Cálculo de Programas con Functores Aplicativos

Tesina de grado presentada
por

Germán Andrés Delbianco
D - 2088 /5

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

3 de Diciembre de 2010

Supervisores

Alberto Pardo

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Julio Herrera y Reissig 565, Montevideo, Uruguay

Mauro Jaskelioff

Departamento de Ciencias de la Computación,
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Av. Pellegrini 250, Rosario, Argentina

Resumen

Una práctica habitual en el paradigma de programación funcional es el uso de *operadores o esquemas* de recursión para estructurar algoritmos. Éstos son funciones de alto orden que capturan patrones de recursión comunes a las estructuras de datos que manipulan e.g. *map, fold*, y que se derivan de la interpretación categórica de los tipos de datos recursivos. El uso de estos operadores para estructurar programas funcionales resulta beneficioso, ya que permite aprovechar las propiedades algebraicas asociados de dichos operadores para derivar leyes genéricas para calcular programas eficientes y correctos.

Los *functores aplicativos* son una abstracción que modela *efectos computacionales* en lenguajes funcionales puros e.g. *entrada/salida, fallas, no determinismo*. Los functores aplicativos generalizan el concepto de *mónada* y favorecen un estilo *aplicativo* de programación.

Este trabajo propone una caracterización de la recursión estructural con *efectos aplicativos*. Presentaremos un operador genérico, al que denominamos *ifold*, que captura la esencia de la recursión estructural aplicativa. Además, presentaremos algunas propiedades *calculacionales* de este operador que permiten fusionar algoritmos aplicativos y mostramos algunos ejemplos de la aplicación de dichas reglas. Finalmente, mostraremos como este operador puede utilizarse para diseñar modularmente algoritmos con *efectos aplicativos* incrementales, agregando funcionalidad en cada etapa de forma sencilla y flexible.

Dedicado a la memoria de mi abuela Lala,
y su Aritmética General de bolsillo.

Farther west than west
beyond the land
my people are dancing
on the other wind.

Ursula K Leguin,
The Other Wind

Índice general

Agradecimientos

> It is good to have an end to
> journey towards, but it is the
> journey that matters in the end
>
> Ursula K Leguin,
> *The Left Hand of Darkness*

Este trabajo constituye el último capítulo de en una historia sobre un viaje que comenzó hace mucho tiempo.

Como en todas las grandes historias sobre viajes, el protagonista de esta historia nunca sabe en un principio cuál es el propósito que los lanza a la aventura, ni es advertido de los peligros que le avecinan, ni de los personajes que tendrá la suerte de conocer en el camino.

Como en todas las grandes historias sobre viajes, este protagonista nunca imaginó tampoco los horizontes extraños que debería recorrer en la búsqueda de su verdad.

Como en todas las grandes historias sobre viajes, y sobre otras cosas también, gran parte de esta historia se ha perdido en las arenas del tiempo y sólo este pequeño relato sobrevive.

Pero, el recuerdo de los que marcaron mi vida a lo largo de estos años perdura conmigo, y esta obra es un tributo a todos ellos: a los dragones, a los ogros, a los sabios de largas barbas y los aprendices de mago, a los fantasmas de vidas pasadas, a los *hetairoi* con los que he compartido victorias y fracasos, a la princesa que espera mi retorno a esta orilla del mar.

Esta historia no sería mi historia si no fuera por todos ustedes, y este es el momento apropiado para agredecerles:

A mi familia por el cariño, el soporte moral y espiritual. A mis padres por enseñarme a nunca rendirme. A mis abuelos por su ejemplo de trabajo y humildad. A mi hermana, por su ejemplo de constancia y voluntad y por ser más que a menudo la hermana mayor.

A mis directores, Alberto y Mauro, por guiarme en esta empresa y confiar en mí, por comprender mis urgencias y por permitirme tener el honor de trabajar con ustedes. He aprendido muchísimo de ustedes durante este proceso.

A los que me han acompañado en las diferentes etapas de este viaje: Juan, Damián, Julián, Pablo, Uciel, Bibi, Enrique, Rodrigo y Franco, por las charlas inspiradores y constructivas, y sobre todo, por las desopilantes; por la solidaridad para ayudarnos a superar escollos *a priori* infranqueables y por tantos momentos de *ñoñez* extrema.

A los docentes y /o alumnos que construyen día a día el *DCC* y hacen de esta una comunidad bastante particular, pero siempre cálida y acogedora. A los *ñoños*, *infras* y *Selenitas* por las discusiones productivas, el café infaltable y la *procrastinación* saludable.

A Fidel y a Guido, por contagiarme el deseo de enseñar y la pasión por el conocimiento; por mostarme el valor de las preguntas correctas, y también el de las preguntas que molestan; y por enseñarme la importancia de encarar la vida con *actitud científica*.

A *los chicos del poli* por dejarme crecer con ustedes y por compartir conmigo tantos momentos inolvidables. Aunque la vida nos ha llevado por caminos diferentes y distantes, voy a recordar siempre mi adolescencia con una sonrisa.

A *los chicos de water*, porque siempre van a ser los hermanos que nunca tuve, mi cable a tierra diario. Por dejarme ser parte de ese inconsciente colectivo capaz de prender fuego la Antártida. Pero, a la vez, al que uno le confiaría la vida sin dudarlo un instante.

Finalmente, a Cande. Por hacerle honor a tu nombre, por ese fuego interior que brilla a través de tu ojos hermosos y es el faro que ilumina mis días y me guía a buen puerto. Porque el recuerdo de tu sonrisa me transporta a vos instantáneamente, aunque nos separen millones de años luz… Sos la compañera perfecta para la aventura que nos espera adelante. ¡Te amo!

A todos, a los que mencioné y a los que olvidé mencionar …¡Muchas gracias!

Capítulo 1

Introducción

> This is a story about magic and where it goes and perhaps more importantly where it comes from and why, although it doesn't pretend to answer ANY of these questions
>
> Terry Pratchet,
> *Equal Rites*

Un problema recurrente en las Ciencias de la Computación es la necesidad de buscar un balance entre el diseño de programas simples que implementen correctamente una especificación y el diseño de programas monolíticos eficientes, que usualmente resultan dificiles de comprender y de mantener y por lo tanto propensos a contener errores.

Uno de los beneficios más promocionados del paradigma de programación funcional es cómo este permite construir fácilmente programas grandes y complejos mediante la combinación de otros más pequeños y simples. [27]. Este enfoque composicional favorece el diseño de algoritmos fáciles de comprender y de modificar.

Sin embargo, esta estrategia *modular* puede dar como resultado programas mucho más ineficientes que sus contrapartes monolíticas, dado que una expresión composicional supone la creación de una estructura de datos cuyo único propósito es ser consumida inmediatamente. Con el fin de resolver este problema se han desarrollado técnicas formales que permiten calcular correctamente definiciones eficientes en las que no se generan las estructuras intermedias.

