b)->(a'->b')->(a,a')->(b,b')
par f g (x,x')=(f x,g x')

mapNest :: (a->b)->(Nest a -> Nest b)
mapNest f NilN = NilN
mapNest f (ConsN(x,xs)) = ConsN(f x,mapNest (par f f) xs)
```

La función `anaNest` *puede ser definida así:*

```
incN :: Nest a->Nest (a,a)
incN NilN = NilN
incN (ConsN(x,xs)) = ConsN((x,x),incN xs)

anaNest :: (b->(a,b))->(b->Bool)->(a->c)->b->Nest c
anaNest theta p f y = if p y then NilN else
ConsN(f x,incN (anaNest theta p f y'))
where (x,y')=theta y
```

Observamos, entonces, que si se desea construir el catamorfismo que, instanciado, represente la función *map* correspondiente, no podemos disponer de una única estructura sino que ésta debe adaptarse a las variaciones implicadas en los tipos anidados. Analicemos esta situación. Consideremos, para ello, el caso general, es decir, K es el funtor que determina un tipo verificando el isomorfismo $in_C : F(C, K(H(C)) \simeq G(C, K(C))$. Dada una estructura (A, B, ξ), con $\xi : F(A, H(B)) \to G(A, B)$ y un morfismo $p : C \to A$, supongamos la existencia de $ExtK_{A,C}\ \xi\ p : K(C) \to B$ satisfaciendo

$$in_C; G(p, ExtK\ \xi\ p) = F(p, ExtK_{H(A),H(C)}\ \rho(\xi)\ \tau(p)); \xi.$$

Como cabía esperar, surgen una nueva función $ExtK_{H(A),H(C)}\ \rho(\xi)\ \tau(p) : K(H(C)) \to H(B)$ que, dada la construcción del tipo, debe provenir de una función $\rho(\xi) : F(H(A), H^2(B)) \to G(H(A), H(B))$ y de otra función $\tau(p) : H(C) \to H(A)$. Esta última, dado que H es un funtor, es $H(p)$. El problema surge con la primera función.

3.12.1 Primer intento de solución

Una solución para garantizar la existencia de $\rho(\xi)$ (basada en los trabajos acerca de computación monádica de Fokkinga [Fokkinga 1994] y Hu&Iwasaki [Hu y Iwasaki 1995]) consiste en contar, para cada par de tipos A B de sendos morfismos $\delta_{A,B} : F(H(A), H^2(B)) \to H(F(A, H(B))$ y $\theta_{A,B} : H(G(A, B)) \to G(H(A), H(B))$. Entonces,

$$\rho(\xi) = \delta_{A,B}; H(\xi); \theta_{A,B}.$$

Aún así, tenemos un problema a resolver: si deseamos definir $mapK\ p : K(C) \to K(A)$ a partir de $f : C \to A$ y del constructor $in_A : F(A, K(H(A)) \simeq G(A, K(A))$ debemos contar con un morfismo $\alpha_A : H(K(A)) \to K(H(A))$, en cuyo caso, podremos escribir

$$mapK\ p = ExtK\ (F(id_A, \alpha_A); in_A)\ p.$$

Es decir:

Corolario 3.3. *Sea K un tipo anidado verificando, para cada tipo C, $F(C, K(H(C)) \simeq G(C, K(C))$. Consideremos, además, la existencia, para cada par de tipos A y B de los morfismos $\delta_{A,B} : F(H(A), H^2(B)) \to H(F(A, H(B))$ y $\theta_{A,B} : H(G(A, B)) \to G(H(A), H(B))$. Entonces, si existe una transformación natural $\alpha : (K; H) \Rightarrow (H; K)$, existe y es único el funcional $ExtK_{A,C}\ \xi\ p : K(C) \to B$ satisfaciendo*

$$in_C; G(p, ExtK\ \xi\ p) = F(p, ExtK\ (\delta_{A,B}; H(\xi); \theta_{A,B})\ H(p)); \xi.$$

Además,

$$mapK\ p = ExtK\ (F(id_A, \alpha_A); in_A)\ p.$$

3.12.2 Segundo intento: signaturas de tipo de rango 2

Ejemplo 3.9. *Consideremos nuevamente el tipo $Nest$ del ejemplo anterior. No resulta sencillo definir de forma suficientemente global la función requerida en el apartado anterior entre $(NestA, NestA)$ y $Nest(A, A)$ sin pérdida de información. Podemos construir la función α(similar al* `zip` *de Haskell)*

$$\alpha_A(NilN, NilN) = NilN;$$

$$\alpha_A(NilN, -) = NilN;$$

$$\alpha_A(-, NilN) = NilN;$$

$$\alpha_A(ConsN(x, xs), ConsN(y, ys)) = ConsN((x, y), \alpha_A(xs, ys))$$

que no satisface todas nuestras expectativas.

Entonces, siguiendo el trabajo de Bird&Meertens [Bird y Meertens 1998], subiremos un escalón en nuestra estructura categórica hasta situarnos en el contexto de la categoría de funtores y transformaciones naturales. Entonces, la estructura que deviene de los funtores que determinan el tipo es, para cualquier funtor funtor T, aquélla que consta de un elemento de tipo $T(A)$ y de un morfismo $T(A) \to T(A, A) \to T(A)$. Sean, pues, $< m|\epsilon >: () + A \times T(A, A) \to T(A)$. Podemos definir la siguiente función:

```
extNest :: (forall x.f x)->(forall x.(x,f(x,x))->f x)->
(c->a)->(Nest c)->f a
extNest m e f NilN = m
extNest m e f (ConsN(x,xs)) = e(f x,extNest m e (par f f) xs)
```

Se tiene la capacidad para definir la función `mapNest` *como instanciación de ésta (en la aproximación de Bird&Meertens tal posibilidad no se da)*

```
mapNest :: (a->b)->Nest a->Nest b
mapNest f = extNest NilN ConsN f
```

Ejemplo 3.10. *El tipo Llist tampoco se acomoda a la idea del caso anterior ya que, nuevamente, no podemos definir de forma natural la función δ (carecemos de imagen de $([\,], l) \in (List\ A) \times (List^2\ B)$ en $List\ (() + A \times (List\ B))$). Entonces, actuando tal como acabamos de hacer con* `Nest`*, podemos construir la función*

```
extLlist :: (forall x.f x)->(forall x.(x,f[x])->f x)->
(c->a)->(Llist c)->f a
extLlist m e f NilL = m
extLlist m e f (ConsL(x,xs)) = e(f x,extLlist m e (map f) xs)
```

Nuevamente, `mapLlist` *es una instancia de este funcional:*

```
mapLlist :: (a->b)->Llist a->Llist b
mapLlist f = extLlist NilL ConsL f
```

3.12.3 Tercer intento de solución

Hasta el momento hemos conseguido logros parciales que no son capaces de aunar los dos propósitos que nos hemos impuesto: derivar el funcional `map` con una instanciación del catamorfismo correspondiente y utilizar estos catamorfismos para obtener funciones (suma, tamaño, etc) sobre ellos. Observando las estructura sobre las que trabajamos podemos obtener una respuesta a las dos cuestiones analizando qué clase de ecuaciones deben satisfacer ambas funciones `fold`:

$$< m|\epsilon >; (foldType < m|\epsilon > h) =! + h \times foldType < m^*|\epsilon^* > h^*.$$

Vemos que el problema radica en las funciones m^*, ϵ^*, h^*. La última proviene de la estructura cambiada de forma que no resulta problemática (es `map h` para el tipo `Llist` y `(h,h)` para `Nest`. ¿ Qué hacer con las otras? ¿ Cómo derivarlas? (Bird & Meertens solventan este problema con la cuantificación universal local pero ya hemos visto los problemas que esto acarrea en cuanto a instanciaciones [Bird y Meertens 1998]). Nuestra solución es la que presentamos a continuación: como parece obvio en la ecuación anterior, debemos restringir el constructor de tipos `f` a aquellos que permitan, dados los valores m, ϵ, obtener los requeridos m^*, ϵ^*. Analizamos primeramente cómo resulta esto con el tipo `Nest`:

```
class ValidNest f where
        fValidNest1 :: (f x)->f (x,x)
        fValidNest2 :: ((x,f(x,x))->f x)->
                       ((x,x),f((x,x),(x,x)))->f(x,x)

data Entero a = N | E a

proy :: Entero a ->a
proy (E x) = x

instance ValidNest Entero where
        fValidNest1 N = N
        fValidNest1 (E x) = E(x,x)
        fValidNest2 k ((a,b),N) =
        E(proy (k(a,N)),proy (k(b,N)))
        fValidNest2 k ((a,b),E((m1,m2),(m3,m4))) =
        E(proy (k(a,E(m1,m2))),proy (k(b,E(m3,m4))))

par :: (a->b)->(a'->b')->(a,a')->(b,b')
par f g (x,x')=(f x,g x')

extNest :: ValidNest f =>
(f a)->((a,f(a,a))->f a)->(c->a)->(Nest c)->f a
extNest m e h NilN = m
extNest m e h (ConsN(x,xs)) =
e(h x,extNest (fValidNest1 m) (fValidNest2 e)
```

```
(par h h) xs)

suma1 :: Num a => (a,Entero(a,a))->Entero a
suma1 (x,N) = E x
suma1 (x,E(m1,m2)) = E (x+m1+m2)

sumaNest :: Num a => Nest a -> Entero a
sumaNest = extNest N suma1 id
```

Para construir `map` debemos proveer al funtor `Nest` de la estructura precisa para poder pertenecer a la clase `Valid` (observemos que, para este caso, sólo nos interesa la extensión de una función singular como es el constructor del tipo `ConsN`, hecho que se refleja en la instanciación procurada):

```
instance ValidNest Nest where
        fValidNest1 N = N
        fValidNest2 k ((a,b),xs) =ConsN((a,b),xs)

mapNest :: (a->b)->Nest a->Nest b
mapNest h = extNest NilN ConsN h
```

Análogamente podemos definir el caso para `Llist`.

```
class Valid f where
    fValid1 :: (f x)->f [x]
    fValid2 :: ((x,f[x])->f x)->([x],f [[x]])->f[x]

flatten :: [[a]]->[a]
flatten [] = []
flatten (x:xs) = x++(flatten xs)

flattenLlist :: Llist[[a]]->Llist [a]
flattenLlist NilL = NilL
flattenLlist (ConsL(l,xs)) =
ConsL(flatten l,flattenLlist xs)

proyLlist :: Llist a ->a
proyLlist (ConsL(x,_)) = x

fValid2E :: ((x,Entero [x])->Entero x)->
([x],Entero [[x]])->[x]
fValid2E k ((x:xs),N) =
(proy (k(x,N))):(fValid2E k (xs,N))
fValid2E k ((x:xs),E (y:ys)) =
(proy (k(x,E(y)))):(fValid2E k (xs,E(ys)))

instance Valid Entero where
```

```
    fValid1 N = N
    fValid1 (E x) = E [x]
    fValid2 k ([],_) = N
    fValid2 k ((x:xs),N) =
    E (fValid2E k ((x:xs),N))
    fValid2 k ((x:xs),E (y:ys)) =
    E (fValid2E k ((x:xs),E (y:ys)))

fValid2L :: ((x,Llist [x])->Llist x)->
([x],Llist [[x]])->[x]
fValid2L k ((x:xs),NilL) =
(proyLlist (k(x,NilL))):(fValid2L k (xs,NilL))
fValid2L k ((x:xs),ConsL(y:ys,NilL)) =
(proyLlist (k(x,ConsL(y,NilL)))):
(fValid2L k (xs,ConsL(ys,NilL)))

instance Valid Llist where
    fValid1 NilL = NilL
        fValid2 k (xs,zs) = ConsL(xs,zs)

extLlist :: Valid f => (f a)->((a,f [a])->f a)->
(c->a)->(Llist c)->f a
extLlist m e h NilL = m
extLlist m e h (ConsL(x,xs)) =
e(h x,extLlist (fValid1 m) (fValid2 e) (map h) xs)

mapLlist :: (a->b)->(Llist a->Llist b)
mapLlist f = extLlist NilL ConsL f

suma1 :: Num a => (a,Entero [a])->Entero a
suma1 (x,N) = E x
suma1 (x,E (l)) = E (x+(sum l))

sumaLlist :: Num a => Llist a -> Entero a
sumaLlist = extLlist N suma1 id
```

En ambos casos vemos que, la estructura debe ser tal que, dados un valor de tipo `f a` y una función de tipo `(a,(f(a,a))->f a` puede generar una función de tipo `((a,a),f((a,a),(a,a)))->f(a,a)` en el caso de `Nest` o `([a],f[[a]])->f[a]` en el caso de `Llist` que es lo que requerimos en la clase `Valid`. Como vemos, ahora sí se puede definir la suma sobre ambos tipos y obtener el `map` como instanciaciones de `fold`. Además, podemos caracterizar las funciones terminales sobre estos tipos por lo cual, como su composición, tenemos los hilomorfismos correspondientes. Como un ejemplo construimos el anamorfismos sobre `Llist`:

```
class AnaValid f where
```

```
    anaValid1 :: (f x->Bool)->f [x]-> Bool
    anaValid2 :: (f a->(a,f [a]))->f [a]->([a],f[[a]])

anaLlist :: AnaValid f =>
(f a->Bool)->(f a->(a,f [a]))->(a->b)-> f a->Llist b
anaLlist p th h y = if p y then NilL else
ConsL( h x,anaLlist (anaValid1 p) (anaValid2 th) (map h) y')
where (x,y')=th y
```

La búsqueda de una semántica que englobe a los tipos regulares y a los anidados sin perder las propiedades de automatización de aquellos es un tema que promete interesantes resultados.

El problema suscitado por la implementación de estas funciones (no derivables en el sistema Hindley-Milner) se resuelve, como hemos visto en las anteriores implementaciones, dentro de los nuevos compiladores de Haskell, como Hugs y GHC, usando signaturas de tipo de rango 2 donde la cuantificación universal se extiende a los constructores de tipo. No es válida todavía esta solución ni en CAMLLight ni en Coq aunque por razones bien diferentes: en CAMLLight el tipo es definible pero las funciones no son derivables; en cambio, en Coq contamos con polimorfismo en constructores pero no podemos definir el tipo por restricciones impuestas sobre estos.

3.13 *Ejemplos*

Consideremos la estructura dialgebraica generada por los funtores $F, G : \mathbf{T} \times \mathbf{Monoids} \rightarrow \mathbf{T}$ definido como $F(A,(B,\theta_B,m_B)) = A$, $G(A,(B,\theta_B,m_B)) = B$. Una F G-diálgebra es, entonces, un par $((A,(B,\theta_B,m_B)),\epsilon)$ siendo $\epsilon : A \rightarrow B$; entonces, dado cualquier morfismo $f : C \rightarrow A$, existe un único morfismo de monoides $ExtMon\ \theta\ m\ \epsilon\ f : C\ list \rightarrow B$, definido como

```
  let rec ExtMon
(theta:'a*'a->'a) m epsilon f=
function  []->m
|(x::xs)
->theta(epsilon(f x),ExtMon theta m epsilon f xs);;
```

de forma que $(f, ExtMon\ \theta\ m\epsilon\ f)$ es un morfismo de F G-diálgebras. La precisión de que la operación binaria `theta` es de tipo `'a*'a->'a` es necesaria para indicar que es una operación interna; de no hacerlo, el chequeador de tipos de CAML le asignaría un tipo `'a*'b->'b`. Podríamos haber creado un tipo específico de monoides para obviar esa indicación. De hecho, como el anamorfismo correspondiente tiene una una estructura más compleja de cara a indicar la conservación de la operación interna, es útil considerar tal tipo:

```
  type 'a monoid=
M
|El of 'a
|Bin of ('a monoid)*('a monoid);;
```

```
  let rec AnaMon theta rho g x=
match x with
 M->[]
 |(El(y))->[g(rho y)]
 |(Bin(y,z))->
 (AnaMon theta rho g y)@(AnaMon theta rho g z);;
```

Comprobamos aquí la versatilidad de las diálgebras para proporcionar estructuras y morfismos de orden superior al no estar supeditadas a endofuntores y poder abarcar un espectro mayor.

3.14 *Propiedades de las diálgebras libres*

3.14.1 Listas y árboles

Comenzaremos esta sección enunciando y probando propiedades del funcional *ExtList*. Estas propiedades generalizarán las leyes de fusión y de deforestación ([Meijer y otros 1991], [Meijer y Jeuring 1995]). Estas leyes, relativas a los catamorfismos, pueden establecerse de la siguiente forma:

1. Comenzamos con la ley de fusión: sean F un endofuntor sobre una categoría y (A, ξ) una F-álgebra; entonces, (sección refinicial) existe un único morfismo de F-álgebras desde la F-álgebra inicial $(\mu F, in)$ a (A, ξ) que denotamos $cata\ \xi : \mu F \to A$. Sean, además, (B, θ) otra F-álgebra y $f : A \to B$ un morfismo de F-álgebras; $cata\ \xi; f : \mu F \to B$ es un morfismo entre μF y B; pero, al ser $(\mu F, in)$ la F-álgebra inicial, entre ella y (B, θ) sólo puede existir un morfismo de F-álgebras, $cata\ \theta$; por lo tanto, $cata\ \theta = cata\ \xi; f$. Es decir, la ley de fusión permite eliminar los valores intermedios producidos por los catamorfismos.

   El siguiente teorema establece una generalización de este resultado:

   Teorema 3.16. [Ley de fusión para listas] *Consideremos los modelos de listas* $S{=}(C, D, \theta, n)$ *y* $T{=}(A, B, \epsilon, m)$*; sea* (f, g) *un morfismo de modelos entre ellos* $(f : C \to A,\ g : D \to B)$*. Tomemos un morfismo* $h{:}E \to C$*; entonces, se verifica que*

   $$(ExtList\ \theta\ n\ h); g = ExtList\ \epsilon\ m\ (h; f).$$

   Demostración. Efectivamente, $ExtList\ \theta\ n\ h$ es el único morfismo entre $E\ list$ y D que hace a $(h, ExtList\ \theta\ n\ h)$ morfismo de $FList$ Π-diálgebras; al ser $\mathbf{DAlg}(FList, \Pi)$ una categoría, $(h; f, (ExtList\ \theta\ n\ h); g)$ es un morfismo de diálgebras entre $(E, E\ list, ::, [\,])$ y (A, B, ϵ, m); pero, por la condición de diálgebra libre sobre E de $(E, E\ list, ::, [\,])$, existe un único morfismo entre ambas diálgebras que es $(h; f, (ExtList\ \epsilon\ m\ (h; g))$. Entonces, ambos morfismos deben coincidir, es decir,

   $$(ExtList\ \theta\ n\ h); g = ExtList\ \epsilon\ m\ (h; f).$$

   □

2. Ley de deforestación (o teorema de la lluvia ácida) ([Meijer y Jeuring 1995]):dadas una F-álgebra (A, ξ) y el catamorfismo $cata\ \xi : \mu F \to A$, sea $g : (F(A) \to A) \to B \to A$ un funcional de orden superior que tiene a *cata* como instanciación; entonces, para la F-álgebra inicial $(\mu F, in)$, tenemos el morfismo $g\ in : B \to \mu F$; además, podemos componer $g\ in; cata\ \xi$; pero, dada la declaración de g, ta mbién tiene sentido $g\ \xi$. La *deforestación* establece que $g\ in; cata\ \xi = g\ \xi$. Así, la deforestación permite elminar los valores intermedios consumidos por los catamorfismos.

   Igual que antes, este teorema puede generalizarse:

   Teorema 3.17. [Ley de deforestación para listas] *Sea G una función de orden superior con tipo $(A * B \to B) \to B \to C \to (C \to A) \to (C \to C) \to C \to B$ con la especificación*

$$G\ \epsilon\ m\ a\ f\ h\ x = \begin{cases} m & if\ x = a \\ \epsilon(f(x), G\ \epsilon\ m\ a\ \ f(h\ x)) & otro\ caso \end{cases}$$

   y consideremos (X, Y, ϵ, m) un modelo de listas [es decir,

$$ExtList : (X * Y \to Y) \to Y \to (A \to X) \to list\ A \to Y;$$

   entonces, cuando $B = list\ A$ y $k : A \to X$, tenemos que

$$(G\ (::)\ [\,]\ c\ f\ h); (ExtList\ \epsilon\ m\ k) =$$

$$G\ \epsilon\ m\ c\ (f; k)\ h,$$

   siendo $G\ \epsilon\ m\ c\ (h; f)\ h : C \to Y$.

   Demostración. Cabe analizar dos casos: Suponiendo que $x = c \in C$, tenemos

$$(ExtList\ \epsilon\ m\ k)(G\ (::)\ [\,]\ c\ f\ hc) =$$

$$(ExtList\ \epsilon\ m\ k)([\,]) = m.$$

   En caso de ser $x \in C$ y $x \neq c$ la igualdad es

$$(ExtList\ \epsilon\ m\ k)(G\ (::)\ [\,]\ c\ f\ hx) =$$

$$(ExtList\ \epsilon\ m\ k)((f\ x) :: (G\ (::)\ [\,]\ c\ f\ h(h\ x)) =$$
$$\epsilon(k(f\ x), (ExtList\ \epsilon\ m\ k)(G\ (::)\ [\,]\ c\ f\ h(h\ x)) =$$
$$(G\ \epsilon\ m\ c\ (f; k)\ hx)).$$

   □

Menciona Gibbons el *teorema de promoción de catamorfismos*; éste es un caso particular de un teorema más general:

Teorema 3.18. *Consideremos los tipos E, C, el modelo canónico de listas con soporte E $list$ y el modelo $(C\ list, C\ list, @, [\ ])$; sea, además, $h{:}E \to C$ un morfismo; entonces, si (A, B, ϵ, m) es un modelo de listas con $f : C \to A$, $(ExtList\ @\ [\ ]\ (map\ h)); (ExtList\ \epsilon\ m\ f) = ExtList\ \theta\ m\ map(h; f)$, donde*

$$\theta([\ ], b) = b,\ b \in B;;$$

$$\theta((x :: xs), b) = \epsilon(x, \theta(xs, b)).$$

De una forma análoga, es posible optimizar, a través de su condición de coigualador, el funcional `Ext_it_List`, obteniendo, como consecuencia, funciones sumamente eficientes. Como simple ejercicio, demostraremos el teorema de fusión.

Teorema 3.19. *Sean (A, B, ϵ, m), (C, D, θ, n) objetos de $\mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi_2})$. Sea X un objeto en $\mathbf{T}$; sean, además, (p, q) un morfismo en $\mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi_2})$ entre ambos objetos y $f : X \to A$. Consideremos los morfismos $g1, g2 : X \times List\ A \times B \to List\ A \times B$ definidos como:*

$$g1(x, l, y) = ((f\ x) :: l, y),$$

$$g2(x, l, y) = (l, \epsilon(f\ x, y).$$

Sean $f : List\ A \times B \to List\ C \times D$ definida como

$$f(l, y) = (maplist\ p\ l, q\ y).$$

Entonces,

$$f1; (ExtitList\ \theta\ n\ (f; p)) = (ExtitList\ e\ m\ f); q.$$

Demostración. En primer lugar, sea G el coigualador de $g1$ y $g2$, que, como ya hemos visto, podemos definir como $ExtitList\ e\ m\ f$. Como

$$f1(g1(x, l, y)) = f1((f\ x) :: l, y)) = (maplist\ p\ ((f\ x) :: l), q\ y)$$

y

$$f1(g2(x, l, y)) = f1((l, \epsilon(f\ x, y)) = (maplist\ p\ l, q(\epsilon(f\ x, y))) =$$
$$(maplist\ p\ l, \theta(p(f\ x), q\ y)),$$

podemos considerar H el coigualador de $g1; f$ y $g2; f$. Entonces, $f1; H$ también coiguala $g1$ y $g2$. Por lo tanto, existe un único morfismo $Q : B \to D$ tal que $G; Q = f1; H$, es decir, $(Extitlist\ \epsilon\ m\ f); Q = ExtitList\ \theta\ n\ (f; p)$. Pero, como q también satisface esa condición, debe ser $Q = q$. En efecto:

$$Q\ m = (Extitlist\ \theta\ n\ (f; p)\ [\ ]) = n = q\ m;$$

$$Q(Extitlist\ \epsilon\ m\ (f\ x) :: l) = (Extitlist\ \theta\ n\ (f; p)\ x :: l) =$$
$$= (Extitlist\ \theta\ \theta(p(f\ x, n))\ (f; p)\ l)$$

$$= (Extitlist\ \theta\ q(\epsilon(f\ x, m))\ (f;p)\ l) = Q(Extitlist\ \epsilon\ \epsilon(f\ x, m))\ f\ l).$$

Por último, veamos que $f1; H = G; q$.

$$H(f1([\,].y)) = H([\,], q\ y) = q\ y = q(G([\,], y));$$

Supongamos que $H(f1(l, y)) = q(G(l, y))$ para cada lista lista l de longitud n y cada $y \in B$. En tal caso:

$$H(f1(x :: l, y)) = H(maplist\ (p\ x) :: l, q\ y) =$$
$$H(maplist\ p\ l, \theta(p\ x, q\ y)) = q(G(l, \theta(p\ x, q\ y))) =$$
$$q(G(l, q(\epsilon(x, y))) = q(G(x :: l, y)).$$

□

Se pueden establecer resultados análogos para el tipo de los árboles de aridad variable.

Teorema 3.20. [Ley de fusión para árboles] *Sean* $S = (C, A, B, \epsilon, \delta, m)$, $T=(C', A', B', \epsilon', \delta', m')$ *dos modelos de tree y (r,p,q) un morfismo entre ellos; sea* $g{:}E \to C$ *un morfismo; entonces,*

$$(ExtTree\ \epsilon\ m\ \delta\ g); p = ExtTree\ \epsilon'\ m'\ \delta'\ (g; r).$$

Demostración. Como sabemos, los modelos de tree y sus morfismos forman una categoría; entonces, (`ExtTree` ϵ m δ g);p es un morfismo entre E *tree* y A'; pero, como el único morfismo entre ellos es `ExtTree` ϵ' m' δ' (g;r), ambas funciones son la misma. □

Teorema 3.21. [Deforestación para árboles] *Sea* G *una función de tipo* $(A * B \to B) \to B \to C \to (C * B \to A) \to (X \to C) \to (X \to X) \to (X \to C) \to X \to A$ *definida como sigue:*

$$G\ \epsilon\ m\ a\ \delta\ f\ k\ h\ x = \begin{cases} \delta(f\ a, m) & \text{if } x = a \\ delta(f(x), \epsilon(G\ \epsilon\ m\ a\ \delta\ f\ k\ h\ (h\ x), m) & \text{otro caso} \end{cases}$$

Entonces,

$$(G\ (::)\ [\,]\ a\ Nodo\ f\ k\ h); (ExtTree\ \epsilon\ m\ \delta\ l) = G\ \epsilon\ m\ a\ \delta\ (f; l)\ k\ h.$$

Demostración. Como antes, tenemos dos casos para analizar:

$$(ExtTree\ \epsilon\ m\ \delta\ l)(G\ (::)\ [\,]\ a\ Nodo\ f\ k\ h\ a) =$$
$$(ExtTree\ \epsilon\ m\ \delta\ l)(Nodo(f\ a, [\,]) =$$
$$\delta(l(f\ a), ExtList\ \epsilon\ m\ (ExtTree\ \epsilon\ m\ \delta\ l)\ [\,]) =$$
$$\delta(l(f\ a), m).$$

Cuando $x \neq a$

$$(ExtTree\ \epsilon\ m\ \delta\ l)(G\ (::)\ [\,]\ a\ Nodo\ f\ k\ h\ x) =$$

$$(ExtTree\ \epsilon\ m\ \delta\ l)(Nodo(f\ x, (G\ (::)\ [\,]\ a\ Nodo\ f\ k\ h\ (h\ x) :: [\,])) =$$
$$\delta(l(f\ x), ExtList\ \epsilon\ m\ (ExtTree\ \epsilon\ m\ \delta\ l)(G\ (::)\ [\,]\ a\ Nodo\ f\ k\ h(h\ x) :: [\,])) =$$
$$\delta(l(f\ x), \epsilon(ExtTree\ \epsilon\ m\ \delta\ l(G\ (::)\ [\,]\ a\ Nodo\ f\ k\ h(h\ x)), m) =$$
$$(G\ \epsilon\ m\ a\ \delta\ (f;l)\ k\ h\ x).$$

□

Utilizando la función `Ext_it_tree` podemos obtener similar capacidad de reducción.

3.14.2 Diálgebras libres

Podemos, ahora, establecer las reglas de optimización genéricas para las diálgebras libres:

Teorema 3.22. [Teorema generalizado de fusión.] *Sean $F, G : \mathbf{C} \times \mathbf{D} \to \mathbf{E}$ dos funtores y sean $((C, D), \phi)$, $((A, B), \epsilon)$ dos F G-diálgebras; sea, además, (p, q) un morfismo de diálgebras entre ellas. Sean, por último, $X \in \mathbf{C}$ y $f : X \to C$; entonces, siendo $((X, \mu_X), \xi_X)$ la F G-diálgebra libre sobre X,*

$$f_{XD}{}^{\phi}; q = (f; p)_{XB}{}^{\epsilon}.$$

Demostración. Al ser F y G funtores, $(f; p, f_{XD}{}^{\phi}; q)$ es un morfismo de F G-diálgebras entre la diálgebra libres sobre X y $((A, B), \epsilon)$; pero, precisamente por ser una diálgebra libre, dado el morfismo $f; p : X \to A$ existe un morfismo $(f; p)_{XB}{}^{\epsilon} : \mu_X \to B$ que verifica $(f; p, (f; p)_{XB}{}^{\epsilon})$ es el único morfismo entre ellas; por consiguiente,

$$f_{XD}{}^{\phi}; q = (f; p)_{XB}{}^{\epsilon}.$$

□

Podemos obtener un corolario estableciendo cuándo la composición de dos catamorfismos es un catamorfismo:

Corolario 3.4. *Sean $F, G : \mathbf{C} \times \mathbf{D} \to \mathbf{E}$ dos funtores y sean $((X, \mu_X), \xi_X), ((Y, \mu_Y), \xi_Y)$ F G-diálgebras libres sobre X e Y, respectivamente y sean $((Y, \mu_Y), \theta)$, $((A, B), \epsilon)$ otras F G-diálgebras. Sean, además, $p : X \to Y$, $q : Y \to A$ morfismos en $\mathbf{C}$. Entonces, si $G(q, q^{\epsilon})$ es el coigualador de ξ_Y, θ se verifica que*

$$p^{\theta}; q^{\epsilon} = (p; q)^{\epsilon}.$$

Es decir, la composición de los catamorfismos p^{θ}, q^{ϵ} es un catamorfismo.

Teorema 3.23. [Teorema generalizado de la lluvia ácida.] *Sea $H : (F(A, B) \to G(A, B)) \to (C \to A) \to D \to B$; consideremos, además, $((X, Y), \epsilon)$ una F G-diálgebra, con $f : C \to A$, $h : A \to X$ morfismos; entonces,*

$$(H\ in_A\ f); h_{AY}{}^{\epsilon} = H\ \epsilon\ (f; h).$$

Dualmente, podemos establecer reglas de optimización para las generalizaciones dialgebraicas de los anamorfismos.

Estudiaremos ahora otra clase de morfismos de orden superior. Estos son los *hilomorfismos* (comenzaremos por las listas y, posteriormente, los generalizaremos) ([Meijer y otros 1991]).

Ejemplo 3.11. *Consideramos, para ello, los funtores FList y* Π *de la sección 3.6. Sean* $((A,B),\rho)$ *una* Π *FList-diálgebra y* $((C,D),\epsilon)$ *una FList* Π*-diálgebra; entonces,* $\rho : B \to () + A * B$ *y* $\epsilon : () + C * D \to D$*. Dada la función f:*$A \to C$*, existe una función* $g : B \to D$*, definida por*

```
  let HilList  theta epsilon m g=
composition (AnaList  theta g) (ExtList  epsilon m I);;
```

Este morfismo, HilList, es una extensión del hilomorfismo descrito en [Meijer y otros 1991].

Podemos optimizarlo utilizando la ley de deforestación:

```
  let HilListOpt  theta epsilon m g b=match (theta b) with
Nil->m
|Prod(a,b')->epsilon(g a,HilListOpt  theta epsilon m g b');;
```

Un ejemplo de su uso lo tenemos en la siguiente función:

```
  let F2=HilList  Theta Epsilon 0
(function (s,i)->if (string_length s)=i then i*i else i+i)
 where
Theta=function ([],[])->Nil
|((x::xs),(y::ys))->Prod((x,y),(xs,ys))
and
 Epsilon(x,y)=x-y;;
```

La composición de hilomorfismos de listas no es, en general, un hilomorfismo de listas. Pero, se verifica la siguiente propiedad:

Teorema 3.24. *Sean* (C,D,ρ), (A,B,θ) *objetos de* $\mathbf{DAlg(\Pi, FList)}$ *(sección 3.8), y sean* (A,B,ϵ,m), (X,Y,ψ,n) *objetos de* $\mathbf{DAlg(FList, \Pi)}$*. Supongamos, además, que* $\epsilon;\theta = id_{A*B}$ *y* $\theta(m) =$`Nil`*. Entonces, dados los morfismos* $f : C \to E$, $g : E \to A$, $h : A \to F$ *y* $k : F \to X$ *(siendo E y F tipos donde construir las listas intermedias) se verifica*

$$(HilList\ \rho\ \epsilon\ m\ (f;g));(HilList\ \theta\ \psi\ n\ (h;k)) =$$
$$= (HilList\ \rho\ \psi\ n\ (f;g;h;k)),$$

es decir, la composición de dos hilomorfismos es un hilomorfismo.

Demostración. La demostración se puede ver recursivamente; sea $a \in D$ y supongamos que $\rho(a)$=`Nil`; entonces:

$$(HilList\ \theta\ \psi\ n\ (h;k))((HilList\ \rho\ \epsilon\ m\ (f;g))a) =$$

$$(HilList\ \theta\ \psi\ n\ (h;k))(m) = n$$

ya que $\theta(m) =$`Nil`; ése es, precisamente, el valor de

$$(HilList\ \rho\ \psi\ n\ (f;g;h;k))a.$$

Analicemos el caso en que $\rho(d) = (c, d')$:

$$(HilList\ \theta\ \psi\ n\ (h;k))((HilList\ \rho\ \epsilon\ m\ (f;g))d) =$$
$$(HilList\ \theta\ \psi\ n\ (h;k))(\epsilon(g(f(c)), (HilList\ \rho\ \epsilon\ m\ (f;g))d')).$$

Como $\theta(\epsilon(x,y)) = (x,y)$, ese valor coincide con

$$\psi(k(h(g(f(c))), (HilList\ \theta\ \psi\ n\ (h;k))$$

$$((HilList\ \rho\ \epsilon\ m\ (f;g))d'),$$

es decir, $(HilList\ \rho\ \psi\ n\ (f;g;h;k))(d)$. □

Para generalizar estos morfismos, debemos reflexionar acerca de qué ocurre cuando, dados dos funtores $F, G : \mathbf{C} \times \mathbf{D} \to \mathbf{E}$, consideramos, a la vez, F G-diálgebras libres y G F-diálgebras colibres.

De hecho, este uso conjunto de F G y G F diálgebras es lo que utiliza P. Freyd ([Freyd 1990]) para reducir los tipos recursivos a tipos inductivos. Considerando el funtor $T : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, contravariante en la primera coordenada, si definimos los funtores $F : \mathbf{C} \to \mathbf{C}$ por $F(A) = T(A, A)$ y $G : \mathbf{C} \to \mathbf{C}$ la idendidad, si (A, a) es una F G diálgebra y (A, a') es una G F diálgebra, (A, a, a') es lo que Freyd denomina una T-diálgebra. Un morfismo de T-diálgebras entre las F G-diálgebras (A, a) y (B, b) y las G F-diálgebras (A, a') y (B, b') es un morfismo de F G-diálgebras entre (A, a) y (B, b) y un morfismo de G F-diálgebras entre (B, b') y (A, a').

Sean, pues, D y A objetos de la categoría $\mathbf{C}$; denotemos por $((D, \mu_D), \xi_D)$, $((A, \mu_A), \xi_A)$ las F G-diálgebras libres sobre D y A, respectivamente y $((D, \mu_D), \rho^D)$, $((A, \mu_A), \rho^A)$ las G F-diálgebras colibres sobre D y A, nuevamente (supuesta su existencia). Entonces, dado el morfismo $f : D \to A$, por ser $((D, \mu_D), \xi_D)$ libre, existe un único morfismo $f_{D\mu_A} : \mu_D \to \mu_A$, verificando $\xi_D; G(f, f_{D\mu_A}) = F(f, f_{D\mu_A}); \xi_A$; además, por ser $((A, \mu_A), \rho^A)$ colibre, existe un único morfismo $f^{\mu_D A} : \mu_D \to \mu_A$, verificando $\rho^D; F(f, f^{\mu_D A}) = G(f, f^{\mu_D A}); \rho^A$.

El teorema 3.24 nos dice que sólo obtendremos hilomorfismos como composición de hilomorfismo cuando, en el paso intermedio, tengamos isomorfismos inversos uno del otro. Supongamos, entonces, que, ρ^A y ξ_A son uno inverso del otro al igual que ρ^D y ξ_D. Entonces:

$$\xi_D; \rho^D; F(f, f^{\mu_D A}); \xi_A =$$
$$\xi_D; G(f, f^{\mu_D A}); \rho^A; \xi_A;$$

por consiguiente,

$$F(f, f^{\mu_D A}); \xi_A = \xi_D; G(f, f^{\mu_D A}).$$

Es decir, $(f, f^{\mu_D A})$ es un morfismo de F G-diálgebras entre $((D, \mu_D), \xi_D)$ y $((A, \mu_A), \xi_A)$. Pero, dado el morfismo f entre D y A, $(f, f_{D\mu_A})$ es el único morfismo entre las diálgebras correspondientes. En consecuencia, se obtiene que

$$f^{\mu_D A} = f_{D\mu_A}.$$

Los hilomorfismos, composiciones de un anamorfismo y un catamorfismo, resultan ser, dados dos funtores F, G, la composición de una G F-diálgebra colibre con una F G-diálgebra libre.

En efecto, sean $((A, B), \rho)$ una G F-diálgebra, $((C, D), \xi)$ una F G-diálgebra y $f : A \to C$ un morfismo. Supongamos que existen $((C, \mu_C), \rho^C)$, la G F-diálgebra colibre sobre C y $((C, \mu_C), \xi_C)$, la F G-diálgebra libre sobre C siendo ρ^C y ξ_C inverso uno del otro; existe un único morfismo $f^{BC} : B \to \mu_C$ verificando

$$\rho; F(f, f^{BC}) = G(f, f^{BC}); \rho^C$$

y existe un único morfismo $id_{\mu_C D} : \mu_C \to D$ verificando

$$\xi_C; G(id_C, id_{\mu_C D}) = F(id_C, id_{\mu_C D}); \xi.$$

Entonces,

$$\rho; F(f, f^{BC}); F(id_C, id_{\mu_C D}); \xi = G(f, f^{BC}); \rho^C; F(id_C, id_{\mu_C D}); \xi =$$

$$G(f, f^{BC}); \rho^C; \xi_C; G(id_C, id_{\mu_C D}).$$

Por consiguiente,

$$\rho; F(f, f^{BC}; id_{\mu_C D}); \xi = G(f, f^{BC}; id_{\mu_C D}),$$

en resumen, f y $f^{BC}; id_{\mu_C D}$ son morfismos compatibles con las estructuras de G F y F G diálgebras.

Ejemplo 3.12. *Analizamos a continuación cómo se puede construir un hilomofismo sobre los árboles. Consideremos, para ello, los funtores* $H_1, H_2 : \mathbf{DAlg}(\mathbf{\Pi}, \mathbf{FList}) \times \mathbf{T} \to \mathbf{T}$ *definidos por*
$H_1((A, B, \theta), C) = C * B$,
$H_2((A, B, \theta), C) = A$.
así como los funtores $G_1, G_2 : \mathbf{DAlg}(\mathbf{FList}, \mathbf{\Pi}) \times \mathbf{T} \to \mathbf{T}$, *definidos por*
$G_1((A, B, \epsilon), C) = C * B$,
$G_2((A, B, \epsilon), C) = A$.

El hilomorfismo surge de los morfismos entre la estructura determinada por una H_1 H_2*-diálgebra y la estructura basada en una* G_1 G_2*-diálgebra. Así, si* $(((' a, ' b, \theta), ' c), \psi)$ *es una* H_1 H_2*-diálgebra,*
$(((' d, ' e, \epsilon), ' f), \delta)$ *es una* G_1 G_2*-diálgebra y* $g :' c \to ' f$ *es un morfismo, el código que permite evaluar el programa es el siguiente, haciendo uso de las funciones previamente definidas:*