Entre estas técnicas se destacan las que permiten *fusionar* funciones o programas. Se conoce como *fusión de programas* al proceso de reemplazar la composición modular de programas por uno monolítico más eficiente. Entre las diferentes técnicas de fusión, nos interesa la conocida como *deforestación* [53]. Ésta consiste en la eliminación de las estructuras de datos intermedias que se generan por la composición de funciones. Un aspecto muy intersante de esta técnica es que involucra reglas *automatizables* que pueden integrarse al compilador de un lenguaje, resultando en optimizaciones transparentes al diseño de un programa.

Otra herramienta común en un lenguaje funcional es el uso de *operadores o esquemas* de recursión. Estos son funciones de alto orden que capturan patrones de recursión comunes a las estructuras que manipulan e.g. *map, fold* [8, 29] y que se derivan de la interpretación categórica de los tipos de datos recursivos. El uso de estos operadores para estructurar programas funcionales resulta beneficioso porque permite aprovechar las propiedades algebraicas asociadas a éstos para derivar leyes genéricas aplicables a la *deforestacion* de programas.

Alternativamente se pueden estructurar los programas funcionales de acuerdo a los *efectos* que estos producen. Una forma de hacer esto es usando *mónadas*. Moggi [44, 45] utilizó *mónadas* como una abstracción para modelar la semántica de diferentes *efectos computacionales* e.g. *entrada/salida, asignaciones de variables, estado*. Luego, Wadler [54, 55] las utilizó como una herramienta para estructurar programas funcionales con efectos computacionales asociados frecuentemente a lenguajes imperativos en lenguajes funcionales puros. Esto ha motivado a diferentes autores a investigar diferentes esquemas de recursión en contextos *mónadicos* [16, 46, 47] y al estudio de técnicas de deforestación que involucran *efectos monádicos* [18, 19, 33, 40].

Recientemente, McBride y Paterson [41] presentaron una nueva estructura para modelar *efectos computacionales* llamada *functores aplicativos*. Éstos resultan una generalización de las *mónadas* que favorecen un estilo *aplicativo* de programación. Gibbons y Oliveira [22] presentaron propiedades calculacionales del operador *traverse*, que realiza *recorridos o traversals* de estructuras de datos con acciones con *efectos aplicativos*.

Este trabajo propone una caracterización genérica de la recursión estructural con *efectos aplicativos*, es decir una caracterización de funciones recursivas estructurales que consumen uniformemente valores de ciertos tipos de datos y producen valores embebidos en un efecto aplicativo. A partir del estudio de patrones comunes que presentan estas funciones, derivamos un operador genérico, al que llamamos *ifold*, que captura la esencia de la recursión estructural aplicativa.

Además, presentaremos algunas propiedades *calculacionales* de este operador, que permiten fusionar algoritmos aplicativos modulares eliminando la generación de estructuras intermedias y mostramos algunos ejemplos de la aplicación de estas reglas. Finalmente, mostraremos como este operador permite separar el cálculo de valores *puros* de la generación de *efectos aplicativos* y como éste puede utilizarse para diseñar modularmente algoritmos con *efectos aplicativos* incrementales, agregando funcionalidad en cada etapa de forma sencilla y flexible.

El resto de este trabajo se organiza como sigue:

- El Capítulo 2 introduce los *Functores Aplicativos*, presentando una reseña de los conceptos y resultados presentados en [41]. También, se presentan los resultados calculacionales para los *traversals o recorridos* sobre *Functores Traversables* desarrollados en [22].

- El Capítulo 3 presenta una reseña del modelo categórico de tipo de datos para *data type generic programming* y el modelo de semántica por álgebras iniciales, presentando el operador *fold* genérico y sus propiedades algebraicas, que constituyen los fundamentos matemáticos formales para el trabajo realizado.

- El Capítulo 4 presenta una caracterización del patrón de *recursión estructural aplicativa*, mediante la derivación del operador genérico *ifold* junto con sus propiedades para calcular programas con efectos aplicativos. Presentamos también algunos ejemplos de la aplicación de estas propiedades.

- El Capítulo 5 analiza el uso del operador *ifold* para diseñar algoritmos aplicativos con efectos incrementales.

- El Capítulo 6 presenta las conlusiones de este trabajo y los posibles trabajos futuros.

- El Apéndice A proporciona las demostraciones de las propiedades del operador *ifold* presentadas en el Capítulo 4.

Capítulo 2

Functores Aplicativos

En este capítulo describiremos el concepto de *Functor Aplicativo*, introducido por McBride y Paterson en [41], una estructura que permite caracterizar en forma abstracta un estilo de programación *aplicativa* en programas funcionales con *efectos computacionales*. Presentaremos también las diferentes clases de functores aplicativos y las herramientas para combinarlos, que los convierten en una abstracción práctica y flexible para diseñar *efectos computacionales* incrementalmente.

Además, presentaremos a los *Functores Traversables*, una clase de tipos de datos relacionadas con los *functores aplicativos* que soportan una noción de *traversal o recorrido* con efectos, mediante la función *traverse*. Repasaremos también, algunas de las propiedades calculacionales de *traverse* presentadas por Gibbons y Oliveira [22] que nos resultarán de gran utilidad en nuestro propósito de estudiar la recursión aplicativa.

Los *functores aplicativos o idioms* intuitivamente modelan la aplicación de funciones puras a valores embebidos en un contexto con efectos, *o idiom*. Antes de dar su definición concreta, mostraremos un par de ejemplos:

Ejemplo 2.1 (Seguimiento en un Mapa)**.** Supongamos un sistema de control que mantiene una lista de objetivos de interés, representados por un tipo abstracto de datos *Point*. El seguimiento de cada objeto se realiza mediante una función $updatePos :: Point \to Maybe\ Point$ que devuelve la nueva posición del objeto o *Nothing*, en el caso de que este se encuentre fuera del alcance del sistema. Modelamos la actualización del sistema con la siguiente función:

$$
\begin{array}{lll}
updateSys & :: [Point] \to Maybe\ [Point] & \\
updateSys\ [\,] & = Just\ [\,] & \\
updateSys\ (x : xs) & = \textbf{case}\ (updatePos\ x, updateSys\ xs)\ \textbf{of} & \\
 & \quad (Just\ p, Just\ ps) \to Just\ (p : ps) & \\
 & \quad \_ \qquad\qquad\qquad\quad \to Nothing &
\end{array}
$$

La segunda ecuación en la definición de *updateSys* realiza análsis por caso del resultado de actualizar la posición del objeto y de la llamada recursiva y recombina los valores si no hubo algún error. Podemos reescribir la definición de la función en un estilo más *aplicativo*:

$$
\begin{array}{ll}
updateSys & :: [Point] \to Maybe\ [Point] \\
updateSys\ [\,] & = Just\ [\,] \\
updateSys\ (x : xs) & = Just\ (:)\ `mbApp`\ updatePos\ x\ `mbApp`\ updateSys\ xs \\
\quad \textbf{where}\ mbApp & :: Maybe\ (s \to t) \to Maybe\ s \to Maybe\ t \\
\qquad mbApp\ (Just\ f)\ (Just\ x) & = Just\ (f\ x) \\
\qquad mbApp\ \_\ \ \_ & = Nothing
\end{array}
$$

Donde *mbApp* extiende la aplicación de funciones puras dentro de *Maybe*, obteniendo una definición de *updateSys* en un estilo *aplicativo*.

Ejemplo 2.2 (Transposición de Matrices [41])**.** Supongamos una representación de matrices usando listas de listas, e..g $[[Int]]$. Sean las siguientes funciones:

$$
\begin{array}{l}
repeat \quad :: a \to [a] \\
repeat\ x = x : repeat\ x
\end{array}
$$

$$
\begin{array}{lll}
zapp & & :: [s \to t] \to [s] \to [t] \\
zapp\ (f : fs)\ (x : xs) & & = f\ x : zapp\ fs\ xs \\
zapp\ \_ \quad \_ & & = []
\end{array}
$$

Donde *repeat* genera una lista infinita de valores puros y *zapp* aplica una lista de funciones a una lista de valores (de idéntica longitud). Utilizando estos dos operadores se puede definir la función que calcula una matriz transpuesta como:

$$
\begin{array}{ll}
transpose & :: [[Int]] \to [[Int]] \\
transpose\ [] & = repeat\ [] \\
transpose\ (xs : xss) & = repeat\ (:)\ `zapp`\ xs\ `zapp`\ transposer\ xss
\end{array}
$$

Analizaremos ciertas características comunes de ambos ejemplos. Primero, reconocemos la acción de una operación que inserta un valor puro en el *idiom* propio de cada ejemplo: *Just* y *repeat*. Si abstraemos el constructor de tipos, los tipos de estas funciones tienen la siguiente forma $a \to c\ a$ donde c representaría respectivamente a *Maybe* y $[]$. Además, reconocemos una segunda operación que modela la aplicación de funciones dentro del contexto: *mbApp* y *zapp*. Con un razonamiento similar al anterior, los tipos de estas resultan $c\ (s \to t) \to c\ s \to c\ t$. Formalizaremos estas operaciones, introduciendo el concepto de *functor aplicativo*.

2.1. Functores Aplicativos

Un *Functor Aplicativo* es un constructor de tipos $f :: * \to *$ con dos operaciones asociadas, representadas en Haskell mediante la siguiente clase:

$$
\begin{array}{l}
\textbf{class}\ (Functor\ f) \Rightarrow Applicative\ f\ \textbf{where} \\
\quad pure :: a \to f\ a \\
\quad (\circledast) :: f\ (s \to t) \to f\ s \to f\ t
\end{array}
$$

Intuitivamente, *pure* introduce valores *puros* dentro del efecto representado mediante f, y $\circledast$ modela la aplicación de funciones dentro del efecto. Se requiere, adicionalmente, que las instancias de éstas verifiquen las siguientes ecuaciones:

$$
\begin{array}{lr}
pure\ \mathsf{id} \circledast\ u = \ u & (\textbf{App.Identity}) \\
pure\ f\ \circledast\ pure\ x = pure\ (f x) & (\textbf{App.Homo}) \\
pure\ (\circ)\ \circledast\ u \circledast\ v \circledast\ w\ =\ u \circledast\ (v \circledast\ w) & (\textbf{App.}\circ) \\
u\ \circledast\ pure\ x\ =\ pure(\lambda f \to (f\ x)) \circledast\ u & (\textbf{App.Intr})
\end{array}
$$

Cualquier instancia de un functor aplicativo resulta un *functor*. En Haskell, un *functor* es un constructor de tipos $f :: * \to *$, que puede instanciarse en la clase *Functor* mediante la definición de su acción sobre flechas, *fmap*:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Functor}\ f\ \mathbf{where} \\
&\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b
\end{aligned}
$$

Profundizaremos sobre este concepto en el Capítulo 3. La función *fmap* para un functor aplicativo debe verifica la siguiente ecuación:

$$fmap\ f\ x\ = pure f \circledast\ x \qquad \textbf{(App.Map)}$$

McBride y Paterson muestran que los functores aplicativos generalizan a las *mónadas*, dado que toda *mónada* define un functor aplicativo y muestran que existen dos clases diferentes de functores aplicativos *no monádicos*. A continuación, presentamos en detalle estas tres clases de functores aplicativos, junto con algunos ejemplos que usaremos en el desarrollo de este trabajo.

2.1.1. Functores Aplicativos Monádicos

Como hemos mencionado anteriormente, el concepto de *mónada* tiene su origen en la Teoría de Categorías,[4, 38] y su uso en el ámbito de semántica de los lenguajes de programación fue propuesto por Moggi [44, 45] para modelar *efectos computacionales* y luego fue utilizado por Wadler [54, 55] y otros [6, 48] como una herramienta para estructurar programas funcionales con efectos en lenguajes *funcionales puros.*

En Haskell, una mónada es un tipo de datos paramétrico junto a las siguientes operaciones:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Monad}\ m\ \mathbf{where} \\
&\mathit{return} :: a \to m\ a \\
&(\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}
$$

Donde *return* tiene el mismo comportamiento que *pure*: insertar un valor puro en el efecto, y $(\gg\!=)$, pronunciada *bind*, secuencializa dos computaciones *monádicas*. *return* y $(\gg\!=)$ deben verificar además ciertas propiedades, las *leyes monádicas.*

Para cualquier *mónada* m podemos definir un operador $ap :: m\ (s \to t) \to m\ s \to mt$ que toma una función embebida en un efecto monádico y un valor monádico y devuelve el resultado de aplicar la función al valor dentro del efecto, luego de *secuencializar* los efectos:

$$
\begin{aligned}
&\mathit{ap} && :: (\mathit{Monad}\ m) \Rightarrow m\ (s \to t) \to m\ s \to m\ t \\
&\mathit{ap}\ \mathit{mf}\ \mathit{mx} && = \mathit{mf} \gg\!= \lambda f \to \mathit{mx} \gg\!= \lambda x \to \mathit{return}\ (f\ x)
\end{aligned}
$$

Esta función tiene el comportamiento esperado para el operador $\circledast$ para un functor aplicativo. Podemos entonces, definir un functor aplicativo para toda mónada m de la siguiente forma:

$$
\begin{aligned}
&\mathbf{instance}\ (\mathit{Functor}\ m, \mathit{Monad}\ m) \Rightarrow \mathit{Applicative}\ m\ \mathbf{where} \\
&\quad \mathit{pure} = \mathit{return} \\
&\quad (\circledast)\ \ = \mathit{ap}
\end{aligned}
$$

Dejamos al lector la demostración de que se verifican las ecuaciones que rigen a los functores aplicativos definidas anteriormente. Los *fu nctores aplicativos monádicos* secuencializan el efecto modelado por la *mónada* pero, la estructura de la computación es fija, a diferencia de la *interfaz monádica* donde, por el tipo de $\gg\!=$, se permite que la segunda operación monádica dependa del valor puro calculado dentro de la primer computación monádica.