```
  let HilTree theta psi epsilon m delta g=
comp (AnaTree theta psi g) (ExtTree epsilon m delta I);;
```

Como ya sabemos, `AnaTree` *produce árboles mientras que* `ExtTree` *los consume; aplicando, entonces, la deforestación, obtenemos la siguiente versión más eficiente de la función* `HilTree`*:*

```
  let rec HilTree  theta psi epsilon m delta f a=
match (psi a) with
(P(c,b)) ->
match (theta b) with
Nil->delta(f c,m)
|(Prod(b',a'))->
delta(f c,
epsilon(HilTree  theta psi epsilon m delta f b',
HilList  theta epsilon m
(HilTree  theta psi epsilon m delta f) a'));;
```

3.15 *Más ejemplos*

Consideremos los funtores $F,\ G : \mathbf{T} \times \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ definidos por $F(A,B,C) = B + A * C$, $G(A,B,C) = C$.

Una $F\ G$-diálgebra es $((A,B,C),\epsilon)$ con $\epsilon : B + A * C \to C$. La $F\ G$-diálgebra libre sobre (C,D) es la cuádrupla $((C,D,C\ list * D),\xi)$ siendo $\xi : D + C * (C\ list * D) \to (C\ list * D)$ el isomorfismo definido por:
$\xi(b) = (\,$`[]`$,b)\ si\ b \in D$;
$\xi(a,(as,b)) = (a :: as, b),\ siendo\ a \in C,\ as \in C\ list,\ b \in D$.

Entonces, dados dos morfismos $(f : C \to A, g : D \to B)$, existe un único morfismo entre $C\ list * D$ y C, $AnaLE(\epsilon,f,g)$ verificando la condición de que $(f,g,AnaLE(\epsilon,f,g))$ es un morfismo de $F\ G$-diálgebras entre $((C,D,C\ list * D),\xi)$ y $((A,B,C),\epsilon)$. La definición de $Cat(\epsilon,f,g)$ es como sigue:
$Cat(\epsilon,f,g)([\,],b) = \epsilon(g(b)),\ si\ b \in D$;
$Cat(\epsilon,f,g)(a :: as, b)) = \epsilon(f(a), Cat(\epsilon,f,g)(as,b),\ siendo\ a \in C,\ as \in C\ list,\ b \in D$.

Pero, además, dados los mismos funtores, podemos considerar cuál es, en caso de existir, la $G\ F$-diálgebra colibre sobre (C,D); en este caso, ésta es la cuaterna $((C,D,C\ list * D + C\ stream),\rho)$, donde $\rho : C\ list * D + C\ stream \to C + D * (C\ list * D + C\ stream)$ viene dado, de una forma complementaria al caso anterior, por:
$\rho($`[]`$,b) = b\ si\ b \in D$;
$\rho(a :: as, b)) = (a,(as,b)),\ siendo\ a \in C,\ as \in C\ list,\ b \in D$;
$\rho(a :: as)) = (a, as),\ siendo\ a \in C,\ as \in C\ stream$.

Entonces, dada la $G\ F$-diálgebra $((A,B,C),\theta)$, y dados los morfismos $(f : A \to C, g : B \to D)$, existe un único morfismo $Ana(\theta,f,g) : D \to C\ list * D + C\ stream$ tal que $(f,g,Ana(\theta,f,g))$ es un morfismo de diálgebras. Su definición, como cabe suponer, es dual de la del catamorfismo anterior:
$Ana(\theta,f,g)(c) = g(b),\ si\ \theta(c) = b,\ c \in C,\ b \in D$;
$Ana(\theta,f,g)(c) = (f(a), Ana(\theta,f,g)(c'),\ si\ \theta(c) = (a,c')\ siendo\ a \in C,\ c,\ c' \in C$.

La $G\ \ F$-diálgebra libre sobre el par `('a,'b)`, `(('a,'b,'a list),P)`, con `P:'a list*b->'b+'a*('a list*'b)` definida por

```
P([],y)=y,
P((x::xs),y)=(x,(xs,y)),
```

permite definir los *paramorfismos* sobre listas, ligados ([Meijer y otros 1991]) con las funciones inductivas. Una función, `f`, inductivamente definida sobre `'a list` se determina a partir de dos funciones, `g:'b→'c` y `h:'c*'a*'a list*'b→'c`, de la siguiente forma:

```
f([],y)=g(y);
f((x::xs),y)=h(f(xs,y),x,xs,y).
```

siendo `y∈'b`, `x∈'a`, `xs∈'a list`. Además, esta función queda unívocamente determinada por `g` y `h` ([Sancho 1990], pag. 149); es decir, `f` es la única función que verifica `P;K=f;[g+h]`, siendo
`K:'b+'a*('a list*'b)->'c+'c*'a*'a list*b` definida por

```
K(y)=y,
K(x,xs,y)=(f(xs,y),x,xs,y),
```

cuando `x∈'a`, `xs∈'a list`, `y∈'b`.

Por ejemplo, a partir de las funciones `g(x)=x, h(y,x,xs,l)=y@l@[x]`, se obtiene la función que invierte el orden de los elementos de una lista. En efecto,

```
 let rec rev=function ([],l)->g(l)
|((x::xs),l)->h(rev(xs,l),x,xs,l);;

 let reversion l=rev(L,[]);;
```

3.16 *Extensión del operador foldl a otros tipos algebraicos*

Es una pretensión de los últimos años generalizar el operador foldl para poder aplicar su condición de tail-recursive a otros tipos algebraicos que no sean las listas. Pero, a pesar de este amplio uso y de este intento de extensión, este funcional no ha sido estudiado tan sistemáticamente como su casi homónimo foldr. De éste se conocen tanto su fundamento como función proveniente del carácter inicial del tipo algebraico definido como punto fijo de un endofuntor como los teoremas de optimización que satisface por ese mismo carácter. Incluso se conoce cuándo un morfismo proveniente de un tipo inductivo es un foldr. No es tan vasto el conocimiento que se tiene del operador que aquí tratamos.

Hemos definido anteriormente el funcional foldl para el tipo de las listas y visto su caracterización categórica. Como función *tail recursive* es bastante eficiente en cuanto a complejidad computacional; entonces, cabe preguntarse si podemos extenderla a otros tipos y en tal caso cómo se hará. La respuesta en principio es que, en las condiciones que hemos analizado en el tipo de las listas, sólo en algunos casos de funtores *simples* tal cosa es posible. Es decir, esa generalización *sin aditamentos extras* sólo es factible en casos especialmente sencillos. En cualquier otro caso, precisaremos dar condiciones extras y, es nuestra pretensión, dar de forma genérica, esos extras.

Veremos inicialmente dos ejemplos analizando la construcción del foldl en cada uno de ellos. Posteriormente, estudiaremos otro caso donde la definición ya no es viable sin argumentos añadidos.

Ejemplo 3.13. *Consideremos* $N : \mathbf{Set} \to \mathbf{Set}$ *definido por* $N(B) = (\ ) + B$. *Entonces, como es sabido, la* N*-álgebra inicial asociada a este funtor es* $(\mathbf{N}, (O, S))$. *Tomemos* $(B, (b, \epsilon))$ *otra* N*-álgebra. Definimos las funciones* $h_1,\ h_2 : \mathbf{N} \times B \to \mathbf{N} \times B$:

$$h_1(n, b) = (S(n), b);$$

$$h_2(n, b) = (n, \epsilon(b)),$$

cuando $n \in \mathbf{N},\ b \in B$. *Entonces, el coigualador de estos dos morfismos,* $\xi(\epsilon) : \mathbf{N} \times B \to B$ *que satisface, para* $n \in \mathbf{N},\ b \in B$*la igualdad*

$$\xi(\epsilon)(S(n), b) = \xi(\epsilon)(n, \epsilon(b)),$$

queda definido por esa condición sólo considerando el caso final

$$\xi(\epsilon)(O, b) = b$$

Es decir, $\xi(\epsilon)(n, b) = \epsilon^n(b)$, [6] *aplicación* n *veces de la función* ϵ *sobre* b. *Por último, definimos*

$$foldl\ \epsilon\ b\ O = b;$$

$$foldl\ \epsilon\ b\ S(n) = foldl\ \epsilon\ \epsilon(b)\ n.$$

Ejemplo 3.14. *Sea ahora, siendo* A *un conjunto fijo,* $S_A : \mathbf{Set} \to \mathbf{Set}$, *con* $S_A(B) = A + B$. *La* S_A*-álgebra inicial asociada a este funtor es* $(\mathbf{S}, [L, I])$. *Si tomamos* $(B, [f, g] : A + B \to B)$ *otra* S_A*-álgebra, el operador foldr viene definido como sigue:*

$$foldr\ f\ g\ L(a) = f(a);$$

$$foldr\ f\ g\ I(s) = g(foldr\ f\ g\ s).$$

Es decir, resumiendo, $foldr\ f\ g\ I^n(L(a)) = g^n(a)$. *Definimos las funciones* $h_1,\ h_2 : A + \mathbf{S} \times B \to \mathbf{S} \times B$:

$$h_1(a) = h_2(a) = (L(a), f(a));$$

$$h_1(s, b) = (I(s), b);$$

$$h_2(s, b) = (s, g(b)),$$

donde $a \in A,\ s \in \mathbf{S},\ b \in B$. *El coigualador de* $h_1,\ h_2,\ \xi(f, g) : \mathbf{S} \times B \to B$ *que satisface, para* $a \in A,\ n \in \mathbf{N},\ b \in B$ *la igualdad*

$$\xi(f, g)(I(s), b) = \xi(f, g)(s, g(b)),$$

queda definido por la función [7]

$$\xi(f, g)(s, b) = \begin{cases} f(a) & si\ s = L(a) \\ g^n(a) & si\ s = I^n(L(a))\ y\ b \neq g(b') \\ g^{n+1}(a) & si\ s = I^n(L(a))\ y\ b = g(b') \end{cases}$$

[6] Huelga decir que no es preciso explicitar esta definición ni ninguna otra concerniente a estas funciones tail-recursive. Basta con su especificación para que queden perfectamente determinadas.

[7] Como ya dijimos tal definición no es necesaria

de forma que $foldl\ f\ g\ L(a) = \xi(f,g)(L(a), g(f(a)))$ *y* $foldl\ f\ g\ I(s) = \xi(f,g)(s, g(f(a)))$ *siendo* $s = I(\ldots(L(a))\ldots)$.

La definición del coigualador no es sencilla pero, vamos a comprobar que es correcta y, además, en este caso, coincide con foldr.
En efecto:

$$foldl\ f\ g\ L(a) = \xi(f,g)(L(a), g(f(a))) = f(a) = foldr\ f\ g\ L(a);$$
$$foldl\ f\ g\ I^{n+1}(L(a)) = \xi(f,g)(I^n(L(a)), g(f(a))) = g^{n+1}(a)$$
$$= foldr\ f\ g\ I^{n+1}(L(a))$$

como vimos anteriormente.

Analicemos qué ocurre con el tipo de los árboles. Observemos, como primera diferencia con respecto a los casos anteriores, que el conjunto soporte del objeto inicial (o del álgebra) aparecen dos veces en un producto. Esto va a provocar ciertas dificultades. El funtor T_A : **Set** $\to$ **Set** definido como $T_A(B) = A + B \times B$ tiene como objeto inicial $Tree\ A$, con isomorfismo $[L,T] : A + (Tree\ A) \times (Tree\ A) \to Tree\ A$. Siendo $(B, [f,g])$ una T_A-álgebra, queremos construir sendos (y diferentes) morfismos $h_1,\ h_2 : A + (Tree\ A) \times (Tree\ A) \times B \times B \to (Tree\ A) \times B$. Hay sólo un aspecto obvio y éste es que, para cada $a \in A$

$$h_1(a) = h_2(a) = (L(a), f(a)).$$

Pero, ¿cómo definimos $h_1(l,r,b_1,b_2),\ h_2(l,r,b_1,b_2)$? Utilizando que $\xi(f,g) : Tree\ A \times B \to B$ es el coigualador que buscamos y, por lo tanto debe satisfacer la igualdad $h_1;\xi(f,g) = h_2;\xi(f,g)$ definimos

$$h_1(l,r,b_1,b_2) = (T(l,r), g(b_1,b_2));$$
$$h_2(l,r,b_1,b_2) = (l, \xi(f,g)(r, g(b_1,b_2))).$$

Entonces, $\xi(f,g)(T(l,r),b) = \xi(f,g)(r, g(b_1,b_2))$, que da lugar a la siguiente definición de foldl (donde fijamos un elemento $b \in B$):

$$foldl\ f\ g\ b\ t\ =\ \xi(f,g)(t,b).$$

Una sencilla implementación de estas ideas es la siguiente (en CamlLight):

```
type 'a Tree = L of 'a|T of 'a Tree*'a Tree;;

let rec xi f g b = function
L(a)->f(a)
|T(l,r)->xi f g (xi f g (g(b,b)) l) r;;

let foldTree f g b = xi f g b;;
```

Intentemos aplicar esta pauta a otro tipo inicial como puede ser, por ejemplo, el de los árboles *imaginarios* [8]. Fijemos dos conjunto A, B y consideramos F_{AB} : **Set** $\to$ **Set**

[8] `fancier trees` es como son denominados en [Greiner 1992]

definido como $F_{AB}(X) = A + A \times (B \to X)B$. El F_A-álgebra inicial es el tipo $(Francier\ Tree_{AB}, in_{F_{AB}})$, con $in_{F_{AB}} = [L, T] : A + A \times (B \to Francier\ Tree_{AB}) \to Francier\ Tree_{AB}$.

Sin más preámbulos, notamos que debemos construir, considerando $(X, [\theta, \epsilon] : A + A \times (B \to X) \to X)$ una F_{AB}-álgebra, dos funciones diferentes $h_1,\ h_2 : A + A \times (B \to Francier\ Tree_{AB}) \times (B \to X) \to Francier\ Tree_{AB} \times X$ para, posteriormente, igualarlas.[9]

Definimos $h_1(a) = (L(a), \theta(a)) = h_2(a)$, $h_1(a, ft, x) = (T(a, ft), \epsilon(a, x))$, $h_2(a, ft, x) = (L(a), \epsilon(a, \lambda y : B.\xi(\theta, \epsilon)(ft(y), f(a))$, siendo, como es obvio, $\xi(\theta, \epsilon) : Francier\ Tree_{AB} \times X \to X$ el morfismo verificando $h_1; \xi(\theta, \epsilon) = h_2; \xi(\theta, \epsilon)$. En tal caso, se verifica que

$$\xi(\theta, \epsilon)(T(a, ft), \epsilon(a, x)) = \xi(\theta, \epsilon)(L(a), \epsilon(a, \lambda y : B.\xi(\theta, \epsilon)(ft(y), f(a)))$$

con $\xi(\theta, \epsilon)(L(a), y0) = y0$, $y0 \in X$. Por lo tanto, declarando $fold\ \theta\ \epsilon\ L(a) = \xi(\theta, \epsilon)(L(a), f(a)) = f(a)$,

$$\begin{aligned} fold\ \theta\ \epsilon\ T(a, ft) &= \xi(\theta, \epsilon)(T(a, ft), f(a)) = \\ &\xi(\theta, \epsilon)(L(a), g(a, \lambda y : B.\xi(\theta, \epsilon)(ft(y), f(a)))) = \\ &g(a, \lambda y : B.\xi(\theta, \epsilon)(ft(y), f(a))). \end{aligned}$$

Por ejemplo, el equipo de Belleanni ha generado la siguiente función foldl (la escribimos en Haskell) para el tipo de los árboles binarios:

```
data Bt a = Leaf a | Tree (Bt a) (Bt a)
foldltbt :: (a -> a) -> (b -> a -> a) -> (Bt b) -> a -> a
foldlbt f g (Leaf x) m = g x m
foldlbt f g (Tree l r) m =
        foldlbt f g r (foldlbt f g l (f m))
```

Vemos que es necesario añadir valores (`f:a -> a, g:b -> a -> a, m::a`) para poder hacer la definición y que ésta, además, no es sino una reconversión del operador foldr convencional, ya que no proviene de ninguna estructura sino que es un morfismo ad-hoc.

3.17 *Análisis de unfoldl sobre las* streams

Análogamente al caso inicial donde hemos definido el operador foldl, es posible definir un operador unfoldl cuando actuamos con un tipo terminal y, entonces, analizar su interpretación. Uno de los casos paradigmáticos de tipo final es el de las *streams*, basado en el funtor $S_A : \mathbf{Set} \to \mathbf{Set}$ definido por $S_A(B) = A \times B$, siendo A un conjunto fijo. En tal caso, $(Stream\ A, Inf)$ es el mayor punto fijo de S_A, siendo $Inf : A \times Stream\ A \to Stream\ A$ su isomorfismo constructor y $Split : Stream\ A \to A \times Stream\ A$ el correspondiente destructor [10] (inverso del anterior). Sea $(B, \theta : B \to A \times B)$ una S_A-coálgebra; existe un único homomorfismo de coálgebras entre (B, θ) y $(Stream\ A, Inf)$ denominado en la literatura sobre el tema $unfold\ \theta$ y que viene definido por

[9] Como podemos notar, surge un problema y es qué elección hacer de entre todas las posibles. Es aquí donde subyace en este momento el quid de la cuestión analizado y estudiado en muchos ámbitos. Nosotros, por asemejar el resultado al que se obtendría usando foldr (en el fondo, en determinadas circunstancias de asociatividad, etc, ambas funciones computan el mismo resultado) hemos realizado la elección indicada.

[10] También llamado observador, aplicación de transición [Jacobs and Rutten 1997]

$unfold\ \theta\ b\ =\ Inf(a, unfold\ \theta\ b')$ donde $(a, b') = \theta(b)$.

¿Quién y qué es, si existe, unfoldl en este contexto? Podemos responder a ambas cuestiones con el siguiente:

Teorema 3.25. *Consideremos* $h_1,\ h_2 : Stream\ A \times B \to A \times Stream\ A \times B$ *definidas por*

$h_1(st, b) = (HS\ st, st, b)$ *donde* $(HS\ st, TS\ st) = Split(st)$;

$h_2(st, b) = (a, st, b)$ *siendo* $(a, b') = \theta(b)$.

Sea $\rho(\theta) : B \to Stream\ A \times B$, *definida por*

$\rho(\theta)(b) = (Inf(a_1, Inf(a_2, \ldots), b)$

siendo $(a_1, b_1) = \theta(b),\ (a_2, b_2) = \theta(b_1),\ \ldots$

Entonces, $(B, \rho(\theta))$ *es el igualador de* h_1 *y* h_2.

Demostración. Hemos de ver que, para cada $b \in B,\ h_1(\rho(\theta)(b)) = h_2(\rho(\theta)(b))$:

$$h_1(\rho(\theta)(b)) = h_1(Inf(a_1, Inf(a_2, \ldots), b) = (a_1, Inf(a_1, Inf(a_2, \ldots), b).$$

Por otra parte,

$$h_2(\rho(\theta)(b)) = h_2(Inf(a_1, Inf(a_2, \ldots), b) = (a_1, Inf(a_1, Inf(a_2, \ldots), b)$$

puesto que $(a_1, b_1) = \theta(b)$.

Así, se satisface la primera condición de igualador. Analicemos su carácter de objeto final dentro de la categoría de los igualadores. Sea, pues, (C, g) otro igualador de $h_1,\ h_2$, es decir, para cada $c \in C,\ h_1(g(c)) = h_2(g(c))$. Definimos $h : C \to B$ tal que $h(c) = b$ siendo $(st, b) = g(c)$. Veamos que para cada $c \in C, \rho(\theta)(h(c)) = g(c)$. Por la definición de $\rho(\theta),\ \rho(\theta)(h(c)) = (Inf(a_1, Inf(a_2, \ldots)), h(c)) = (Inf(a_1, Inf(a_2, \ldots)), b)$. Entonces, la segunda componente, b, coincide en ambas expresiones. Sabemos, además, que $(a_1, b_1) = \theta(b),\ (a_2, b_2) = \theta(b_1), \ldots$ Como $h_1(g(c)) = h_2(g(c)),\ h_1(st, b) = h_2(st, b)$, es decir, igualando las definiciones de h_1 y h_2 sobre ese par tenemos $(HS\ st, st, b) = (a_1, st, b)$. Razonando del mismo modo sobre el resto de los elementos de la cola de la *stream* vemos que, efectivamente, también la primera coordenada coincide en ambas expresiones. La unicidad de la función h queda reflejada por su construcción y condición. Queda por determinar quién es $unfoldl\ \theta$. Pues bien, si $b \in B, unfoldl\ \theta\ b = \Pi_1(\rho(\theta(b)))$, coincidente, en este caso, con unfold. □

3.18 *Unfoldl definido en otros tipos algebraicos*

Ejemplo 3.15. *Retomemos el funtor N del Ejemplo 3.13. Sea* (B, θ) *una N-coálgebra; entonces, sean* $h_1,\ h_2 : \mathbf{N} \times B \to (\ ) + \mathbf{N} \times B$ *definidos como:*

$h_1(n, b) = h_2(n, b) = (\ )$ *si* $\theta(b) = (\ )$;

$h_1(n, b) = (pred\ n, b;)$

$h_2(n, b) = (n, b')$ *con* $b' = \theta(b)$. *La función* $\rho(\theta) : B \to \mathbf{N}$ *definida como*

$\rho(\theta)(b) = (O, b)$ *si* $\theta(b) = (\ )$; $\rho(\theta)(b) = (O, b_k)$ *siendo* $b_1 = \theta(b),\ b_2 = \theta(b_1), \ldots \theta(b_k) = (\ )$ *es el igualador de* $h_1,\ h_2$.

Observemos que en estas computaciones con tipos terminales (infinitos) debemos hacer evaluaciones lazy para no incurrir en errores, ya que, como se afirma en [Gibbons 1994], el uso de semánticas no estrictas (como ocurre en el caso de Haskell) permite que el tipo recursivo generado por el endofuntor correspondiente sea, a la vez, su menor y mayor punto fijo.

4 Aplicaciones de la Optimización

4.1 *Introducción*

Éste es un capítulo esencialmente de programación. Presentamos y resolvemos tres problemas atractivos (alguno con un gran interés práctico como puede ser el primero [Franco 1996]) donde ponemos en juego las estrategias teóricas planteadas en el capítulo anterior tanto en cuanto a las funciones a utilizar como a las optimizaciones oportunas.

El primer problema que analizamos es el del cálculo del número cromático de un grafo, importante de cara a su aplicación en la resolución de sistemas sparse surgidos en la discretización de ecuaciones diferenciales.

El segundo problema pretende la resolución (funcional) del famoso problema P-S de McCarthy.

El tercero calcula el spanning-tree de un grafo.

4.2 *Cálculo del número cromático de un grafo*

Iniciamos esta sección con la definición de los conceptos necesarios.

Un **grafo** `G`=(S,A) consta de un conjunto S de *nodos* y un conjunto A de *aristas*, subconjunto de S×S. Los nodos pertenecientas a una arista se denominan *nodo inicial* y *nodo final*. Una arista determina dos nodos *adyacentes*. El *grado* de un nodo es la cantidad de nodos adyacente a él. Un conjunto de aristas uniendo dos nodos es denominado un *camino*. Un grafo es *conexo* si, para cada par de nodos peretenecientes a S, existe una camino uniéndolos. Un *circuito* en un grafo es un camino donde los grafos inicial y final coinciden. Un grafo es *simple* si ningún nodo aparece en él más de una vez; un circuito es *simple* si ningún nodo, salvo el inicial-final, aparece más de una vez y este nodo inicial-final no vuelve a aparece a los largo del circuito. Un grafo es *libre de circuitos* si no contiene ningún circuito simple.

El grafo `G'`=(S',A') es un *subgrafo* del grafo G si S'⊂S y A'⊂A. Un *arbol* es un grafo conexo libre de circuitos.

Una *función de transición* es una función de tipo S→S list, adjudicando a cada nodo sus nodos adyacentes. Dado un grafo, su número cromático es la menor cantidad de colores precisos para etiquetar todos sus nodos de forma que dos nodos adyacentes tengan coloraciones diferentes. Es sabido [Even 1979] que no hay ningún algoritmo óptimo para la coloración de un grafo; de hecho, éste es un problema **NPC**, es decir, no polinómico completo. A pesar de ello, el más recomendado [aunque no sea óptimo] consiste en seleccionar primeramente los nodos con mayor grado.

Dado un grafo por su función de transición, el algoritmo que usamos es como sigue:

1. Calculamos una sublista con los nodos de mayor grado; el primer elemento de esta lista, x, será el pivote de la primera sublista de coloración, $l_1 = [x]$;

2. hallamos la lista de nodos no adyacentes a la lista l_1 y añadimos a ésta el nodo que resulta de aplicarle la regla anterior;

3. Continuamos el proceso descrito en 1 y 2 hasta que la lista de nodos no adyacentes a l_1 esté vacía;

4. Con los vértices restantes, repetimos los pasos 1-2-3 formando otra sublista l_2 y así sucesivamente;

5. Repetimos el proceso hasta completar todos los vértices del grafo. A los nodos de la lista l_i les asignamos el color i.

La implementación de este algoritmo es como sigue [construimos el tipo `nodos` como un tipo suma para poder aplicar pattern-matching]:

```
type nodos=X1|X2|X3|X4|X5|X6|X7|X8|X9|X10|X11
|X12|X13|X14;;

let nodes=[X1;X2;X3;X4;X5;X6;X7;X8;X9;X10;X11;
X12;X13;X14];;

let p l=match l with X1->
[X3;X5;X8;X10;X12;X14]|X2->[X4;X7;X9;X11;X12]
|X3->[X1;X4;X7;X8;X9;X10;X11;X13]
|X4->[X2;X3;X5;X6;X7;X8;X9;X10;X11;X12;X13]
|X5->[X1;X4;X6;X8;X10;X11]
|X6->[X4;X5;X7;X9;X10;X13;X14]
|X7->[X2;X3;X4;X6;X7;X8;X9]
|X8->[X1;X3;X4;X5;X9;X11;X13;X14]
|X9->[X1;X3;X4;X5;X8;X10]
|X10->[X2;X3;X5;X6;X8;X13;X14]
|X11->[X1;X2;X3;X4;X6;X7;X8;X9;X10;X11]
|X12->[X1;X3;X13;X14]
|X13->[X1;X2;X6;X10;X11;X12]
|X14->[X1;X3;X4;X5;X10;X13;X14]
;;
```

Hemos introducido los nodos y las aristas, éstas por medio de la función de transición. En lo que sigue hacemos uso continuado de la función `ExtList` con la consiguiente ventaja en tiempo de computación para calcular el grado de cada vértice junto con la lista de vértices adyacentes:

```
let no_pert y = ExtList (fun (x,z)->(x<>y)&z) true I;;

let pert y = ExtList (fun (x,z)->(x=y) or z) false I;;

let rec elimina l = ExtList (prefix @) []
(fun x->if pert x l then [] else [x]);;
```

```
let head l=match l with []->[]|(x::xs)->[x];;

let grado n=(n,comp p longitud n);;

let gr=ExtList (prefix (::)) [] grado;;

let s n=comp p (ExtList suma 0 (comp p (ExtList suma 0
(comp (fun x->if x=n then [] else [x]) (fun x->1))))) n;;

let ady n=(n,s n);;

let sr=ExtList (prefix (::)) [] ady;;

let union(x,y)=x or y;;

let menor y=ExtList (union) false (fun x->y<x);;

let mayor y=ExtList (union) false (fun x->y>x);;

let igual y=ExtList (union) false (fun x->y=x);;

let rec maximo l r k=match k with []->r
|((y,x)::xs)->if menor x l then maximo l r xs else
if mayor x l then maximo [x] [y] xs else
maximo (x::l) (r@[y]) xs;;

let max l=maximo [] [] (gr l);;

let pivote l=if longitud l=1 then head l else
head (maximo [] [] (sr (max l)));;
```

El último paso del proceso es calcular el número cromático, es decir, la cantidad mínima de colores [según este algoritmo] precisa para su coloreado:

```
let rec color l=if l=[] then [] else
(pivote l)@(color (elimina ((pivote l)
@((ExtList append [] p)(pivote l))) l));;

let  rec coloracion l=if l=[] then [] else
(color l)::(coloracion (elimina (color l) l));;

let numero_cromatico=composition (coloracion list_lenght);;
```

En el caso que proponemos, obtenemos la siguiente respuesta:

```
#coloracion nodes;;
```

```
- : nodos list list
- = [[X4; X14]; [X11; X5; X13]; [X3; X6; X2];
[X8; X7; X10; X12]; [X9]; [X1]]

#numero_cromatico nodes;;
- : int
- = 6
```

4.3 *El problema P-S de McCarthy*

Otro ejemplo donde la aplicación conjunta de las leyes de fusión y de la lluvia ácida permite una enorme simplificación es el de la implementación del problema P-S de McCarthy, cuyo enunciado es como sigue:

un árbitro elige dos números naturales distintos entre 2 y b, b un número cualquiera [mayor que 2]; el árbitro calcula su producto y su suma, los escribe en dos papeles distintos y entrega el papel con el producto a la persona PROD y el papel con la suma a la persona SUM; dichas personas, después de leer cada una su papel, mantienen la siguiente conversación:

PROD: No puedo adivinar tu suma.

SUM: Ya lo sabía; yo tampoco puedo adivinar tu producto.

PROD: Entonces ya sé tu suma.

SUMA: En ese caso, yo también conozco tu producto.

El problema consiste en hallar los dos números elegidos por el árbitro o, en su defecto, el valor de su producto y de su suma.

El programa ya optimizado es el siguiente (donde usamos algunas de las funciones previamente construidas):

```
 let pertenece a=ExtList (fun (x,y)->x or y)
false (fun x->x=a);;

 let rec rad n m=if (m*m<=n)&((m+1)*(m+1)>n)
then m else rad n (m+1);;

 let raiz n=rad n 1;;

 let y(x,z)=x&z;;

let lista_vacia = function
[]->true
|(x::xs)->false;;

let tail = function []->[]
|(x::xs)->xs;;
```

```
 let lista inicio=G (fun (x,y)->y@[x]) []
(fun x->x) inicio (fun x->x-1);;

 let factores p=G (fun (x,y)->if x=(0,0)
then y else x::y) []
(fun x->if p mod x=0 then (x,p/x)
else (0,0)) (raiz p) (fun n->n-1) (p/2);;

 let s_f p=G (fun (x,y)->if x=0 then y else x::y) []
(comp (fun x->if p mod x=0 then (x,p/x) else (0,0))
(fun (x,y)->x+y))
(raiz p) (fun n->n-1) (p/2);;

 let productos_incon s=G cons []
(comp (fun x->(x,s-x)) (fun (x,y)->x*y))
(s/2) (fun n->n-1) (s-2);;

 let v1 p=if (longitud (factores p)=1) then []
else s_f p;;

 let card s=if (s=2 or s=3 or s=4) then false
 else
 G y true
(comp (comp (comp (fun x->(x,s-x)) (fun (x,y)->x*y))
factores) (fun x->(longitud x)>1))
(fun x->x=(s/2)) (fun n->n-1) (s-2);;

let filtro = g append []
(function x->if (card (head x))
then [head x]
else [])
(function x->(lista_vacia x))  tail;;

let lista_posibles n = filtro (lista 5 n);;

 let v2_1 s=G append []
(comp (comp (fun x->(x,s-x)) (fun (x,y)->x*y))
(fun x->(if pertenece s (v1 x) then [x] else [])))
(fun x->x=(s/2)) (fun n->n-1) (s-2);;

 let v2 s=if (card s)&((longitud (productos_incon s))>1)
then v2_1 s else [];;

 let v3_1 p=ExtList append []
(fun x->(if pertenece p (v2 x) then [x] else []));;
```

```
 let v3 p=if (longitud (v3_1 p (v1 p)))=1 then
(v3_1 p (v1 p)) else [];;

 let rec v4_1 s=ExtList append []
(fun x->if pertenece s (v3 x) then [x] else []);;

 let v4 s=if (longitud (v4_1 s (v2 s)))=1
then  (v4_1 s (v2 s)) else [];;

let solucion_PS n = ext append []
(function s-> if (lista_vacia (v4 s))
then []
else
[head (v4 s),s]) (lista_posibles n);;


let rec suma s = function []->[]
|(x::xs)->if (mas(x)=s) then x::(suma s xs) else suma s xs;;

let rec solucion_ref = function []->[]
|(x::xs)->
(suma (snd x) (factores (fst x)))::(solucion_ref xs);;

let solucion_referee n = (solucion_ref (solucion_PS n));;
```

El objetivo de cada una de las funciones anteriormente construidas es el que sigue:

1. `factores` calcula las factorizaciones formadas por dos factores propios del número a evaluar;
2. `s_f` halla la suma de las factorizaciones referidas en el ítem anterior;
3. `productos_incon n` halla el producto de los dos sumandos distintos (a partir de 2) en que se puede descomponer n;
4. `v1` genera una lista vacía con aquellos números que sólo se pueden factorizar de una forma como producto de dos factores propios y, en otro caso, crea la lista con todas las sumas de esas diferentes posibilidades;
5. `card s` determina si el producto de cada descomposición de s formada por dos sumandos da como resultado un número con, al menos, dos factores;
6. `v2_1 s` genera una lista indicando qué productos se descomponen en dos sumandos diferentes en las condicione anteriores satisfaciendo la condición establecida en v1 (es decir, cuando la lista resultante no es la vacía.

Las otras funciones involucradas en la resolución son aplicaciones bastante obvias en su significado de las anteriormente descritas.

La función `solucion_PS` nos indica la lista de pares (producto,suma) verificando las condiciones requeridas mientras que `solucion_referee` nos aporta la lista con los números que el árbitro ha elegido. Utilizando la deforestación podemos optimizar el programa resultando la siguiente función solución:

```
let solucion_PS_optimizada  n = g append []
(comp (function x->if (card (head x)) then
[head x]
else [])
(function s-> if (lista_vacia (v4 (head s)))
then []
else [(head (v4 (head s))),(head s)]))
(function x->(lista_vacia x))
 tail (lista_posibles n);;
```

La primera solución computada es el par (52, 17) reflejando, respectivamente, el producto y la suma de los dos números solución, siendo, por consiguiente, los números elegidos por el árbitro 4 y 13.

Una función que nos permite analizar si un determinado par es o no solución del problema es la siguiente:

```
let es_solucion = function  n1->function n2->
(longitud (v4 (n1+n2))=1)&(longitud (v3 (n1*n2))=1);;
```

4.4 *Generación de los spanning trees de un grafo conexo*

Retomamos los conceptos definidos en la sección 4.2. Además, necesitamos añadir la siguiente noción: un subgrafo de un grafo G que contiene todos los nodos de G y es, además, un árbol, se llama un *spanning tree* de G.

Pretendemos, entonces, obtener todos lo spanning trees de un grafo conexo. Para ello, seguiremos el siguiente proceso:

1. Crear un tipo para representar el grafo;
2. Representar ese grafo a través de su función de transición;
3. Definir una función que nos permita filtrar aquellos elementos de una lista verificando una cierta condición booleana;
4. Establecer todos los caminos que se inician en un nodo fijado previamente;
5. Borrar todos los caminos que no sean maximales.

Dependiendo de si usamos modelos de listas o modelos de arbol, reuniremos esos caminos bien en una lista de listas bien en una lista de arbols.

Iniciamos el trabajo usando modelos de listas; el criterio utilizado en la construcción de las funciones es el ya citado en el la sección 1.2:

```
let append(x,y)=x@y;;

let pert x=ExtList E false K where E(z,y)=z or y
and K z=(z=x);;

let filter p=ExtList append [] K where K x=if p x
then [x] else [];;

 let elimina L=filter p where p x=(pert x L=false);;

 let rec paths y l=ExtList append [] K where
 K x=(map (cons y) (paths x (l@[y])
 (elimina (l@[y]) (P x))));;

 let tree y=paths y [] (P y);;

 let TREE=ExtList (fun (x,y)->x::y) [] tree;;

 let filtro=ExtList append [] (fun x->if list_length x=
list_length nodes then [x] else []);;

 let SPAN=ExtList append [] filtro;;

 let spanning_tree=comp (TREE) (SPAN);;
```

De nuevo, aplicando deforestación o fusión, podemos simplificar este operador:

```
 let SPANNING_TREE=ExtList append [] (comp (tree) (filtro));;
```

De hecho, revisando el algoritmo de la obtención de caminos maximales, podemos obtener una versión más eficiente:

```
 let no_pert y=ExtList E true (fun x->x)
where E(x,z)=(x<>y)&z;;

 let rec no_nul=fun []->[]|(x::xs)->if x=[]
then (no_nul xs) else x::(no_nul xs);;

 let rec PATHS max n y l=ExtList append [[]] K where
 K x=if (n+1=max) then
 (if no_pert x (l@[y]) then [l@[y]@[x]]
 else [])
 else
 (if no_pert x (l@[y]) then
```

```
 (PATHS max (n+1) x (l@[y]) (p x))
 else []);;

 let arbol_max y=
no_nul(PATHS (list_length nodes) 1 y [] (p y));;

 let spanning_tree=ExtList append [] arbol_max;;
```

Hasta aquísólo hemos usado listas; a continuación expresamos este programa por medio de modelos de árbol. Explotamos, con este fin, las funciones previamente definidas.