A continuación, mostramos algunos *functores aplicativos monádicos* que utilizaremos en los ejemplos de esta tesina:

Ejemplo 2.3 (Functor Identidad)**.** El functor aplicativo monádico *Identidad* es simplemente un *wrapper* para un valor puro. Se define mediante el siguiente constructor de tipos:

$$\textbf{data}\ \mathit{Identity}\ a = \mathit{Id}\ \{\mathit{unId} :: a\}$$

La función de los operadores de interfaz aplicativa es simplemente *empaquetar y desempaquetar* los valores puros:

$$\begin{array}{l} \textbf{instance}\ \mathit{Applicative}\ \mathit{Identity}\ \textbf{where} \\ \quad \begin{array}{ll} \mathit{pure} & = \mathit{Id} \\ (\mathit{Id}\ f) \circledast (\mathit{Id}\ x) & = \mathit{Id}\ (f\ x) \end{array} \end{array}$$

Si bien *Identity* no introduce ningún efecto computacional, resultará de gran utilidad para almacenar valores puros cuando se combinan diferentes efectos.

Ejemplo 2.4 (Mónada *Maybe*)**.** El functor aplicativo *monádico Maybe* modela las *fallas* como un efecto computacional. Se define mediante el siguiente constructor de tipos:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

Es decir, un valor *Maybe a* puede ser un valor de tipo *a* o una *falla*. La interfaz aplicativa resulta:

$$\begin{array}{l} \textbf{instance}\ \mathit{Applicative}\ \mathit{Maybe}\ \textbf{where} \\ \quad \mathit{pure} = \mathit{Just} \\ \quad \begin{array}{ll} (\mathit{Just}\ f) \circledast (\mathit{Just}\ x) & = \mathit{Just}\ (f\ x) \\ (\mathit{Just}\ f) \circledast \mathit{Nothing} & = \mathit{Nothing} \\ \mathit{Nothing} \circledast \_ & = \mathit{Nothing} \end{array} \end{array}$$

Donde *pure* simplemente introduce un valor puro en el efecto y $\circledast$ combina los valores puros dentro del functor aplicativo, propagando las *fallas*. Podemos reescribir el Ejemplo 2.1, para que refleje esta instancia:

$$\begin{array}{ll} \mathit{updateSys} & :: [\mathit{Point}] \to \mathit{Maybe}\ [\mathit{Point}] \\ \mathit{updateSys}\ [\,] & = \mathit{pure}\ [\,] \\ \mathit{updateSys}\ (x : xs) & = \mathit{pure}\ (:) \circledast \mathit{updatePos}\ x \circledast \mathit{updateSys}\ xs \end{array}$$

Ejemplo 2.5 (Mónada [ ])**.** El *functor aplicativo* monádico de listas modela el efecto computacional de *no determinismo o posibilidad no determinística*. La interfaz aplicativa resulta:

$$\begin{array}{l} \textbf{instance}\ \mathit{Applicative}\ [\,]\ \textbf{where} \\ \quad \begin{array}{ll} \mathit{pure} & = (:[\,]) \\ \mathit{fs} \circledast \mathit{xs} & = [(f\ x) \mid f \leftarrow \mathit{fs}, x \leftarrow \mathit{xs}] \end{array} \end{array}$$

pure simplemente introduce un valor puro, i.e. determinístico, en el efecto. Para definir $\circledast$, hemos utilizado la notación de *comprensión de listas* [8]. Esta definición debe interpretarse como sigue: dada una lista de posibles funciones *fs*, y una lista de posibles valores (del tipo apropiado) *xs*, $\circledast$ genera la lista de resultados de todas las posibles combinaciones entre los elementos de *fs* y *xs*. Dada la semántica de las compresiones de listas, si *fs* o *xs* es la lista vacía [ ], modelando un efecto de *computación sin alternativa válida*, el resultado de la combinación será [ ].

2.1.2. Functores Aplicativos *Neperianos o Vectorizados*

Una clase de functores aplicativos no monádicos es la que constituyen los functores aplicativos *vectorizados o neperianos*, que se caracterizan por modelar un efecto de *transposición de valores*. El ejemplo clásico de esta clase lo consituyen las *ziplists*:

Ejemplo 2.6 (Ziplists)**.** En el Ejemplo 2.2 hemos descripto a una matriz por una lista de listas, utilizando dos operaciones peculiares: *repeat* y *zapp*. Estas definen una interfaz aplicativa para las listas, diferente a la *monádica* presentada en el Ejemplo 2.5. Llamaremos *ziplists* a estas listas vectorizadas. La instancia en la clase *Applicative* resulta:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Applicative}\ [\,]\ \textbf{where} \\
\quad \mathit{pure} = \mathit{repeat} \\
\quad (\circledast)\ \ = \mathit{zapp}
\end{array}
$$

Dejamos al lector verificar que ésta cumple con las ecuaciones asociadas a un functor aplicativo. Podemos reescribir *transpose* reflejando las definiciones de la interfaz aplicativa, obteniendose la siguiente definición:

$$
\begin{array}{ll}
\mathit{transpose} & :: [[a]] \to [[a]] \\
\mathit{transpose}\ [\,] & = \mathit{pure}\ [\,] \\
\mathit{transpose}\ (\mathit{xs} : \mathit{xss}) & = \mathit{pure}\ (:) \circledast \mathit{xs} \circledast \mathit{transpose}\ \mathit{xss}
\end{array}
$$

Esta construcción podría ser extendida para otros tipos: pares, vectores de longitud fija, árboles binarios infinitos, etc, obteniéndose un *efecto de transposición* especializado a la *estructura* particular de estos tipos de datos.

2.1.3. Acumuladores de monoides

Una tercer familia de ejemplos interesantes de *functores aplicativos* presentada en [41] es la que constituyen los *acumuladores de monoides o functores aplicativos monoidales*. Un *monoide* se implementa en Haskell mediante un tipo de datos junto con un *elemento neutro* y una *suma*, definidos como una instancia de la siguiente clase:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monoid}\ o\ \textbf{where} \\
\quad \varnothing \quad :: o \\
\quad (\oplus) :: o \to o \to o
\end{array}
$$

Naturalmente se requiere además que $\varnothing$ y $\oplus$ definidas en esta instancia, satisfagan las ecuaciones de un *monoide*:

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z \qquad (\oplus.\textbf{Assoc})$$
$$x \oplus \varnothing = x \qquad (\varnothing.\textbf{NeutR})$$
$$\varnothing \oplus x = x \qquad (\varnothing.\textbf{NeutL})$$

Todo monoide induce un functor aplicativo *acumulador*, definido mediante el siguiente constructor:

$$\textbf{newtype}\ \mathit{Acc}\ o\ x = \mathit{Acc}\ \{\mathit{acc} :: o\}$$

La interfaz aplicativa de *Acc o* queda determinada por la siguiente instancia:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monoid}\ o) \Rightarrow \mathit{Applicative}\ (\mathit{Acc}\ o)\ \textbf{where} \\
\quad \mathit{pure}\ \_ \qquad\qquad\quad = \mathit{Acc}\ \varnothing \\
\quad (\mathit{Acc}\ p) \circledast (\mathit{Acc}\ q) = \mathit{Acc}\ (p \oplus q)
\end{array}
$$