```
 let arbol_of_element x=Nodo(x,[]);;

 let rec select=fun (Nodo(x,[]))->[]
|(Nodo(x,y::ys))->
(if (arbol_height y)=((list_length nodes)-1) then
[Nodo(x,[y])] else [])@(select (Nodo(x,ys)));;

 let rec path_of_node  L x=
Nodo(x,map (path_of_node (x::L)) (el (x::L) (P x)));;

 let paths=Tree_ExtList append [] (fun (x,y)->(select x)@y)
(fun x->path_of_node [] x);;

 let span x=paths (arbol_of_element x);;

 let spanning_tree=ExtList append [] span;;
```

Nuevamente, es posible obtener un algoritmo más eficiente; para ello, sólo precisamos añadir un nodo singular y aislado *inicio* únicamente como referencia de un principio:

```
 let rec no_pert x (Nodo(y,xs))=
(x<>y)&(ExtList E true (no_pert x) xs) where
E(y,z)=y&z;;

 let rec path_of_node max n L x=
if n=max then
(if no_pert x L then
(if L=Nodo(inicio,[]) then
[Nodo(x,[])] else [Nodo(x,[L])])
 else [])
else
(if no_pert x L then
(if L=Nodo(inicio,[]) then
ExtList (append) [] (path_of_node max (n+1)
(Nodo(x,[]))) (p x) else
```

```
ExtList (append) [] (path_of_node max (n+1)
(Nodo(x,[L]))) (p x))
else []);;

 let PATH x=path_of_node (list_length nodes) 1
(Nodo(inicio,[])) x;;

 let spanning_tree=ExtList append [] PATH;;
```

Aplicando este algoritmo a un grafo conexo suficientemente pequeño para poder obtener todas las respuestas tenemos el siguiente resultado:

```
type Nodes=x1|x2|x3|x4|x5;;

let nodes=[x1;x2;x3;x4;x5];;

let p=fun x1->[x1;x2;x4]|x2->[x1;x4]|x3->[x1;x4;x5]
|x4->[x1;x3]|x5->[x2;x4];;

spanning_tree nodes;;
- : Nodes list list
- = [[x1; x2; x4; x3; x5]; [x1; x4; x3; x5; x2];
[x2; x1; x4; x3; x5];[x3; x5;x2; x1; x4];
[x3; x5; x2; x4; x1]; [x3; x5; x4; x1; x2];
[x4; x3; x5; x2; x1]; [x5; x2; x1; x4; x3];
 [x5; x2; x4; x3; x1];[x5; x4; x3; x1; x2]]
```

5 Categorías y Coq

5.1 *Introducción*

Introducimos ahora una metodología para construir programas sobre tipos de datos definidos inductiva o coinductivamente (cotipos). Esta forma de introducir los tipos de datos aparece en un contexto donde puedan coexistir las construcciones de nuevos objetos (parte conjuntista) y sus propiedades (parte proposicional). Uno de estos contextos es Coq, una implementación del Cálculo de Construcciones Inductivas (CIC) de Gerard Huet, realizada en el INRIA por [Dowek 1998]. Este sistema es un probador automático de teoremas dirigido por objetivos (*goal-directed*) y por tácticas (*tactic-driven*) en el estilo del LCF [Gordon y otros 1979], que permite la verificación formal de programas e, igualmente, su generación automática. También tiene soporte automático para la búsqueda de pruebas; igualmente, podemos definir tipos inductivamente de forma que los constructores de las pruebas inductivas y las funciones pueden derivarse automáticamente.

Nuestro objetivo consiste en mostrar la potencia del constructor de programas asociado a un tipo inductivo, y utilizarla para definir programas directamente a partir de sus especificaciones. Demostramos los teoremas de fusión y deforestación cuyo uso permite simplificar los morfismos entre estructuras homogéneas (tanto inductivas como coinductivas). Vemos una generalización del teorema de fusión para tipos inductivos dependientes de un parámetro. Por último, implementamos el conjunto de las funciones primitivo recursivas. En ese contexto vemos que la función de Ackermann, no primitivo recursiva, es un catamorfismo. Sin embargo, cuando estos se restringen [Gregorio y otros 1995] ya no es un catamorfismo generalizado.

5.2 *Tipos y Cotipos*

Hablamos en la introducción de este capítulo de tipos inductivos y tipos coinductivos. Su significado lo precisamos a continuación.

Definición 5.1. *Un tipo* **inductivo** *es el álgebra inicial de un endofuntor cocontinuo. Por consiguiente, el menor punto fijo de este funtor. Un tipo* **coinductivo** *es la coálgebra final de un endofuntor continuo. Por lo tanto, el mayor punto fijo del funtor.*

En Coq estos tipos pueden definirse y, en el caso de los tipos inductivos, las funciones correspondientes a su inicialidad son derivadas automáticamente por el sistema.

5.3 *Listas y árboles revisitados*

Así, por ejemplo, el tipo de las listas será definido inductivamente; en cambio, surgen problemas que serán mencionados más adelante para definir el tipo de los árboles generales como inductivo, por ello su declaración será coinductiva. Para determinar el tipo de las listas actuamos como hemos indicado al principio: tras fijar un objeto A de una categoría subyacente, $\mathbf{C}$, con objeto terminal (), productos y coproductos finitos, se define el funtor F_A asignando a cada objeto B de $\mathbf{C}$ el objeto $F_A(B) = () + A * B$. En tal caso, $\mu_{F_A} = list\ A$, e $in_F = [Nil; Cons]$, tal como ya hemos visto. En Coq utilizamos la palabra reservada `Inductive` para precisar que hacemos una definición inductiva:

```
Inductive list [A:Set]:Set :=
Nil:(list A) | Cons:A->(list A)->(list A).
```

`Nil` y `Cons` son denominados los *constructores* del tipo inductivo.

Para definir el tipo de los árboles, utilizamos el funtor G definido como $GA = A * listA$; entonces, la coálgebra terminal es $(tree\ A, out_G)$, con $out_G : tree\ A \to A * list(tree\ A)$ también un isomorfismo (su inverso, $Nodo : A * list(tree\ A) \to tree\ A$ permite construir los términos de $tree\ A$). La definición de este cotipo en Coq está precedida de la palabra `CoInductive`:

```
CoInductive tree [A:Set]:Set :=
Nodo:A->(list (tree A))->(tree A).
```

Hemos de definir, posteriormente, los dos *destructores* de este cotipo (que representan la función out_G anterior), que permiten obtener, a partir de un elemento de `tree A`, sus componentes respectivos en `A` y en `list (tree A)`.

Hay una diferencia sustancial entre el tratamiento de los tipos inductivos y los tipos coinductivos en Coq: como ya dijimos, tras la definición de los primeros el sistema genera automáticamente las funciones que provienen de su condición de inicial. No ocurre lo mismo con los cotipos. Las funciones que determinan su condición de terminal debe construirlas el programador. Análoga situación tenemos en el caso de las demostraciones: en las inductivas debemos apoyarnos en los diferentes casos que devienen de sus constructores mientras que en las coinductivas no tenemos tal posibilidad. Después de la construcción inductiva de las listas el sistema informa de que ha incorporado al entorno las siguientes definiciones:

```
list_ind is defined
list_rec is defined
list_rect is defined
list is defined
```

Por contra, la definición del cotipo de los árboles sólo añade al entorno la siguiente construcción:

```
tree is defined
```

Naturalmente, cabe la posibilidad de saltarse estas limitaciones estructurales y definir los árboles como un tipo (tal como se ha hecho en [Freire y Blanco 1996]):

```
Inductive tree2 [A:Set]:Set :=
Nodo2:A->(list (tree2 A))->(tree2 A).
```

Automáticamente, el sistema genera las funciones correspondientes a la definición inductiva:

```
tree2_ind is defined
tree2_rec is defined
tree2_rect is defined
tree2 is defined
```

Veremos más adelante que estas funciones no son eficientes pues prescinden en cierto grado del proceso recursivo generado por `list (tree A))`. Analizaremos una forma de solventar ese obstáculo por medio de los tipos mutuamente inductivos, estrategia que permite resolver esas limitaciones del automatismo propiciado por Coq.

5.4 *Automatización de funciones en* Coq

Vamos a analizar la rémora que constituye el no contar con las funciones terminales en el caso de los tipos coinductivos y ver cómo puede paliarse ese problema. Para ello, trabajamos con dos casos paradigmáticos dentro de cada una de las clases : el coproducto como tipo inductivo y el producto como tipo coinductivo. Para definir el coproducto (o suma) partimos de su representación dialgebraica: entonces, aquél proviene de los funtores $F,\ G : \mathbf{C} \times \mathbf{C} \times \mathbf{C} \to \mathbf{C} \times \mathbf{C}$ definidos por $F(A,B,C) = (A,B)\ G(A,B,C) = (C,C)$. Entonces, dada una FG-diálgebra $((C,D,E),\xi)$ con $(p,q) : (C,D) \to (E,E)$, para cada par de objetos $A,\ B\ \in \mathbf{C}$ y para cada par de morfismos $f : A \to C,\ g : B \to D$ existe un único morfismo $[f,g] : A + B \to E$ definido por $[f,g](inA\ a) = p(f(a)),\ [f,g](inB\ b) = q(g(b))$, siendo $a \in A, b \in B$.

```
Inductive coprod [A,B:Set]:Set :=
inA:A->(coprod A B)
|inB:B->(coprod A B).
```

El sistema responde

```
coprod_ind is defined
coprod_rec is defined
coprod_rect is defined
coprod is defined
```

Al ser éste un tipo inicial el sistema nos provee, automáticamente, de las funciones que lo tienen como dominio. Por ejemplo,

```
Print coprod_rec.
coprod_rec =
[A,B:Set]
 [P:(coprod A B)->Set]
  [f:(y:A)(P (inA A B y))]
```

```
[f0:(y:B)(P (inB A B y))]
 [c:(coprod A B)]Cases c of
                   (inA x) => (f x)
                 | (inB x) => (f0 x)
                 end
```

Analicemos qué derivación ha sido realizada: en primer lugar, para cada elemento del coproducto, se determina un conjunto; seguidamente, el sistema, basándose en los constructores, determina la necesidad de dos funciones, una correspondiente al constructor `inA`, parametrizada sobre los elementos sobre los que éste actúa, es decir, debemos, dado un elemento `y` de `A`, decir qué elemento de `P (inA A B y)` le corresponde y otra, análogamente construida, correspondiente al segundo constructor. Entonces, dado un objeto del tipo `coproducto`, dependiendo de cuál sea su constructor, será pasado a una u otra función.

No hay, a priori, obligación alguna de conocer el carácter inicial o final de un tipo, de forma que cualquiera puede ser construido tanto inductiva como coinductivamente. La diferencia, a posteriori, radica en la utilidad que obtengamos de tal declaración. Por ejemplo, no hay problema en declara como inicial el tipo producto:

```
Inductive prod [A:Set; B:Set] : Set :=
pair : A->B->(prod A B)
prod_ind is defined
prod_rec is defined
prod_rect is defined
prod is defined
```

Surgen automáticamente las funciones derivadas del (supuesto) carácter inicial del tipo. Consideremos el catamorfismo asociado:

```
Definition cata_prod:=[A,B,C:Set](prod_rec A B [_:(prod A B)]C).
Eval Simpl in cata_prod.
```

El tipo de esta función

```
cata_prod
 : (A,B,C:Set)(A->B->C)->(prod A B)->C
```

indica el bien conocido proceso de descurrificación: si se desea tener una función entre `prod A B` y un conjunto `C`, es necesario conocer una función que a un valor de `A` y a un valor de `B` asigne un valor de `C`. Naturalmente, como cabía esperar al no disponer el producto de capacidad inicial, no aporta nada nuevo a las posibilidades de actuación sobre él.

Veamos, entonces, qué ocurre con los tipos coinductivos y analicemos cómo debería ser, por dualidad, la función terminal requerida. Primeramente, dada la obligación de tener tipos definidos positivamente, al no poder declararlos en función de sus destructores, estos deben ser definidos singularmente (su derivación automática no es problemática al representar los constructores isomorfismos). El producto se corresponde con la GF-diálgebra colibre siendo los funtores F y G los construidos para el coproducto.

```
CoInductive prod [A,B:Set]:Set :=
conj:A->B->(prod A B).

Definition proy1 := [A,B:Set][x:(prod A B)]
Cases x of (conj a b)=>a end.

Definition proy2 := [A,B:Set][x:(prod A B)]
Cases x of (conj a b)=>b end.
```

La parametrización sobre las bases de los constructores, en este caso, carece de sentido, al ser el producto final de flecha y, entonces, no afectar al dominio de éstas. Por lo tanto, dado un conjunto y dos flechas (una a cada uno de los objetos que conforman el producto) podemos obtener la flecha terminal:

```
Definition prod_corec := [A,B,X:Set]
[f:X->A][g:X->B][x:X](conj A B (f x) (g x)).
```

Por ejemplo, la función `mapProd` se describe como sigue:

```
Definition map_prod :=[A,B,C,D:Set][f:A->C][g:B->D]
(prod_corec C D (A*B) [x:(A*B)](f(proy1 A B x))
[x:(A*B)](g(proy2 A B x))).
```

Consideremos el tipo exponencial como otro ejemplo para ver si realmente esta dualidad funciona tal como esperamos. Este tipo es la FG-diálgebra colibre de los funtores $F,\ G : \mathbf{C}^{op} \times \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ definidos por $F(A,B,C) = C \times A,\ G(A,B,C) = B$. En efecto, consideremos la FG-diálgebra $((C,D,E),\xi)$ con $\xi : E \times C \to D$; entonces, para cada par de objetos $A,\ B \in \mathbf{C}$ y cada par de morfismos $f : A \to C,\ g : D \to B$ existe un único morfismo $< f,g >^* : E \to (exp\ A\ B)$ definido como $< f,g >^*(e)(a) = g(\xi(e, f(a)))$, donde $a \in A, e \in E$. Por consiguiente, por comodidad notacional, definimos el tipo y las funciones resultantes como sigue:

```
CoInductive exp [A,B:Set] : Set := E:(A->B)->(exp A B).

Definition IE := [A,B:Set][f:(exp A B)]
Cases f of (E f1) =>f1 end.

Definition curry := [A,B,C:Set][f:(prod C A)->B][c:C][a:A]
(f (conj C A c a)).

Definition eval := [A,B:Set][x:(prod (exp A B) A)]
Cases ((proy1 (exp A B) A) x) of f =>
Cases f of (E f1) =>(f1 ((proy2 (exp A B) A) x))
end end.

Definition exp_corec :=[A,B,X:Set][f:(prod X A)->B][x:X]
(E A B ((curry A B X f) x)).
```

Como antes, la función `mapExp` será

```
Definition  mapExp := [A,B,C,D:Set][f:C->A][g:B->D]
(exp_corec C D (exp A B)
[x:(prod (exp A B) C)]
Cases ((proy1 (exp A B) C) x) of (E h1) =>
(g(h1(f( (proy2 (exp A B) C) x)))) end).
```

Observamos, pues, que no parece resultar excesivamente complicado, después de indicar claramente de qué estructura funtorial provienen, automatizar estos morfismo coinductivos.

5.5 *Definiciones inductivas*

5.5.1 Los números naturales

Provienen los naturales del funtor (cocontinuo) definido sobre los objetos por $F(A) = () + A$. En tal caso, $((nat, [O; S])$ es el objeto inicial de la categoría de las F-álgebras; O corresponde al natural 0 y $S : nat \to nat$ es la función sucesor. Dada otra F-álgebra $(A, [\theta; \epsilon])$, donde θ es un elemento de A y $\epsilon : A \to A$, existe un único homomorfismo de F-álgebras entre $((nat, [O; S])$ y $(A, [\theta; \epsilon])$, $cata_{nat}\ [\theta; \epsilon])$; como tal verifica las siguientes especificaciones:

$$(cata_{nat}\ [\theta; \epsilon])\ O = \theta;$$
$$(cata_{nat}\ [\theta; \epsilon])\ (S\ n) = \epsilon(cata_{nat}\ [\theta; \epsilon]\ n).$$

Podemos definir este tipo en Coq como un tipo inductivo:

```
Inductive nat : Set := O :nat | S : nat-> nat.
nat_ind is defined
nat_rec is defined
nat_rect is defined
nat is defined
```

El sistema añade el tipo `nat` y los constructores O y S al entorno.

Aparecen, tal como ocurrió anteriormente con las listas, los nuevos términos correspondientes al carácter inicial de `nat`:
`nat_ind, nat_rec, nat_rect`. El primer funcional, `nat_ind`, se usa para hacer pruebas por inducción acerca de proposiciones sobre números naturales (`P:nat->Prop`). Si se quiere demostrar la aserción $\forall n : nat.(P(n))$ (es decir, en lógica intuicionista, para construir un elemento de tipo `(n:nat)(P n)`) bastará con tener una prueba de `(P O)` y un procedimiento de tipo `(n:nat)(P n)->(P (S n))` (una prueba de $\forall n : nat.(P(n)) \Rightarrow (P(S(n)))$), o sea, una estrategia que, para cada natural `n`, nos permitirá definir una función que devuelva una prueba de `P (S n)` a partir de una prueba de `P n`.

El segundo (`nat_rec`) expresa que, si se tiene una familia numerable de conjuntos `(P n)` y se quiere determinar un criterio para elegir un elemento en cada uno de ellos bastará con tener un elemento del conjunto `P O` y un procedimiento (de tipo

`(n:nat)(P n)->(P (S n))`) para, dado un elemento en el conjunto `P n` poder elegir uno en `P (S n)`). En tal caso, dado un diagrama (grafo dirigido cuyos vértices son tipos y cuyas aristas son morfismos entre ellos) $P_0 \stackrel{F_1}{\to} P_1 \stackrel{F_2}{\to} P_2 \to \cdots \to P_n \stackrel{F_{n+1}}{\to} P_{n+1} \cdots$, y fijado un elemento $c_0 \in P_0$, `nat_rec` determina una familia de morfismos, $c_n : () \to P_n$, verificando, para cada $n \in nat$, la conmutatividad: $c_{n+1} = F_{n+1}(c_n)$. Esta familia $((),c_0, c_1, \cdots, c_n, \cdots))$ se denomina *cono*. Es, entonces, sencillo construir funciones a partir de `nat_rec`.

La suma, por ejemplo, ha de verificar las dos siguientes especificaciones:

$$(suma\ O)\ m = m;$$
$$(suma\ (S\ n))\ m = F_{n+1}((suma\ n)\ m);$$

Como (S n)+m=S((n+m)), definimos $F_{n+1} : (nat \to nat) \to nat \to nat$ por $F_{n+1}\ f\ n = S\ (f\ n)$, $n \in nat$, $f : nat \to nat$.

En este caso notamos que el tipo destino y el procedimiento de transición no cambian; independientemente del natural al que se apliquen, siempre son los mismos. En cambio, si queremos definir la función factorial, la especificación a verificar es

$$factorial\ (S\ n) = F_{n+1}(factorial\ n);$$

pero, si (S n)!=(S n) × n!, la función $F_{n+1} : nat \to nat$ debe venir definida por $F_{n+1}\ m = producto\ (S\ n)\ m$, dependiente del número n, es decir, tenemos un procedimiento diferente para cada transición. Cuando todos los conjuntos P_n son coincidentes (por ejemplo el tipo C), fijamos un elemento $a \in C$ y consideramos un único procedimiento de transición $F : C \to C$ obtenemos con `nat_rec` el catamorfismo proveniente de los naturales; su semántica ($F(c_n) = c_{n+1},\ n \in nat$) implica que $c_{n+1} = F^{n+1}(c_0),\ n \in nat$. Así, la suma de números naturales es un catamorfismo; en cambio, ya que no hay forma de definir el factorial de un número usando un único procedimiento de transición sino que éste dependerá, en cada caso, del argumento al cual aplicamos la función factorial, ésta no es un catamorfismo hacia `nat` ([Meijer y otros 1991], pg. 128).

Las dos funciones anteriores se definen en Coq de la siguiente forma:

```
Definition Suma :=
  ( nat_rec
  ([_:nat]nat->nat)
  ([n:nat]n)
  ([_:nat][f:nat->nat][m:nat](S (f m)))).

Suma is defined

Definition Factorial :=
       ( nat_rec
       ([_:nat]nat)
       (S O)
       ([n:nat][m:nat](Producto (S n) m)).

Factorial is defined
```

Un hecho que ha suscitado equívocos es el de la expresividad de este catamorfismo de números naturales. Por ejemplo, en
[Cockett y Fukushima 1992], definen la función de Ackermann como una lista infinita de listas infinitas debido a (sic) ... *los tipos iniciales nos han restringido al dominio de las funciones primitivo recursivas.* Entonces, limitan los catamorfismos a esta subclase de las funciones recursivas. Pero, de hecho, la capacidad expresiva de los catamorfismos va más alla de la recursión primitiva. Demostraremos en una próxima sección que la función de Ackermann es un catamorfismo. Esto está basado en el siguiente resultado de Marie-France Thibault:

Teorema 5.1. *([Lambeck y Scott 1989], pág. 258) Sea L un λ-cálculo tipado: entonces, cualquier función primitivo recursiva y la función de Ackermann son representables en L.*

Pero comprobaremos algo más: restringiendo la definición de catamorfismo tal como se hace en [Gregorio y otros 1995], ya no se tiene capacidad para definir la función de Ackermann.

Es sabido que Dedekind, con la definición de las funciones primitivo recursivas, pretendía abarcar todas las funciones numéricas definible sobre nat^k, $k \in nat$ con valores en nat. Hilbert, en 1926, cuestionó a la comunidad matemática acerca de la posibilidad de que hubiera funciones numéricas que no fuesen primitivo recursivas. Ackermann, en 1928, definió la denominada función exponencial generalizada (de Ackermann), demostrando, desde un punto de vista *cuantitativo*, que ésta era una función total no primitivo recursiva ya que superaba en tamaño a cualquiera que sí lo fuese. Dede entonces, la bibliografía sobre sus propiedades es incesante. Un tema estudiado ampliamente es el de su representabilidad en determinados contextos (λ-cálculo tipado, λ-cálculo extendido, etc). Recientemente, ([Manes y Arbib 1986], [Szasz 1991], [Cockett y Fukushima 1992], [Dowek 1998]) analizan algún aspecto relativo a ella (para ser más precisos, trabajan con la la función de Ackermann simplificada definida por Rózsa Péter), cada uno en un contexto distinto. En el primer caso se analiza desde la perspectiva de los tipos terminales; en el segundo como aplicación de la teoría de pruebas; el tercero la estudia desde la óptica del cálculo de construcciones inductivas y, por último, el cuarto la trata como algo inherente a las especificaciones recursivas.

La demostración de por qué no es primitivo recursiva, como hemos mencionado anteriormente, suele basarse en el hecho de que es mayor que cualquier función primitivo recursiva posible: es decir, que si f es una función primitivo recursiva y $(n_1, ..., n_k)$ un vector de naturales, existe un natural C que verifica

$$f(n_1, ..., n_k) < Ackermann(C, \sum_{i=1}^{i=k} n_i).$$

Entonces, un argumento por reducción al absurdo nos conduce a la contestación afirmativa a la cuestión propuesta por Hilbert. Así puede verse en [Yasuhara] y usando este criterio se demuestra en el sistema lógico ALF [Szasz 1991].

En ([Cockett y Fukushima 1992], pag. 23) se afirma que los tipos de datos iniciales allí definidos, restringen su aplicación al dominio de las funciones primitivo recursivas. Entonces, y como aplicación de la riqueza provista por los tipos terminales, se define la

función de Ackermann como una lista infinita de listas infinitas de naturales, donde la computación de la $(n+1)$-ésima columna (correspondientes a Ackerman $n+1$) depende de la columna anterior. La afirmación hecha en este trabajo sobre la no expresividad de esta función como un fold (catamorfismo sobre un tipo paramétrico, [Pardo 1997]) no es correcta (como podemos ver en [Dowek 1998] y como nosotros comprobaremos construyéndola como un catamorfismo).

En ([Dowek 1998], pag. 113) se muestra que esta función puede definirse en el sistema T de Gödel, es decir, haciendo uso de la recurrencia. La expresión en este contexto de tal función es como sigue:

$$Ackermamm \;=\; \lambda\, n : nat\; (Rec^{nat \to nat}\; S$$

$$\lambda\, p : nat\; \lambda\, f : nat \to nat \lambda\, m : nat\; (Rec^{nat}\; (f\; (S\; O))$$

$$\lambda\, q : nat\; \lambda\, s : nat\; (f\; s)\; m)\; n).$$

Usando las herramientas provistas por Coq en las declaraciones inductivas e interpretando su significación, puede obtenerse una especificación ligeramente diferente de la función de Ackermann y, por consiguiente, una definición inductiva simplificada. Además, tras este análisis, puede verse la función de Ackermann como un caso singular de la iteración. En efecto, con la definición que damos de esta función se ve que

$$Ackerman\; 0\; m = \; m+1;$$

$$Ackermann\; n+1\; m = (Ackermann\; n)^{m+1}\; 1$$

lo cual revela el porqué del rápido crecimiento de esta función. Así, por ejemplo, $Ackermann\; 4\; 4 = 2^{2^{2^{2^{16}}}} - 3$. Además, se ve que, para cada natural n, $Ackermann\; n$ es una función primitivo recursiva. Por ejemplo,

$$(Ackermann\; 1)\; m = m+2;$$

$$(Ackermann\; 2)\; m = 2m+3;$$

$$(Ackermann\; 3)\; m = 2^{m+3} - 3.$$

Un sencillo razonamiento inductivo lo permite ver. Nuestra intención es dar una demostración *cualitativa* de que, en cambio, la función de Ackermann no lo es. Para ello, construiremos, también en Coq, las funciones primitivo recursivas. Por último, usando Coq, veremos que tampoco es un catamorfismo generalizado [Gregorio y otros 1995].

5.6 *Demostración de Teoremas en* Coq

Hasta el momento no se ha mostrado la capacidad de Coq para probar teoremas. Se prueba a continuación la condición de inicialidad que tienen los catamorfismos.

El catamorfismo sobre los naturales se define fijando el tipo destino y el procedimiento de transición sobre él: es decir,

```
Definition cata_nat :=
[C:Set][a:C][F:C->C](nat_rec [_:nat]C a [_:nat]F).
```

Entonces, se verifica el siguiente teorema:

Teorema 5.2. *Sean (C,a,F1), (D,b,F2) dos álgebras con una estructura semejante al álgebra de los naturales anteriormente definida; sea, además, h un morfismo entre C y D que preserva sus estructuras (es decir, (h a)=b y, si x ∈ C, F2 (h x)=h (F1 x)). Entonces,* `(cata_nat C a F1);h=cata_nat D b F2`*, donde con* `;` *se indica la composición.*

El enunciado y la demostración son como sigue:

```
Theorem Fusion:(C,D:Set) (a:C) (b:D) (h: C->D)
(F1:C->C)(F2:D ->D)
((h a)=b)->
((x:C)(F2 (h x))=(h (F1 x)))->
((m:nat)(h (cata_nat C a F1  m))=(cata_nat D b F2 m)).
1 subgoal

Fusion < Intros.
Fusion < Elim m.
Fusion < Simpl.
Fusion < Trivial.
Fusion < Intros.
Fusion < Simpl.
Fusion < Replace (cata_nat D b F2 n)
Fusion < with (h (cata_nat C a F1 n)).
Fusion < EAuto.
Subtree proved!
Fusion < Qed.
Fusion is defined
```

5.7 *Funciones Recursivas*

5.7.1 Funciones primitivo recursivas. Función de Ackermann. Catamorfismos generalizados. Definiciones

El conjunto de las funciones *primitivo recursivas* es el menor que contiene a las funciones cero, sucesor y las proyecciones, y, además, es cerrado bajo la composición y bajo el esquema de recursión (definido más adelante). Entonces, si se considera el menor conjunto de funciones totales que contiene al anterior y es cerrado bajo el proceso de *minimización* (acotada o no) se tiene el conjunto de las funciones *μ-recursivas*. Añadiendo, por último, la posibilidad de que una función no esté definida en algunos elementos del conjunto original, se llega al denominado conjunto de funciones *parciales recursivas* que, según la tesis de Church, conforma el conjunto de las funciones efectivamente computables.

Como anunciamos previamente, el esquema de recursión se construye de la siguiente forma:
sean f y g dos funciones primitivo recursivas, $f : N^n \to N$, $g : N^{n+2} \to N$; la función $h : N^{n+1} \to N$ es definida siguiendo el *esquema de recursión* si verifica, para cada n-tupla

de naturales $(x_1, ...x_n)$:

$$h(0, x_1, ..., x_n) = f(x_1, ..., x_n)$$
$$h(n+1, x_1, ..., x_n) = g(n, x_1, ..., x_n, h(n, x_1, ..., x_n)),\ n \in N.$$

Como se mencionó en la Introducción, en el año 1926 Hilbert cuestionó a la comunidad matemática acerca de la posibilidad de la existencia de funciones totales que no fueran primitivo recursivas. Ackermann, para contestar a esta pregunta, a partir de una sucesión de funciones primitivo recursivas, demostró que la función que las codificaba a todas no era primitivo recursiva: ésta es la denominada exponencial generalizada de Ackermann. Partiendo de la función sucesor la construyó de la siguiente forma:

$$sucesor \equiv f_0(n, x) = s(n);$$

$$suma = \begin{cases} f_1(0, x) = x \\ f_1(n+1, x) = f_0(f_1(n, x), x) \end{cases}$$

$$producto = \begin{cases} f_2(0, x) = x \\ f_2(n+1, x) = f_1(f_2(n, x), x) \end{cases}$$

$$exponencial = \begin{cases} f_3(0, x) = 1 \\ f_3(n+1, x) = f_2(f_3(n, x), x) \end{cases}$$

En general,

$$\begin{cases} f_{n+1}(0, x) = g_{n+1}(x) \\ f_{n+1}(n+1, x) = f_n(f_{n+1}(n, x), x) \end{cases}$$

De hecho, podemos definir un método para construir funciones que no sean primitivo recursivas. Dada su construcción, es evidente que éstas forman un conjunto numerable. Sea, pues, Q su enumeración (es decir, una función $Q : N \to FPR$). Consideremos Q_n la n-ésima derivación de esta lista y g_n la función primitivo recursiva correspondiente. Construyamos la función h definida por $h(n) = g_n(n) + 1$. Es obvio que h no puede ser primitivo recursiva ya que, si lo fuese, se verificaría que, para algún natural k, $h = g_k$. Por consiguiente, $g_k(k) = h(k) = g_k(k) + 1$ lo cual es una clara contradicción.

En [Gregorio y otros 1995] se define, constructivamente, lo que denominan los autores los **catamorfismos generalizados** sobre los naturales. Demuestran, además, que estos dan lugar a las funciones primitivo recursivas. Veremos que la función de Ackermann no es un catamorfismo generalizado.

Definición 5.2. *La función $h : N^n \to N$ es un catamorfismo generalizado sobre la clase de funciones* C *si existen funciones $g_i : N^{n-1} \to N$, $1 \leq i \leq n$ y $g : N \to N$ en* C *tales que:*

$$h(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) = g_i(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n);$$
$$h(x_1 + 1, \ldots, x_n + 1) = g(h(x_1, \ldots, x_n)).$$

con la restricción sobre las g_i de ser compatibles.

Teorema 5.3. *Sea* C *una clase de funciones que contiene a los catamorfismos. Consideremos la función* $h : N^2 \to N$ *de forma que* h *es un catamorfismo en cada una de sus variables: (es decir, si* $n \in N$, $h_n(m) = h(n,m)$ *y* $h^m(n) = h(n,m)$ *son catamorfismos): entonces* h *es un catamorfismo generalizado.*

En primer lugar, que $\forall n, m \in N$, h_n y h^m sean catamorfismos implica que existen valores $a_n,\ b_m \in N$ tales que $h_n(O) = a_n,\ h^m(O) = b_m$. Definimos, entonces, las funciones $g_1,\ g_2 : N \to N$ como $g_1(m) = b_m,\ g_2(n) = a_n$ que, obviamente, pertenecen a C.

Además, como h_n es un catamorfismo, existe una función $k_1 : N \to N$ tal que $h_n(S\ m) = k_1(h_n(m))$; análogamente, al ser h^m un catamorfismo, existe $k_2 : N \to N$ tal que $h^m(S\ n) = k_2(h^m(n))$. Entonces, $h(S\ n, S\ m) = h_{S\,n}(S\ m) = k_1(h_{S\,n}(m)) = k_1(h(S\ n, m)) = k_1(h^m(S\ n)) = k_1(k_2(h^m(n)) = k_1(k_2(h(n,m))$, de donde, existe $g = k_2; k_1 \in$ C tal que $h(S\ n, S\ m) = g(h(n,m))$ como queríamos ver.

Definición 5.3. *La clase de funciones* $\mathbf{C}^w$ *es el menor conjunto que contiene al cero, a la función sucesor y es cerrada bajo composición y esquema de catamorfismo generalizado.*

Teorema 5.4. *([Gregorio y otros 1995], pag. 480) La clase de funciones* C^w *es la clase de las funciones recursivas primitivas.*

5.8 *Construcción en* Coq *de las funciones primitivo recursivas*

Para efectuar tal construcción debemos expresar, en primer lugar, el conjunto de las n-tuplas de números naturales, N^n. Dada la facilidad que proporciona Coq para definir tipos dependientes, este conjunto lo declaramos

```
Inductive Power [A:Set]:nat->Set:= Power_inic:(Power A 0)
|Power_S:(n:nat)A->(Power A n)->(Power A (S n)).
```

Deben quedar claras varias cosas:
utilizamos un parámetro `A` para poder hacer declaraciones más vastas (de tipos funcionales y no sólo de potencias de naturales);
en segundo lugar, es evidente que, para $n \geq 1$, `Power A` $n \equiv A^n$, siendo `Power A 0` $\equiv$ `Unit`, tipo con un único elemento;
por último, es obvio por la definición que, para cada $n \in$ `nat`, $A^{n+1} = A \times A^n$.
(Es preciso aclarar que, a efectos de mejorar la legibilidad, hemos decidido que el tipo de salida sea `nat` en lugar de `Power nat (1)`. Es por ello que construimos el isomorfismo

```
Definition inclusion := [n:nat]
(Power_S nat 0 n (Power_inic nat)).
```

que relaciona cada natural con su equivalente en `Power nat (1)` de forma que el resultado de componer por su derecha un morfismo primitivo recursivo da lugar a una función primitivo recursiva.

Introducimos, pues, las funciones primitivo recursivas básicas, cero, proyección y composición, teniendo en cuenta que la función sucesor ya forma parte de la construccón del tipo `nat`.

```
Inductive Unit : Set := unit: Unit.

Definition zero := [x:Unit] O.
```

La proyección se define inductivamente a partir de la correspondiente especificación:
siendo n, $i \in N$, $i \leq n$ y $x \in N^n$
projection $n+1$ (`Power_S` $n\ a\ x$) $0\ = a$;
projection $n+1$ (`Power_S` $n\ a\ x$) $i+1\ =$ *projection* $n\ x\ i$.
Como las funciones en `Coq` son totales indicamos
projection O `Power_inic` $i\ =\ a$ siendo a un elemento arbitrario del conjunto A.

La definición es como sigue (considerando que hacemos una definición global (para cualquier tipo no sólo para tuplas de naturales) que pueda ser usada en el caso de familias de funciones de tipo $(N^n \to N)^m$ en el momento de declarar la composición):

```
Definition proj := [A:Set] [a:A] Fix F
{F [n:nat;x:(Power A n)]:nat->A :=
Cases x of
  Power_inic  => [_:nat]a
  |(Power_S n0 y p) =>
     Cases n0 of
        O => [i:nat]
           Cases i of
               O => y
               |_ => (hd A a n0 p)
           end
        |(S n1) => [i:nat]
              Cases i of
                 O => y
                 |(S i0) => (F n0 p i0)
              end
     end
end}.

Definition projection :=[n:nat][i:nat][x:(Power nat n)]
(proj nat O n x i).
```

La siguiente función a definir es, pues, la composición. Para ello, recordemos que su especificación es:
dadas m+1 funciones $G_i : N^n \to N, 1 \leq i \leq m,\ f : N^m \to N$, su composición es la función $G : N^n \to N$ definida, para $x \in N^n$, como $G(x) = f(G_1(x), \ldots, G_m(x))$. Observemos, en la definición de la función `app,` el uso de `mapPower,` la cual, tomando como argumentos dos tipos, `((Power nat n)->nat)` y `nat,` y un morfismo entre ellos, `aplication n x`, da como salida una función de tipo
`(n0:nat)(Power ((Power nat n)->nat) n0)->(Power nat n0).`

```
Definition aplication := [n:nat] [x:(Power nat n)]
```

```
[G:(Power nat n)->nat] (G x).

Definition app := [n:nat] [x:(Power nat n)] mapPower

((Power nat n)->nat) nat (aplication n x)).

Definition Composition := [n,m:nat] [f:(Power nat m)->nat]
[G:(Power ((Power nat n)->nat) m)] [x:(Power nat n)]
(f (app n x m G)).
```

Resta construir el esquema de recursión de especificación :
sean f y g dos funciones primitivo recursivas, $f : N^n \to N$, $g : N^{n+2} \to N$; la función $h : N^{n+1} \to N$ es definida siguiendo el *esquema de recursión* si verifica, para cada n-tupla de naturales $(x_1, ...x_n)$:

$$h(0, x_1, ..., x_n) = f(x_1, ..., x_n)$$
$$h(n+1, x_1, ..., x_n) = g(n, x_1, ..., x_n, h(n, x_1, ..., x_n)),\ n \in N.$$

Su implementación es

```
Definition Rec :=[n:nat][g:(Power nat n)->nat]
[h:(Power nat (S(S n)))->nat]
Fix F { F [m:nat]:(Power nat n)->nat :=
   Cases m of
       O =>[x:(Power nat n)](g x)
       |(S m0) =>[x:(Power nat n)]
                 (h (Power_S nat (S n) m0
                 (Power_S nat n (F m0 x) x)))
   end}.

Definition Recursion := [n:nat] [g:(Power nat n)->nat]
[h:(Power nat (S(S n)))->nat] [x:(Power nat (S n))]
(Rec n g h (projection (S n) O x) (tl (S n) x)).
```

Puede ser interesante explicitar la función constante

```
Definition cons := [A:Set][n:nat][x:A]n.
```

Podemos, ya, declarar cuándo es primitivo recursiva una función según la indicación dada en la sección 5.7.1:

```
Inductive isprim : (A:Set)(A->nat)-> Prop :=

Ax1: (isprim Unit cero)

|Ax2:(isprim nat S)
```

```
|Ax3:(n,m:nat)(isprim (Power nat n) (projection n m))

|Ax4:(m:nat)(g:(Power nat m)->nat)
(f:(Power nat (S(S m)))->nat)
(isprim (Power nat m) g)->
(isprim (Power nat (S(S m))) f)->
(isprim (Power nat (S m)) (Recursion m g f))

|Ax5:(n,m:nat)(f:(Power nat m)->nat)
(G:(Power ((Power nat n)->nat)
m)) (isprim (Power nat m) f)->
(isprim (Power nat n) (Composition
n m f G))

|Ax6:(n,m:nat)(isprim (Power nat n) (cons n m))

|Ax7:(g:(Power nat (1))->nat)(isprim (Power nat (1)) g)->
 (isprim nat (composition nat (Power nat (1)) nat inclusion g))

|Ax8:(A:Set)(f:A->nat)(g:nat->nat)(isprim A f)->(isprim nat g)->
(isprim A (composition A nat nat f g)).
```

Analicemos un par de ejemplos.

Ejemplo 5.1. *Comenzamos con la suma de naturales:*

```
Dado (x:(Power nat (1))

sum 0 n   = n                  => g x= (projection 0 0 x)
sum (S m) n = S (sum m n)      => h a b c=(S b)
```

La construcción y la propiedad son las siguientes:

```
Definition sum := [x:(Power nat (2))]
(Recursion (1) (projection (1) 0)
(composition (Power nat (3)) nat nat (projection (3) (1))
S) x).

Lemma one: (isprim (Power nat (2))
(Recursion (1) (projection (1) 0)
(composition (Power nat (3)) nat nat (projection (3) (1)) S))).
Apply Ax4.
Apply Ax3.
Apply Ax8.
Apply Ax3.
Apply Ax2.
Qed.
```

Ejemplo 5.2. *Otro caso, algo más complicado en cuanto a la definición de la función que lo soporta, es el del producto:*
siendo (x:(Power nat (1))
product 0 n = 0 =¿g x= (constante O)
*product (S m) n = sum m (product m n) =¿h a b c= sum b c *)*

```
Definition Product := [x:(Power nat (2))]
(Recursion (1) (cons (1) 0) (Composition (3) (2) sum
(Power_S ((Power nat (3))->nat) (1) (projection (3) (2))
(Power_S ((Power nat (3))->nat) 0 (projection (3) (1))
(Power_inic ((Power nat (3))->nat))))  ) x).