Donde *pure* devuelve el cero del monoide, $\varnothing$, y $\circledast$ reduce los valores acumulados mediante $\oplus$. El efecto modelado por esta familia de functores aplicativos es el de *acumular* los valores del monoide, de ahí surge su nombre. Ilustramos su aplicación con un ejemplo:

Ejemplo 2.7. Consideramos el monoide de las listas , con la lista vacía ($[\,]$) como *neutro* y la concatenación ($+\!\!+$) como *suma*:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monoid}\ [a]\ \textbf{where} \\
\quad \varnothing \quad = [\,] \\
\quad (\oplus) = (+\!\!+)
\end{array}
$$

Utilizaremos este monoide para acumular *aplicativamente* los elementos de una lista que cumplen con cierta propiedad. Definimos entonces la función *check* como sigue:

$$
\begin{array}{ll}
\mathit{check} & :: (\mathit{Eq}\ a) \Rightarrow (a \to \mathit{Bool}) \to [a] \to \mathit{Acc}\ [a]\ [a] \\
\mathit{check}\ p\ [\,] & = \mathit{pure}\ [\,] \\
\mathit{check}\ p\ (x : \mathit{xs}) & = \mathit{pure}\ (\textbf{if}\ (p\ x)\ \textbf{then}\ (\mathit{Acc}\ [x])\ \textbf{else}\ (\mathit{Acc}\ \varnothing)) \circledast \mathit{check}\ \mathit{xs}
\end{array}
$$

Por ejemplo, dada una lista de *strings* tomamos aquellas de longitud mayor a 42

$$
\begin{array}{l}
\mathit{checkStr} :: [\mathit{String}] \to \mathit{Acc}\ [\mathit{String}]\ [\mathit{String}] \\
\mathit{checkStr} = \mathit{check}\ ((>42) \circ \mathit{length})
\end{array}
$$

2.2. Combinación de Functores Aplicativos

Los functores aplicativos proveen diferentes formas de combinarse, permitiendo construir incrementalmente efectos más complejos a partir de efectos más simples, favoreciendo un diseño de efectos incrementales donde se puede agregar diferentes efectos aislando las características atómicas de cada uno. En el Capítulo 5 mostraremos un ejemplo de diseño incremental donde haremos uso intensivo de estos combinadores.

Composición A diferencia de las *mónadas*, los functores aplicativos resultan cerrados bajo la composición i.e. dados dos functores aplicativos C y D el functor $C \bullet D$ también es un functor aplicativo. En Haskell, esto se refleja mediante el *constructor de tipos* $\bullet :: (* \to *) \to (* \to *) \to (* \to *)$:

$$
\begin{array}{l}
\textbf{data}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow (f \bullet g)\ a = \mathit{Comp}\ \{\mathit{unComp} :: f\ (g\ a)\} \\
\textbf{instance}\ (\mathit{Applicative}\ f, \mathit{Applicative}\ g) \Rightarrow \mathit{Applicative}\ (f \bullet g)\ \textbf{where} \\
\quad \mathit{pure} \qquad\qquad\qquad\quad = \mathit{Comp} \circ \mathit{pure} \circ \mathit{pure} \\
\quad (\mathit{Comp}\ \mathit{fs}) \circledast (\mathit{Comp}\ \mathit{xs}) = \mathit{Comp}\ (\mathit{pure}\ (\circledast) \circledast \mathit{fs} \circledast \mathit{xs})
\end{array}
$$

El hecho de ser *cerrado bajo la composición* hace de ésta una herramienta *simple y práctica* para combinar efectos, permitiendo analizar granularmente las características de cada *efecto computacional*, favoreciendo un diseño *incremental* a nivel de los efectos. En el caso de las *mónadas*, no siempre resulta que la composición de los functores subyacentes resulte una mónada, por lo que se recurren a otras formas de combinar efectos como son los *transformadores de mónadas* [31, 35].

Producto de Functores Aplicativos Otra forma de combinar functores aplicativos es mediante el producto de dos de éllos:

$$
\begin{array}{l}
\textbf{data}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow (f \times g)\ a = \mathit{Prod}\ \{\pi_1 :: f\ a, \pi_2 :: g\ a\} \\
\textbf{instance}\ (\mathit{Applicative}\ f, \mathit{Applicative}\ g) \Rightarrow \mathit{Applicative}\ (f \times g)\ \textbf{where} \\
\quad \mathit{pure}\ x \qquad = \mathit{Prod}\ (\mathit{pure}\ x)\ (\mathit{pure}\ x) \\
\quad \mathit{mf} \circledast \mathit{mx} = \mathit{Prod}\ (\pi_1\ \mathit{mf} \circledast \pi_1\ \mathit{mx})\ (\pi_2\ \mathit{mf} \circledast \pi_2\ \mathit{mx})
\end{array}
$$

Combinadores de Acciones Aplicativas Las dos formas de combinar functores aplicativos arriba descriptas proveen también combinadores de *acciones aplicativas o cómputos aplicativos*; estos permiten definir modularmente las acciones aplicativas para los efectos compuestos.

$$
\begin{array}{ll}
(\odot) & :: (\mathit{Functor}\ m, \mathit{Functor}\ n) \Rightarrow (b \to n\ c) \to (a \to m\ b) \to a \to (m \bullet n)\ c \\
f \odot g & = \mathit{Comp} \circ \mathit{fmap}\ f \circ g \\
(\otimes) & :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow (a \to f\ b) \to (a \to g\ b) \to a \to (f \times g)\ b \\
(f \otimes g)\ a & = \mathit{Prod}\ (f\ a)\ (g\ a)
\end{array}
$$

$\odot$ y $\otimes$ reciben el nombre de *composición secuencial* y, respectivamente, *paralela* de acciones aplicativas.

2.3. Functores Traversables

Los functores *traversables* tienen una relación estrecha con los functores aplicativos. Un *functor traversable* es un tipo de datos paramétrico que soporta *traversals o recorridos* con efectos, donde se puede entretejer una acción aplicativa paramétrica a travez del tipo de datos, obteniendo como resultado la estructura dentro del efecto aplicativo. Cuando la acción aplicativa es la identidad, el recorrido es simplemente una distribución de la estructura de datos con respecto al functor aplicativo. Representamos a los *functores traversables* mediante la siguiente clase:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Traversable}\ f\ \textbf{where} \\
\quad \mathit{traverse} :: (\mathit{Applicative}\ m) \Rightarrow (a \to m\ b) \to f\ a \quad \to m\ (f\ b) \\
\quad \mathit{dist} \quad\ :: (\mathit{Applicative}\ m) \Rightarrow \qquad\qquad\quad f\ (m\ a) \to m\ (f\ a) \\
\quad \mathit{dist} \quad\ = \mathit{traverse}\ \mathit{id}
\end{array}
$$