Lemma two: (isprim (Power nat (2))
(Recursion (1) (cons (1) 0) (Composition (3) (2) sum
(Power_S ((Power nat (3))->nat) (1) (projection (3) (2))
(Power_S ((Power nat (3))->nat) 0 (projection (3) (1))
(Power_inic ((Power nat (3))->nat)) ))))) .

Proof.
Apply Ax4.
Apply Ax6.
Apply Ax5.
Apply one.
Qed.
```

5.9 *Sucesiones de funciones primitivo recursivas*

Las funciones primitivo recursivas forman un conjunto numerable (de hecho, es numerable el conjunto de funciones recursivas del cual es un subconjunto propio). La prueba de la existencia de funciones recursivas no primitivas se fundamenta en el proceso de diagonalización. Ackermann, con la función exponencial, pretende llegar al mismo punto pero, utilizando únicamente, una colección numerable de funciones primitivo recursivas. Podemos definir el esquema de recursión de una forma algo más amigable para su uso y manejo:

```
Definition EsqRec:=[A:Set][g:A->nat][h:nat->nat->A->nat]
Fix F {F [n:nat]:A->nat :=
    Cases n of
        0 =>[x:A](g x)
        |(S n0)=>[x:A](h n0 (F n0 x) x) end}.
```

Esta función queda encuadrada dentro de las definibles con la función estructural inducida por la declaración del tipo de los naturales, `nat_rec`. En efecto, `EsqRec` coincide con la función:

```
Definition EsqRecInd:=[A:Set][g:A->nat][h:nat->nat->A->nat]
(nat_rec [_:nat]A->nat g [n:nat][f:A->nat][a:A](h n (f a) a)).
```

Se puede ver con la siguiente demostración:

```
Lemma P1:(A:Set)(g:A->nat)(h:nat->nat->A->nat)(n:nat)(a:A)
((EsqRec A g h n a)=(EsqRecInd A g h n a)).
Intros;Elim n;Simpl;Auto;Intros.
Rewrite H.
Trivial.
Qed.
```

Por ejemplo, la función factorial puede definirse como sigue:

```
Definition factorial :=
(EsqRec nat [n:nat](S O) [n,m,x:nat](mult (S n) m)).
```

Como es inmediatamente comprobable, esta función `EsqRec` tiene un espectro de actuación más amplio que el catamorfismo sobre los naturales. Podemos denominarlo fold siguiendo la nomenclatura de [Cockett y Fukushima 1992] (siendo foldr y foldl casos singulares de este funcional). Con este esquema, la composición, la función constante O y la función sucesor (las proyecciones van asociadas a la capacidad de definición) se tienen determinadas todas las funciones primitivo recursivas. De hecho, aprovechando en toda su capacidad la función `nat_rec`, se puede establecer un esquema de recursión de espectro más amplio:

```
Definition EsqRecGen:=[P:nat->Set][g:(P O)->nat]
[h:(n:nat)nat->nat->(P n)->nat][T:(n:nat)(P (S n))->(P n)]
(nat_rec [n:nat](P n)->nat g
[n:nat][f:(P n)->nat][x:(P (S n))]
 (h n n (f ((T n) x )) ((T n) x))).
EsqRecGen is defined
```

La función `AckermannOr` definida a continuación permite codificar cualquier sucesión de funciones funciones primitivo recursivas provenientes de otras funciones primitivo recursivas primarias (denotada, en este caso, por `h` y `g`). La especificación a verificar es la siguiente:

$$AckermannOr\ O\ x\ m\ =\ h\ x\ m;$$

$$AckermannOr\ (S\ n0)\ x\ O\ =\ g\ (S\ n0)\ x;$$

$$AckermannOr\ (S\ n0)\ x\ (S\ m0)\ =\ F1\ n0\ x\ (F1\ (S\ n0)\ x\ m0).$$

Para su definición se usa una función auxiliar:

```
Definition H0:=[A:Set][g:A->nat][alfa:A->nat->nat][x:A]
(nat_rec [_:nat]nat (g x) [_:nat][m:nat](alfa x m)).

Definition AckermannOr:=[A:Set][g:(n:nat)A->nat]
[h:A->nat->nat]
(nat_rec [_:nat](A->nat->nat) h [n:nat](H0 A (g (S n)))).
```

El único requisito establecido para la obtención de funciones recursivo primitivas es que lo sean `h` y cada una de las g_n, $n \in nat$. Considerando, por ejemplo, `h` la función sucesor, se obtienen las operaciones elementales sobre los naturales. Así, definiendo

```
Definition h1:=[n:nat][m:nat](S n).
Definition g1:=[n:nat]
(Cases n of
O=>[x:nat](S x)
|(S O)=>[x:nat]x
|(S(S O))=>[x:nat]O
|(S(S(S n )))=>[x:nat](S O)
end).
```

es posible representar las operaciones básicas sobre los naturales como sigue:

```
Definition Gen:=(AckermannOr nat g1 h1).
```

Esta función tiene como argumento tres naturales: el primero indica el lugar que ocupa la función que se quiere representar según la anterior ordenación; los otros dos son los operandos (siendo el último el argumento recursivo).

Entonces, la suma, el producto y la exponenciación se obtienen, respectivamente como `Gen (S O), Gen (S(S O)), Gen (S(S(S O)))`. El valor correspondiente a `Gen O n m` es `(S n)`.

Tal como se ha construido la función `Gen` se ve que, a medida que aumenta el argumento indicativo de la numeración, los valores van aumentando exponencialmente de tamaño. Coq no da esos valores así que, para poder evaluar esas cantidades, se puede desarrollar alguna para ver realmente su alcance. Por ejemplo, `Gen 4 3 2`:

```
Gen 4 3 2 -> Gen 3 3 (Gen 4 3 1) ->
Gen 3 3 (Gen 3 3 (Gen 4 3 0))  ->
Gen 3 3 (Gen 3 3 1)
```

Aprovechando el hecho de saber que `Gen 3` representa la función exponencial, se obtiene el resultado $3^3 = 27$. Haciendo `Gen 5 6 2`, tenemos

```
Gen 5 6 2 -> Gen 4 6 (Gen 5 6 1) ->
Gen 4 6 (Gen 4 6 (Gen 5 6 0))
```

Como, por la construcción de `Gen`, `Gen 5 6 0=1`, y, además, `Gen 4` representa la torre exponencial, el resultado que se obtiene es

`Gen 4 6 (Gen 4 6 1)=Gen 4 6 6=`$6^{6^{6^{6^6}}}$. Se puede enunciar el siguiente teorema:

Teorema 5.5. *Dados números naturales n, p, q tales que $p \geq 3$, $q \geq 3$, se verifica que Gen n p q < Gen (S n) p q.*

Ya que `Gen (S n) p (S q)=Gen n p (Gen (S n) p q)` basta, para demostrarlo, probar el siguiente:

Lema 5.1. *Dados números naturales n, p, q tales que $p \geq 2$, Gen n p q < Gen n p (S q).*

Pero, `Gen (S n) p (S q)=Gen n p (Gen (S n) p q)>Gen (S n) p q`.

Se puede construir una función de una sola variable con el mismo carácter no primitivo de AckermannOr:

```
Definition Seq:=[n:nat](Gen n n n).
```

Es importante notar que `AckermannOr` permite codificar **todas** las funciones primitivo recursivas. Esto es obvio porque, si `k:A->nat` es una de estas funciones, tomando como `h:nat->A->nat` la función que a `n:nat, x:A` le asigna `k x`, tenemos que `k=AckermannOr A [n:nat][x:A]n h O`. Esto queda aseverado de la siguiente forma:

```
Definition HPR:=[A:Set][h:A->nat][n:nat][x:A](h x).
Definition GPR:=[A:Set][n:nat][x:A]n.
Definition FPR:=[A:Set][h:A->nat](Funcion A (GPR A) (HPR A h) O).
Theorem PR1:(A:Set)(h:A->nat)(x:A)(n:nat)
((FPR A h n x)=(h x)).
Auto.
Qed.
```

5.10 *La función exponencial generalizada de Ackermann es un catamorfismo*

En la sección 5.5.1, al tratar las funciones `Suma, Producto, Factorial`, se vio una sustancial diferencia en su construcción: las dos primeras eran catamorfismos sobre el tipo de los naturales mientras que la segunda no. Analicemos con más detalle este hecho. Para definir la suma de naturales, se construye un álgebra de la siguiente forma: $(nat \to nat, (id_{nat}; \epsilon)$, donde $id_{nat} : nat \to nat$, $id_{nat}(n) = n$ y $\epsilon : (nat \to nat) \to nat \to nat$ está definida como $\epsilon\ h\ n = S(h\ n)$, con $h : nat \to nat$ y $n : nat$; análogamente, para el producto el álgebra adecuada es $(nat \to nat, (cero; \delta)$, donde $cero : nat \to nat$, $cero(n) = O$ y $\delta : (nat \to nat) \to nat \to nat$ queda determinada por $\delta\ h\ n = (h\ n) + n$, donde, de nuevo, $h : nat \to nat$ y $n : nat$.

En cambio, no se puede determinar ningún álgebra simple con `nat` como soporte para la función factorial. Entonces, como principal hecho diferencial con respecto a los catamorfismos, `nat_rec` no tiene un tipo destino fijo sino que éste depende del argumento natural sobre el cual actúe. Es decir, el catamorfismo fija un álgebra de salida, (A, ξ), mientras que `nat_rec` permite trabajar con una familia numerable de tipos relacionados entre sípor una familia numerable de funciones (procedimientos de transición entre ellos).

La función `cata_nat` anteriormente definida representa el esquema de recursión presentado en la sección 5.7.1; pero, a pesar de ello, no determina, en general, únicamente funciones primitivo recursivas. Para que ocurra esto, es preciso que los argumentos en la aplicación del antedicho esquema recursivo, sean primitivo recursivas. En la definición de `cata_nat` no se ha impuesto tal restricción. La función `f:C->C` puede ser o no primitivo recursiva por lo cual, la función resultante, también lo será o no.

Por lo tanto, habrá funciones no primitivo recursivas expresables a partir de `cata_nat`.

Anteriormente se ha introducido la función de Ackermann y ya sabemos que no es primitivo recursiva. Tampoco es un catamorfismo. Como se ve en su definición, la función

de transición depende del índice de paso (como en el caso del factorial). Una función que tampoco es primitivo recursiva y sí es un catamorfismo es la siguiente, una simplificación de la función de Ackermann con especificación:

```
Ackermann   0   m   =   m+1
Ackermann       n+1 0   =   Ackermann   n   1
Ackermann   n+1 m+1 =   Ackermann   n   (Ackermann n+1  m)
```

Una definición en CAML de esta función es como sigue:

```
let rec Ackermann=function 0->(function m->m+1)
|n->(function 0->(Ackermann (n-1) 1)
|m->(Ackermann (n-1) (Ackermann n (m-1))));;
```

Esta función satisface la especificación, pero no se puede garantizar. Por ejemplo, la función de Collatz, de especificación

```
collatz     0   =   1
collatz     n   =   collatz (n/2)   si n es par
collatz     n   =   collatz (3n+1)  si n es impar
```

es definible en CAML aunque su terminación no está garantizada:

```
let rec collatz=function 0->1
|n->if (n mod 2=0) then (collatz (n/2))
else (collatz (3*n+1));;
```

Coq tiene la posibilidad de definir funciones recursivas:

```
 Recursive Definition Ack:nat->nat->nat :=
0      n    =>     (S n)
|(S n)     0    => (Ack n (S 0))
|(S n)     (S m)    => (Ack n (Ack (S n) m)).
Ack_eq1 is defined
Ack_eq2 is defined
Ack_eq3 is defined
Ack is recursively defined.
```

Para asegurar la existencia de esta función, se puede usar otra característica de Coq: si el sistema acepta una definición inductiva se tiene asegurada su terminación (y, por consiguiente, su corrección). Formalizando en Coq la especificación tenemos:

```
 Inductive Ackermann:nat->nat->nat->Prop :=
Ack0   : (n:nat)(Ackermann 0 n (S n))
|Ackn0 : (n,p:nat)(Ackermann n (S 0) p)->
              (Ackermann (S n) 0 p)
|AckSS : (n,m,p,q:nat)
              (Ackermann (S n) m q)->(Ackermann n q p)
              ->(Ackermann (S n) (S m) p).
Ackermann_ind is defined
Ackermann is defined
```

Veamos que, en efecto, se puede definir esta función de Ackermann como un catamorfismo. Como el resultado debe ser una función de `nat` en `nat`, se debe hallar un álgebra `(nat->nat,(a,epsilon))` tal que `Ackermann=cata_nat nat->nat m epsilon`. Las condiciones que deben verificar son las siguientes:

$$\begin{aligned} (catanat\ O)\ m &= a\ m; \\ (catanat\ (Sn))\ m &= (epsilon(catanat\ n))\ m. \end{aligned}$$

Queda establecido en la especificación que `a=`S. Resta ver cómo es `epsilon`. En la especificación de la función de Ackermann se ha visto que $(catanat\ (Sn))\ O = (catanat\ n)\ (SO)$. Por consiguiente, `(epsilon f) O=f (S O)`, para cualquier $f : nat \to nat$. La última condición de la función de Ackermann informa acerca de que

$$(catanat\ (Sn))\ (Sm) = (catanat\ n)\ ((catanat\ (Sn))\ m.$$

Entonces,
`(epsilon (catanat n)) (S m)=catanat n (catanat (S n) m.`
Pero, `(catanat (S n)) m=(epsilon (catanat n)) m.`
Por consiguiente,
`epsilon (catanat n) (S m)=catanat n (catanat (S n) m=`
`catanat n ((epsilon (catanat n)) m)`".
Es decir, dada una función $f : nat \to nat$,
`(epsilon f) (S m)=f ((epsilon f) m).`
Por consiguiente, la función de Ackermann es un catamorfismo. Definamos esto en Coq, usando una función auxiliar:

```
Definition omega := (cata_nat ((nat->nat)->nat)
([f:nat->nat](f (S O)))
([h:(nat->nat)->nat][k:nat->nat](k (h k)))).
omega is defined
```

Observando esta función se ve que puede interpretarse como la iteración: en efecto, `omega n f=`f^{n+1} 1. El teorema siguiente confirma que, en efecto, la función de Ackermann es el catamorfismo anunciado:

Teorema 5.6. *Sea* `iteracion` *la función definida como*

```
Definition iteracion:=[f:nat->nat][n:nat](omega n f).
```

Entonces,la función

```
Definition HAck :=(cata_nat nat->nat S iteracion).
HAck is defined
```

coincide con la función simplificada de Ackermann.

Notemos los sencillos mimbres con los que se construye esta función: el sucesor y la iteración.

```
Theorem ACKermann:(n,m:nat)(Ackermann n m (HAck n m)).
```

Su demostración es como sigue:

```
ACKerman < Induction n; Induction m.
ACKerman < Simpl.
ACKermann < Auto.
ACKermann < Auto.
ACKermann < Simpl.
ACKermann < Auto.
ACKermann < Apply Ackno.
ACKermann < Simpl.
ACKermann < Apply (H (S O)).
ACKermann < Intros.
ACKermann < Replace (HAck (S n0) (S n1))
ACKermann < with (HAck n0 (HAck (S n0) n1)).
ACKermann < Apply AckSS with (HAck (S n0) n1).
ACKermann < Assumption.
ACKermann < Apply (H (HAck (S n0) n1)).
ACKermann < Simpl.
ACKermann < Trivial.
Subtree proved!
```

Se ha obtenido, entonces, la siguiente especificación de esta versión simplificada de la función de Ackermann:

$$Ackermann\ O\ m\ =\ (S\ m);$$

$$Ackermann\ (S\ n)\ m\ =\ (iteracion\ (S\ m)\ (Ackermann\ n))\ 1.$$

5.11 *Los catamorfismos generalizados en el contexto de* Coq

Pueden definirse esas funciones (aunque Coq, en este aspecto, no ayuda en exceso debido al carácter limitado de construir funciones recursivas). De hecho, la siguiente es la construcción de estos catamorfismos generalizados donde las funciones auxiliares tienen el importante papel de permitir `eliminar` la componente nula de un vector (una proyección de N^n en N^{n-1}), restar componente a componente del vector original el menor valor de sus elementos para llegar al valor nulo en una coordenada y, por último, iterar la función ese mismo número de veces:

```
Definition CatGen := [x:(Lista nat n)]
Cases (Isnull n x) of
true => ( (proy ((Lista nat (pred n))->nat)
(nula (pred n)) n G (Location n x))
(UnConversion (pred n)
(Eliminar (Location n x) (Conversion n x))))
|_ => ( Iteration
g (Is_minor (Conversion n x))
```

```
((proy ((Lista nat (pred n))->nat)
(nula (pred n)) n G
(Posicion_a (Is_minor (Conversion n x)) n x))
(UnConversion (pred n) (Eliminar (Location n x)
(Conversion n (predecesorL n x) ) ) ) ) )
end.
```

5.12 *La función de Ackermann no es un catamorfismo generalizado*

Con esta caracterización de la función simplificada de Ackermann, podemos ver que no cumple las condiciones exigibles para ser catamorfismo generalizado según la caracterización descrita en [Gregorio y otros 1995]. Para serlo, deberíamos contar con tres funciones primitivo recursivas $g_1,\ g_2,\ g : N \to N$ verificando lo siguiente:

$$Ackermann\ O\ m\ =\ (g_1\ m);$$

$$Ackermann\ (S\ n)\ O\ =\ (g_2\ (S\ n));$$

$$Ackermann\ (Sn)\ (Sm)\ =\ (g(Ackermann\ n\ m).$$

Es inmediato de la definición de *Ackermman* que $g_1 = S$ y que, para cada $n \in nat$, $(g_2\ (S\ n) = Ackermann\ n\ 1$ función previamente definida (en Charity [Cockett y Fukushima 1992] construyen una matriz de forma que la n-ésima columna representa el cálculo de $Ackermann\ n$; entonces, con cada columna pueden obtener la siguiente, de forma que, al intentar calcular $Ackermann\ (S\ n)$ ya tenemos creadas previamente las n columnas que le preceden). El problema para la consecución de nuestro objetivo estriba en la tercera ecuación: tal función g no existe. En efecto, apliquemos la anterior caracterización a los valores $Ackermann\ 3\ 1$, cuyo valor es 13 y a $Ackermann\ 2\ 2$ que vale 7. Debería ocurrir lo siguiente:
por una parte $13 = Ackermann\ 3\ 1 = g(Ackermann\ 2\ 0) = g(3)$; y, por la otra, $7 = Ackermann\ 2\ 2 = g(Ackermann\ 1\ 1) = g(3)$.

La demostración en Coq es como sigue:

```
Variable G:(Lista ((Lista nat (pred (2)))->nat) (2)).
Variable g:nat->nat.

Hypothesis H1 : (n,m:nat)
(CatGen (2) G g (Form n m))=(Ackermann (Form n m)).

Lemma PL1 : (Ackermann (Form (3) (1)))=(13).
Trivial.
Qed.
Hint PL1.

Lemma PL2 : (Ackermann (Form (2) (2)))=(7).
Trivial.
Qed.
```

```
Hint PL2.

Lemma PL3 : (Ackermann (Form (2) 0))=(3).
Trivial.
Qed.
Hint PL3.

Lemma C1 : (CatGen (2) G g (Form (3) (1)))=
(g (Ackermann (Form (2) 0))).
Proof.
Unfold CatGen.
Simpl.
Replace (Ackermann (Form (2) (0)))
with (CatGen (2) G g (Form (2) 0)).
Trivial.
Apply (H1 (2) 0).
Qed.
Hint C1.

Lemma C2 : (CatGen (2) G g (Form (2) (2)))=
(g (Ackermann (Form (1) (1)))).
Proof.
Unfold CatGen.
Simpl.
Replace (Ackermann (Form (1) (1)))
with (CatGen (2) G g (Form (1) (1))).
Trivial.
Apply (H1 (1) (1)).
Qed.
Hint C2.

Lemma C3 : (CatGen (2) G g (Form (2) (2)))=(7).
Apply (H1 (2) (2)).
Qed.
Hint C3.

Lemma C4 : (CatGen (2) G g (Form (3) (1)))=(13).
Apply (H1 (3) (1)).
Qed.
Hint C4.

Theorem Contradiccion : (7)=(g (3)) /\(13)=(g (3)).
Split.
Rewrite <- C3.
Rewrite <- PL3.
```

```
Trivial.
Rewrite <- C4.
Rewrite <- PL3.
Trivial.
Qed.
```

En efecto, tal como habíamos predicho, si tal función g existe, alcanza dos valores diferentes en 3 lo cual, obviamente, es una contradicción.

Se han conseguido varios propósitos:
declarar, en Coq, el conjunto de las funciones primitivo recursivas;
mostrar que la función original de Ackermann se puede definir inductivamente (y es más, sin apelar a tipados dependientes) a pesar de no ser primitivo recursiva usando el sucesor y la iteración;
comprobar que el catamorfismo `cata_nat`, correspondiente al esquema de recursión, genera funciones que no tienen por qué ser primitivo recursivas; depende, en cualquier circunstancia, de los argumentos que le sean aplicados y cómo estos sean considerados. En nuestro caso, la función utilizada en el patrón de la recursión (`iteracion`) no es, obviamente, primitivo recursiva al tener como argumento una función;
por último, ver, constructivamente, que, a pesar de ser un catamorfismo, la función de Ackermann simplificada no es un catamorfismo generalizado: es decir, aplicando una leve restricción a los catamorfismos, se elimina la capacidad para generar funciones no primitivo recursivas.

5.13 *Los números ordinales*

A continuación se analiza un tipo con una estructura más compleja que la de los naturales. Los ordinales se definen a partir del funtor F siguiente: dado un objeto A, su imagen es el objeto $() + A + (nat \to A)$ y, dado un morfismo $f : A \to B$, lo transforma en el morfismo $F(f) : () + A + (nat \to A) \to () + A + (nat \to A)$, $F(f) =! + f + (\lambda x : nat \to A.\lambda n : nat(f(xn))$. Puede definirse en Coq de la siguiente forma:

```
Inductive ord:Set :=Fst:ord|Next:ord->ord|Lim:(nat->ord)->ord.
```

La definición del catamorfismo correspondiente a la inicialidad del tipo se puede definir como sigue:

```
Definition cata_ord:=[C:Set](ord_rec [_:ord]C).
```

El siguiente ejemplo permite ver cómo construir funciones sobre este tipo:

```
Definition translation:=
(ord_rec [_:ord]nat  (S(S(S O)))
[_:ord][n:nat](S(S(S(S n))))
[o:nat->ord][f:nat->nat](f (S(S O)))).
```

Considerando la función siguiente entre los naturales y los ordinales

```
Definition ord_1=Fix F {F [n:nat]:ord Cases n of
O=>Fst
|(S n0)=>(Next (F n0)) end}.
```

y el siguiente elemento de tipo `ord`

```
Definition ord1:=(Lim ord_1).
```

la función `traslation` convierte el ordinal anterior en el natural
`translation ord1=`
`(S (S (S (S (S (S (S (S (S (S (S O)))))))))))`.

Cuando se precisa definir un morfismo sobre el tipo de los ordinales, se debe establecer una imagen para el elemento `Fst`, una función de transición proveniente de `Next` y, como caso singular, para cada elemento-función entre naturales y ordinales (`o:nat->ord`) una función entre el tipo imagen de ese ordinal (`(P (o n))`) y `(P (Lim o))`. En este caso, se tienen dos conos, uno surgido del constructor `Next`, con una función de transición $(o : ord)(F_o : (P\ o) \to (P\ (Next\ o))$ y el otro proveniente de que, dado un elemento $\alpha : nat \to ord$, para cada número natural n se obtiene un ordinal $(\alpha\ n)$, el cual tiene asociado un conjunto $P\ (\alpha\ n)$, por lo que se necesita una función de transición adecuada para poder definir la función en el caso límite.

5.14 *Tipos Inductivos*

Son estos tipos objetos iniciales de las categorías de las álgebras determinadas por endofuntores cocontinuos. A raíz de su definición, quedan determinados los únicos homomorfismos de álgebras (catamorfismos) existentes entre ellos y las demás álgebras de la categoría. Analicemos un caso paradigmático de tipo inductivo ya tratado previamente en varios aspectos complementarios al que sigue: las listas. Su definición (ya declarada en la sección 5.3) determina sus constructores y tipos respectivos.

Las tres funciones `list_ind, list_rec, list_rect` corresponden a la generalización de las estructuras catamórficas derivadas de la declaración de tipo inductivo. La definición de `list_rec` se amolda a la condición de cono como anteriormente habíamos mencionado. La condición de cono queda plenamente determinada en la definición anterior: dado un diagrama (sobre los conjuntos A y $List\ A$) $P_i \overset{F_{ij}}{\to} P_j,\ i,j \in A$), y un objeto fijo en P_{Nil}, `list_rec` determina una familia de morfismos (indicada, de nuevo, por A y por $List\ A$) $c_i : () \to P_i$ verificando la conmutatividad $F_{ij}(c_i) = c_j$.

Al igual que ocurría con `nat_rec`, esta función es más general que un catamorfismo. De hecho, el catamorfismo de listas puede ser definido como sigue, indicando las restricciones impuestas (concernientes al recorrido de la función (un solo tipo) y a las funciones usadas como argumento (también una sola)):

```
Definition cata_list :=
 [A,C:Set][a:C][F:C->C](list_rec A [_:(list A)]C
 a [_:A][_:(list A)]F).
```

La función `maplist`, que precisaremos para trabajar con el cotipo de los árboles, se define de la siguiente forma:

```
Definition maplist :=[A,B:Set][f:A->B]
(List _rec A [_:(List  A)](List  B)
(Nil B) [y:A][l:(List  A)][m:(List  B)]
(Cons B (f y) m)).
```

La condición que debe verificar la función `maplist` para ser la asociada a un funtor es la siguiente:

```
Theorem funtor:(A,B,C:Set)(f:A->B)(g:B->C)
(xs:(List  A))
((maplist B C g (maplist A B f xs))=
(maplist A C (comp A B C f g) xs)).
```

Una demostración de que tal hecho es verdadero es la que mostramos a continuación:

```
funtor < Intros;Elim xs;Simpl;Trivial;Intros.
funtor < Simpl;Elim H;Trivial.
```

Al igual que las funciones predecesor y factorial son ejemplos de funciones no definibles como catamorfismos sobre `nat`, podemos definir con `list_rec` funciones más allá de los catamorfismos. Así ocurre con el denominado **rightwards fold** [Gibbons 1994]. Su especificación es como sigue:

$$(f, \oplus) \rightsquigarrow [a_1, ..., a_n] = ((a_1 \oplus a_2) \oplus \ldots) \oplus (f\ a_n)$$

En general, no es un catamorfismo sobre las listas; puede, en cambio, definirse como:

```
Definition RF:=[A:Set][f:A->A][b:A][h:A->A->A]
(list_rec A [_:(list A)]A b  ([y:A][l:(list A)][x:A]
Cases l of  Nil=>(f y) |(Cons a l')=>(h y x) end)).
```

Se presta especial atención en [Freire y Blanco 1996] a las posibilidades de optimización (teoremas de fusión, deforestación) que tienen los catamorfismos. De hecho, usando la estructura libre en la categoría de las diálgebras proveniente de dos funtores, se define (ver [Freire y Blanco 1996]) un morfismo *extensión* natural de los catamorfismos que, dado un morfismo entre dos tipos, da lugar a una función entre las estructuras de listas que esos tipos generan; en Coq podemos definir estas funciones a partir del catamorfismo

```
Definition Ext_list :=[A,B,C:Set][b:B][f:C->A][e:A->B->B]
(List _rec C [_:(List  C)]B b [y:C][_:(List  C)][m:B]
(e (f y) m)).
```

siendo `(A,B,b,e)` una estructura en la misma diálgebra que las listas.

Sean, entonces, (A, B, b, ϵ) y (C, D, d, δ) dos objetos, con $b : B$, $\epsilon : A \to B \to B$ y $d : D$, $\delta : C \to D \to D$; sean, además, $p : A \to C$, $q : B \to D$ dos morfismos verificando

$(q\ b) = d$

$(q\ (\epsilon\ a\ b)) = (\delta\ ((p\ a)\ (q\ b)),\ a : A,\ b : B.$

Entonces, dado un morfismo $f : C \to A$, se verifica que

$(q\ (Ext_{list}\ b\ f\ \epsilon x) = (Ext_{list}\ d\ (f; p)\ \delta\ x)$

siendo $x : (ListC)$. Se puede enunciar en Coq en los siguientes términos:

```
Theorem Fusion_list:
(A,B,C,D,E:Set)(b:B)(e:A->B->B)(d:D)(t:C->D->D)(f:A->C)
(g:B->D)(h:E->A)(xs:(List  E))
((g b)=d)->((x:A)(y:B)(g(e x y))=(t (f x) (g y)))->
((g(Ext_list A B E b h e xs))=
(Ext_list C D E d ([x:E](f(h x))) t xs)).
```

La comprobación del aserto se puede llevar a cabo con las siguientes tácticas:

```
Fusion_list < Intros;Elim xs;Simpl;Trivial;Intros;Simpl.
Fusion_list <  Elim H1;Elim H0;Trivial.

Subtree proved!
```

Como se ve, el teorema de fusión permite eliminar los valores intermedios producidos por los catamorfismos.

Como corolario, podemos establecer bajo qué condiciones la composición de dos catamorfismos es un catamorfismo:

Corolario 5.1. *Sean* $[a, \xi] : 1 + A \times List\ A \to (List\ A,\ [d, \theta] : 1 + B \times D \to D$ *dos* $FList$*–álgebras. Sean, además,* $f : C \to A, g : A \to B$ *dos morfismos. Entonces, si* $\forall x : A,\ l : (ListA),\ (Ext_L ist\ d\ g\ \theta)(\xi\ x) = \theta(g\ x, Ext_L ist\ d\ g\ \theta\ xs)$,

$$(Ext_L ist\ a\ f\ \xi); (Ext_L ist\ d\ g\ \theta) = Ext_L ist\ d\ (f; g)\ \theta.$$

Un teorema que nos permite eliminar los valores intermedios consumidos por los catamorfismos es el denominado teorema de la lluvia ácida (o deforestación). Su enunciado se establece como sigue: sean C un tipo, (A, B, b, ϵ) una estructura con $b : B$ y $\epsilon : A \to B \to B$ y $f : C \to A$; consideremos, además, G una funcional de orden superior de tipo $(X \to Y \to Y)\ Y \to Z \to (Z \to X) \to (Z \to Z) \to Z \to Y$, que verifica, si $\delta : (X \to Y \to Y)$, $y0 : Y$, $z0 : Z$, $f : Z \to X$, $t : Z \to Z$,

$(G\ \delta\ y0\ z0\ f\ z0) = y0$; $(G\ \delta\ y0\ z0\ f\ z) = (\delta\ (f\ z)\ (G\ \delta\ y0\ z0\ f\ (t\ z))$,

siendo $z : Z$. Entonces, el teorema afirma que, dados $h : D \to C$, $d0 : D$, $d : D$,

$(Ext_{list}\ b\ f\ \epsilon(G\ (::)\ []\ d0\ h\ d)) = (G\ \epsilon\ b\ d0\ (h; f)\ d))$.

Podemos enunciar y demostrar este teorema en Coq de la siguiente forma:

```
Parameter G :
(A,B,C,X:Set)(A->B->C)->C->X->(X->A)->(X->X)->
(C->B)->X->C.

Axiom G1 :(A,B,C,X:Set)(e:A->B->C)(c0:C)(x0:X)
(f:X->A)(g:X->X) (h:C->B)
(c0=(G A B C X e c0 x0 f g h x0)).

Hint G1.

Axiom G2 :(A,B,C,X:Set)(e:A->B->C)(c0:C)(x0:X)
```

```
(f:X->A)(g:X->X)(h:C->B)(x:X)
(~(x=x0))->
((e (f x) (h (G A B C X e c0 x0 f g h (g x))))=
(G A B C X e c0 x0 f g h x)).

Hint G2.


Lemma acidrain1 :
(A,B,C,X:Set)(e:A->B->B)(b0:B)(c0:C)(x0:X)(f:X->C)
(g:X->X)(h:C->B)(k:C->A)
( (Ext_list A B C e b0 k (G C (list C) (list C) X
[x:C][l:(list C)](Cons C x l) (Nil C) x0 f g
[x:(list C)]x x0)) =
(G A B B X e b0 x0 [x:X](k(f x)) g [x:B]x x0)).

Intros.

Replace (G C (list C) (list C) X
[x:C][l:(list C)](Cons C x l)
(Nil C) x0 f g [x:(list C)]x x0) with (Nil C).

Simpl.

Apply (G1  A B B X e b0 x0 [x:X](k (f x)) g [x:B]x).

Trivial.

Qed.

Lemma acidrain2 :
(A,B,C,X:Set)(e:A->B->B)(b0:B)(c0:C)(x0:X)(f:X->C)
(g:X->X)(k:C->A)(x:X)(~(x=x0))->
((x:X) (G A B B X e b0 x0 [x:X](k(f x)) g[x:B]x (g x))=
(Ext_list A B C e b0 k (G C (list C) (list C) X
[x:C][l:(list C)](Cons C x l) (Nil C) x0 f g
[x:(list C)]x (g x)))) ->
( (Ext_list A B C e b0 k (G C (list C) (list C) X
[x:C][l:(list C)](Cons C x l) (Nil C) x0 f g
[x:(list C)]x x)) =
(G A B B X e b0 x0 [x:X](k(f x)) g [x:B]x x)).

Intros.

Replace (G C (list C) (list C) X
```

```
[x1:C][l:(list C)](Cons C x1 l) (Nil C) x0 f g
[x1:(list C)]x1 x)
with (Cons C (f x) (G C (list C) (list C) X
[x1:C][l:(list C)](Cons C x1 l) (Nil C) x0 f g
[x1:(list C)]x1 (g x))).

Replace (G A B B X e b0 x0 [x1:X](k (f x1)) g [x1:B]x1 x)
with (e (k(f x)) (G A B B X e b0 x0 [x1:X](k (f x1)) g
[x1:B]x1 (g x))).

Simpl.

Replace (Ext_list A B C e b0 k
 (G C (list C) (list C) X [x1:C][l:(list C)](Cons C x1 l)
(Nil C) x0 f g [x1:(list C)]x1 (g x)))
with (G A B B X e b0 x0 [x1:X](k (f x1)) g [x1:B]x1 (g x)).

Trivial.

Apply H0.

Apply (G2 A B B X e b0 x0 [x1:X](k (f x1)) g [x1:B]x1).

Assumption.

Apply (G2 C (list C) (list C) X
[x1:C][l:(list C)](Cons C x1 l)
(Nil C) x0 f g [x1:(list C)]x1).

Qed.
```

Se verifica también el siguiente teorema que permite aplicar una versión más general de la deforestación (consistente en la aplicación de deforestación sobre el conjunto intermedio):

```
Lemma Ar1 : (A,X,Y:Set)(P:(list A)->Set)(f:X->A)(g:X->X)
(x0:X)(t:Y->X)(k:Y->Y)(y0:Y) (a0:(P (Nil A)))(q:(l:(list A))
(m:(list A))(P l)->(P m)) ((t y0)=x0)-> ((G A (list A)
(list A) Y [x:A][l:(list A)](Cons A x l) (Nil A) y0
[y:Y](f(t y)) k [x:(list A)]x y0)
=
(G A (list A) (list A) X [x:A][l:(list A)](Cons A x l)
(Nil A) x0 f g [x:(list A)]x (t y0))).

Intros.

Replace (G A0 (list A0) (list A0) Y [x:A0]
```

```
[l:(list A0)](Cons A0 x l) (Nil A0) y0 [y:Y](f0 (t y))
k [x:(list A0)]x y0) with (Nil A0).

Replace (t y0) with x1.

Trivial.

Trivial.

Qed.


Lemma AC2 : (A,X,Y:Set)(P:(list A)->Set)(f:X->A)(g:X->X)
(x0:X)(t:Y->X)(k:Y->Y)(y0:Y) (a0:(P (Nil A)))(q:(l:(list A))
(m:(list A))(P l)->(P m)) ((t y0)=x0)-> ((y:Y)(t(k y))
=
(g(t y)))->((y:Y)(~(t y)=x0))-> ((y:Y) (G A (list A)
(list A) X [x:A][l:(list A)](Cons A x l) (Nil A) x0 f g
[x:(list A)]x (t (k y)))
=
(G A (list A) (list A) Y [x:A][l:(list A)](Cons A x l)
(Nil A) y0 [y:Y](f(t y)) k [x:(list A)]x (k y)) ) ->
 ((y:Y)(~y=y0)) -> ((y:Y)(G A (list A) (list A) Y
 [x:A][l:(list A)](Cons A x l) (Nil A) y0
[y:Y](f(t y)) k [x:(list A)]x y)
=
(G A (list A) (list A) X [x:A][l:(list A)](Cons A x l)
(Nil A) x0 f g [x:(list A)]x (t y))).

Intros.

Replace (G A0 (list A0) (list A0) Y
[x:A0][l:(list A0)](Cons A0 x l) (Nil A0) y0
[y1:Y](f0 (t y1)) k [x:(list A0)]x y)
with (Cons A0 (f0(t y)) (G A0 (list A0) (list A0) Y
[x:A0][l:(list A0)](Cons A0 x l) (Nil A0) y0
[y1:Y](f0 (t y1)) k [x:(list A0)]x (k y))).

Replace (G A0 (list A0) (list A0) X0
[x:A0][l:(list A0)](Cons A0 x l) (Nil A0) x1 f0 g0
[x:(list A0)]x (t y))
with (Cons A0 (f0(t y)) (G A0 (list A0) (list A0) X0
[x:A0][l:(list A0)](Cons A0 x l) (Nil A0) x1 f0 g0
[x:(list A0)]x (g0(t y)))).
```

```
Replace (g0 (t y)) with (t (k y)).

Replace (G A0 (list A0) (list A0) Y
[x:A0][l:(list A0)](Cons A0 x l) (Nil A0) y0
[y1:Y](f0 (t y1)) k [x:(list A0)]x (k y))
with (G A0 (list A0) (list A0) X0
[x:A0][l:(list A0)](Cons A0 x l) (Nil A0) x1 f0 g0
[x:(list A0)]x (t (k y))).

Trivial.

Apply H2.

Apply H0.

Apply (G2 A0 (list A0) (list A0) X0
[x:A0][l:(list A0)](Cons A0 x l) (Nil A0)
x1 f0 g0 [x:(list A0)]x).

Apply H1.

Apply (G2 A0 (list A0) (list A0) Y
[x:A0][l:(list A0)](Cons A0 x l) (Nil A0) y0
[y1:Y](f0 (t y1)) k [x:(list A0)]x).

Apply H3.