Una condición suficiente para que pueda definirse la función *traverse* es que el tipo de datos sea *regular*. Consideramos, a modo de ejemplo las instancias para algunos tipos de datos paramétricos inductivos, como las listas y los árboles binarios:

Ejemplo 2.8 (Listas)**.** Para listas, la instancia de *traverse* resulta:

$$
\begin{array}{ll}
\mathit{traverse} & :: (\mathit{Applicative}\ c) \Rightarrow (a \to c\ b) \to [a] \to c\ [b] \\
\mathit{traverse}\ f\ [\,] & = \mathit{pure}\ [\,] \\
\mathit{traverse}\ f\ (x : \mathit{xs}) & = \mathit{pure}\ (:) \circledast f\ x \circledast \mathit{traverse}\ f\ \mathit{xs}
\end{array}
$$

De esta definición, podemos observar que la transposición de matrices expresadas con *ziplists* del Ejemplo 2.2, *transposer*, es simplemente una instancia de la distribución *dist* para listas, especializada para el functor aplicativo:

$$
\begin{array}{l}
\mathit{transposer} :: [[a]] \to [[a]] \\
\mathit{transposer} = \mathit{traverse}\ \mathit{id}
\end{array}
$$

En el caso de la función *updateSys* del Ejemplo 2.1, esta resulta un *traverse* de listas, con la *acción aplicativa* que actualiza la posición de los objetos, *updatePos*.

$$\begin{array}{l} \mathit{updateSys} :: [\mathit{Point}] \to \mathit{Maybe}\ [\mathit{Point}] \\ \mathit{updateSys} = \mathit{traverse}\ \mathit{updatePos} \end{array}$$

Ejemplo 2.9 (Árboles Binarios)**.** Consideramos árboles binarios con datos en los nodos y hojas vacías:

$$\mathbf{data}\ \mathit{Bin}\ a = \mathit{Leaf} \mid \mathit{Node}\ (\mathit{Bin}\ a)\ a\ (\mathit{Bin}\ a)$$

La instancia de *traverse* para éstos resulta:

$$\begin{array}{ll} \mathit{traverse} & :: (\mathit{Applicative}\ c) \Rightarrow (a \to c\ b) \to \mathit{Bin}\ a \to c\ (\mathit{Bin}\ b) \\ \mathit{traverse}\ \iota\ \mathit{Leaf} & = \mathit{pure}\ \mathit{Leaf} \\ \mathit{traverse}\ \iota\ (\mathit{Node}\ \mathit{tl}\ x\ \mathit{tr}) & = \mathit{pure}\ \mathit{Node} \circledast \mathit{traverse}\ \iota\ \mathit{tl} \circledast \iota\ x \circledast \mathit{traverse}\ \iota\ \mathit{tr} \end{array}$$

Si consideramos el functor aplicativo *acumulador de monoides*, se observa que toda *acumulación* es en realidad un *recorrido* mediante *traverse*. Esto nos permite definir funciones *accumulate* y *reduce* para cualquier *Functor Traversable*:

$$\begin{array}{l} \mathit{accumulate} :: (\mathit{Traversable}\ f, \mathit{Monoid}\ o) \Rightarrow (x \to o) \to f\ x \to o \\ \mathit{accumulate}\ m = \mathit{acc} \circ \mathit{traverse}\ (\mathit{Acc} \circ m) \\ \mathit{reduce} :: (\mathit{Traversable}\ f, \mathit{Monoid}\ o) \Rightarrow f\ o \to o \\ \mathit{reduce} = \mathit{accumulate}\ \mathit{id} \end{array}$$

Ejemplo 2.10. Considerando las definiciones anteriores, podemos redefinir la función *checkStr* del Ejemplo 2.7, obteniendo directamente la lista de cadenas cuya longitud sea mayor a 42

$$\begin{array}{l} \mathit{checkStr} :: [\mathit{String}] \to [\mathit{String}] \\ \mathit{checkStr} = \mathit{accumulate}\ ((>42) \circ \mathit{length}) \end{array}$$

2.3.1. Propiedades de Traverse

Gibbons y Oliveira [22] presentan propiedades calculacionales de *dist* y *traverse*. Enunciamos algunas de ellas, que nos resultarán de utilidad en los capítulos subsiguientes para probar propiedades del operador *ifold*. Destacamos entre ellas los *free-theorems* [52] asociados a sus tipos polimórficos:

$$\mathit{dist} \circ \mathit{fmap}\ (\mathit{fmap}\ k) = \mathit{fmap}\ (\mathit{fmap}\ k) \circ \mathit{dist} \qquad (\textbf{Dist.MapMap})$$

$$\mathit{traverse}\ (g \circ f) = \mathit{traverse}\ g \circ \mathit{fmap}\ f \qquad (\textbf{Traverse.Map})$$

$$\mathit{traverse}\ (\mathit{fmap}\ k \circ f) = \mathit{fmap}\ (\mathit{fmap}\ k) \circ \mathit{traverse}\ f \qquad (\textbf{MapMap.Traverse})$$

$$\mathit{traverse}\ \mathit{pure} = \mathit{pure} \qquad (\textbf{Traverse.pure})$$

También, se enuncian las siguientes propiedades para *traverse* cuando se utilizan los combinadores de acciones aplicativas de la sección anterior:

$$
\begin{aligned}
&\mathit{dist} \circ \mathit{fmap}\ \mathit{Id} = \mathit{fmap}\ \mathit{Id} \circ \mathit{dist} && (\textbf{Dist}.\mathit{Id})\\
&\mathit{dist} \circ \mathit{fmap}\ \mathit{Comp} = \mathit{Comp} \circ \mathit{fmap}\ \mathit{dist} \circ \mathit{dist} && (\textbf{Dist}.\mathit{Comp})\\
&\mathit{traverse}\ (\mathit{Id} \circ f) = \mathit{Id} \circ \mathit{fmap}\ f && (\textbf{Traverse}.\mathit{Id})\\
&\mathit{traverse}\ (f \odot g) = \mathit{traverse}\ f \odot \mathit{traverse}\ g && (\textbf{Traverse}.\odot)\\
&\mathit{traverse}\ (f \otimes g) = \mathit{traverse}\ f \otimes \mathit{traverse}\ g && (\textbf{Traverse}.\otimes)
\end{aligned}
$$

2.4. Más sobre Functores Aplicativos

Los siguientes trabajos muestran la aplicación de *functores aplicativos* en diferentes ámbitos no relacionados con nuestro propósito de estudiar la *recursión aplicativa*. Esta lista no debe suponerse exhaustiva.

Relación entre Functores Aplicativos y *Arrows* McBride y Paterson [41] sostienen incialmente que los functores aplicativos resultan una generalización de *mónadas* menos abstracta (y por lo tanto, *más fuerte*) que los *Arrows* [28]. Sin embargo, Lindley *et allis* [37, 36] desarrollan un metalenguaje llamado *Arrows Calculus* para estudiar la relación entre estas tres abstracciones, probando que en realidad los Arrows resultan más *fuertes o específicos* que los functores aplicativos.

Combinadores de Parsers Swiestra [50] presenta ciertos *combinadores de parsers* que no pueden definirse usando *mónadas* y muestra también como, aún en el caso de trabajar con combinadores de parsers monádicos, las operaciones del functor aplicativo permiten describir combinadores de parsers flexibles y versátiles.

***Formlets* en Links** Cooper *et allis* [11, 12] presentan *Formlets* implementadas en Links [10], un lenguaje funcional *tipado y estricto* pensado específicamente para diseñar *aplicaciones Web*. Los *Formlets* son *templates* composicionales que cuando son instanciados devuelven por un lado el código fuente de un *formulario HTML* y una función llamada *colector* que transforman datos crudos procedentes de un formulario en datos estruturados e.g. en XML. Los autores muestran que los Functores Aplicativos resultan la mejor abstracción para modelar la semántica de los *Formlets* e implementarlos en Links.

Idris Idris [9] es un lenguaje funcional con tipos dependientes experimental implementado en Haskell, fuertemente orientado a *efectos* y pensado para desarrollar sistemas que interaccionen con el *mundo exterior* i.e. con soporte para *I / O*, redes, concurrencia, manejo de archivos etc. La sintaxis de Idris soporta notación **do** monádica y functores aplicativos implementando *idiom brackets*, una notación introducida por McBride y Paterson para escribir computaciones con efectos aplicativos.

Capítulo 3

Modelo Categórico de Tipos de Datos

Este capítulo introduce una breve reseña del modelo categórico de tipos de datos para *datatype-generic programming* [2, 7, 15, 21] y del modelo de *semántica por álgebras iniciales* [26, 34]. Presentaremos algunos conceptos que son fundamentales para comprender el trabajo realizado, haciendo hincapié en cómo se reflejan dichos conceptos en la implementación.

Este modelo está basado en *Teoría de Categorías*, que ha resultado una herramienta fructífera para expresar modelos semánticos para lenguajes de programación y para razonar algebraicamente con programas funcionales [13, 14, 25, 39].

Suponemos que el lector conoce ciertas nociones básicas de *Teoría de Categorías*, que pueden consultarse en diversos textos introductorios destinados a las Ciencias de la Computación, e.g. [5, 49]. Para un estudio más profundo, la referencia tradicional es el conocido libro de Saunders MacLane [38].

3.1. Preliminares

En el *modelo categórico de tipos de datos* se trabaja sobre una categoría base donde los objetos de la categoría modelan los *tipos de datos* y las flechas modelan las funciones, i.e. una función $f :: A \to B$ se representa mediante una flecha $A \xrightarrow{f} B$.

Los constructores de tipos son representados mediante *functores*:

Definición 3.1 (Functor)**.** Dadas dos categorías $\mathcal{C}$ y $\mathcal{D}$, un *functor* $\mathsf{F} : \mathcal{C} \to \mathcal{D}$ está definido por un mapeo de objetos de $\mathcal{C}$ en objetos de $\mathcal{D}$ y un mapeo de flechas de $\mathcal{C}$ en flechas de $\mathcal{D}$, de modo que se verifiquen las siguientes propiedades:

$$A \xrightarrow{f} B \in \mathcal{A}\mathsf{rr}(\mathcal{C}) \implies \mathsf{F}\ A \xrightarrow{\mathsf{F}\ f} \mathsf{F}\ B \in \mathcal{A}\mathsf{rr}(\mathcal{D}) \qquad (\textbf{Functor.Cova})$$

$$\mathsf{F}\ f \circ \mathsf{F}\ g = \mathsf{F}\ (f \circ g) \qquad (\textbf{Functor.}\circ)$$

$$\mathsf{F}\ \mathsf{id}_A = \mathsf{id}_{\mathsf{F}A} \qquad (\textbf{Functor.}\mathsf{id})$$

En Haskell, implementamos los *functores* y su acción sobre flechas como una instancia de la clase *Functor*, mediante la definición de la función *fmap*, presentada en el Capítulo 2.

Las *transformaciones naturales*, en el contexto de programación genérica, modelan a las funciones polimórficas paramétricas [52].

Definición 3.2 (Transformación Natural)**.** Sean dos functores $\mathsf{F}, \mathsf{G} : \mathcal{C} \to \mathcal{D}$, una *transformación natural* $\tau : \mathsf{F} \Rightarrow \mathsf{G}$ es una familia de morfismos $\tau_X : \mathsf{F}\ X \to \mathsf{G}\ X \in \mathcal{A}\mathsf{rr}(\mathcal{D})$ tal que, para toda $f : A \to B \in \mathcal{A}\mathsf{rr}(\mathcal{C})$ el siguiente diagrama conmuta:

$$\begin{tikzcd}
\mathsf{F}\ A \arrow[r, "\tau_A"] \arrow[d, "\mathsf{F}\ f"'] & \mathsf{G}\ A \arrow[d, "\mathsf{G}\ f"] \\
\mathsf{F}\ B \arrow[r, "\tau_B"] & \mathsf{G}\ B
\end{tikzcd}$$

Los conceptos de F-álgebras y categorías de F-álgebras, son ubicuos en la literatura de *datatype-generic programming*, e.g. [2, 7, 21]:

Definición 3.3. Sea $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ un endofunctor. Un F-álgebra es una flecha $\phi : \mathsf{F}\ A \to A$ para algún objeto A en $\mathcal{C}$, el *carrier o conjunto soporte* del álgebra.

Definición 3.4 (Homomorfismo de F-álgebras)**.** Un **homomorfismo** entre dos F-álgebras ϕ, ψ es una flecha $f : A \to B$ en $\mathcal{C}$ tal que el siguiente diagrama conmuta:

$$\begin{tikzcd}
\mathsf{F}\ A \arrow[r, "\mathsf{F}\ f"] \arrow[d, "\phi"'] & \mathsf{F}\ B \arrow[d, "\psi"] \\
A \arrow[r, "f"'] & B
\end{tikzcd}$$

i.e. tal que $\psi \circ \mathsf{F}\ f = f \circ \phi$.

Definición 3.5 ($\mathcal{A}lg_{\mathsf{F}\,(\mathcal{C})}$)**.** Podemos construir la categoría de las F-álgebras sobre $\mathcal{C}$, $\mathcal{A}lg_{\mathsf{F}\,(\mathcal{C})}$, de la siguiente forma:

- $\mathcal{O}\mathsf{bj}(\mathcal{A}lg_{\mathsf{F}\,(\mathcal{C})})$, los objetos de $\mathcal{A}lg_{\mathsf{F}\,(\mathcal{C})}$, son las F-álgebras i.e. pares de la forma (A, ϕ) donde $\phi : \mathsf{F}\ A \to A \in \mathcal{A}\mathsf{rr}(\mathcal{C})$ y $A \in \mathcal{O}\mathsf{bj}(\mathcal{C})$.