Qed.
```

Observemos que el único requisito `extra` que hemos impuesto para este teorema es que, para cualquier elemento $y : Y$, $y \neq y0$, $(t\ y \neq x0$.

Es más, si queremos trabajar con el funcional `list_rec` (que, en general, no es un catamorfismo, como se ha visto), también tenemos la ventaja de poder utilizar el teorema de fusión. Se puede describir y demostrar como sigue:

```
Theorem Fusion_list_Gen:(A,B:Set)(P:(list A)->Set)
(Q:(list B)->Set)
(a:(P (Nil A))) (b:(Q (Nil B))) (f:A->B)
(h:(l:(list A))((P l)->(Q (maplist A B f l))))
(t1:(x:A)(l:(list A))((P l)->(P (Cons A x l))))
(t2:(y:B)(m:(list B))((Q m)->(Q (Cons B y m))))
(((h (Nil A) a)=(b))->
(((x:A)(l:(list A))((y:(P l)) (t2 (f x)
(maplist A B f l) (h l y) ) =
(h (Cons A x l) (t1 x l y) )))->
((l:(list A) ) (h l ( (list_rec A P a t1) l) )=
```

```
((list_rec B Q b t2) (maplist A B f l) ))  )).

Fusion_list_Gen < Intros;Elim l;Simpl;Trivial.
Fusion_list_Gen < Intros;Simpl.
Fusion_list_Gen < Replace (h (Cons A y l0)
Fusion_list_Gen < (t1 y l0 (list_rec A P a t1 l0)))
Fusion_list_Gen < with (t2 (f y) (maplist A B f l0)
Fusion_list_Gen < (h l0 (list_rec A P a t1 l0))).
Fusion_list_Gen < Replace (list_rec B Q b t2 (maplist A B f l0))
Fusion_list_Gen < with (h l0 (list_rec A P a t1 l0)).
Fusion_list_Gen < Trivial.
Fusion_list_Gen < Apply (H0 y l0 (list_rec A P a t1 l0)).
Fusion_list_Gen < Qed.
```

A continuación mostramos dos simples ejemplos de la utilización de este teorema, cada uno con una versión no optimizada y otra ya optimizada: Versión no optimizada:

```
Definition suma :=
Fix F {F [l:(list nat)]:nat :=
Cases l of
Nil  => O
|(Cons  x l') =>(plus x (F l')) end}.

Definition P := [l:(list nat)](list nat).
Definition Q := [b:(list nat)]nat.

Definition a := (Nil nat).
Definition b := O.

Definition h := [l:(list nat)](suma l).

Definition t1 := [x:nat][l:(list nat)][m:(list nat)]
(Cons nat x m).

Definition t2 := [x:nat][l:(list nat)][n:nat](plus n x).

Definition H1 := (list_rec nat P a t1).
Definition nonoptimizedsum :=[l:(list nat)] (h (H1 l)).
```

Como vemos, primeramente la función `H1` construye la lista de enteros y, posteriormente, se aplica el morfismo `h`. En la siguiente versión optimizada, a medida que se va construyendo la lista se aplica la función:

```
Definition optimizedsum := (list_rec nat Q b t2).
```

Un ejemplo un tanto más complejo es el siguiente donde usamos tipos que dependen de la lista utilizada en ese momento:

```
Definition length :=[A:Set] (list_rec A [l:(list A)]nat O
[_:A][_:(list A)][n:nat](S n)).

Definition ispar := (nat_rec [_:nat]bool true
[_:nat][b:bool](if b then false else true)).

Inductive ListN [A:Set]:nat->Set:= lista_0:A->(ListN A O)
|lista_S:(n:nat)A->(ListN A n)->(ListN A (S n)).

Definition P := [xs:(list nat)]
(ListN nat (length nat xs)).
Definition Q := [xs:(list nat)]
(ListN bool (length nat xs)).

Definition mapListN := [A,B:Set][f:A->B]
(ListN_rec A [n:nat][_:(ListN A n)](ListN B n)
[y:A](ListN_0 B (f y))
[n:nat][y:A][_:(ListN A n)][x:(ListN B n)]
(ListN_S B n (f y) x)).

Definition f := [x:nat][l:(list nat)][xs:(P l)]
(ListN_S nat (length nat l) (plus x (length nat l)) xs).

Definition g := [x:nat][l:(list nat)][ys:(Q l)]
(if (ispar (plus x (length nat l)))
then
(ListN_S bool (length nat l) true ys)
else
(ListN_S bool (length nat l) false ys)).

Definition h :=[l:(list nat)]
(mapListN nat bool ispar (length nat l)).

Theorem alpha: (x:nat)(l:(list nat))(xs:(P l))
((h (Cons nat x l) (f x l xs))=(g x l (h l xs))).

Intros.

Unfold h; Unfold f; Unfold g.

Simpl.

Case (iseven (plus x (length nat l))).

Trivial.
```

```
Trivial.

Qed.

Definition FN :=[l:(list nat)]
(h l (list_rec nat P (ListN_0 nat O) f l)).

Definition FB :=[l:(list nat)]
(list_rec nat Q (ListN_0 bool true) g l).

Theorem beta : (l:(list nat))(FN l)=(FB l).

Induction l.

Trivial.

Intros.

Simpl.

Unfold FN.

Unfold list_rec.

Rewrite alpha.

Rewrite <- H.

Trivial.

Save.
```

La utilización abstracta de las listas permitirá, en la próxima sección, acercarse al cotipo de los árboles; se verá que, para poder abarcar todas sus posibilidades de abstracción, es preciso hacer uso de ella. Además se comprobará (ver [Freire y Blanco 1996]), lo útil que resulta poder trabajar simultáneamente con estructuras algebraicas y coalgebraicas asociadas a un mismo tipo.

En la siguiente sección mostramos primeramente un ejemplo más general que los previamente descritos abarcando todas las situaciones que podemos encontrarnos en una declaración de tipo inductivo y, posteriormente, enunciamos y demostramos el teorema de fusión generalizado sobre tipos dependientes.

5.15 *El teorema de fusión para tipos dependientes*

```
Inductive test [A,B:Set]:Set :=
```

```
uno:A->(test A B)
|dos:A->B->(test A B)
|tres:A->(test A B)->B->(test A B).


Definition maptest := [A,B,C,D:Set][f:A->C][g:B->D]
Fix F {F [l:(test A B)]:(test C D) :=
Cases l of
   (uno x) => (uno C D (f x))
   |(dos x y) =>(dos C D (f x) (g y))
   |(tres x l y) =>(tres C D (f x)
                  (F l) (g y))
end}.

Theorem fusion : (A,B,C,D:Set)(f:A->C)(g:B->D)
(P:(test A B)->Set)(Q:(test C D)->Set) (f0:(x:A)
(P (uno A B x)))(f1:(x:A)(y:B)(P (dos A B x y)))
(f2:(x:A)(l:(test A B))(P l)->(y:B)(P (tres A B x l y)))
 (g0:(x:C)(Q (uno C D x)))
 (g1:(x:C)(y:D)(Q (dos C D x y)))
 (g2:(x:C)(l:(test C D))(Q l)->
 (y:D)(Q (tres C D x l y))) (h:(l:(test A B))(P l)->
 (Q (maptest A B C D f g l)))
 ( (y:A)(h (uno A B y) (f0 y))  =(g0 (f y)) ) ->
 ( (y:A)(z:B)(h (dos A B y z) (f1 y z))=(g1 (f y) (g z)) )
 -> (((x:A)(l:(test A B))(y:(P l)) (z:B) (g2 (f x)
 (maptest A B C D f g l) (h l y) (g z) )
 = (h (tres A B x l z) (f2 x l y z))))
 ->((l:(test A B)) (h l (test_rec A B P f0 f1 f2 l))
 = (test_rec C D Q g0 g1 g2 (maptest A B C D f g l )) ).

Intros; Elim l; Simpl.
EAuto.
EAuto.
Replace (test_rec C D Q g0 g1 g2 (maptest A B C D f g t))
with (h t (test_rec A B P f0 f1 f2 t)).
EAuto.
Qed.
```

Se pretende hacer una demostración global del teorema de fusión para cualquier tipo definido inductivamente. Para poder entender el significado del teorema hacemos un pequeño resumen de los tipos inductivos introduciendo los términos que vayamos a necesitar en el decurso de la exposición.

Partimos, pues, de I, un tipo inductivo y polimórfico con n parámetros. Consideremos sus k constructores $N_1, \ldots N_r, R_1, \ldots R_s$, siendo los r primeros no recursivos y recursivos

los s siguientes. Recordemos que esto implica que cada tipo N_i es de la forma (usando expresiones curryficadas) ${A_1}^i \to {A_2}^i \to \ldots \to {A_{n_i}}^i \to (I\ A_1 \ldots\ A_n)$ mientras que el tipo R_j es de la forma ${A_1}^j \to {A_2}^j \to \ldots \to (I\ A_1 \ldots\ A_n) \to \ldots {A_{n_j}}^j \to (I\ A_1 \ldots\ A_n)$ (observemos que I tiene los los mismos argumentos con los que previamente ha sido definido, lo cual excluye la posibilidad de generar tipos anidados). Entonces, al definir el tipo inductivo, el sistema **Coq** genera automáticamente el funcional `I_{rec}`, que representa la expansión de la recursión primitiva sobre el tipo I; es decir, dados una familia de conjuntos $P : (I\ A_1\ \ldots A_n) \to Set$ y morfismos f_i (uno por cada constructor) cuyos tipos denotaremos por $\Delta(T_i, P)$ [Paulin y Werner 1998] (siendo T_i el nombre del constructor) estos suplirán a los constructores correspondientes. Los tipos de estas funciones son los siguientes según sea el caso de que los argumentos del constructor sean o no recursivos.

1. Si los argumentos no son recursivos el tipo de f_i es

$$\Delta(N_i, P) \equiv$$

$$(x_1{}^i : A_1{}^i)(x_1{}^i : A_1{}^i) \ldots (x_{n_i}{}^i : A_{n_i})(P(N_i\ A_1 \ldots A_n\ x_1{}^i\ x_2{}^i \ldots x_{n_i}{}^i)).$$

2. Suponiendo que el el k-ésimo argumento del constructor R_j es recursivo (o sea, es $(I\ A_1 \ldots A_n)$); entonces, tenemos que el tipo de f_j es:

$$\Delta(R_j, P) \equiv$$

$$(x_1{}^j : A_1{}^j) \ldots (x_{k-1}{}^j : A_{k-1}{}^j)(x : (I\ A_1 \ldots\ A_n)$$
$$(P\ x) \to \ldots (x_{n_j}{}^j : A_{n_j}{}^j) \to$$

$$(P\ (R_j A_1 \ldots A_n\ x_1{}^j\ \ldots x_{k-1}{}^j\ x \ldots x_{n_j}{}^j)).$$

Entonces, la semántica de `I_{rec}` es como sigue:

1. $$I_{rec}\ A_1\ \ldots\ A_n\ P\ f_1\ \ldots\ f_k\ (N_i\ u_1\ \ldots\ u_{n_i}) \triangleright f_i\ u_1\ \ldots\ u_{n_i};$$

2. $$I_{rec}\ A_1\ \ldots\ A_n\ P\ f_1\ \ldots\ f_k\ (R_j\ u_1\ \ldots\ x \ldots\ u_{n_i}) \triangleright$$

$$f_j\ u_1\ \ldots\ x\ (I_{rec}\ A_1\ \ldots\ A_n\ P\ f_1\ \ldots\ f_k\ x) \ldots\ u_{n_i}.$$

Consideremos, a continuación, una familia de conjuntos $B_1, \ldots, B_n$ y morfismos $t_i : A_i \to B_i$, $1 \leq i \leq n$. Tomemos, además, $Q : (I\ B_1\ \ldots\ B_n) \to Set$ otra familia de conjuntos y $\{g_i\}$ una familia de morfismos similar a la $\{f_i\}$ supliendo la familia P por la Q. Por último, sea h una familia de morfismos, $h : (x : (I\ A_1\ \ldots\ A_n))(P\ x) \to (Q\ (mapI\ t_1 \ldots t_n)\ x)$, satisfaciendo las conmutatividades siguientes:

$\forall ({x_1}^i : {A_1}^i) \ldots ({x_{n_i}}^i : {A_{n_i}}^i)$

1.
$$(h\ N_i\ A_1\ \ldots A_n\ {x_1}^i \ldots {x_{n_i}}^i)(f_i\ {x_1}^i \ldots {x_{n_i}}^i)$$

$$= (g_i\ ({t_1}^i\ {x_1}^i) \ldots ({t_{n_i}}^i\ {x_{n_i}}^i));$$

$\forall ({x_1}^i : {A_1}^i) \ldots ({x_{n_i}}^i : {A_{n_i}}^i),\ (l : I\ {A_1}^i \ldots {A_{n_i}}^i),\ (y : (P\ l)),$

2.
$$(h\ R_j\ A_1 \ldots A_n\ {x_1}^j \ldots l\ y\ \ldots {x_{n_j}}^j)\ (f_j\ {x_1}^j \ldots l\ y \ldots\ {x_{n_j}}^j)$$

$$= (g_j\ ({t_1}^j\ {x_1}^j) \ldots (mapI\ A_1\ \ldots\ A_n\ B_1\ \ldots\ B_n\ t_1\ \ldots\ t_n\ l)\ (h\ l\ y) \ldots$$
$$({t_{n_j}}^j\ {x_{n_j}}^j)).$$

Podemos ya enunciar y demostrar el teorema generalizado de fusión que afirma que, en el contexto previamente establecido, para cada elemento $x : (I\ A_1\ \ldots\ A_n)$ se verifica

$$h\ x\ (I_{rec}\ P\ f_1\ \ldots\ f_k\ x)$$
$$= (I_{rec}\ Q\ g_1\ \ldots\ g_k\ (mapI\ A_1\ \ldots\ A_n\ B_1\ \ldots\ B_n\ t_1\ \ldots\ t_n\ x)).$$

La demostración consta de dos parte bien diferenciadas dependiendo de que el constructor sea o no recursivo. Supongamos, pues, en primer lugar, que actuamos sobre un constructor no recursivo. Tenemos la siguiente cadena de razonamientos:

$$\forall\ ({x_1}^i : {A_1}^i) \ldots ({x_{n_i}}^i : {A_{n_i}}^i),$$

$$h\ (N_i\ {A_1}^i\ {A_{n_i}}^i\ {x_1}^i \ldots {x_{n_i}}^i)(I_{rec}\ P\ f_1\ \ldots\ f_k\ (N_i\ {A_1}^i\ {A_{n_i}}^i\ {x_1}^i \ldots {x_{n_i}}^i))$$

$$= h\ (N_i\ {A_1}^i\ {A_{n_i}}^i\ {x_1}^i \ldots {x_{n_i}}^i)\ (f_i\ {x_1}^i \ldots {x_{n_i}}^i)$$

$$= (g_i\ ({t_1}^i\ {x_1}^i) \ldots ({t_{n_i}}^i {x_{n_i}}^i))$$

por la hipótesis (1.) asumida sobre la función h. Pero esta última expresión coincide con

$$(I_{rec}\ Q\ g_1\ \ldots\ g_k\ (mapI\ A_1\ \ldots\ A_n\ B_1\ \ldots\ B_n\ t_1\ \ldots\ t_n$$
$$({N_i}\ {A_1}^i\ {A_{n_i}}^i\ {x_1}^i \ldots {x_{n_i}}^i))).$$

Veamos cómo actuar sobre el constructor recursivo anteriormente descrito $R_j : {A_1}^j \rightarrow \ldots (I\ A_1\ \ldots\ A_n) \rightarrow \ldots {A_{n_j}}^j \rightarrow (I\ A_1\ \ldots\ A_n)$.

Queremos ver que,

$$\forall\ ({x_1}^j : {A_1}^j) \ldots (x : (I\ A_1\ \ldots\ A_n)) \ldots ({x_{n_j}}^j : {A_{n_j}}^j),$$

$$h(R_j\ A_1 \ldots A_n\ {x_1}^j \ldots x\ \ldots {x_{n_j}}^j)$$

$$(I_{rec}\ A_1 \ldots A_n\ P\ f_1\ \ldots\ f_k\ (R_j\ A_1 \ldots A_n\ {x_1}^j \ldots x\ \ldots {x_{n_j}}^j))$$

$$= (I_{rec}\ B_1 \ldots B_n\ Q\ g_1\ \ldots\ g_k$$

$$(mapI\ A_1 \ldots A_n\ B_1 \ldots B_n\ t_1 \ldots t_n\ (R_j\ A_1 \ldots A_n\ {x_1}^j \ldots x\ \ldots {x_{n_j}}^j))).$$

Reduciendo ambas expresiones, equivale a ver que

$$h(R_j\ A_1 \ldots A_n\ {x_1}^j \ldots x\ \ldots {x_{n_j}}^j)$$
$$(f_j\ {x_1}^j \ldots x\ (I_{rec}\ A_1 \ldots A_n\ P\ f_1\ \ldots\ f_k\ x) \ldots {x_{n_j}}^j)$$

$$= g_j\ ({t_1}^j\ {x_1}^j) \ldots (mapI\ A_1 \ldots A_n\ B_1 \ldots B_n\ h_1 \ldots h_n\ x)$$

$$(I_{rec}\ B_1 \ldots B_n\ Q\ g_1\ \ldots\ g_n$$
$$(mapI\ A_1 \ldots A_n\ B_1 \ldots B_n\ h_1 \ldots h_n\ x)) \ldots ({t_{n_j}}^j\ {x_{n_j}}^j)).$$

Pero, utilizando la hipótesis de inducción asumida sobre estos constructores cuando aplicamos la expresión sobre elemento x, podemos reemplazar el término

$$(I_{rec}\ B_1 \ldots B_n\ Q\ g_1\ \ldots\ g_n\ (mapI\ A_1 \ldots A_n\ B_1 \ldots B_n\ h_1 \ldots h_n\ x))$$

por

$$h\ x\ (I_{rec}\ A_1 \ldots A_n\ P\ f_1\ \ldots\ f_n\ x),$$

quedando la igualdad buscada de la siguiente forma:

$$h(R_j\ A_1 \ldots A_n\ {x_1}^j \ldots x\ \ldots {x_{n_j}}^j)$$

$$(f_j\ {x_1}^j \ldots x\ (I_{rec}\ A_1 \ldots A_n\ P\ f_1\ \ldots\ f_n\ x) \ldots {x_{n_j}}^j)$$

$$= g_j\ ({t_1}^j\ {x_1}^j) \ldots (mapI\ A_1 \ldots A_n\ B_1 \ldots B_n\ t_1 \ldots t_n\ x)$$

$$(h\ x\ (I_{rec}\ A_1 \ldots A_n\ P\ f_1\ \ldots\ f_n\ x))\ \ldots ({t_{n_j}}^j\ {x_{n_j}}^j)$$

que es la hipótesis (2.) asumida sobre la función h, quedando, entonces, la proposición demostrada.

5.16 *Tipos coinductivos*

Hemos visto en la sección 5.2 que los cotipos corresponden a las coálgebras terminales de funtores continuos. Entonces, dado un funtor continuo F, que la coálgebra (ν_F, out_F) (out_F un isomorfismo) sea terminal significa que, dada una coálgebra arbitraria, (A, ξ), con $\xi : A \to F(A)$, existirá un único homomorfismo de F-coálgebras $ana\ \xi : A \to \nu_A$ (es decir, verificando la igualdad $\xi; F(ana\ \xi) = (ana\ \xi); out_F$. Lo primero que debemos observar es que, al construir un cotipo, no damos sus constructores, sino que definimos unos *destructores*; en efecto, al definir el cotipo de los árboles de tipo A hemos utilizado el funtor $F_A(B) = A * (ListB)$; por ello, la coálgebra (B, ξ) determina el morfismo $\xi : B \to A * (ListB)$. El isomorfismo $out_{F_A} : tree\ A \to A * list(tree\ A)$ no indica cómo construir los elementos de $tree\ A$ sino que, dado uno de ellos, lo descompone en sus constructores (uno de tipo A y el otro de tipo $list(tree\ A)$). Ahora bien, el hecho de indicar que la coálgebra terminal conlleva un isomorfismo nos permite describir el cotipo (coinductivo) como un tipo (inductivo) no especificando sus destructores (provenientes de out_F) sino sus constructores (a partir de ${out_F}^{-1}$). Se ve en [Freire y Blanco 1996] la ventaja que tiene poder usar las dos condiciones en un tipo; su carácter libre y colibre da lugar a la generación automática de funciones desde y hacia él, enriqueciendo su relación con el entorno. Utilizando, además, la capacidad de abstracción de las diálgebras (tal como se puede ver en [Freire y Blanco 1996] o en [Erwig 1998]) podemos ampliar el uso de estas técnicas a tipos no algebraicos.

Volviendo a las cuestiones de implementación en Coq y considerando las restricciones impuestas en mor del denominado *carácter positivo*, en el sistema no está permitido declarar el cotipo de los árboles como su carácter de coálgebra indica:

```
CoInductive tree [A:Set]:Set :=
HT: (tree A)->(A->(list(tree A))).
Error: Non positive Ocurrence in
(A:Set)(tree A)->A->(list (tree A))
```

Entonces, si queremos definir el cotipo de los árboles, debemos hacer uso de sus constructores no de sus destructores:

```
CoInductive tree [A:Set]:Set :=
Nodo:A->(list (tree A))->(tree A).
tree defined
```

A primera vista vemos que hemos perdido la generación automática de funciones hacia el cotipo; en cambio, podemos construir sus términos del mismo modo que anteriormente:

```
Definition termino :=
(Nodo nat (S(S(S O)))  (Nil (tree nat))).
```

Ya que en la declaración coinductiva hemos determinado los constructores del tipo, cabe pensar en definir éste inductivamente:

```
Inductive tree [A:Set]:Set :=
Nodo:A->(list (tree A))->(tree A).
```

```
tree_ind is defined
tree_rec is defined
tree_rect is defined
tree is defined
```

Ahora, como cabía esperar, sí se generan funciones iniciales (desde el tipo inductivo). Pero, si analizamos detenidamente la definición, vemos que ésta no se corresponde con lo que pretendemos de una función con pretensión de generalizar a los catamorfismos. Efectivamente, como ya sabemos, el catamorfismo correspondiente al tipo indcutivo anterior debe verificar la siguiente ecuación: $catatree\ \epsilon\ (Nodo\ y\ l) = \epsilon\ y\ (map_{list}\ (catatree\ \epsilon)\ l)$. Pues bien, la parte correspondiente a la aplicación de `maplist` no se lleva a cabo. Y no sólo eso: es imposible, en estas condiciones, forzar su definición. Esta característica limitativa la podemos ver, también, en los tipos mutuamente inductivos `arbol` y `bosque` :

```
Section tree.

Variable A,B:Set.
Inductive arbol : Set := nodo : A-> bosque -> arbol
with bosque : Set :=
|hoja : B -> bosque
|cons: arbol -> bosque ->bosque.

End tree.

Definition maparbol := [A,B,C,D:Set] [f:A->C] [g:B->D]
Fix F {F [t:(arbol A B)]:(arbol C D) :=
Cases t of
(nodo a bs) => (nodo C D (f a) (mapbosque bs))
end
with
mapbosque [bs':(bosque A B)]:(bosque C D) :=
Cases bs' of
(hoja b) => (hoja C D (g b))
|(cons t' bs'') => (cons C D (F t') (mapbosque bs''))
end}.
```

La función `maparbol` no puede ser definida como una instanciación de `arbol_rec` porque en ésta se prescinde de la condición recursiva del tipo `arbol`. Es por ello preciso definirla ad-hoc. En este caso la declaración sí es aceptada por el sistema al contrario de lo que ocurre con el tipo anterior.

Esto nos incita a pensar si tal solución es aceptable pare el tipo de los árboles. ¿Cabe la posibilidad de declararlo mutuamente con las listas? Y en tal caso, ¿podremos definir la función maprtree? La respuesta a ambas cuestiones es positiva y proficua:

```
Section Tree.
```

```
Variable A : Set.

Inductive tree : Set :=
Nodo:A-> List -> tree
with
List : Set :=
|Nil : List
|Cons: tree -> List -> List .
End Tree.

Definition maptree := [A,B:Set] [f:A->B]
Fix F {F [t:(tree A)]:(tree B) :=
Cases t of
(Nodo a bs) => (Nodo C (f a) (mapList bs))
end
with
mapList [bs':(List A) ]:(List B) :=
Cases bs' of
Nil => (Nil B)
|(Cons x xs) => (Cons B (F x) (mapList xs))
end}.
```

Esta declaración permite utilizar tal como cabe esperar el tipo de los árboles, solventando la limitación que su declaración *embebida* [McBride 1999] imponía.

¿Qué ventajas aporta utilizar la definición coinductiva con respecto a su isomorfa inductiva? Fundamentalmente, que podemos definir objetos infinitos y, además, caracterizar a estos tipos por medio de su condición terminal. En el caso de los árboles podríamos retomarlos y declarar el tipo como sigue:

```
Inductive excep [A,B:Set] : Set :=
Null: (excep A B)
|Ok:A->B->(excep A B).

CoInductive tree [D:Set] : Set :=
Nodo: (A,B,C:Set)
(B->(excep A B))->(A->D)->(C->(List (tree D)))->B->(tree D).
```

Consideremos el cotipo de las *streams*, cadena infinita de elementos de un tipo determinado (utilizamos, por comodidad notacional y como gran ventaja a la hora de la demostración de los teoremas, el tipo `producto`, anteriormente definido inductivamente pero, en esta ocasión, con su propio carácter final). Añadimos las funciones correspondientes a los destructores de ambos cotipos (`Fst, Snd` en el caso de `producto` y `hd, tl` para `stream`).

```
CoInductive prod [A,B:Set]:Set:=
Pair:A->B->(prod A B).
Definition Fst:=[A,B:Set][x:(prod A B)]
```

```
Cases x of (Pair a b)=>a end.
Definition Snd:=[A,B:Set][x:(prod A B)]
Cases x of (Pair a b)=>b end.

CoInductive stream [A:Set]: Set :=
listinf :A->(stream A)->(stream A).
Definition hd:= [A:Set][x:(stream A)]
Cases x of (listinf a _) => a end.
Definition tl:= [A:Set][x:(stream A)]
Cases x of (listinf _ s) => s end.
```

Dado el carácter final de `stream`, podemos definir una función que, siendo única, relaciona cualquier estructura análoga a la de `stream` (es decir, un objeto `(A,B,t)`, donde `A, B` son un par de tipos y `f` es un morfismo `t:B->A*B`) en éste.

```
Definition ana_stream :=[A,B,C:Set][t:B->(prod A B)]
[f:A->C]
(CoFix F {F:B->(stream C):=
[b:B]
(listinf C (f (Fst A B (t b))) (F (Snd A B (t b))))}).
```

Ese carácter final de `stream` queda reflejado en las dos Siguientes propiedades:

```
Lemma streams_hd:(A,B,C,D,E:Set)
(h1:D->A)(h2:E->B)(s:E->(prod D E))
(t:B->(prod A B))(f:A->C)
((e:E)(Fst A B (t (h2 e)))= (h1 (Fst D E (s e))))->
((e:E) (Snd A B (t (h2 e)))=(h2 (Snd D E (s e))))->
((x:D) (f (h1 x)) =(comp D A C h1 f x))->
((e:E)(hd C (ana_stream A B C t f (h2 e)))=
(hd C (ana_stream D E C s (comp D A C h1 f) e))).
streams_hd < Intros;Simpl.
streams_hd < Replace (comp D A C h1 f (Fst D E (s e)))
streams_hd < with (f (h1 (Fst D E (s e)))).
streams_hd < Replace (h1 (Fst D E (s e)))
streams_hd < with (Fst A B (t (h2 e))).
streams_hd < Trivial;Apply H. Apply (H1 (Fst D E (s e))).
streams_hd < Qed.

Lemma streams_tl:(A,B,C,D,E:Set)
(h1:D->A)(h2:E->B)(s:E->(prod D E))
(t:B->(prod A B))(f:A->C)
((e:E)(Fst A B (t (h2 e)))= (h1 (Fst D E (s e))))->
((e:E) (Snd A B (t (h2 e)))=(h2 (Snd D E (s e))))->
((x:D) (f (h1 x)) =(comp D A C h1 f x))->
((e:E)(hd C (tl C ((ana_stream A B C t f) (h2 e))))=
```

```
(hd C (tl C ((ana_stream D E C s (comp D A C h1 f)) e))).
streams_tl < Intros;Simpl.
streams_tl < Replace
streams_tl < (comp D A C h1 f (Fst D E (s (Snd D E (s e)))))
streams_tl < with (f (h1 (Fst D E (s (Snd D E (s e)))))).
streams_tl < Replace
streams_tl < (h1 (Fst D E (s (Snd D E (s e))))) with
streams_tl < (Fst A B (t (h2 (Snd D E (s e))))).
streams_tl < Replace
streams_tl < (h2 (Snd D E (s e))) with (Snd A B (t (h2 e))).
streams_tl < Trivial.
streams_tl < Apply H0; Apply H; Apply H1; Qed.
```

Contrariamente a lo que ocurre con las funciones definidas inductivamente, esta función no va disminuyendo su argumento; en cada iteración hay que proveer un método de construcción efectivo de un nuevo elemento del objeto infinito.

Para tipos más complejos (como por ejemplo el de los árboles anteriormente construido) esta definición no es válida por las limitaciones impuestas por el sistema: éste no permite utilizar el operador `CoFix` si no es "protegido"por el constructor del tipo: en el caso que nos ocupa, la declaración del anamorfismo que le corresponde debería ser como sigue:

```
Definition ana_tree :=[A,B,C:Set][t:B->A*(List  B)]
[f:A->C] CoFix F {F:B->(tree C):= [b:B]
   Cases (t b) of
     (a,bs) => (Nodo (f a) (map_list B (tree C) F bs))
   end}.
```

El sistema nos informa de que hemos cometido un error; éste surge porque la aplicación correcursiva de la definición está integrada dentro de `map_list`, no limitada al constructor del tipo como es el requisito imperante. Ésta es una severa limitación de cara a la construcción de tipos coinductivos por medio de otros tipos previamente declarados. El porqué de esta protección lo tenemos en el carácter exigible a un tipo coinductivo: debe estar constantemente protegido o guardado por sus constructores. Ello se consigue precaviendo en cada caso la aplicación iterativa de la función con el constructor del cotipo. Por ejemplo, la siguiente función no se puede definir en el sistema:

```
CoFixpoint filter:(nat->bool)->(list nat)->(list nat) :=
[p:nat->bool][s:(list nat)]
Cases (proy nat s) of
   I =>(Nil nat)
   |(PairI x xs) =>if (p x)
   then (Cons nat x (filter p xs))
    else (filter p xs)
end.
Error during interpretation of command:
CoFixpoint filter:(nat->bool)->(list nat)->
```

```
(list nat) :=
[p:nat->bool][s:(list nat)]
Cases (proy nat s) of
   I =>(Nil nat)
   |(PairI x xs) =>if (p x) then (Cons nat x (filter p xs))
                    else
                   (filter p xs)
end.
Error: Unguarded  recursive call
 The recursive definition
 [p:nat->bool]
  [s:(list nat)]
   Cases (proy nat s) of
     I => (Nil nat)
   | (PairI x xs) =>
      if (p x) then (Cons nat x (filter p xs))
      else (filter p xs)
   end
   is not well-formed
```

Notemos que, en la segunda aplicación de `filter` ésta no está protegido por el constructor: entonces, la estructura no está preservada. Podemos, por contra, definir las siguientes funciones:

```
CoFixpoint filter:(nat->bool)->(nat->nat)->(nat->nat)->
(list nat)->(list nat) :=
[p:nat->bool][f,g:nat->nat][s:(list nat)]
Cases (proy nat s) of
   I =>(Nil nat)
   |(PairI x xs) => if (p x) then
                    (Cons nat (f x) (filter p f g xs))
                    else
                    (Cons nat (g x) (filter p f g xs))
end.

CoFixpoint filter2:(nat->bool)->(nat->nat)->(nat->nat)->
(stream nat)->(stream nat) :=
[p:nat->bool][f,g:nat->nat][s:(stream nat)]
Cases (p (hd nat s)) of
   true   =>(listinf nat (f (hd nat s))
             (filter2 p f g (tl nat s)))
   |false =>(listinf nat (g (hd nat s))
             (filter2 p f g (tl nat s)))
end.
```

Vemos, en ambos casos, que la aplicación recursiva de la función está protegida por el constructor correspondiente, de ahí su correción.

Un problema, entonces, a estudiar, es cómo resolver estas limitaciones de Coq. Otra tarea interesante es relajar la restricción explícitamente indicada en la construcción de tipos relativa a la no definición de tipos anidados. Por ejemplo, si intentamos declarar el tipo

```
Inductive Rope [A,B:Set] := Nothing: (Rope A B)|
Tor:A->(Rope B A).
```

el sistema nos avisa de una mala utilización de los parámetros (la diferente posición de las variables determina este error). Este caso singular es de fácil solución; el tipo anterior es equivalente al siguiente que no presenta problemas de cara a su definición inductiva:

```
Inductive Rope1 [A,B:Set] := N: (Rope1 A B)| E:A->(Rope1 A B)
|Tor:A->B->(Rope1 A B)->(Rope1 A B).
Rope1_ind is defined.
Rope1_rec is defined.
Rope1_rec is defined.
Rope1 is defined.
```

La definición de `cataRope` es como sigue:

```
Definition cata_rope := [A,B,C:Set] [c0:C] [f:A->C]
[theta:C->C]
(Rope_rec A B [_:(Rope A B)]C c0 f
[_:A][_:B][_:(Rope A B)]theta).
```

Vemos que esta definición construida a través de `Rope_rec` coincide con la previamente obtenida en la sección dedicada a los tipos anidados.

Como anécdota, diremos que el hecho de trabajar con tipos curryficados permite una mayor expresividad en sus funciones derivadas. Si definimos el tipo `Rope` de la forma

```
Inductive Rope2 [A,B:Set] :=
N2: (Rope2 A B)
| E2:A->(Rope2 A B)
|Tor2:A*B*(Rope2 A B)->(Rope2 A B).
Warning: Ignoring recursive call
Rope2_ind is defined.
Rope2_rec is defined.
Rope2_rec is defined.
Rope2 is defined.
```

Esta limitación concierne a la recursión invocada en el constructor `Tor` que, en este caso, no es respetada. Comparemos las dos acciones:

```
Print Rope1_rec.
Rope1_rec =
[A,B:Set]  [P:(Rope1 A B)->Set] [f:(P (N A B))]
[f0:(y:A)(P (E A B y))]
```

```
[f1:(y:A)(y0:B)(r:(Rope1 A B))(P r)->(P (Tor A B y y0 r))]
     Fix F
       {F [r:(Rope2 A B)] : (P r) :=
          Cases r of
            N => f
          | (E y) => (f0 y)
          | (Tor y y0 r0) => (f1 y y0 r0 (F r0))
          end}
     : (A,B:Set)
        (P:(Rope2 A B)->Set)
         (P (N A B))
          ->((y:A)(P (E A B y)))
             ->((y:A)(y0:B)(r:(Rope1 A B))(P r)->
               (P (Tor A B y y0 r)))
                ->(r:(Rope1 A B))(P r)

Print Rope2_rec.
Rope2_rec =
[A,B:Set]
 [P:(Rope2 A B)->Set]
  [f:(P (N2 A B))]
   [f0:(y:A)(P (E2 A B y))]
    [f1:(p:A*B*(Rope2 A B))(P (Tor2 A B p))]
     Fix F
       {F [r:(Rope2 A B)] : (P r) :=
          Cases r of
            N2 => f
          | (E2 y) => (f0 y)
          | (Tor2 p) => (f1 p)
          end}
     : (A,B:Set)
        (P:(Rope2 A B)->Set)
         (P (N2 A B))
          ->((y:A)(P (E2 A B y)))
             ->((p:A*B*(Rope2 A B))(P (Tor2 A B p)))
                ->(r:(Rope2 A B))(P r)
```

En la aplicación de `Rope1_rec` vemos que al elemento `Tor y y0 r0` le asigna el valor `f1 y y0 r0 (F r0)`, preservando la recursividad en `r0`; en cambio, al comprobar cómo actúa `Rope2_rec` coprobamos que al mismo elemento (en forma de producto `p`) le asigna `f1 p` omitiendo la componente recursiva.

Naturalmente, no todos los tipos tienen esta capacidad de equivalencia con uno inductivo: esos, en Coq, no son definibles.

6 Mónadas y sus Álgebras

6.1 *Introducción*

Durante los últimos años ha sido muy activa la utilización de las mónadas (o triples o teorías algebraicas) en computación [Wadler 1987], [Moggi 1989], [Wadler 1993], [Wadler 1994], [Fokkinga 1994], [Gordon 1994], [Meijer y Jeuring 1995], [Hu y Iwasaki 1995]. La idea consiste en utilizar la estructura que proveen las mónadas para emular efectos imperativos en programación funcional. Así, en [Gordon 1994] se usan para proveer de mecanismos de entrada/salida funcional sin cambios de estado. Tanto su utilización en semántica denotacional como en programación se han vislumbrado sumamente útiles.

Nosotros estamos interesados no tanto en las mónadas en sí como en las álgebras asociadas a ellas (Sección 2.3.4). El porqué de esta atención surge del hecho de que los tipos de datos son presentaciones ecuacionales, es decir, pueden ser caracterizados por medio de triples de Kleisli [Manes 1976]. En tal caso, es sabido que su semántica es la categoría de sus álgebras. Por consiguiente, semánticas equivalentes definen tipos de datos Morita–equivalentes y es sumamente útil saber si dos teorías algebraicas distintas tienen o no el mismo significado.

Otros resultados que resaltan la importancia de estas álgebras vienen reseñados a continuación:

Definición 6.1. *([Lambeck y Scott 1989], pág. 30)*
Dada una mónada (T, η, μ) *sobre una categoría* $\mathbf{C}$, *una resolución* $(\mathbf{D}, U, F, \epsilon)$ *de este triple consta de una categoría* $\mathbf{D}$ *y un par de funtores adjuntos* $F : \mathbf{C} \to \mathbf{D}$ *y* $U : \mathbf{D} \to \mathbf{C}$ *que verifica* $F; U = T$ *con adjunciones* η *y* ϵ *tal que* $F; \epsilon; U = \mu$.

Teorema 6.1. *([Lambeck y Scott 1989], pág. 31)*
La categoría de Eilenberg-Moore $\mathbf{C}^T$ *del triple* (T, η, μ) *sobre* $\mathbf{C}$ *da lugar a una resolución* $(\mathbf{C}^T, U^T, F^T, \epsilon^T)$ *la cual es un objeto terminal en la categoría de las resoluciones, siendo*

$$U^T : \mathbf{C}^T \to \mathbf{C},\ U^T(A, \xi) = A,\ U^T(f) = f;$$

$$F^T : \mathbf{C} \to \mathbf{C}^T,\ F^T(A) = (T(A), \mu_A),\ F^T(f) = T(f),$$

donde $\epsilon^T(A, \xi) = \xi$.

Teorema 6.2. *([Lambeck y Scott 1989], pág. 33)*

La categoría de Kleisli $\mathbf{C}_T$ *del triple* (T, η, μ) *sobre* $\mathbf{C}$ *da lugar a una resolución* $(\mathbf{C}_T, U_T, F_T, \epsilon_T)$ *la cual es un objeto inicial en la categoría de las resoluciones siendo*

$$U_T : \mathbf{C}_T \to \mathbf{C},\ U_T(A) = T(A),\ U_T(f) = T(f); \mu_B;$$

$$F_T : \mathbf{C} \to \mathbf{C}_T,\ F_T(A) = A,\ F_T(f) = f; \eta_B,$$

donde $\epsilon_T(A) = 1_{T(A)}$.

Corolario 6.1. *([Lambeck y Scott 1989], pág. 35)*
La categoría de Kleisli de una mónada es equivalente a la subcategoría plena de la categoría de Eilenberg-Moore que consta de todas las álgebras libres.

Como se menciona en [Taylor 1999] *los funtores y las mónadas pueden ser usadas para codificar la misma teoría libre pero de formas diferentes (...) El funtor (...) codifica un sistema de condiciones de clausura. Es dinámico: podemos ver la génesis del álgebra libre desde sus coálgebras bien fundadas(...) La mónada (...) generaliza la operación de clausura y es estática (...) Una mónada es necesaria para codificar leyes.*

6.2 *El triple de las excepciones*

En todos los lenguajes (imperativos y declarativos) juegan un papel muy importante las excepciones, desde el momento en que, por ejemplo, son una herramienta útil para conservar el polimorfismo. Podemos hacer uso de ellas siguiendo varias estrategias: en CAML o se definen localmente o recurriendo al genérico `raise (failure...)`; otra posibiloidad radica en construir, para cada tipo A, otro tipo, `excep A`, que sirva para retomar y reconducir los errores. Esta opción tiene la ventaja de que no se abandona la computación correspondiente al tipo original. Por consiguiente, todas las operaciones que se pueden llevar a cabo con sus valores se preservan con las excepciones de forma que es posible volver a ellos y evaluarlos. Es decir, la excepción deja de ser una huida en caso de error o fallo de algún tipo y se convierte en una herramienta de reconducción de cara a otras áreas del programa. La definición del tipo de las excepciones sobre el tipo `A` es (en sintaxis de CAML) como sigue:

```
type 'a excep=fallo|succ of 'a;;
```

Veamos, con un par de ejemplos, la simplificación que introducen estas excepciones .

Ejemplo 6.1. *Construyamos una función que tome el primer elemento de una lista:*

```
let head=function  []->fallo
| x::l->succ(x);;
```

función polimórfica de tipo `excep A`.

Ejemplo 6.2. *La función que busca en qué posición, dentro de una lista hallamos un valor satisfaciendo una determinada condición:*

```
let rec find p n=function  []->fallo
| x::l->if p(x) then succ(n) else
(find p (n+1) l);;

let find_index p l=find p 1 l;;
```

donde `find_index` *es una función de tipo* `bool*'a list->'a excep`.

La estructura de triple que definimos sobre esta transformación de tipos, nos permite relacionar el tipo original con su trasladado. Es, en este caso, sencillo construir tal triple [de Kleisli]. Cuando $f : A \to B$ es un morfismo, definimos, siendo $x \in A$ y $a \in T(A)$:

$$(excepf)a = \begin{cases} succ(fx) & \text{if a=succ(x)} \\ fallo & \text{a=fallo} \end{cases}$$

Definimos la unidad de forma obvia:

$$\eta_A(x) = succ(x),\ x \in A.$$

Por último, creamos el operador de Kleisli. Siendo $a \in T(A)$, $f : A \to T(B)$,

$$\rho_{A,B}\ a\ f = \begin{cases} succ(fx) & \text{if a=succ(x)} \\ fallo & \text{if a=fallo} \end{cases}$$

Es inmediato comprobar que $\mathbf{T}=(T, \eta, -^*)$ es un triple de Kleisli, donde, si $f : A \to T(B)$, $f^*a = \rho_{A,B}\ a\ f$ verificando $-^*$ las ecuaciones requeridas. A partir de estas definiciones, se puede derivar la construcción de la multiplicación:

$$\mu_A(a) = \rho_{T(A),A}\ a\ 1_{T(A)}.$$

Por lo tanto, siendo $x \in A$ y $a \in TT(A)$,

$$\mu_A(a) = \begin{cases} fallo & \text{if a=fallo} \\ fallo & \text{if a=succ(fallo)} \\ succ(x) & \text{if a=succ(succ(x))} \end{cases}$$

Entonces, (T, η, μ) es un triple.

6.2.1 Categoría de Eilenberg-Moore

Sea, entonces, (A,ξ) una $\mathbf{T}$-álgebra sobre este triple. En tal caso, $\xi : T(A) \to A$ es un morfismo que debe satisfacer las dos reglas ya vistas. La primera, proveniente del hecho de que $\eta_A; \xi = 1_A$, impone que

$$\xi(succ(x)) = x,\ x \in A.$$

La segunda nos dice que

$$\mu_A; \xi = T(\xi); \xi.$$

Pero, esta última ecuación no representa, realmente, condición alguna, como se ve a continuación:

$$(\mu_A;\xi)(fallo) = \xi(\mu_A(fallo)) = \xi(fallo);$$
$$(T(\xi);\xi)(fallo) = \xi(T(\xi)(fallo)) = \xi(fallo).$$

$$(\mu_A;\xi)(succ(fallo)) == \xi(fallo);$$
$$(T(\xi);\xi)(succ(fallo)) = \xi(fallo).$$

$$(\mu_A;\xi)(succ(succ(x)) = \xi(succ(x)) = x;$$
$$(T(\xi);\xi)(succ(succ(x)) = \xi(succ(x)) = x.$$

Por lo tanto, la única imposición condicionando a las **T**-álgebras es que $\xi(succ(x)) = x,\ x \in A$. Sea, entonces, $x_A = \xi(fallo), x_A \in A$. Supongamos,además, que $f : A \to B$ es un morfismo de **T**-álgebras. Debe verificarse la siguiente conmutatividad:

$$\xi_A; f = \xi_B; (excep\ f).$$

Sea, por consiguiente, $succ(x) \in TA$. La conmutatividad anterior implica que $f(x) = f(x)$, lo cual es necesariamente cierto al ser f una función. ¿Qué ocurre con `fallo`? Debe suceder que $f(\xi_A(fallo)) = \xi_B((excep\ f)(fallo))$, es decir, $f(x_A) = x_B$. En conclusión, la categoría de las **T**-álgebras contiene como objetos los de la categoría original, $A \in \mathbf{C}$, en los cuales hemos singularizado un valor $x_A \in A$. Los morfismos son funciones en **C** que conservan ese elemento singular. Resumiendo, si $f \in C^T(A,B),\ f \in C(A,B)\ y\ f(x_A) = x_B$.

Definición 6.2. . *La categoría de conjuntos con punto, C_*, es aquélla que tiene como objetos pares (A, x_A), siendo A un conjunto y x_A un elemento de A. Una flecha en esta categoría, $f : (A, x_A) \to (B, x_B)$ es una aplicación $f : A \to B$ que verifica $f(x_A) = x_B$.*

Por consiguiente, la categoría de Eilenberg-Moore sobre el triple de las excepciones es la categoría de conjuntos con punto. Tiene, por tanto, objeto cero, a saber, los conjuntos con un solo punto. Toda esta información queda condensada en la siguiente implementación en Coq.

6.2.2 La categoría de Eilenberg-Moore de las excepciones y Coq

Inicialmente definimos el tipo y las funciones precisas:

```
Inductive excep  [A:Set]:Set :=N:(excep  A)|E:A->(excep  A).

Definition mapexcep :=[A,B:Set][f:A->B][x:(excep  A)]
(Cases x of N=>(N B) | (E y)=>(E B (f y)) end).

Definition unit:=[A:Set][x:A](E A x).

Definition asoc:=[A:Set][x:(excep  (excep  A))]
```

```
(Cases x of N=>(N A)|(E y)=>y end).

Definition bind:=[A,B:Set][f:A->(excep  B)][x:(excep  A)]
(Cases x of N=>(N B) |(E y)=>(f y) end).
```

A continuación expresamos los teoremas que indican el carácter mondico de la definición:

```
Theorem uno:(A:Set)(x:(excep  A))
((asoc A (unit (excep  A) x))=x).

uno < Intros;Trivial.
uno < Qed.

Theorem dos:(A:Set)(x:(excep  (excep  (excep  A))))
((asoc A (mapexcep  (excep  (excep  A)) (excep  A) (asoc A)
x))
=
(asoc A (asoc (excep  A) x))).

dos < Intros.
dos < Elim x.
dos < Trivial.
dos < Simpl;Trivial.
dos < Qed.
```

Por último, mostramos los teoremas indicativos de que, efectivamente, la categoría de Eilenberg-Moore coincide con la de conjuntos con punto:

```
Theorem tres:(A,B:Set)(f:A->B)(a:A)(b:B)
(H:(excep  A)->A)(K:(excep  B)->B)
((x:A)(H (unit A x))=x)->
((x:(excep  (excep  A)))(H (asoc A x))=
(H (mapexcep  (excep  A) A H x)))->
((x:B)(K (unit B x))=x)->
((x:(excep  (excep  B)))(K (asoc B x))=
(K (mapexcep  (excep  B) B K x)))->
((x:(excep  A))(f (H x))=(K (mapexcep  A B f x)))->
((H (N A))=a)->((K (N B))=b)->((f a)=b).

Replace a with (H (N A)).
Replace b with (K (N B)).
Replace (N B) with (mapexcep  A B f (N A)).
Trivial.
Trivial.
Qed.
```

6.2.3 Categoría de Kleisli

Si $f \in C_T(A,B)$, sabemos que $f \in C(A,T(B))$. Así pues, f asigna a cada valor de tipo A o un valor de la forma succ(y), con $y \in B$, o la excepción (fallo). Sea $g \in C_T(B,C)$. Analicemos los casos posibles de cara a su composición. Siendo $x \in A$:
si $f(x) = y \in B$, $(f;g)(x) = g(y) \in T(C)$; es decir, $g[f(x)] = succ(z)$ o $g[f(x)] = fallo$;
si $f(x) = fallo$, $(f;g)(x) = \mu_C(Tg[fallo]) = \mu_C(fallo) = fallo$.
En resumen, la categoría de Kleisli de este triple de las excepciones es isomorfa a la categoría con los mismos objetos más un elemento singular y los morfismo son los mismos que en la categoría original con la condición de que conservan ese elemento.

Definición 6.3. *Sea A un conjunto; denominamos $lift(A)$ al conjunto A al que hemos añadido un elemento singular $\perp$.*

Entonces, estas consideraciones remiten a las denominadas funciones estrictas (definidas sobre los dominios): siendo f una función definimos la función `strict` f como aquélla que, sobre x $\neq\perp$ vale f x y sobre $\perp$ es $\perp$. Esto implica que sólo es posible calcular el valor de f sobre un valor x después de que éste esté reducido a su formar normal débil principal (WHNF), y nunca antes. Se usa esta exigencia para controlar el orden de computación en una evaluación lazy: así, podemos reducir las expresiones f y x en paralelo, mas no podremos acceder al resultado mientras que x no esté en WHNF.

Finalizamos esta sección reseñando que, si A es un objeto de C, $A \cup C_A$ es un objeto de la categoría de Kleisli y los morfismos son tales que $f(C_A) = C_B$. Es decir, la categoría de Kleisli sobre el triple de las excepciones es la categorá ıa de las conjuntos `liftados` con un elemento singular.

Ejemplo 6.3. *Supongamos que uno de los tipos es int. A la hora de dividir, bien podemos obtener valores fuera del tipo (float) bien podemos llegar a situaciones imposibles (división por 0). Ambos casos se arreglan con ese valor C_{int} que nos elimina esas adversidades:* $f : int \times int \to int \cup C_{int}$,

$$f(a) = \begin{cases} \frac{a}{b} & if\ a \dot{=} b \\ C_{int} & otro\ caso \end{cases}$$

6.2.4 La categoría de Kleisli de las excepciones y Coq

Debemos definir la composición en esta categoría para demostrar que, si la salida de la primera función es `N` la salida final es `N` (Propiedad cuatro) y que, si es un valor `E x0` la salida final es la imagen de ese valor `x0` (Propiedad cinco).

```
 Definition composition := [A,B,C:Set] [f:A->B] [g:B->C]
[x:A](g(f(x))).

Definition Kleisli_comp := [A,B,C:Set] [f:A->(MB B)]
[g:B->(MB C)]
(composition A (MB (MB C)) (MB C)
(composition A (MB B) (MB (MB C)) f (mapMB  B (MB C) g))
(asoc C)).
```

```
Theorem cuatro : (A,B,C:Set) (f:A->(MB B)) (g:B->(MB C))
(x:A) ((f x)=(N B))->(((Kleisli_comp A B C f g) x)=(N C)).

Intros.

Unfold Kleisli_comp.

Unfold composition.

Rewrite H.

Trivial.

Qed.

Theorem cinco : (A,B,C:Set) (f:A->(MB B)) (g:B->(MB C))
(x:A){y:B|(f x)=(E B y)}->
{y:B | (((Kleisli_comp A B C f g) x)=(g y))}.

Intros.

Elim H.

Intros.

Exists x0.

Unfold Kleisli_comp.

Unfold composition.

Rewrite y.

Trivial.

Qed.
```

6.3 *El triple de las continuaciones*

6.3.1 Introducción

Sea R un tipo fijo; introducimos una continuación de tipo A como un morfismo que, asigna, a cada función entre A y R, un valor de R. Es decir,

```
type 'A cont=val of ('A->'R)->'R;;
```

tipo que indicaremos con que denotaremos por $T_R(A)$. Este tipo es conocido en la literatura algebraica clásica como *doble dualidad*
[Freire 1980]. Nuestro primer objetivo es construir, sobre este tipo, un esquema de triple. Para ello, comenzamos definiendo cómo debe ser $T_Rf : T_RA \to T_RB$, siendo $f : A \to B$. Notemos, primeramente, que si $b \in T_RB$ y $a : (A \to R) \to R$, $(T_Rf)(a) : (B \to R) \to R$. Entonces, dada una función $h : B \to R$, debemos ver cómo es $((T_Rf)(a))h$. Ahora bien, $f;h : A \to B \to R$ es un morfismo entre A y R sobre el cual puede actuar a: por lo tanto definimos $((T_Rf)(a))h = a(f;h)$. Por consiguiente, $(T_Rf)(a) = \lambda h.(a(f;h))$, para cada $a \in T_RA$. Veamos, ahora, la definición de $\eta_A : A \to T_RA \equiv (A \to R) \to R$; o sea, si $h : A \to R$, $(\eta_A(a))h = h(a)$. Por lo tanto, $\eta_A(a)$ define una función, C, definida por $C(h) = h(a)$. Concluyendo, $\eta_A(a) = \lambda h.(ha)$.

Analicemos cómo debe ser el operador de Kleisli, $\rho_{A,B} : T_RA \to (A \to T_RB) \to T_RB$. Considerando, entonces, $a \in T_RA$ y $f : A \to T_RB$, $\rho_{A,B}\ a\ f) : (B \to R) \to R$. Pero, si $h : B \to R$, resulta que, para cada $x \in A$, $(f(x))(h) \in R$. Entonces, entre A y R tenemos el morfismo f_* definido por $f_*(x) = (f(x))(h)$. Pero, entonces, $a(f_*) \in R$. En conclusión, definimos $\rho_{A,B}\ a\ f = \lambda h.a(\lambda x.(f(x))(h))$. Además $\mu_A(a) = \rho_{A,T_RA}(a\ 1_{T_RA})$. Tal como hemos definido los morfismos, es inmediato comprobar que $\mathbf{T}=(T_R, \eta, \mu)$ satisface las condiciones de triple. En cuanto a su definición práctica tenemos que $\mu_A(a) = \lambda h.a(\lambda k.(1_{T_RA}(k))(h)) = \lambda h.a(\lambda k.(k(h)))$, que, escrito de forma más simplifcada representa que, fijando $h : A \to R$, se considera $C : ((A \to R) \to R) \to R$, definida como $C(k) = k(h)$, para cada $k : (A \to R) \to R$. Entonces, $\mu_A(a) = \lambda h.a(C)$.

6.3.2 Estructura de T-álgebra

Antes de iniciar este estudio, intentemos representar más claramente quién es $T(A)$. Consideremos, para ello, el tipo funcional $A \to R$ como R^A; sabemos, entonces, que contamos con la función $eval_{A,R} : A \times R^A \to R$ propia de la exponencial. Así, intuitivamente hablando (veremos más adelante que esta intuición tiene sentido) podemos anticipar que el elemento $a \in A$, que corresponde a la función de orden superior $k : (A \to R) \to R$ es aquél que verifica que $k(h) = h(a)$, $h : A \to R$. El problema surgirá con aquellas funciones k que no satisfagan esta condición (es decir, aquéllas para las que no existe tal valor a verificando una ecuación semejante a la anterior). La no existencia de tal valor de tipo A es un hecho ya que, si A tiene cardinal $\sharp A$, el cardinal de R^A es $(\sharp R)^{(\sharp A)}$, mayor que el anterior por el teorema de Cantor (sobreentendiendo que $\sharp R \geq 2$).

Supongamos, pues, que (A, ξ) es una $\mathbf{T}$-álgebra. Entonces, ξ es un morfismo de $T(A)$ en A, o sea, una función de $(A \to R) \to R$ en A: asigna, por tanto, un valor de tipo A a cada continuación de tipo A. Es psoible interpretar $(A \to R)$ como las filas de una array con índice en A y valores en R. En tal caso, $(A \to R) \to R$ representa la selección, en cada fila, de un elemento. Deben verificarse las dos ecuaciones siguientes:

$$\eta_A;\xi = 1_A;$$

$$T(\xi);\xi = \mu_A;\xi.$$

¿Qué representan ambas igualdades? Para responder a esta cuestión, introducimos algunas nociones.

Dada $k : R^A \to R$, puede existir un subconjunto S de A de forma que h tenga una factorización a través de R^S. Consideremos, en estas condiciones, $res : R^A \to R^S$ la aplicación que envía al morfismo $h : A \to R$ en su S-restricción.

Definición 6.4. *Dada una función $k : R^A \to R$ denominamos* soporte *de k a un subconjunto de A de forma que la anterior factorización existe y, además, k es independiente de los elementos de A que no están en S. Denominamos sop(k)=$\{S \subset A/$ S soporte de $k\}$.*

Definición 6.5. *Un* cuasifiltro *sobre un conjunto A es una colección no vacía $\mathcal{F}$ de subconjuntos de A verificando las siguientes condiciones:*

1. *cada superconjunto de un elemento de $\mathcal{F}$ está en $\mathcal{F}$;*
2. *la intersección de dos elementos de $\mathcal{F}$ está en $\mathcal{F}$.*

Teorema 6.3. *Siendo $k : R^A \to R$, $Sop(k)$ es un cuasifiltro.*

Demostración. La primera condición de cuasifiltro es obvia así que prestaremos atención sólo a la segunda. Sean, pues, U, $V \subset Sop(k)$. Debemos analizar dos posibilidades:

$$U \cap V = \emptyset.$$

Como $U \subset sop(k)$, k factoriza a través de A^U y, además, k no depende de loss elementos de X que no están en U. En particular, como $V \subset X \setminus U$, k no depende de V. Análogamente, razonando sobre V vemos que k no depende de U. Ya que k factoriza a través de A^U o de A^V, y k no depende ni de U ni de V, colegimos que k es constante.

$$U \cap V \neq \emptyset.$$

Razonando igual que antes, vemos que k depende sólo de las partes comunes a U y a V. Notemos que, si k factoriza a través de un subconjunto $\mathcal{F}$ de X, podemos establecer entre X y $\mathcal{F}$ un morfismo $p_X : X \to \mathcal{F}$ dado por

$$p_X(x) = \begin{cases} x & \text{if(x)} \in \text{U} \\ x_0 & \text{if(x)} \notin \text{U, } x_0 \in \text{U, fijo} \end{cases}$$

En tal caso, el morfismo $f : X \to A$ se puede factorizar a través de p_X y f_U (la restricción de f a U), de forma que $k(f) = k(p_X; f_U)$. □

No resulta difícil trasladar estas ideas a la situación planteada con las álgebras de las continuaciones. Sea $N_k = min\{n \in N : \exists S \in Sop(k),\ card(S) = n\}$, es decir, el cardinal del menor subconjunto de A que sirve de soporte para k. Sea I_k un subconjunto de A soporte de k de cardinal N_k. Si hubiera varios subconjuntos I_k con este cardinal mínimo, al ser $Sop(k)$ un cuasifiltro, su intersección o es $\emptyset$ es soporte de k. Pero, en este último caso, como ambos I_k son mínimos en su cardinalidad, deben ser el mismo. En

resumen, $I_k = \emptyset$ o I_k es un subconjunto no vacío de A. Si $I_k = \emptyset$, k es constante, es decir, $k(h) = r_k,\ \forall h : A \to R$.

Definimos, entonces, $\xi(k) = a_k$, un elemento arbitrario de A. Sea, en el otro supuesto, ese I_k minimal en el cardinal de los soportes de k. Definimos $a_k = supa \in I_k$. Consideramos, entonces, $\xi(k) = a_k$. La primera restricción de **T**-álgebra se satisface, pues, si $\eta_A; \xi = 1_A$, $\xi(\lambda h.h(a)) = a$ y en este caso, a es el soporte minimal de k. La condición propia de álgebra no precisa ser vista pues es bastante obvia. En efecto, $(\eta_A; \xi)x = x$, cualquiera que sea $x \in A$. Es decir, $\xi(\lambda h.h(x)) = x$. Esta es la confirmación de la intuición expresada inicialmente. Si C es una computación en $T(A)$ (es decir, $C : (A \to R) \to R$), de forma que existe $a \in A$ tal que $\gamma(h) = h(a)$, cualquiera que sea el morfismo $h : A \to R$, entonces, la estructura de **T**-álgebra, ξ, lleva γ en a. O sea, cuando γ es tal que en cada fila toma el elemento indicado por a, el valor de tipo A correspondiente es, precisamente, a. Analicemos la segunda que, como es natural nada más verla, es bastante más prolija en su especificación. Deben verificarse las dos siguientes igualdades:

$$(\mu_A; \xi)(\overline{\overline{x}}) = \xi(\lambda h.\overline{\overline{x}}(\lambda k.k(h))) \text{ [1]};$$

$$(T(\xi); \xi)(\overline{\overline{x}}) = \xi(\lambda h.\overline{\overline{x}}(\xi; h)) = \xi(\lambda h.\overline{\overline{x}}(\lambda k.h(\xi(k)))) \text{ [2]}.$$

Notemos que $\xi; h : T(A) \to A \to R$ y $\overline{\overline{x}} : (T(A) \to R) \to R$.

En resumen, la igualdad establece que

$$\xi(\lambda h.\overline{\overline{x}}(\lambda k.k(h))) = \xi(\lambda h.\overline{\overline{x}}(\lambda k.h(\xi(k)))).$$

[1]: Fijando, por ejemplo, la función, h_1, se da a la primera columna un valor; variando h, cada columna corresponde a un valor de tipo R, tal como se puede ver en el cuadro siguiente:

$$\begin{array}{cccc} k_1(h_1) & k_1(h_2) & k_1(h_3) & \cdots \\ k_2(h_1) & k_2(h_2) & k_2(h_3) & \cdots \\ \cdots & & & \\ k_n(h_1) & k_n(h_2) & k_n(h_3) & \cdots \\ r_1 & r_2 & r_3 & \cdots \end{array}$$

[2]: En este caso, conocemos el soporte de la función que hemos definido. Viene dado como la unión de $\xi(k)$, para cada $k : R^A \to R$. Es decir, sea cual sea $h : A \to R$, queda restringida a $S = \cup\xi(k) : k : R^A \to R$. Cada fila, entonces, nos proporciona un valor, fruto de la aplicación de cada función h a los elementos de S, como vemos a continuación:

$$\begin{array}{cccc} h_1(a_1) & h_1(a_2) & h_1(a_3) & \cdots r^1 \\ h_2(a_1) & h_2(a_2) & h_2(a_3) & \cdots r^2 \\ \cdots & & & \\ h_n(a_1) & h_n(a_2) & h_n(a_3) & \cdots r^n \\ \cdots & & & \end{array}$$

Vemos, pues, que, en la signatura que estamos utilizando, hemos llegado a la siguiente conclusión:

[1] Si calculamos primeramente el soporte minimal de cada $k : R^A \to R$ y calculamos el supremo correspondiente a la unión de todos debe coincidir con [2] el valor supremo de la unión de todos los valores supremos correspondientes a los soportes minimales de todos los funcionales $k : R^A \to R$. Expresándolo en forma más legible, hemos llegado a la siguiente igualdad:

$$sup(\cup_{k:R^A \to R}(sup_{x \in I_k}(x)) = sup(\cup_{k:R^A \to R} I_k.$$

Es decir, siendo $\xi : (R^A \to R) \to A$ un morfismo de $\mathbf{T}$-álgebras, hemos visto que, para un morfismo $k : R^A \to R$ podemos considerar un elemento $a_k \in A$ de forma que $\xi(k) = a_k$. La construcción que hemos hecho nos remite a la noción de semiretículo completo [conjunto parcialmente ordenado en el cual cada subconjunto finito tiene un supremo].

6.3.3 Categoría de Kleisli de las continuaciones

Para realizar la modelización de la categoría de Kleisli, precisamos trabajar en una categoría cartesiana cerrada. En ella, sabemos que, para cada par de objetos A, R, existe un objeto, R^A, denominado exponencial, con una flecha $eval : A \times R^A \to R$, verificando que, si tenemos otra flechas $f : A \times C \to R$, existe una única flecha $f^\sharp : C \to R^A$ que verifica $f = eval < 1, f^\sharp >$. Para ello, vamos a formalizar este argumento poniendo las condiciones precisas para su aplicación. Sea, entonces, $f : A \to T(C)$ un morfismo en la categoría de Kleisli; para cada $a \in A$, $f(a) : (B \to R) \to R$. Vamos a ver, como primer paso, que $(R^{(R^B)})^A$es isomorfo a $R^{R^B \times A}$. Definimos, para ello, dos funcionales [que veremos isomorfos] ϕ y ψ de la siguiente manera:

$$\phi : (R^{(R^B)})^A \to R^{R^B \times A},\ \phi(g)(g, a) = (ha)(g);$$

$$\psi : R^{R^B \times A} \to (R^{(R^B)})^A,\ \psi(h)(a) = \lambda g.h(g, a).$$

$$(\phi; \psi)(h) = \psi(\phi(h)) = \lambda a.\lambda g.(\phi(h)(g, a)) =$$
$$\lambda a.\lambda g.(ha)(g) \equiv h$$

por la η-reducción.

$$(\psi; \phi)(h) = \phi(\psi(h)) = \lambda(g, a).(\psi(h)(a))g =$$

$$\lambda(g, a).h(g, a) \equiv h$$

por la misma razón.

Luego, $\phi; \psi$ y $\psi; \phi$ son identidades, así que, en efecto, se verifican los isomorfismos mencionados. Ahora ya tenemos todo preparado para el proceso que hemos anticipado con las exponenciales. R^A es la exponencial correspondiente a R y A. Contamos con el morfismo $eval : A \times R^A \to R$. Pero, por otra parte, hemos visto que tenemos otra flecha $\phi(h) : A \times R^B \to R$; por la propiedad de la exponencial, existe una única flecha $(\phi(h))^\sharp : R^B \to R^A$ que verifica $eval < 1_A, (\phi(h))^\sharp >= \phi(h)$. Hemos, pues, visto que hay un isomorfismo entre $A \to ((B \to R) \to R)$ y $R^B \to R^A$. La correspondencia entrambas

las dos construcciones es obvia: sea $h : A \to ((B \to R) \to R)$; $(\phi(h))^\sharp(g) = \lambda a.h\ a\ g$; sea, ahora, $p : R^B \to R^A$; $(p^*)\ a\ g = (p\ g)\ a$.

En resumen, cada flecha $f : A \to ((B \to R) \to R)$ induce de forma única, una flecha $(\phi(h))^\sharp : R^B \to R^A$. Así, pues, la categoría de Kleisli de las continuaciones viene dada por la categoría cuyos objetos son las exponenciales de base R, R^C, y las flechas las originadas por la propiedad de exponencial de aquéllos.

6.4 *El triple de los lectores de estado*

6.4.1 Introducción

Sea **C** la categoría que hemos definido sobre un lenguaje funcional y, sea, en esta categoría S un tipo fijo, que denominaremos, indistintamente, de estados o de registros. Consideramos el siguiente morfismo sobre los objetos de **C**: $T(A) = S \to A$, que llamaremos el tipo de los lectores de estado de tipo A. Así, pues, un lector de estados de tipo A es un morfismo, f, que, a cada estado $s \in S$ asigna un valor $f(s) \in A$.

Podemos ampliar la definición de T para que sea un funtor, de forma que, si $a : A \to B$ es un morfismo, $Ta : T(A) \to T(C)$ viene dado, en notación del λ-cálculo, por $(Ta)(f) = \lambda s.a(f\ s)$. Es decir, primero obtenemos el valor de tipo A que determina f sobre s y luego lo trasladamos por a. Con esta definición, $T : \mathbf{C} \to \mathbf{C}$, es un funtor. Construyamos la mónada correspondiente.

La unidad del triple tiene un expresión muy simple: $\eta_A(x) = \lambda s.x$, $\eta_A : A \to T(A)$. Es decir, $\eta_A(x)$ es la función que, dado un valor de tipo A fijo, a cada estado le asigna ese valor. Definamos el operador de Kleisli, que denotamos $\rho_{A,B} : T(A) \to (A \to T(C)) \to T(C)$. Si $f \in T(A)$ y $a : A \in T(C)$, $\rho_{A,B}(f\ a) = \lambda s.[(a(f\ s))\ s]$. Es decir, primeramente calculamos el valor de tipo A que viene dado por f s; como a asigna a (f s) un valor de tipo T(C), aplicamos a(f s) al estado inicial, s. Con la unidad y este operador, tenemos un triple de Kleisli, donde el operador $-^*$ viene dado por la siguiente expresión: si $a : A \to T(C)$, $a^* : T(A) \to T(C)$, $a^*(f) = \rho_{A,B}(f\ a)$. Comprobamos las ecuaciones que deben ser satisfechas para poder hablar de triple de Kleisli.

$$\eta_A{}^* = 1_{T(A)},\ ya\ que$$

$$\eta_A{}^*(f) = \rho_{A,A}(f\ \eta_A) = \lambda s.(\eta_A(f\ s))\ s) = \lambda s.(f\ s) = 1_{T(A)}(f)$$

por la definición de η.

$$\eta_A; a^* = a,\ a : A \to T(C).$$

$$a^*(\eta_A(x)) = \rho_{A,B}(\eta_A(x)\ a) = \lambda s.(a(\eta_A(x)\ s)\ s) = \lambda s.(a\ x)\ s) = a(x).$$

$$(a; b^*)^* = a^*; b^*,\ a : A \to T(C),\ b : B \to T(C).$$

$$(a; b^*)^* = a^*; b^*,\ a : A \to T(C),\ b : B \to T(C).$$

$$(a; b^*)^*(f) = \rho_{A,C}(f\ (a; b^*)) = \lambda s.(b^*(a(f\ s))\ s) = \lambda s.(b(a(f\ s)))\ s).$$

$$(a^*; b^*)(f) = b^*(a^*(f)) = b^*(\rho_{A,B}(f\ a)) =$$

$$\rho_{B,C}(\rho_{A,B}(f\ a))\ b) = \lambda s.(b(a(f\ s))\ s).$$

Con esta definición, la multiplicación viene dada, siendo $h \in T^2(A) = S \to (S \to A)$, por $\mu_A(h) = \rho_{T(A),A}(h\ 1_{T(A)})$; es decir, $\mu_A(h) = \lambda s.((f\ s)\ s)$. Por consiguiente, $\mathbf{T}=(T, \eta, \mu)$ es una mónada.

6.4.2 Dominios computacionales

Singularizamos el caso en que S, el conjunto de los estados, es numerable. Introducimos en esta sección los elementos necesarios para poder intepretar las operaciones indicadas en el álgebra de Eilenberg-Moore, aunque, algunas nociones no serán definidos por su obviedad: relación de orden, orden parcial, conjunto parcialmente ordenado, cadena, supremo, etc.

Definición 6.6. *Un conjunto parcialmente ordenado D es w-completo si cualquier cadena $d \in D^w$ tiene supremo en D. Ello quiere decir que, siendo $i,\ j \in w,\ a_i,\ a_j \in D,\ a_i \leq_D a_j$, cuando $i \leq_w j$, entonces, hay un elemento $e \in D$ tal que $a_i \leq e,\ \forall i \in w$ y, si $f \in D$ es tal que $\forall i \in w,\ a_i \leq f$, entonces $e \leq f$. Al supremo de una cadena $d \in D^w$ le llamaremos punto límite y será denotado por*

$$\coprod_{i \in w} a_i,$$

donde $d(i) = a_i,\ i \in w$. Naturalmente, si la cadena d es tal que, para cada

$$i \in w,\ a_i = a, \coprod_{i \in w} a_i = a$$

. En este caso, la cadena se denomina sin interés.

Definición 6.7. *Un dominio computacional es un conjunto parcialmente ordenado y w-completo.*

Definición 6.8. *Si A es un conjunto y D es un dominio, entonces $A \to D$ es el dominio cuyo conjunto base es tal que $\mid A \to D \mid = A \to \mid D \mid$ y la relación de orden que definimos viene dada por:*

$$f \leq_{A \to D} f' \text{ si, para cada } a \in A, f(a) \leq_D f'(a).$$

Teorema 6.4. *Siendo A un conjunto y D un dominio, si $f \in (A \to D)^w$, existe*

$$\coprod_{i \in w} f_i$$

y viene dado, siendo $a \in A$, por

$$(\coprod_{i \in w} f_i)(a) = \coprod_{i \in w} (f_i(a)).$$

Demostración. Hemos de ver dos cosas: En primer lugar que

$$\forall i \in w,\ f_i \leq_{A \to D} \coprod_{i \in w} f_i.$$

Pero esto es inmediato a partir de la definición de

$$\coprod_{i \in w} f_i,$$

ya que

$$(\coprod_{i \in w} f_i)(a)$$

es el punto límite de la cadena formada por $f_i(a),\ i\ inw$. En segundo lugar, consideremos h tal que $\forall i \in w,\ f_i \leq_{A \to D} h$. Para cada $a \in A,\ f_i(a) \leq_D h(a)$. Pero, por ser la menor cota superior de las f_i, se verifica que

$$\coprod_{i \in w} (f_i(a)) \leq_D h(a)$$

. Entonces, por la definición,

$$\coprod_{i \in w} f_i \leq_{A \to D} h.$$

□

Teorema 6.5. *Si el dominio D tiene un elemento minimal,*$\perp$*, el dominio* $A \in D$ *también lo tiene, siendo éste la función definida por* $m(a) = \perp, \forall a \in A$.

Demostración. Veamos que, para cualquier $f \in A \to D,\ m \leq_{A \to D} f$.. Por ser $\perp$ minimal en D, $\perp \leq_D f(a),\ \forall f \in A \to D,\ \forall a \in A$. Como $\perp = m(a),\ m(a) \leq_D f(a),\ \forall f \in A \to D, a \in A$. Es decir, $m \leq_{A \in D} f$. □

Enunciamos y demostramos, a continuación, dos teoremas que nos permiten relacionar todas estas ideas con nuestro problema inicial.

Teorema 6.6. *Siendo D un dominio y siendo* $a,\ b \in D^w$, *cadenas tales que, para cada* $i \in w$ *existe un* $j \in w$ *tal que* $a_i \leq_D b_j$, *entonces,*

$$\coprod_{i \in w} a_i \leq_D \coprod_{i \in w} b_i.$$

Demostración. Bastará con ver que para cada

$$i \in w,\ a_i \leq_D \coprod_{i \in w} b_i.$$

Sean $i \in w$ y $a_i \in D$. Existe, por la hipótesis, $k \in w$ tal que $a_i \leq_D b_k$. Como

$$b_k \leq_D \leq_D \coprod_{i \in w} b_i,$$

$$a_i \leq_D \coprod_{i \in w} b_i.$$

□

Definición 6.9. *Sea D un dominio; entonces, $d \in D^{w\times w}$ se denomina cadena doble si se verifica lo siguiente:*

1. $d_{i0} \leq d_{i1} \leq d_{i2} \ldots$, *para cada* $i \in w$;
2. $d_{0j} \leq d_{1j} \leq d_{2j} \ldots$, *para cada* $j \in w$.

Teorema 6.7. *Siendo d una cadena doble en un dominio D, se verifica que, siendo*

$$d_{i\infty} = \coprod_{j\in w} d_{ij},\ d_{\infty j} = \coprod_{i\in w} d_{ij},$$

$$\coprod_{i\in w} d_{i\infty} = coprod_{j\in w} d_{\infty j} = coprod_{k\in w} d_{kk}.$$

Demostración. Veremos, en primer lugar, que tanto $d_{i\infty}$ como $d_{\infty j}$ tienen sentido. Al ser d una cadena doble, $d_{ij} \leq d_{i(j+1)}$, para cada $i \in w$. Entonces, la cadena tiene un supremo, $d_{i\infty}$. Análogamente ocurre para $d_{\infty j}$. Además, $d_{i\infty} \leq d_{(i+1)\infty}$, ya que, por la segunda condición de la definición de cadena doble, para cualquier $j \in w$, $d_{ij} \leq d_{(i+1)j}$. Entonces, $d_{i\infty},\ i \in w$, es una cadena y tiene supremo en D. Lo mismo ocurre con $d_{\infty j}$. Además, para cada $k \in w, d_{kk} \leq d_{k(k+1)} \leq d_{(k+1)(k+1)}$, con lo que d_{kk} también es una cadena en D y, por lo tanto, tiene supremo en D. Veremos las igualdades descritas haciendo uso de la propiedad antisimétrica de la relación de orden. Así, comencemos viendo que

$$\coprod_{i\in w} d_{i\infty} \leq \coprod_{j\in w} d_{\infty j}.$$

Por la definición dada al inicio del teorema, para cualquier $i \in w$, $d_{ij} \leq d_{\infty j}$, $\forall j \in w$; entonces, por el teorema anterior,

$$\coprod_{i\in w} d_{ij} \leq \coprod_{j\in w} d_{\infty j}.$$

Ya que esto ocurre para cada $i \in w$, y al ser

$$\coprod_{i\in w}\coprod_{j\in w} a_{ij}$$

la menor cota superior, obtenemos la desigualdad deseada.

Resolviendo recíprocamente, obtenemos la otra desigualdad con lo que, aplicando la antisimetría, concluimos que

$$\coprod_{i\in w} d_{i\infty} = \coprod_{j\in w} d_{\infty j}.$$

Veamos la otra igualdad. Es obvio que

$$\coprod_{k\in w} d_{kk} \leq \coprod_{i\in w} d_{i\infty}$$

ya que

$$d_{kk} \leq d_{k\infty} \leq \coprod_{k\in w} d_{k\infty}.$$

La otra desigualdad se obtiene inmediatamente si observamos que, dado un d_{ij} cualquiera, existe $k = max(i,j)$ tal que $d_{ij} \leq d_{kk}$; por lo tanto,

$$\coprod_{j \in w} d_{ij} \leq \coprod_{k \in w} d_{kk}.$$

□

Definición 6.10. *Sean A y B dominios; una función $h : A \to B$ es monótona (o preserva el orden) si, cuando $a \leq_A a'$ resulta que $h(a) \_B h(a')$.*

Definición 6.11. *Siendo A, B dominios, la función $h : A \to B$ es w-continua (o preserva límites) si, para cada cadena $d \in A^w$, se verifica que*

$$h(\coprod_{i \in w}^{A} d_i) = \coprod_{i \in w}^{B} h(d_i).$$

6.4.3 Categoría de Eilenberg-Moore

Supongamos, pues, que (A, ξ) es una $\mathbf{T}$-álgebra. $\xi : T(A) \to A$ es, entonces, un morfismo que asigna, a cada función $f : S \to A$ un valor de tipo A. Así, denotemos por $\xi(f) = a_f$. Como $f(s) = a_{f,s}$, podemos representar f como $(a_{f,s})$, con $s \in S$. En tal caso, $\xi(f) = \coprod a_{(f,s)}$. Las dos condiciones que deben verificarse vienen dadas por:

$$\eta_A ; \xi = 1_A.$$

Es decir, $\xi(\lambda s.x) = x$, $\forall x \in A$. Entonces,

$$\coprod_{s} x = x, \ \forall x \in A.$$

$$T(\xi); \xi = \mu_A ; \xi.$$

Sea, $f \in T^2 A = S \to (S \to A)$. $\mu_A(f) = lambdas.((f\ s)\ s) \equiv a_{s,s}$. Por consiguiente,

$$(\mu_A ; \xi)(f) = \coprod_{s} a_{s,s}.$$

Por otra parte,

$$(T(\xi))\ f = \lambda s.(\xi(f\ s)) = \lambda s. \coprod_{s'} a_{s,s'},$$

ya que $(f\ s) : S \to A$, $(f\ s)\ s' = a_{s,s'}$. Por lo tanto,

$$T(\xi); \xi)(f) = \coprod_{s} (\coprod_{s'} a_{s,s'}).$$

Esto significa, definitivamente, que

$$\coprod_{s} (\coprod_{s'} a_{s,s'}) = \coprod_{s} a_{s,s}.$$

Puesto en función del morfismo $f : S \to (S \to A)$, quiere significar que: $\xi(\lambda s.((f\ s)\ s)) = \xi(\lambda s.\xi((fs)))$. ξ determina, entonces, una operación sobre A, $\coprod$, de aridad $\sharp S$. Veremos, más abajo, una interpretación de estas ecuaciones y qué estructura se amolda perfectamente a ellas. Pero, primeramente, analicemos el punto de vista funcional (relativo a las funciones). Estas ecuaciones entonces, determinan el comportamiento de ξ de la siguiente forma:

1. primeramente, aplicada a una función constante ($f\ s = a, s \in S$), retorna el valor a;
2. en segundo lugar, si $f\ s_0 = \xi(g)$, con $: S \to A$, $\xi(f) = \xi(f')$, siendo

$$f(s) = \begin{cases} f(s) & \text{if s} \neq\ s_0 \\ g(s_0) & \text{otro caso} \end{cases}$$

   Es decir, cuando una computación $f : S \to A$ depende, en un estado s_0, de otra valoración previa ($f(s_0) = \xi(g)$, con $g : S \to A$), a la hora de evaluar $\xi(f)$, no precisamos determinar el valor de $\xi(g)$, sino que basta con tomar como elemento representativo $g(s_0)$.

Ya podemos, con estas nociones, atacar el problema inicial: ¿qué estructura observamos en las ecuaciones que hemos obtenido de la condición de **T**-álgebra? Si recordamos aquellas ecuaciones, observamos la absoluta igualdad con las anteriores. Además, ?'qué ocurre si $h : (A, \xi_A) \to (B, \xi_B)$ es un morfismo de **T**-álgebras? Como ya sabemos, esto implica que

$$h(\coprod_{s \in S}^{A} a_s) = \coprod_{s \in S}^{B} h(a_s).$$

En resumen: una **T**-álgebra (A, ξ) induce una estructura de dominio computacional sobre A. Además, la categoría de Eilenberg-Moore sobre el triple de los lectores de estado, es la categoría de los dominios computacionales. y los morfismo que preservan límites. Añadimos, a continuación, una visión ligeramente simplificada de esta estructura. Para ello, precisamos los siguientes conceptos y definiciones: sea $\Sigma = \sigma$ un tipo formado por un símbolo de operación de aridad k.

Definición 6.12. *Un tipo finito es un conjunto Σ de símbolos de operación, σ, junto a una aplicación que asigna a cada uno de estos símbolos, un número natural, $\mid \sigma \mid = n$, que se denomina la aridad de la operación. Denotaremos por Σ_n el conjunto de todos los símbolos de operación n-aria.*

Definición 6.13. *Una Σ-álgebra consta de un conjunto Q con operaciones indicadas por las aridades de los símbolos contenidos en Σ, $\sigma : Q^n \to Q$, siendo $\sigma \in \Sigma$, $\mid \sigma \mid = n$.*

Entonces, prescindiendo de las ecuaciones que se obtienen de la estructura de **T**-álgebra, vemos que la categoría de Eilenberg-Moore de las **T**-álgebras de los lectores de estado (con registros un conjunto de cardinal k), es la categoría de las Σ-álgebras, con $\Sigma = \sigma$, σ símbolo de operación k-aria.

6.4.4 La mónada de los lectores de estado y Coq

Igual que en el caso de las excepciones, no resulta prolija interpretar estos hechos en Coq. Lo presentamos sin comentarios:

```
Inductive LS [S,A:Set]:Set:=R:(S->A)->(LS S A).

Definition prv:=[S,A,B:Set][h:S->A][f:A->B][n:S]
(f (h n)).
Definition mapLS:=[S,A,B:Set][f:A->B][x:(LS S A)]
(Cases x of (R h)=>(R S B (prv S A B h f)) end).

Definition uno:=[S,A:Set][x:A][n:S]x.
Definition unit:=[S,A:Set][x:A](R S A (uno S A x)).

Definition prv2:=[S,A:Set][h:S->(S->A)][n:S](h n n).
Definition extract:=[S,A:Set][x:(LS S A)]
(Cases x of (R h)=>h end).
Definition sucesion:=[S,A:Set][x:(LS S A)][n:S]
(Cases x of (R h)=>(h n) end).

Definition suc_diagonal:=[S,A:Set][x:(LS S (LS S A))]
[n:S]
(Cases x of (R h)=>((extract S A (h n)) n) end).
Definition suc_diagonalLS:=[S,A:Set][x:(LS S (LS S A))]
(R S A (suc_diagonal S A x)).
Definition suc_doble:=[S,A:Set][x:(LS S (LS S A))]
[n,n':S]
(Cases x of (R h)=>((extract S A (h n)) n') end).
Definition asoc:=[S,A:Set][x:(LS S (LS S A))]
(R S A (suc_diagonal S A x)).
Definition Gen:=[S,A:Set][H:(LS S A)->A]
[x:(LS S (LS S A))][n:S]
(Cases x of (R h) =>(H (h n)) end).
Definition GenLS:=[S,A:Set][H:(LS S A)->A]
[x:(LS S (LS S A))]
(R S A (Gen S A H x)).


Lemma primero:(S,A:Set)(x:(LS S (LS S A)))
((asoc S A x)=(suc_diagonalLS S A x)).
primero < Intros; Elim x;EAuto.
primero < Qed.

Lemma segundo:(S,A:Set)(H:(LS S A)->A)
(x:(LS S (LS S A)))
```

```
((mapLS S (LS S A) A H x)=(GenLS S A H x)).
segundo < Intros; Elim x;EAuto.
segundo < Qed.

Theorem tres:(S,A:Set)(H:(LS S A)->A)
((x:A)(H (unit S A x))=x)->
((x:(LS S (LS S A)))(H (asoc S A x))=
(H (mapLS S (LS S A) A H x)))->
( (x:(LS S (LS S A)))
(H (suc_diagonalLS S A x))=(H (GenLS S A H x))).
tres < Intros; Replace (suc_diagonalLS S A x)
tres < with (asoc S A x).
tres < Replace (GenLS S A H x) with
tres < (mapLS S (LS S A) A H x).
tres < Apply H1.
tres < Apply segundo.
tres < Apply primero.
tres < Qed.
```

6.4.5 Categoría de Kleisli

Sea $\alpha : A \to T(C)$ un morfismo de la categoría de Kleisli. Eso significa que a cada valor de A se le asocia un morfismo $f : S \to B$. Recordamos que, en una categoría cartesiana cerrada C, son isomorfos $C(A, B^S)$ y $C(A \times S, B)$, viniendo el isomorfismo dado por: si $\alpha \in C(A, B^S)$, $\phi(\alpha)(a, s) = (\alpha(a))(s)$; si $h \in C(A \times S, B)$, $(\psi(h)(a))(s) = h(a, s)$. Vemos que, con esta consideración, la categoría de Klesili sobre el triple de los lectores de estado coincide con la categoría de Kleisli sobre el cotriple $C = (C_S, \epsilon_S, \delta_S)$ definido por: $C(A) = A \times S$; $\epsilon_S : T \Rightarrow 1$ dado por $\epsilon_S(A) = \pi_{A,S}$. Por último, $\delta_S : T \Rightarrow T^2$ viene definido como $\delta_S(A) =< 1_{A\times S}, \pi'_{A,S} >$.

Con este isomorfismo comprobamos la afirmación hecha en [Lambeck-Scott] de que, siendo C una categoría cartesiana cerrada, la categoría de Klesili sobre el triple de los lectores de estado en tal categoría C, es isomorfa a la categoría de polinomios C[x], con la suposición $x : 1 \to S$. Como ya sabemos tal isomorfismo viene dado por $\phi : C_T \to C[x]$, siendo $\phi(\alpha : A \times S \to B) =< 1_A, 0_A; x >; \alpha$, donde $0_A : A \to 1$ es la única flecha de A al objeto terminal 1; $x : 1 \to S$ es la flecha indeterminada sobre la que construimos la categoría de polinomios. Su funtor inverso es $\psi : C[x] \to C_T$, dado por $\psi(\phi(x) : A \to B)$ la única flecha que existe en la categoría C, denotada por $\kappa_{x\in S}(\phi(x))$, que verifica $\kappa_{x\in S}(\phi(x))) < 1_A, 0_A; x >= \phi(x)$.

Otra posibilidad para hallar un modelo para la categoría de Kleisli, es la siguiente. Podemos asociar a cada $h \in C(A, B^S)$ un morfismo $\kappa(h) \in C(A \times S, B \times S)$, donde $\kappa(h)(a, s) = ((\phi(h)](a, s), s))$. Entonces, la categoría de Kleisli de los lectores de estados tiene por objetos $A \times S$, con $A \in C$ y S el objeto de los estados. Analicemos los morfismos. Para ello, sean $\alpha : A \to (S \to B)$ y $B : B \to (S \to C)$. La composición en la categoría de

Kleisli, $\alpha; B$ viene dada por

$$(\alpha; B)(a) = \mu_C((T(B))(\alpha(a))) = \mu_C[\lambda t.B(\alpha(a))(t))$$
$$= \lambda s.(((\lambda t.B[(\alpha(a))t)s)s) = \lambda s.(B((\alpha(a))s)s).$$

Coincide con nuestra concepción de κ ya que ésta viene dada como sigue:

$$(\kappa(\alpha); \kappa(B))(a, s) = \kappa(B)(\kappa(\alpha(a, s)) =$$
$$\kappa(B)((\alpha(a))(s), s) = (B(\alpha(a)(s)s, s).$$

Conclusión: la categoría de Kleisli del triple de los lectores de estado es isomorfa a la categoría cuyos objetos son, para cualquier objeto A de la categoría, productos cartesianos $A \times S$, S objeto fijo de los estados y las flechas morfismos entre $A \times S$ y $B \times S$ que dejan el estado fijo. Entonces, construimos, por una parte, $\phi : C(A, B^S) \to C(A \times S, B \times S)$ de la siguiente forma: si $\alpha : A \to (S \to B)$, $\phi(\alpha) = \lambda(a, s).((\alpha(a)])s, s)$.

Por otra, $\psi : (C(A \times S, B \times S) \to C(A, B^S)$, definida como: si $\beta : A \times S \to B \times S$, $\psi(\beta) = \lambda a.\lambda s.\pi_1(\beta(a, s))$. Veamos que $(\phi; \psi) = 1_{C(A,B^S)}$ y que $\psi; phi = 1_{C(A\times S,B\times S)}$.

Sea $\alpha : A \to (S \to B))$;

$$(\phi; psi)(\alpha) = \psi(\phi(\alpha)) = \lambda a.\lambda s.\pi_1(\phi(\alpha)(a, s)))$$
$$= \lambda a.\lambda s.\pi_1(\alpha(a)\ s, s) = \lambda a.\lambda s.(\alpha(a))\ s$$
$$\equiv \alpha$$

por la η-reducción. Sea $\beta : A \times S \to B \times S$ de forma que deja el estado fijo.

$$(\psi; \phi)(\beta) = \phi(\psi(\beta)) = \lambda(a, s)(\psi(\beta)a)s, s)$$
$$= \lambda(a, s)(\pi_1(\beta(a, s), s) = \lambda(a, s).\beta(a, s) \equiv \beta$$

de nuevo por η-reducción.

6.5 *El triple de las salidas*

6.5.1 Introducción

Fijando un tipo S, consideramos la transformación $T(A) = (A, S \to S)$. Vamos a construir una estructura de triple sobre ella. En este caso singular las funciones son especialmente sencillas:

$$\eta_A : A \to T(A),\ \eta_A(x) = (x, id_S);$$

si $f : A \to B$, $T(f) : T(A) \to T(B)$ queda definido como:

$$T(f)(x, h) = (f(x), h),\ (x, h) : T(A);$$

por último, definimos $\mu_A : T^2(A) \to T(A)$ por:

$$\mu_A(X_2) = (X_0, f_1; f_2)$$

siendo $X_2 = (X_1, f_1),\ X_1 = (X_0, f_2)$. Resta por definir la función $\rho_{A,B} : T(A) \to (A \to T(B)) \to T(B)$. Ésta es obvia:

$$\rho_{A,B}\ (x, h)\ f = (y, h; g),\ \textit{siendo}\ f(x) = (y, g).$$

Es un simple ejercicio de cálculo verificar las ecuaciones de mónada y, ya que éste no es nuestro objetivo, no lo haremos.

6.5.2 Categoría de Eilenberg-Moore

Consideremos (A, ξ), nuevamente, una T-álgebra de Eilenberg-Moore. Deben verificarse las dos ecuaciones siguientes (concernientes a η y μ, respectivamente:

$$\xi(x, id_S) = x,\ x \in A;$$

$$\xi(x, f_1; f_2)) = \xi(\xi(x, f_2), f_1)$$

donde $x \in A, f_1,\ f_2 : S \to S$.

La expresión es tan clara y suficientemente expresiva que, sin más preámbulos, nos conduce a la siguiente:

Definición 6.14. *Sea $(M, 1, \otimes)$ un monoide. Un M-conjunto es un conjunto A con un morfismo $M \times A \to A$ que verifica $1a = a$ y $(m \otimes m')a = m(m'a)$ para cada $a \in A,\ m, m' \in M$.*

Corolario 6.2. *La categoría de Eilenberg-Moore de la mónada de las salidas es la categoría de los $S \to S$-conjuntos.*

6.5.3 Categoría de Kleisli

Consideramos $f : A \rightharpoonup B$ un morfismo en la categoría de Kleisli (es decir, $f : A \to T(B)$). Entonces, para cada elemento $a \in A$, obtenemos un elemento $b \in B$ y un morfismo $f_a : S \to S$. Esto indica que trabajamos en una categoría cuyos objetos son pares (A, S), con S el tipo prefijado de los estados, y siendo los morfismos pares $(f_1 : A \to B, f_2 : (A, S) \to S)$. La identidad es el par $id_{(A,S)} = (id_A, \lambda(a, s).s)$. La composición de dos morfismos, $(f_1, f_2),\ (g_1 : B \to C, g_2 : (B, S) \to S)$, viene dada por

$$(f_1, g_1);_K (g_1, g_2) = (f_1; g_1, \lambda(a, s).g_2(f_1(a), f_2(a, s))).$$

Comprobemos la condición de elemento neutro (por la izquierda y por la derecha) que tiene la identidad :

$$id_{(A,S)};_K (f_1, f_2) = (id_a; f_1, \lambda(a, s).f_2(id_A(a), (\lambda(x, t).t)(a, s))) =$$
$$(f_1, \lambda(a, s).f_2(a, s)) = (f_1, f_2);$$

$$(f_1, f_2);_K id_{(B,S)} = (f_1; id_B, \lambda(a, s).(\lambda(b, t).t)(f_1(a), f_2(a, s))) =$$
$$(f_1, \lambda(a, s).f_2(a, s)) = (f_1, f_2),$$

donde, en el último caso, hemos aplicado la η–reducción.

6.6 *El triple de los transformadores de estado*

6.6.1 Introducción

Cuando definimos el tipo de los lectores de estado, veíamos que, a cada estado (o registro, como también le llamábamos) le asignábamos un valor de forma que el estado no sufría alteración alguna. A la hora de analizar autómatas, sabemos que, generalmente, hay un cambio de estado a la hora de obtener un valor: es decir, dados un estado y un valor del alfabeto de entrada, obtenemos un valor de un alfabeto de salida y un nuevo estado. En tales condiciones, el triple de los lectores de estado no nos es excesivamente útil (salvo en el hecho de permitir una cierta çonmutatividad", al no verse afectados los estados unos por otros, lo cual habilita para prescindir en ciertos aspectos del orden de análisis de los registros). Para suplir esta carencia, construimos el tipo de los transformadores (o manipuladores de estado), donde, aparte de un valor de un tipo cualquiera, alteramos el registro. Podemos definirlo como sigue, donde S es el tipo de los estados: `type` $ST(A) = S \to (A, S)$. Así construido, un transformador de estados de tipo A es un morfismo que a cada estado le asigna un valor (de tipo A) y un nuevo estado. Podemos determinar un funtor para esta definición: $T(A) = ST(A)$ para los objetos y, si $f : A \to B$ es un morfismo, $T(f) : T(A) \to T(B)$ viene dado por:

$$(Tf)(\alpha) = \lambda s.[((f(x), s')|(x, s') \leftarrow (\alpha(s)].$$

Para entender esta definición es preciso recordar que si $\alpha \in T(A)$, $\alpha : S \to (A, S)$, es decir,$\alpha(s) = (x, s')$, siendo $x \in A$, $s' \in S$. La unidad se construye igualmente: $\eta_A : A \to T(A)$, es tal que $\eta_A(x) = \lambda s.(x, s)$. Construyamos el operador de Kleisli: $\rho_{A,B} : T(A) \to (A \to T(C)) \to T(C)$. Sean, entonces, $\alpha \in T(A)$ y $f : A \to T(C)$. Definimos:

$$\rho_{A,B}\ \alpha\ f = \lambda s.[(((f(x))s'), s'')|(x, s') \leftarrow \alpha(s)].$$

En tales condiciones, si $f : A \to T(C)$, $f^* : T(A) \to T(C)$ viene dado por $f^*(\alpha) = \rho_{A,B}(\alpha\ f)$. No requiere trabajo comprobar las ecuaciones precisas para que $(T, \eta, ^*)$ sea un triple de Kleisli. Para construir el triple, nos falta la multiplicación $\mu_A : TT(A) \to T(A)$. Recordando que $TT(A) = S \to (T(A), S)$, $\mu_A(\tau)) = \rho_{T(A),A}(\tau\ 1_{T(A)})$, con lo cual tenemos que

$$\mu_A(\tau) = \lambda s.[(x, s'')|(x, s'') \leftarrow \alpha(s'), (\alpha, s') \leftarrow \tau(s)].$$

Sin dificultad, (T, η, μ) es un triple. Además, por las ecuaciones descritas, observamos que, como cabía prever, su comportamiento es muy similar al de los lectores de estado. La única diferencia ostensible la tenemos en que, cuando hay una segunda acción de un valor del tipo T(A) no se hace, como ocurría antes, sobre el nuevo estado, sino sobre el estado resultante de la anterior acción. Es decir, si usamos el superíndice SR para denotar la acción en los lectores y ST para los transformadores, vemos esa distinción que, a la par, muestra la analogía:

$$\mu_A^{ST}(\tau) = \lambda s.[(x, s'')|(x, s'') \leftarrow \alpha(s'), (\alpha, s') \leftarrow \tau(s)];$$

$$\mu_A^{SR}(\tau) = \lambda s.[x|x \leftarrow \alpha(s),\ \alpha \leftarrow \tau(s)].$$

Cabe pensar, entonces, que no habrá grandes diferencias entre las categorías de Eilenberg-Moore y de Kleisli sobre ambos triples.

6.6.2 Categoría de Eilenberg-Moore

Consideremos (A, ξ) una T-álgebra, $\xi : T(A) \to A$. Lo que hace ξ, entonces, es transformar un morfismo que, a medida que recorre los estados de S va obteniendo valores de A y nuevos estados, en un valor de tipo A. Sea $\xi(f) = a_f,\ f \in T(A)$, es decir, $f : S \to (A, S)$. Las reglas ya conocidas de las T-álgebras confirman que:

$$\xi(\lambda s.(x, s)) = x,$$

siendo $x \in A$ (ésta es, exactamente, la misma condición vista en las T-álgebras sobre los lectores de estado);

$$\xi(\lambda s.[(x, s'')|(\alpha, s') \leftarrow \tau(s), (x, s'') \leftarrow \alpha(s')]) =$$

$$\xi(\lambda s.((\xi(\alpha), s')|(\alpha, s') \leftarrow \tau(s)]) :$$

o sea,

$$\xi(\lambda s.[(x, s'')|(\alpha, s') \leftarrow \tau(s), (x, s'') \leftarrow \alpha(s')]) =$$

$$\xi(\lambda s.((\xi(\lambda s'.(x, s'')|(x, s'') \leftarrow \alpha(s')]|(\alpha, s') \leftarrow \tau(s)]).$$

Indicando la operación ξ por medio de $\coprod$, lo anterior puede expresarse más sucintamente como sigue:

$$\coprod_s(\phi_1(s)_1(\phi_2(s)), \phi_1(s)_2(\phi_2(s))) = \coprod_s(\coprod_t(\phi_1(s)_1(t), \phi_1(s)_2), \phi_2(s))$$

donde $\phi : S \to (S \to (A, S)), S)$ queda descrito por $\phi_1 : S \to (S \to (A, S))$ y $\phi : S \to S$, quedando el primer morfismo descrito por $\phi_1(s)_1 : S \to (A, S)$, $\phi_1(s)_2 : S \to S$.

Toda esta prolija expresión conduce al siguiente resultado: si $f \in T(A)$ es tal que, para algún valor $s_0 \in S$, fs viene dada como el valor ligado a $\xi(g)$, con $g \in T(A)$, no es preciso hacer la valoración de esa computación para obtener la de f, sino que basta con obtener el par (x, s_2) resultado de calcular $g(s_1)$, siendo s_1 el nuevo estado propiciado por f en s_0. [Recordemos que $f : S \to (A, S)$ y que $f(s_0)) = (\xi(g), s_1)$, donde $\xi(g) \in A$, $g : S \in (A, S)$].

Para lo que sigue, precisamos algunos conceptos que nos ayuden a aproximarnos a donde deseamos.

Definición 6.15. *Sea Σ un conjunto no vacío, que denominaremos alfabeto de entrada. Un Σ-autómata secuencial es una quíntupla*
A=$(S, \delta, A, \gamma, q_0)$ donde:
S es un conjunto, denominado conjunto de los estados;
$\delta : Sx\Sigma \to S$ es una aplicación que reporta, dado un estado y una entrada, el estado siguiente;
A es un conjunto que llamaremos alfabeto de salida;
$\gamma : S \to A$ es la denominada aplicación de salida;
q_0 es un elemento de S denominado estado inicial.

Definición 6.16. *A partir del alfabeto de entrada, construimos el conjunto de las palabras Σ^* de la siguiente forma: $\emptyset$ es una palabra, denominada palabra vacía; si $\sigma \in \Sigma$, σ es una palabra; si $\sigma_1, \sigma_2 \in \Sigma^*$, $\sigma_1\sigma_2$ es una palabra.*

Definición 6.17. *La función de recorrido del autómata A es* $\rho : \Sigma^* \to S$, *definida como:* $\rho(\emptyset) = q_0$, *el estado inicial; si* $\rho(sigma_1 \ldots \sigma_n) = q_n$, *entonces,* $\rho(sigma_1 \ldots \sigma_n\sigma_{n+1}) = \delta(q_n, \sigma_{n+1})$, *siendo* $\sigma_i \in \Sigma, 1 \leq i \leq n, q_n \in S$.

Definición 6.18. *El comportamiento del autómata A es el morfismo* $\beta : \Sigma^* \to A$, *donde* $\beta = \rho;\gamma$.

El paralelismo con lo que nosotros tenemos es absoluto, considerando, en nuestro caso, como alfabeto de entrada el tipo unit, formado por el único valor `()` [de hecho, para eso tenemos este tipo que nos permite crear constantes en funciones sin argumentos]. Para ver esto, realizamos dos comprobaciones: 1. vemos que dado un Σ-autómata secuencial con alfabeto de salida A, podemos definir un valor de tipo ST A (es decir, un morfismo $f : S \to (A, S)$); 2. dado un valo de tipo ST A, f, cosntruimos un Σ-autómata secuencial con alfabeto de salida A. 1. Sea, entonces, A=$(S, \delta, A, \gamma, q_0)$ un Σ-autómata secuencial. Sabemos, entonces, que $\delta : S \times \Sigma \to S$ es el morfismo que retorna el nuevo estado; para encajar S en $S \times Sigma$ podemos usar $< 1_S, () >$, definido por: $< 1_S, () > (s) = (s, ())$, $s \in S$. Como tenemos $\gamma : S \to A$, construimos el morfismo $f =< \gamma, < 1_S, () >; \delta >: S \to A \times S$, valor de tipo ST A. 2. Igualmente, sea $f \in STA$, $f : S \to (A, S)$. Tomamos, igual que antes, $\Sigma = ()$. Obviamente, con $\tau : S \times \Sigma \to S$, dada por $\tau(s, ()) = s$ determinamos $\delta_f = \tau; ?; f; \pi_2 : S \times \Sigma \to S$. Más fácilmente, $\gamma_f = f;\pi_1 : S\ A$. El estado inicial podemos considerarlo como el resultado de aplicar $\lambda : I \to S$, siendo $I = 0$. Los morfismos de recorrido y comportamiento son sencillos en este contexto, ya que $\Sigma^* = \emptyset \cup \sigma^n | n \in N$, siendo σ=`()`. En primer lugar, la construcción de $\rho_f : \Sigma^* \to S$ es obvia; $\rho_f(\emptyset) = \lambda(0)$; siendo $\sigma_i = (), 1 \leq i \leq n$ y considerando que $\rho_f(\sigma_1 \ldots \sigma_n) = q_n$, entonces $\rho_f(\sigma_1 \ldots \sigma_n\sigma_{n+1}) = \delta_f(q_n, \sigma_{n+1})$.

Igualmente lo es, a continuación, la obtención de $\beta_f : \Sigma^* \to A$, $\beta_f = \rho_f;\gamma_f)$ definidos ρ_f, γ_f como anteriormente hemos hecho.

Así, pues, podemos afirmar que, siendo A un tipo, y $f \in STA$, el tipo de los transformadores de estados de tipo A, construimos un Σ-autómata secuencial, donde Σ=unit=`()` y viceversa, dado un Σ-autómata secuencial, con Σ-unitario, con estados en S y alfabeto de salida en A, podemos construir un valor del tipo de transformaciones de estado de tipo A y estados en S. [No representa, en el proceso abstracto que hacemos, problema alguno asignar a un valor de tipo S como registro inicial, q_0, tal como lo hemos construido a partir de I].

Aún más, dado el tipo ST A de los morfismos de S en (A,S), podemos construir un Σ-autómata que verifique todos los comportamientos anteriormente descritos. Construyamos $A_\sharp = \beta_f : \Sigma^* \to A$, donde $f : S \to (A, S)$. Vamos a considerar $A_\sharp$ como el tipo de los estados. La función que determina el estado siguiente es $\delta : A_\sharp \times \Sigma \to A_\sharp$, definida por $\delta(\beta_f, \sigma) = \beta_f(\sigma\_)$, que a cada $v \in \Sigma^*$ le asigna $\beta_f(\sigma v)$.

El morfismo de salida será $\gamma : A_\sharp \to A$, definido, en este caso, como: $\gamma(\beta_f) = \xi(f)$. Con esta definición, dado un morfismo de T-álgebras, tenemos un morfismo de autómatas. Intentemos extrapolar las dos reglas que verifica la estructura de la T-álgebra al morfismo deas salida γ.

Regla 1. Sabemos que, siendo $f = \lambda s.(a, s)$, $\xi(f) = a$. Veamos cuánto vale $\gamma(\beta_f)$. Primeramente, calculemos β_f para esta f peculiar: $\beta_f = \rho_f;\gamma_f$. Pero $\gamma_f = f;\pi_1$; luego, $\gamma_f(s) = a$, $a \in S$; Para calcular ρ_f precisamos conocer $\delta_f = \tau; f; \pi_2$; entonces, tenemos:

que $\delta_f(s,()) = s, \forall s \in S$. En tal caso, $\rho_f(\emptyset) = s_0$, $\rho_f(()) = \delta_f(s_0,()) = s_0$; En general, $\rho_f(w) = s_0, \forall w \in \Sigma^*$. Es decir, $\beta_f(w) = a, \forall a \in \Sigma^*$. Luego, $\gamma(\beta_f) = a$.

Regla 2. Cuando tenemos $f : S \to AxS$ tal que, para un estado s_0, $f(s_0) = (\xi(g), s_1)$, hemos visto que $\xi(f) = \xi(f')$, siendo $f'(s) = f(s)$, $sneqs_0$, $f'(s_0) = g(s_1)$. Por lo tanto, la regla trasladada a γ dice que $\gamma(\beta_f) = \gamma(\beta_{f'})$.

Para concluir este proceso establecemos morfismos de Σ-autómatas. La forma de definir tal morfismo es simple: sabemos que, si $\Omega : A \to B$ es un morfismo de T-álgebras, dado $f \in (S \to (A,S))$, se verifica que $\Omega(\xi_A(f)) = \xi_B(T(\Omega(f)))$. El objetivo es ver cómo definimos $\Omega_\sharp$ a partir de Ω.

Sabemos que los elementos de $A_\sharp$ son de la forma β_f, con $f \in T(A)$; entonces, podemos definir $\Omega_\sharp(\beta_f) = \beta_f; \Omega$, morfismo entre Σ^* y B, es decir, así definido, $\Omega_\sharp(\beta_f) \in B_\sharp$. Pero, por ser $\beta_f = \rho_f; \gamma_f$, escribimos $\Omega_\sharp(\beta_f) = \rho_f; (\gamma_f; \Omega)$. Pero, obviamente, $\gamma_f; \Omega = (T(\Omega)(f))_\sharp$, siendo $(T(\Omega))(f) : S \to (B,S)$, definida como $(T(\Omega))(f))(s) = (f(x), s')$, siendo $(x,s') = f(s)$. Por lo tanto, $\gamma_B(\Omega_\sharp(f)) = \gamma_B(f_\sharp; \Omega) = \xi_B((T(\Omega)(f))$. Entonces, por ser Ω morfismo de T-álgebras, $\Omega(\gamma_A(f)) = \gamma_B(\Omega_\sharp(f))$, como queríamos ver. Pero, hay un detalle importante: si $f(q_k) = (\xi(g), q_{k+1})$, no es necesario desarrollar el autómata generado por g hasta su estado final (tal como hemos reproducido el de f) sino que se obtiene la misma computación para f tomando, como representante para $\xi(g)$, el valor de tipo A que resulta de $g(q_{k+1})$. La noción que de aquí se infiere recuerda mucho al concepto de realización minimal: es decir, dado el morfismo $f : S \to (S,A)$, hacer uso del menor número posible de estados para desarrollar la función de comportamiento que resulta de f, β_f. Inicialmente, para la realización de β_f, precisamos de todo S, pero, si a la hora de computar $\xi(g)$ para obtener $\xi(f)$, podemos hacer uso únicamente, del estado q_{k+1} (en lugar de todo S) hemos acotado el uso de estados. Entonces, vemos que, para nuestra T-álgebra (A,ξ), $T(A) = S \to (A,S)$ se limita a ser isomorfa al Σ-autómata secuencial recién definido, siendo, en tal contexto, ξ representada por la función de salida del autómata, γ.

6.6.3 Categoría de Kleisli

Si $f \in C_T(A,B)$, $f : A \to T(C)$. Como $T(C) = S \to (B,S)$, podemos interpretar f como $f^* : (A,S) \to (B,S)$, donde $f^*(a,s) = (f\ a)\ s$. Entonces, la categoría de Kleisli sobre este triple tiene como objetos productos de la forma (A,S), con S el tipo de los estados y morfismos $f : (A,S) \to (B,S)$. Supongamos $g \in C_T(B,C)$, $g : B \to C$, representable como $g^* : (B,S) \to (C,S)$. Entonces, sea $(b,s') = f^*(a,s) = (f\ a)\ s$. Como $b \in B$, $g(b) : S \in (C,S)$. Luego, sea $(c,s'') = (g\ b)\ s' = g^*(b,s')$. Obviamente, $(c,s'') = (f;g)(a)(s)$. Entonces, globalizando, la categoría de Kleisli sobre este triple es isomorfa a la categoría con objetos pares de la forma (A,S) y morfismos $f : (A,S) \to (B,S)$.

7 Tipos Monádicos

7.1 *Catamorfismos monádicos en* Coq

Hasta ahora hemos visto una forma de estructurar programas *dirigida a la sintaxis* ([Meijer y Jeuring 1995]); a continuación presentamos otra posibilidad *dirigida a la semántica*. Para ello, seguimos precisando mónadas. Sea, pues, $(M, unit, bind)$ una mónada, con M endofuntor [de forma que, dado un tipo A, $M(A)$ será llamado tipo de las computaciones de tipo A], $unit$ la unidad que lleva valores de tipo A en computaciones de tipo A y $bind$ un funcional de orden superior que asigna, a cada morfismo $f : A \to M(B)$ un morfismo $bind\ f : M(A) \to M(B)$.

Entonces, dada m una computación de tipo A, $bind\ f$(m) actúa como sigue: evalúa m hasta obtener x, un valor de tipo A, al cual aplica la función f, dando lugar a una computación de tipo B. Notemos que, al igual que en el uso de las diálgebras, esta estructura tiene reminiscencias de concepto libre.

Para obtener la asociatividad de las mónadas habituales, notemos que, dada la función $id_{M(A)}$,

$$bind\ id_{M(A)} : M(M(A)) \to M(A).$$

Cabe pensar, siguiendo la línea que han marcado Meijer y Jeuring [Meijer y Jeuring 1995], Fokkinga [Fokkinga 1994], Hu e Iwasaki [Hu y Iwasaki 1995], cómo integrar ambas perspectivas: la proveniente de las diálgebras y la monádica. Curiosamente, a pesar de utilizar una estructura más general que la usada por ellos, no es preciso añadir ninguna restricción.

Consideremos, pues, F, G dos funtores definidos entre las categorías $\mathbf{C} \times \mathbf{T}$ y $\mathbf{T}$, siendo $\mathbf{T}$ la categoría de tipos y programas definida anteriormente.

Sea, además, $(M, unit, bind)$ una mónada en $\mathbf{T}$. La idea que subyace en la teoría de los catamorfismos monádicos consiste en pasar del ámbito de las $F\ G$-diálgebras en $\mathbf{T}$ a $F(GM)$-diálgebras en la categoría de Kleisli generada por M, $\mathbf{T^M}$.

Supongamos, por lo tanto, que $((A, B), \xi_1)$ y $((C, D), \xi_2)$ son $F\ G$-diálgebras en $\mathbf{T^M}$; dados los morfismos $f : A \to C$, $g : B \rightharpoonup D$, nuestro objetivo es poder definir morfismos $F^*(f, g) : F(A, B) \rightharpoonup F(C, D)$, $G^*(f, g) : G(A, B) \rightharpoonup G(C, D)$ (notemos que debemos extender nuestro contexto de actuación de $\mathbf{T}$ a $\mathbf{T^M}$) verificando la regla compositiva propia de las diálgebras,

$$\xi_1 \circ G^*(f, g) = F^*(f, g) \circ \xi_2.$$

Analizando los posibles formas de funtores polinómicos que pueden adoptar F y G, vemos que las construcciones aportadas por Fokkinga ([Fokkinga 1994]) y Hu&Iwasaki ([Hu y Iwasaki 1995]) son perfectamente adaptables a nuestra situación, con lo cual, sea

como sea la mónada M, es posible siempre obtener esa extensión; de hecho, al tratar con diálgebras, no precisamos que los funcionales F^* y G^* sean funtores (requisito que sí impone Fokkinga), estando, en tal caso, más próximos a la línea de Hu&Iwasaki.

Para ello, debemos contar con dos transformaciones d_{FM}, d_{GM}, definidas, para cada par (B, C), como

$$d_{FM}(B,C) : F(B, M(C)) \to M(F(B,C)),$$

$$d_{GM}(B,C) : G(B, M(C)) \to M(G(B,C)),$$

de forma que $F^*(f,g) = F(f,g); d_{FM}$ y $G^*(f,g) = G(f,g); d_{GM}$.

Por consiguiente, si $((A, \mu_A), \xi_A)$ es la diálgebra libre sobre A y $((C,D),\xi)$ es una F G-diálgebra en $\mathbf{T^M}$, dado cualquier morfismo $f : A \to C$, existe un único morfismo $Ext_H(f) : \mu_A \rightharpoonup D$ verificando que $(f, Ext_H(f))$ es un morfismo de F G-diálgebras en $\mathbf{T^M}$.

Ejemplo 7.1. *Consideremos la FList Π_2-diálgebra definida para la estructura de las listas. Construimos, primeramente, la mónada de los transformadores de estado, donde tras fijar un tipo de posibles estados,* `State`*, declaramos*

`type 'a ST=st of State->'a*State;;`.

[*En lo que sigue prescindiremos, por comodidad, del constructor* `st`].

Sean, además, $unit(x) = function\ s \to (x, s)$ y, siendo f:`'a` $\to$ `'b ST`*, con m un valor de tipo* `'a ST`*,*

$$bind\ f\ m = function\ s \to (b, s'')\ where\ let\ (a, s') = m\ s$$

$$in\ let\ (b, s'') = f(a)(s').$$

En tales condiciones, la definición de d_{FListM} viene dada por: $d_{FListM}('a,'b)$: `()+'a*'b ST`$\to$ `(()+'a*'b) ST`*, con*

$$d_{FListM}('a,'b)\ () = function\ s \to ((), s)$$

$$d_{FListM}('a,'b)\ (x, m) = function\ s \to ((a, b), s')\ where\ (b, s') = m\ s.$$

Naturalmente, $d_{\pi_2 M}('a,'b) = id_{'bState}$.

Consideremos, nuevamente, F, G endofuntores; una F G-diálgebra (A, ξ) es *regular* si existen morfismos $\xi_F : F(A) \to A$, $\xi_G : A \to G(A)$ de forma que $\xi = \xi_F; \xi_G$. Un morfismo entre dos F G-diálgebras regulares (A, ξ), (B, ψ) es un morfismo de F G-diálgebras verificando $\xi_F; f = F(f); \psi_F$ y $f; \psi_G = \xi_G; G(f)$.

Las F G-diálgebras regulares y sus morfismos forman una subcategoría de $\mathbf{DAlg}(F,G)$. Además, la categoría de las F G-diálgebras regulares genera una categoría de F-álgebras y otra categoría de G-coálgebras. Una F G-diálgebra regular (A, ξ) es *singular* si $\xi = in; out$ siendo (A, in) una F-álgebra inicial y (A, out) una G-coálgebra final.

Analicemos estas construcciones en el contexto de `Coq`. Primeramente construimos la estructura propia de una mónada para, a posteriori, definir el catamorfismo monádico generalizado que podemos declarar sobre el tipo de las listas:

```
Section cat.

Variable M:Set->Set.

Variable eta :(A:Set)(A ->(M A)).
Variable mu : (A:Set)((M (M A))->(M A)).
Variable map: (A,B:Set)(f:A->B)((M A)->(M B)).
Definition comp := [A,B,C:Set][f:A->B][g:B->C][x:A]
(g(f(x))).
Definition id := [A:Set][x:A]x.

Definition ley1 := (A:Set)
(comp (M A) (M (M A)) (M A) (eta (M A)) (mu A)) =
(id (M A)).

Definition ley2 := (A:Set)
(comp (M A) (M (M A)) (M A) (map A (M A) (eta A)) (mu A)) =
(id (M A)).

Definition ley3 := (A:Set)
(comp (M (M (M A))) (M (M A)) (M A)  (mu  (M A)) (mu A)) =
(comp (M (M (M A))) (M (M A)) (M A)
(map (M (M A)) (M A) (mu A)) (mu A)).
End cat.

Structure Monada : Type :=
{Carrier :> Set;
M:Set->Set;
map : (A,B:Set)(A->B)->((M A)->(M B));
eta : (A:Set)(A ->(M A));
mu : (A:Set)((M (M A))-> (M A));
l1 : (ley1 M eta mu);
l2 : (ley2 M eta mu map);
l3 : (ley3 M mu map)}.

Inductive List [A:Set] : Set := Nil :(List A)
|Cons: A->(List A)->(List A).

Definition Extmon := [T:Monada][A,B,C:Set]
[epsilon:B->(M T C)->C][c0:C][f:A->B]
Fix F {F [l:(List A)] : (M T C) :=
Cases l of
Nil => (eta T C c0)
|(Cons x xs) => (eta T C (epsilon (f x) (F xs))) end}.
```

Podemos ir más allá, trabajando no sólo con mónadas sino también con sus duales las comónadas. Supongamos, en tal caso, que (F, ϵ, δ) es una comónada y (G, η, μ) es una mónada en una categoría $\mathbf{T}$; entonces, para cualquier objeto A de $\mathbf{T}$, tenemos la F G-diálgebra $(A, \epsilon_A; \eta_A)$ que, por la definición, es regular.

7.2 *Anamorfismos comonádicos*

Analicemos qué problema encontramos para definir la construcción dual de la anterior. Consideremos, como antes, el caso del tipo de las listas con la estructura anamórfica asociada. Sea, además, $\mathbf{M} = (M, \delta, \epsilon)$ una comónada (es decir, $\delta : M \Rightarrow 1$, $\epsilon : M \Rightarrow M^2$ transformaciones naturales satisfaciendo las conmutatividades duales a las definidas con las mónadas. En estas condiciones podemos definir el morfismo `anacom` siguiendo el procedimiento dual del anterior:

```
Section cat.

Variable C:Set->Set.

Variable delta :(A:Set)((C A) -> A).

Variable epsilon : (A:Set)((C A)->(C(C A))).
Variable map: (A,B:Set)(f:A->B)((C A)->(C B)).
Definition comp := [A,B,C:Set][f:A->B][g:B->C][x:A](g(f(x))).
Definition id := [A:Set][x:A]x.

Definition ley1 := (A:Set)
(comp (C(C A)) (C A) (C(C A)) (delta (C A)) (epsilon A)) =
(id (C (C A))).

Definition ley2 := (A:Set)
(comp (C(C A)) (C A) (C(C A)) (map (C A) A (delta A))
(epsilon A)) =(id (C (C A))).

Definition ley3 := (A:Set)
(comp (C A) (C (C A)) (C (C (C A))) (epsilon A)
(epsilon (C A))) = (comp (C A) (C (C A)) (C (C (C A)))
(epsilon A) (map (C A) (C (C A)) (epsilon A))).
End cat.

Structure Comonada : Type :=
{Carrier :> Set;
C:Set->Set;
map : (A,B:Set)(A->B)->((C A)->(C B));
delta : (A:Set)((C A) -> A);
epsilon : (A:Set)((C A)->(C(C A)));
l1 : (ley1 C delta epsilon);
```

```
l2 : (ley2 C delta epsilon map);
l3 : (ley3 C epsilon map)}.


CoInductive List [A:Set] : Set := Nil :(List A)
|Cons: A->(List A)->(List A).

Inductive AnaList [A:Set] : Set := ANil: (AnaList A)
|Pair : A*B->(AnaList A).

Definition Anacom := [H:Comonada] [A,B,C:Set]
[theta:C->(AnaList B (M H C))] [f:B->A]
CoFix F {F: (M H C) -> (List A) := [x:(M H C)]
Cases (theta(delta H C x)) of

ANil => (Nil A)
|(Pair(b,y)) => (Cons A (f b) (F y)) end}.
```

Comprobamos como usamos la doble condición de inductivas y coinductivas que tienen las listas. En el primer caso, al utilizarlas como argumento de entrada, usamos su caracterización inductiva para utilizar la recursión. En el segundo, apelamos a su carácter de objeto final para aplicar la coinducción.

8 Cuantificación Existencial

8.1 *Introducción*

Hemos tratado en los capítulos precedentes temas referentes a, básicamente, dos clases de tipos: no paramétricos (cuyos representantes son los tipos delos números naturales o de los ordinales) y tipos paramétricos (listas, árboles, tipos anidados y monádicos). Un tema que también nos ha interesado sobremanera es el concerniente a los tipos cuantificados existencialmente, es decir, tipos donde el parámetro que usamos en su definición no está explícitamente declarado antes de ésta. Los tipos abstractos son tipos cuantificados existencialmente ya que nosotros conocemos una especificación de ellos pero no conocemos una implementación concreta. Entonces, llevado esto al extremo, el tipo existencial es el tipo abstracto por excelencia. Por ejemplo, el tipo

```
data Lista = forall a. Cons (a,Lista) | Nil
```

que representa el tipos de las listas con elementos de *cualquier tipo*. [Ya hemos utilizado previamente el constructor `forall` para indicar un polimorfismo local en las definiciones correspondientes a los tipos anidados.] En este caso la aplicación es diferente. El parámetro `a` no es local: en la definición indicamos que, en cada aplicación del constructor `Cons` podemos utilizar un valor de un tipo cualquiera. Así, el término

```
term=Cons (1, Cons ([1,2], Cons(True, Nil)))
```

es perfectamente correcto: tenemos, pues, listas heterogéneas. Observemos la diferencia existente entre esa declaración y la siguiente:

```
data Llista =  LNil | LCons (forall a.a) Llista
```

En este caso sí hemos declarado un polimorfismo local de forma que, ahora, siempre que utilicemos el constructor `Cons` debemos tener en cuenta esa condición de universalidad. Este constructor `Llista` es lo que previamente hemos denominado tipo polimórfico de rango 2, imprescindible para poder definir funciones sobre los tipos anidados. Es decir, podemos definir una función como

```
long :: Llista -> Integer
long LNil = 0
long (LCons x xs) = 1 + (long xs)
```

pero no es posible establecer el término

```
term2 =LCons 1 (LCons [1,2] (LCons True LNil))
```

puesto que el sistema nos informa de que hemos determinado una restricción no prevista en la declaración. Debemos prescindir de la realización de restricciones de tipo que no queden determinados en la definición de éste. Si hubiéramos hecho la declaración

```
data RLista b =  RCons (forall a.a->a->a) b
```

sí podríamos haber definido el morfismo

```
h :: RLista a -> a -> a
h (RCons f k) x = f k x
```

Observemos que se produce una instanciación de las variables `f, k` al aplicarlas al valor `x` de tipos `a` quien está perfectamente determinado. Las aplicaciones que tienen los tipos cuantificados existencialmente son inmediatas: podemos usarlos en la circunstancia de obtener un valor cuyo tipo no sea conocido previamente a su consecución. Consideremos, por ejemplo un sistema cliente-servidor donde el cliente emite demandas que el servidor correspondiente debe responder [Gulías 1994] : éstas van conformando una lista. Naturalmente, es imposible saber, en el momento de la construcción de ésta, cuáles van a ser los tipos de las respuestas. Una solución la tenemos, aprovechando el tipo `Maybe` aportado por el sistema, con el siguiente tipo:

```
data Remota =  forall a. Val (Bool,Integer,Maybe a)
type ListaRemota = [Remota]
```

El tipo `Remota` consta de tres parámetros: el boolenao indica el carácter de lleno o vacío (según haya llegado o no la respuesta del servidor); el entero es el identificador de la demanda y, por último, tenemos un campo para la respuesta que, en previsión de que no haya llegado, lo cuestionamos usando el tipo `Maybe`; además, al no tener un tipo conocido de antemano, lo dejamos con una cuantificación existencial. Observemos que esto permite no hacer uso de tipos *mutables* ni de guardar valores inconsistentes en espera de la llegada de la respuesta. El tipo `ListaRemota` contiene una lista donde, por orden de llegada, van acumulándose las diferentes respuestas de los servidores a las demandas de los clientes. En el estado actual de la cuestión existencial, el tipo cuantificado existencialmente no puede ir más allá del alcance del pattern matching [Haskell 1998]: es decir, el valor de un tipo existencial nunca puede ser el resultado de una computación (a fin de cuentas,¿ qué tipo le asignaríamos si nos es desconocido pues es una variable? [Aunque tengamos un valor concreto con un tipo determinado, al ser existencial es un tipo abstracto, sin concreción]. Por ello, precisamos un conjunto de funciones que, empaquetadas con el valor existencial, manipulen esos elementos heterogéneos de forma que podamos recuperarlos y tratarlos a todos ellos de una manera útil. Un ejemplo de esta idea la tenemos en el siguiente caso:

```
data Key = forall a. K a (a->Integer)

f :: Key -> Integer
```

```
f (K val fn) = fn val

car :: Bool -> Integer
car True = 1
car False = 0

elemento1 :: Key
elemento1 = K 5 (\x->x+10)

elemento :: [Key]
elemento = [K 1 id,K False car]
```

Observemos que el valor `5` con que hemos definido `elemento1`, en ese momento, no es un entero sino que, su tipo es abstracto. En cambio, cuando calculamos `f elemento1`, el 5 ya es un número entero y la salida que obtenemos es, como corresponde por la función que acompaña al número, 15. También, si calculamos `map f elemento` obtenemos, como parece obvio por las declaraciones precedentes, `[1,0]`. Es por tanto imprescindible acompañar el tipo existencial con una función que lo manipule, permitiendo obtener un entero como valor representativo de su computación.

Otra forma de aplicar estas enormes posibilidades de los tipos existenciales surge con los tipos dinámicos [Duggan 1999]. Se define los dinámicos como tipos existenciales de la forma $\exists A.A$.

Para poder trabajar con estos se introduce un tipo base denominado `Dynamic` con una expresión `dynamic` para construr valores de tipo `Dynamic` y una expresión `typecase` para inspeccionarlos de acuerdo a las siguientes reglas:

$$\frac{C \vdash a \in T}{C \vdash dynamic(a : T) \in Dynamic}$$

$$\frac{C \vdash d \in Dynamic,\ C, x : P \vdash b \in T\ C \vdash c \in T}{C \vdash typecase\ d\ of\ (x : P)\ b\ else\ c \in T}$$

La potencia expresiva de esta construcción es inmediatamente constrastable [Duggan 1999].

Observemos que estamos hablando de tipos *lazy*, es decir, tipos que son inferidos sólo cuando es necesario hacerlo: estamos hablando de un *lenguaje polimórfico con tipos lazy*. Así, pues, el sistema de inferencia de tipos, encargado de deducir el tipo de cualquier expresión, debe cubrir dos propósitos:

1. ver y analizar el buen tipado de los valores;
2. en el caso de hallar un tipo cuantificado existencial, obviarlo, asignándole como valor una nueva variable [no es posible unificar dos tipos existenciales distintos], hasta que sea requerido y demandado concretamente: ésta es, entonces, la filosofía de los tipos dinámicos.

Recordemos que en el sistema de tipos de Hindley-Milner, los tipos cuantificados tanto universal como existencialmente no pueden ser sustituidos por variables de tipo: se

conjetura [es un problema abierto] que la reconstrucción de tipos en esas condiciones en las que los cuantificadores pueden afectar a tipos polimórficos es indecidible. Pongamos un ejemplo para ver los riesgos que conlleva considerar variables sobre los tipos cuantificados. Consideremos el tipo $\exists A.A \times A$; consta de pares de valores **del mismo tipo**. Si alguna de las A fuese una variable, la segunda, por ejemplo, podría tomar el valor $\exists A.A$, de donde, el tipo anterior quedaría $\exists A.A \times \exists A.A$. Como vemos, ya no resultan pares del mismo tipo.

En múltiples ocasiones, al definir funciones, hacemos uso de sus tipos cuando estos, realmente, son prescindibles. Por ejemplo, cuando definimos la función `append` para pegar dos listas, nos resulta totalmente indiferente el tipado de esos valores. La computación sin los tipos existenciales nos obliga a considerar siempre listas con elementos del mismo tipo [aunque, como acabamos de decir, para el pegado los tipos de elos elementos es irrelevante]. La cuantificación existencial elimina esa restricción al poder definir:

```
append :: Lista -> Lista -> Lista

append Nil l = l
append l Nil = l
append (Cons x xs) l = Cons x (append xs l)
```

Al ser el tipo `Lista` heterogéneo, podemos pegar listas cualesquiera sin restricción:

```
lista1=Cons 1 (Cons True (Cons 'c' Nil))
lista2=Cons False (Cons "Luego" (Cons 3.6 (Cons 15 Nil)))
append lista1 lista2 = Cons 1 (Cons True (Cons 'c'(Cons False
(Cons "Luego" (Cons 3.6 (Cons 15 Nil))))))
```

9 Conclusiones y trabajo futuro

A lo largo de esta memoria hemos investigado diversas construcciones surgidas del entorno de la programación funcional observadas desde una perspectiva categórica.

Por medio de las nociones ligadas al concepto de diálgebra introducimos el concepto de $\mu\nu$-tipo regular, caracterizado por la dualidad inicial–terminal. Esta construcción nos permite declarar tipos sobre estructuras previamente establecidas (como, por ejemplo, una generalización de la definición del álgebra de los árboles de aridad variable fundamentada en la abstracción de las listas) y los morfismos que surgen (y llegan) de ellos. Tal definición da lugar a la preservación de las propiedades iniciales y/o finales en los morfismos (la ley de fusión para catamorfismos y anamorfismos o la ley de deforestación), añadiendo nuevas posibilidades de optimización tal y como se ha demostrado mediante ejemplos ilustrativos. En este mismo contexto se aporta una condición para que la composición de hilomorfismos dé lugar a un hilomorfismo.

Es nuestro propósito seguir estudiando estas estructuras y sus propiedades con el objeto de aportar mayor abstracción a los programas y, por ende, usar morfismos más genéricos entre diferentes instancias de tipos algebraicos. Un objetivo inmediato, por ejemplo, es ver cómo integrar estas nociones en el ámbito de la programación genérica o *polytypic programming*. Además, es interesante analizar qué ventajas pueden hacer estas estructuras al estudio de los tipos con leyes, no interpretables estrictamente dentro de los algebraicos.

Estudiamos y aportamos una solución al problema de la construcción de funciones recursivas (catamorfismos, anamorfismos, hilomorfismos) desde y hacia tipos no regulares. Observamos como este problema puede verse desde dos perspectivas según sea *regularizable* en un número finito de pasos o no el tipo anidado. En el primer caso la solución la aporta el tipo regular equivalente al anidado y tal solución es implementable en cualquier lenguaje polimórfico sin ninguna construcción adicional a las convencionales. Para el segundo, en cambio, necesitamos un lenguaje con requisitos singulares: debe permitirnos la cuantificación existencial local. Tal posibilidad la provee, por ejemplo, Haskell con las signaturas de rango 2. En estas condiciones, usando las clases e instancias que Haskell permite declarar, podemos implementar las funciones buscadas. Es nuestra pretensión estudiar en más profundidad estos tipos anidados y las funciones recursivas definidas sobre ellos siguiendo los trabajos de Hinze, Gibbons, Paterson y Bird.

Un tema de creciente interés es el estudio de cómo extender el uso del operador foldl, función tail–recursive definida sobre el tipo de las listas, a otros tipos algebraicos. Tal y como hemos visto en varios casos particulares, una buena aproximación a esa extensión la aporta el uso de la noción de coigualador. Pretendemos seguir indagando acerca de la utilización de este operador de cara a obtener una solución global.

Implementamos, usando varias de las funciones previamente analizadas, soluciones a varios problemas: el cálculo del número cromático de un grafo usando un algoritmo de tie–breaking demostrado sumamente eficiente [Franco 1996]; la resolución del problema P–S de McCarthy; el cálculo del spanning–tree de un grafo usando la estructura de árbol de aridad variable previamente estudiada.

Comprobamos que definiendo los cotipos de forma coinductiva, tal como su condición de objeto terminal reclama, podemos construir el tipo de las *streams* o el tipo exponencial de forma isomorfa a sus análogos inductivos pero proveyéndolos de las funciones que los tienen como salida las cuales, desde la perspectiva inductiva, no son expresables. Pretendemos utilizar estos estudios sobre cotipos (tipos coinductivos) en la implementación de un sistema concurrente y sus propiedades lógicas en Coq. Para probar la equivalencia de programas funcionales los cuales, posiblemente, consuman y generen estrucuras de tipos de datos infinitas, se define coinductivamente la bisimulación aplicativa. Por consigueinte, en la verificación de sistemas concurrentes con conjuntos de estados previsiblemente infinitos, las estructuras coinductivas y sus propiedades juegan un papel relevante [Reichel 1995].

Implementamos el conjunto de las funciones primitivo recursivas en el ámbito del Cálculo de Construcciones Inductivas. Vemos, entonces, que la función de Ackermann, siendo un catamorfismo, no es una función primitivo recursiva. Mas, restringiendo ligeramente las condiciones de catamorfismo, demostramos que esa función ya no forma parte de este subconjunto de los catamorfismos.

Analizamos, además, cómo resolver el problema de la definición de funciones recursivas sobre tipos recursivos que hacen referencia, en su construcción, a otros tipos que los tienen como argumentos. Comprobamos que, definiéndolos en una construcción mutua, podemos declarar esas funciones que, en otro caso, son inaccesibles.

En esta memoria, por último, se identifican las álgebras sobre las mónadas utilizadas en los lenguajes funcionales para emular la programación imperativa. Así por ejemplo, la categoríade Eilenberg–Moore sobre la mónada de las continuaciones resulta ser la de los semiretículos completos mientras que la correspondiente a la mónada de los lectores de estado es la de los Dominios Computacionales. Como trabajo futuro queda la utilización de estos resultados para mejorar las distintas implementaciones actuales de las mónadas en lenguajes funcionales. Ello debido a que el conocimiento de las álgebras y las resoluciones correspondientes de las mónadas, permiten dicha mejora. La importancia de las adjunciones monádicas radica en el hecho de que las construcciones universales con álgebras pueden ser computadas por los *carriers*, como se desprende del trabajo de Beck [Manes 1976].

Referencias

[Aguado 1992] F. Aguado, Construcción de Elementos de Categorías en Computación. *Tesis Doctoral. Universidad de La Coruña.* (1992)

[Altenkirch 2001] T. Altenkirch, Representations of first function types as terminal coalgebras. *TLCA 2001. LNCS 2044* (2001)

[Asperti y Longo 1991] A. Asperti, G. Longo, Categories, Types and structures. An introduction to category theory for the working computer scientist. *Foundation of Computing Series. The MIT Press* (1991)

[Asperti 1992] A. Asperti,A categorical understanding of environment machines. *Journal of Functional Programming. Cambridge University Press* (1992)

[Backhouse y Hoogendijk 1999] R. Backhouse, P. Hoogendijk, Final Dialgebras: from categories to allegories. *Theoretical Informatics and Applications, 33* (1999)

[Barendregt 1984] H.P. Barendregt, The lambda calculus: its syntax and semantics. *North Holland* (1984)

[Barr y Wells 1999] M. Barr, C. Wells, Category Theory for Computing Science. *Les publications Centre de recherches mathématiques* (1999)

[Bellé 1996] G. Bellé, C.B. Jay, E. Moggi, Functorial ML. Advanced Functional Programming. *Lecture Notes in Computer Science, 1200. Springer–Verlag* (1996)

[Bird y de Moor 1993] R. Bird, O. de Moor, Solving Optimisation Problems with Catamorphims. *Lecture Notes in Computer Sciencie, 669. Springer Verlag* (1993)

[Bird y Meertens 1998] R. Bird, L. Meertens, Nested datatypes. *Mathematical foundations for computing. Lecture Notes in Computer Science. Springer–Verlag* (1998)

[Bird y Paterson 1999] R. Bird, R. Paterson, de Bruijn notation as a nested datatype. *Journal of function al programming,* **9** *(1). Cambridge University Press* (1999)

[Bird y Paterson 1999 (2)] R. Bird, R. Paterson, Generalised folds for nested datatypes. *Formal Aspects of Computing, 11.* (1999)

[Cardelli 1986] L. Cardelli, A polymorphic λ–calculus with **Type** : **Type**. *SRC Research Report 10. Digital Equipment Corporation* (1986)

[Castéran 1993] P. Castéran, Pro[gramm,v]ing with continuations: A development in Coq. *Technical Report. URA CNRS 1304. Universidad de Burdeos* (1993)

[Cockett y Fukushima 1992] R. Cockett, T. Fukushima, About **Charity**. *Technical Report. Calgary University*(1992)

[Cousineau y Mauny 1995] G. Cousineau, G. M. Mauny, Approche fonctionnelle de la programmation. *Ediscience International* (1995)

[Crole 1993] R. L. Crole, Categories for Types. *Cambridge Mathematical Textbooks* (1993)

[Davie 1992] A. Davie, An introduction to Functional Programming systems using Haskell. *Cambridge Computer Science Texts. Cambridge University Press*(1992)

[Dowek y otros 1993] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murty, C. Parent, C. Paulin-Mohring, B. Werner, The Coq proof assistant user's guide. *Technical Report 134. INRIA* (1993)

[Dowek 1998] G. Dowek, Théories des types. *Technical Report. INRIA-Rocquencourt* (1998)

[Duggan 1999] D. Duggan, Dynamic typing for distributed programming in polymorphic languages. *ACM Transations on Programming Languages and Systems, vol. 21, n 1* (1999)

[Erwig 1998] M. Erwig, Categorical Programming with Abstract Data Types. AMAST'98. *Lecture Notes in Computer Science 1548, Springer-Verlag* (1998)

[Erwig 2000] M. Erwig, Programs are Abstract Data Types. *Technical Report. Oregon State University* (2000)

[Erwig 2000(2)] M. Erwig, Random Access to Abstract Data Types. *AMAST'2000. Lecture Notes in Computer Science 1816, Springer-Verlag* (2000)

[Even 1979] S. Even, Graph Algorithms. *Computer Science Press* (1979)

[Field y Harrison 1988] A.J. Field, P.G. Harrison, Functional programming. *International Computer Science Series. Addison-Wesley, Publishing Company* (1988)

[Fokkinga 1992] M.M. Fokkinga, Calculate categorically ! *Formal Aspects de Computing, 4(4)* (1992)

[Fokkinga 1994] M.M. Fokkinga, Monadic Maps and Folds for Arbitrary types. *Memoranda Informatica , 94-28* (1994)

[Fokkinga 1996] M.M. Fokkinga, Datatype laws without signatures. *Mathematical Structures in Computer Science, vol. 6. Cambridge University Press* (1996)

[Fernández y Freire 1985] R. Fernández, J.L. Freire, La teoría algebraica de los retículos completamente distribuidos. *X Jornadas Hispano Lusas de Matemáticas. Universidad de Murcia*(1985).

[Franco 1996] J.R. Franco, Reducción del efecto fill-in en sistemas lineales sparse de matriz simétrica. *Tesis Doctoral. Departamento de Matemáticas. Universidad de Las Palmas de Gran Canaria* (1996).

[Freire 1972] J.L. Freire, Propiedades Universales en Triples de Grado Superior. *Álxebra 11. Departamento de Álgebra y Fundamentos. Universidad de Santiago de Compostela* (1972).

[Freire 1980] J.L. Freire, Sobre subteoriías de la doble dualidad. *Álxebra 26. Departamento de Álgebra y Fundamentos. Universidad de Santiago de Compostela* (1980).

[Freire y Blanco 1996] J.L. Freire, A. Blanco, On the abstraction process. *In Brain Processes and Models. MIT Press* (1996).

[Freire y otros 2001] J. L. Freire Nistal, A. Blanco Ferro, J. E. Freire Brañas, J. J. Sánchez Penas, Fusion and Deforestation in Coq. Lecture Notes in Computer Science. Vol. 2178, Pags. 583-596. Ed. Springer-Verlag.

[Freire y otros 2003] J. L. Freire Nistal, A. Blanco Ferro, V. Gulías, J. E. Freire Brañas, The Reduction Lemma in Coq. Proc. Eurocast 2003.

[Freire y otros 2003] J. L. Freire Nistal, A. Blanco Ferro, V. Gulías, J. E. Freire Brañas On the Strong Co–induction in Coq. To appear in *Lecture Notes in Computer Science. Ed. Springer-Verlag.*

[Freyd 1990] P. Freyd, Recursive types reduced to inductive types. *IEEE* (1990)

[Gibbons 1994] J. Gibbons, An Introduction to the Bird-Meertens Formalism. *Technical Report, University of Auckland* (1994)

[Gibbons 1996] J. Gibbons, The Third Homomorphism Theorem. *Journal of Functional Programming, 6 (4) Cambridge University Press* (1996)

[Gibbons 2000] J. Gibbons, G. Hutton, Proof methods for corecursive programs. *Technical Report, Oxford UNiversity* (2000)

[Gibbons, Hutton, Altenkirch 2001] J. Gibbons, G. Hutton, T. Altenkirch, When is a function a fold or an unfold? *Coalgebraic Methods in Computer Science. Electronic Notes in Theoretical Computer Science, 44.1* (2001).

[Goguen 1990] J. A. Goguen, A Categorical Manifesto. *Technical Report. Programming Research Group. University of Oxford* (1990)

[Gordon 1994] A.D. Gordon, Functional Programming and input/output. *Distinguished dissertations in computer science. Cambridge University Press* (1994)

[Gordon 1994] A.D. Gordon, A Tutorial on Co–Induction and Functional Programming. *Glasgow Workshop on Functional Programming. Springer Workshops in Computing* (1994)

[Gordon y otros 1979] M. J. Gordon, A. J. Milner, C. P. Wadsworth, Edinburgh LCF: la Mechanised Logic of Computation. *Lecture Notes in Computer Science 78, Springer–Verlag* (1979)

[Gordon y Felham 1993] M. J. C. Gordon, T. F. Felham, HOL. *Cambridge Univ. Press* (1993)

[Gray 1991] J.W. Gray, Products in **PER**: an elementary treatment of the semantics of the polymorphic lambda calculus. *Category Theory at work. Heldermann Verlag* (1991)

[Gregorio y otros 1995] C. Gregorio, M. Nú ñez, P. Palao, La potencia expresiva de los catamorfismos. *GULP-PRODE 1995* (1995)

[Greiner 1992] J. Greiner, Programming with Inductive and C-Inductive Types. *Technical Report. School of Computer Science. Carnegie Mellon University. Pittsburgh* (1992)

[Gulías 1994] V.M. Gulías, Interfaz Gráfica para una implementación del λ–cálculo construida sobre entornos heterogéneos. *Proyecto fin de carrera de Licenciatura. Universidad de A Coruña.* (1994)

[Gunter 1992] C.A. Gunter, Semantics of programming languages. Structures and techniques. *The MIT Press* (1992)

[Hagino 1987] T. Hagino, A categorical programming language. *Tesis de doctorado. Edinburgh University* (1987)

[Hagino 1989] T. Hagino, Codatatypes in ML. *, Journal of Symbolic Computation, 8. Academic Press Limited* (1989)

[Hall y otros 1996] C.V. Hall, K. Hammond, S.L. Peyton Jones, P.L. Wadler, Type classes in Haskell. *ACM Transactions on Programning Languages and Systems* (1996)

[Hasegawa 1994] R. Hasegawa, Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science* (1994)

[Hasegawa 1996] R. Hasegawa, The theory of twiners and linear parametricity (Note). *Technical Report. Graduate School of Mathematical Science. University of Tokyo* (1996)

[Haskell 1998] Explicit quantification in Haskell. *Technical Report. Glasgow University* (1998)

[Hinze 1999] R. Hinze, Numerical Representation as higher-order nested datatypes. (1998)

[Hinze 1999] R. Hinze, Polytypic programming with ease. *In proceedings of the FLOPS'99. Lecture Notes in Computer Science 1722. Springer–Verlag* (1999)

[Hinze 2000] R. Hinze, A new approacho to generic functional programming. *In proceedings of the 27th Annual acm sigplan-sigact symposium on principles of programming languages. ACM Press* (2000)

[Howard 1980] W. la Howard, The formulae-as-type notion of construction. *In H.B.Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism. Academic Press* (1980)

[Hu, Iwasaki, Takeichi 1994] Z. Hu, H. Iwasaki, M. Takeichi, Catamorphism–based transformation of functional programs. *METR 94–06* (1994)

[Hu y Iwasaki 1995] Z. Hu, H. Iwasaki, Promotional Transformation of Monadic Programs. *Fuji International Workshop on Functional and Logic Programming* (1995)

[Hu, Iwasaki, Takeichi 1995] Z. Hu, H. Iwasaki, M. Takeichi, Deriving Structural Hylomorphisms from recursive definitions. *METR 95-11* (1995)

[Hudak 1989] P. Hudak, Conception, Evolution and Application de Functional Programming Languages. *ACM Computing Surveys* Vol. 21, No. 3, pp.361–411, (1989).

[Huet y Amokrane 2000] G. Huet, A. Saïbi, Constructive Category Theory. *Proof, Language and Interaction. Essays in Honour of Robin Milner. The Mit Press* (2000)

[Hughes 1990] J. Hughes, Why Functional Programming Matters. *Research Topics in Functional Programming pp.17–42. Addison-Wesley* (1990).

[Hutton 1999] G. Hutton, A tutorial on the uiversality and expressiveness of fold. *Technical Report. University of Nottingham* (1999)

[Jacobs y Rutten 1997] B. Jacobs, J. Rutten, A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin 62* (1997)

[Jay y otros 1998] C.B. Jay, G. Bellé, E. Moggi, Functorial ML. *Journal of Functional Porgramming, Cambridge University Press* (1998)

[Jeuring y Jansson 1996] J. Jeuring, P. Jansson, Polytypic Programming. *Lecture Notes in Computer Science, Spronger Verlag* (1996)

[Jones 1995] M.P. Jones, Functional Programming with overloading and Higher-Order Polymorphism. *Advanced Functional Programming, Lecture Notes in Computer Science 925* (1995)

[Jürgensen 2000] C. Jürgensen, A formalization of hylomorphism based deforestation with an application to an extended typed λ-calculus. *Technical Report. Deparment of Computer Science. Dresden University of Technology* (2000)

[Kieburtz y Lewis 1995] R.B. Kieburtz, J. Lewis, Programming with algebras. *Advanced Functional Programming. Lecture Notes in Computer Science, 925, Springer–Verlag* (1995)

[Kieburtz 1999] R.B. Kieburtz, Codata and Comonads in Haskell. *Technical Report. Oregon Graduate Institute* (1999)

[Krivine 1990] J.L. Krivine, Lambda-calcul. Types et modeles. *Études et recherches en informatique. Masson* (1990)

[Lambeck y Scott 1989] J. Lambeck, P.J. Scott, Introduction to higher order categorical logic. *Cambridge studies in advanced mathematics 7, Cambridge University Press* (1989)

[Lämmel, Visser, Kort 2001] R. Lämmel, J. Visser, J. Kort, Dealing with large bananas. *Technical Report. University of Amsterdam* (2001)

[Läufer y Odersky] K. Läufer, M. Odersky, An extension of ML with first-class abstract types. *Proceeding of the 1992 workshop on ML ans its applications* (1992)

[Läufer y Odersky 1993] K. Läufer, M. Odersky, Type classes are signatures de abstract types. *Journal de function al programming, 2* (1993)

[Läufer y Odersky 1994] K. Läufer, M. Odersky, Polymorphic Type Inference and Abstract Data Types. *ACM Transactions on Programming Languages and Systems* (1994)

[Läufer 1996] K. Läufer, Type classes with existential types. *Journal de function al programming, 6* (1996)

[Lewis y otros 2000] J.R. Lewis, M.B. Shields, E. Meijer, J. Launchbury, Implicit parameters: dynamic scoping with static types. *POPL'00: the* 27^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM* (2000)

[McBride 1999] C. McBride, Dependently typed functional programs and their proofs. *Tesis Doctoral. University of Edinburgh* (1999)

[Mac Lanes 1971] S. Mac Lane, Categories for the working mathematician. *Graduate Texts in Mathematics 5. Springer-Verlag* (1971)

[Manes 1976] E.G. Manes, Algebraic Theories. *Graduate Texts in Mathematics, 26. Springer-Verlag* (1976)

[Manes y Arbib 1986] E.G. Manes, M.A. Arbib, Algebraic approaches to program semantics. *The AKM Series in Theoretical Computer Science, Springer-Verlag* (1986)

[Martin y Gibbons 2001] C. Martin, J. Gibbons , On the semantics of nested datatypes. *Information Processing Letters, 80* (2001)

[Martin, Gibbons y Bayley 2001] C. Martin, J. Gibbons, I. Bayley, Disciplined, efficient, generalised folds for nested datatypes. *Technical Report. Oxford University Computing Laboratory* (2002)

[Mauny y Pottier 1993] M. Mauny, F. Pottier, An implementation of CAML-Light with existential types. *Rapport Techniques. INRIA* (1993)

[Mauny 1995] M. Mauny, Types in programming. *Notes of course. A Coruña* (1995)

[Meertens 1990] L. Meertens, Paramorphisms. *Formal Aspects of Computing 4(5):413-424* (1990)

[Meertens 1996] L. Meertens, Calculate Polytypically. *Lecture Notes in Computer Science, Springer Verlag* (1996)

[Meijer y otros 1991] E. Meijer, M.M. Fokkinga, R. Paterson, Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. *FPCA91, Lecture Notes in Computer Science 523, Springer Verlag pp.124–144* (1991)

[Meijer y Hutton 1994] E. Meijer, G. Hutton, Bananas in space: extending fold and unfold to exponential types. *Technical Report. University of Utrech* (1994)

[Meijer y Jeuring 1995] E. Meijer, J. Jeuring, Merging Monads and Folds for Functional Programming. *Advanced Functional Programming, Lecture Notes in Computer Science 925, Springer Verlag pp. 28-266* (1995)

[Mitchell y Plotkin 1988] J.C. Mitchell, G. Plotkin, Abstract types have Existential type. *ACM Transactions on programming languages and systems* (1988)

[Mitchell 1996] J.C. Mitchell, Foundations for programming languages. *Foundations de Computing Series, The MIT Press* (1996)

[Moggi 1989] E. Moggi, Computational lambda calculus and monads. *IEEE Conference on Logic in Computer Science* (1989)

[Pardo 1997] A. Pardo, A Calculational Approach to Strong Datatypes. *Technical Report. Technische Hochschule Darmstadt. Germany* (1997)

[Paulin y Werner 1998] C. Paulin-Mohring, B. Werner, Calcul des Constructions Inductives. *DEA Sémantique, Preuves et Programmation 1998-99. Technical Report* (1998)

[Paulson 1998] L. C. Paulson, A Fixedpoint approach to (Co)Inductive and (Co)Datatype Definitions. *Technical Report. Computer Laboratory. University of Cambridge* (1998)

[Peyton Jones 1987] S.L. Peyton Jones, The implementation of the functional programming languages. *Prentice Hall International Series in Computer Science. Prentice Hall* (1987)

[Pierce 1993] B.C. Pierce, F-Omega-Sub User's Manual Version 1.3. *Technical Report* (1993)

[Pitts 1995] A. M. Pitts, Categorical Logic. *Technical Report. Cambridge University* (1995)

[Plotkin y Abadi 1993] G. Plotkin, M. Abadi, A logic for parametric polymorphism. *Lecture Notes in Computer Science 664. Springer–Verlag* (1993)

[Poigné 1992] A. Poigné, Basic category Theory. *Handbook of Logic in Computer Science pp. 416–634. Oxford Science Public* (1992)

[Poll y Zwanenburg 2001] E. Poll, J. Zwanenburg, From Algebras and Coalgebras to Dialgebras. *Coalgebraic Methods in Computer Science 2001* (2001)

[Power 1998] J. Power, Categories with algebraic structure. *Lecture Notes in Computer Science 1414. Springer-Verlag* (1998)

[Reichel 1995] H. Reichel, An approach to object semantics based on terminal co–algebras. *Mathematical Structures in Computer Science, Vol. 5, N. 2* (1995)

[Robin y otros 1992] J. Robin, B. Cockett, D. Spencer, Strong categorical datatypes I. *Canadian Mathematical Society, Conference Proceeding, Volume 13* (1992)

[Jacobs and Rutten 1997] B. Jacobs, J. Rutten, A Tutotial on (Co)Algebras and (Co)Induction. *EATCS Bulletin 62.* (1997)

[Rydeheard y Burstall 1988] D.E. Rydeheard, R.M. Burstall, Computational category Theory. *Prentice Hall International Series in Computer Science* (1988)

[Rydeheard 1985] D.E. Rydeheard, Adjunctions. *Lecture Notes in Computer Science 200. Springer–Verlag* (1985)

[Sancho 1990] J. Sancho San Román,Lógica Matemática y Computabilidad. *Ediciones Díaz de Santos* (1990)

[Stoy 1977] J.E. Stoy, Denotational semantics: the Scott-Strachey approach to programming language theory. *The MIT Press Series in Computer Science. The MIT Press* (1977)

[Szasz 1991] N. Szasz, A Machine Checked Proof that Ackermann's Fucntion is not Primitive Recursive. *Technical Report, Chalmers University of Technology*(1991)

[Taylor 1999] P. Taylor, Practical Foundations of Mathematics. *Cambridge University Press* (1999)

[Underwood 1995] J. Underwood, Typing Abstract data types. *Technical Report. University of Edinburgh* (1995)

[Wadler 1987] P. Wadler, Deforestation: transforming programs to eliminate trees. *Technical Report* (1987)

[Wadler 1993] P. Wadler, Comprehending monads. *ACM Conference on Lips and Functional Programming* (1993)

[Wadler 1994] P. Wadler, Monads and composable continuations. *LISP and SYMBOLIC COMPUT(A)TION. Kluwer Academic Publishers* (1994)

[Wadler 1997] P. Wadler, How to declare an imperative. *ACM Computing Surveys, Vol. 29* (1997)

[Wadler 2001] P. Wadler, The Girard-Reynolds Isomorphism. *TACS 2001* (2001)

[Weide y Ogden 1994] B. W. Weide, W. F. Ogden, Recasting Algorithms to Encourage Reuse. *IEEE Software Sept., pp.80–88* (1994)

[Winskel 1996] G. Winskel, The formal semantics of programming languages. An introduction. *Foundations of Computing Series. The MIT Press* (1996)

[Wraith 1989] G.C. Wraith, A note on categorical types. *In D.E. Rydeheard, P. Dybjer, A.M. Pits, and A. Poigné, editors, Category Theory and Computer Science. Lecture Notes in Computer Science, 389. Springer–Verlag* (1989)

[Yasuhara] A. Yasuhara, Recursive Function Theory and Logic. *Academic Press* (1971)

[Xi1999] H. Xi, Dependently typed data structures, Technical Report. *Oregon Graduate University* (1999)

Índice alfabético