- $\mathcal{A}\mathsf{rr}(\mathcal{A}lg_{\mathsf{F}\,(\mathcal{C})})$, las flechas de $\mathcal{A}lg_{\mathsf{F}\,(\mathcal{C})}$, son los homomorfismos de F-álgebras. Como éstos se preservan por composición en $\mathcal{C}$, la operación de composición será $\circ_{\mathcal{C}}$. Las identidades de $\mathcal{C}, \mathsf{id}_A : A \to A$, son trivialmente homorfismos de F-álgebras y constituyen las identidades de $\mathcal{A}lg_{\mathsf{F}\,(\mathcal{C})}$.

Los *álgebras iniciales* modelan los constructores de las estructuras de datos

Definición 3.6 (Álgebra Inicial)**.** Un *álgebra inicial* es un objeto inicial en $\mathcal{A}lg_{\mathsf{F}\,(\mathcal{C})}$ i.e. un F-álgebra tal que exista un único homomorfismo entre éste y cualquier otro F-álgebra.

Suponemos trabajar en una categoría donde existe el F-álgebra inicial para todo functor F. Para esto, alcanza con suponer que la categoría base $\mathcal{C}$ es *co-completa* y los functores son *endofunctores co-continuos*. No avanzaremos en este sentido, dado que en el resto del trabajo no se requiere analizar detalles específicos de la semántica del modelo. Éstos pueden consultarse en [3, 26, 34].

3.2. Semántica por F-álgebras iniciales

En el contexto de las semánticas por F-álgebras iniciales [26, 34], un *endofunctor* $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ modela la *signatura* recursiva de un tipo *inductivo* de datos. Los tipos recursivos resultan la menor solución de la ecuación de punto fijo: $X \cong \mathsf{F}\ X$ [1, 3]. Dicha solución está dada por un *álgebra inicial*: $(\ \mu\mathsf{F}\ , in_{\mathsf{F}}\ : \mathsf{F}\ \mu\mathsf{F} \to \mu\mathsf{F}\ )$.

En Haskell, codificamos a los tipos de datos como puntos fijos de sus signaturas mediante el constructor de tipos $\mu :: (* \to *) \to *$:

$$\mathbf{newtype}\ \mu\, f = \mathit{In}\ \{\mathit{unIn} :: f\ (\mu\, f)\}$$

Las signaturas de los tipos de datos, son functores que capturan la estructura recursiva de los mismos. Estos se implementan mediante constructores de tipos no recursivos, e.g. :

Ejemplo 3.1 (Naturales)**.** Dada la representación de los números naturales mediante el siguiente tipo de datos:

$$\mathbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Suc}\ \mathit{Nat}$$

La *signatura* de *Nat* es capturada por el functor N, implementado como sigue:

$$\begin{array}{l}
\mathbf{data}\ N\ x = Z \mid S\ x \\
\mathbf{instance}\ \mathit{Functor}\ N\ \mathbf{where} \\
\quad \mathit{fmap}\ f\ Z \qquad\ = Z \\
\quad \mathit{fmap}\ f\ (S\ n) = S\ (f\ n)
\end{array}$$

Para representar la *signatura* de tipos de datos *polimórficos paramétricos*, necesitamos extender el concepto de *functor* al de *bifunctor*:

Definición 3.7 (Bifunctor)**.** Un *bifunctor o functor binario* es un functor sobre dos argumentos, i.e. un *bifunctor* $\mathsf{F} : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ está definido por mapeos de objetos de $\mathcal{C}$ y $\mathcal{D}$ en objetos de $\mathcal{E}$ y respectivamente, mapeos de flechas de $\mathcal{C}$ y $\mathcal{D}$ en flechas de $\mathcal{E}$, de modo que se verifiquen conjuntamente las condiciones de functorialidad presentadas en la Definición 3.1, resumidas en las siguientes propiedades:

$$A \xrightarrow{f} B\ \in \mathcal{A}\mathsf{rr}(\mathcal{C}),\ X \xrightarrow{g} Y\ \in \mathcal{A}\mathsf{rr}(\mathcal{D}) \implies \mathsf{F}\ (A, X) \xrightarrow{\mathsf{F}\ (f,g)} \mathsf{F}\ (B, Y)\ \in \mathcal{A}\mathsf{rr}(\mathcal{E}) \qquad (\textbf{BiFun.Cova})$$

$$\mathsf{F}\ (f, k) \circ \mathsf{F}\ (g, j) = \mathsf{F}\ (f \circ g, k \circ j) \qquad (\textbf{BiFun.}\circ)$$

$$\mathsf{F}\ (\mathsf{id}_A, \mathsf{id}_X) = \mathsf{id}_{\mathsf{F}(A,X)} \qquad (\textbf{BiFun.}\mathsf{id})$$

En Haskell, representamos a un *bifunctor* mediante un constructor de tipos $s :: * \to * \to *$ y su acción sobre flechas, definida como una instancia de la clases *Bifunctor*:

$$\begin{array}{l}
\mathbf{class}\ \mathit{Bifunctor}\ f\ \mathbf{where} \\
\quad \mathit{bimap} :: (a \to b) \to (c \to d) \to f\ a\ c \to f\ b\ d
\end{array}$$

De la definición anterior, resulta inmediato que un *bifunctor* induce un *functor* cuando se fija el primer parámetro:

Definición 3.8 (Functor Paramétrico)**.** Sea un *bifunctor* $\mathsf{F} : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ y $X \in \mathcal{O}\mathsf{bj}(\mathcal{C})$, definimos al *functor* $\mathsf{F_A} : \mathcal{D} \to \mathcal{E}$ como:

$$\begin{aligned}\mathsf{F_A}\ X &= \mathsf{F}\ (A, X)\\ \mathsf{F_A}\ f &= \mathsf{F}\ (\mathsf{id}_A, f)\end{aligned}$$

En la implementación, esto se refleja mediante la siguiente *instancia* genérica:

$$\begin{array}{l}\mathbf{instance}\ (\mathit{Bifunctor}\ f) \Rightarrow \mathit{Functor}\ (f\ \ a)\ \mathbf{where}\\ \quad \mathit{fmap}\ \phi = \mathit{bimap}\ \mathit{id}\ \phi\end{array}$$

Ahora, estamos en condiciones de representar a los tipos de datos polimórficos paramétricos mediante los *bifunctores* que capturan su signatura. A modo de ejemplo presentamos los bifunctores que capturan las signaturas de las listas y otras estructuras que utilizaremos en este trabajo, mediante su implementación:

Ejemplo 3.2 (Listas)**.** Capturamos la *signatura* de las listas $[a]$ mediante el bifunctor L, definido como sigue:

$$\begin{array}{l}\mathbf{data}\ L\ a\ x = N \mid C\ \ a\ \ x\\ \mathbf{instance}\ \mathit{Bifunctor}\ L\ \mathbf{where}\\ \quad \mathit{bimap}\ \phi\ \xi\ N \qquad\quad = N\\ \quad \mathit{bimap}\ \phi\ \xi\ (C\ \ a\ \ b) = C\ (\phi\ \ a)\ (\xi\ \ b)\end{array}$$

La instancia genérica para el functor parametrizado resulta:

$$\begin{array}{l}\mathbf{instance}\ \mathit{Functor}\ (L\ \ a)\ \mathbf{where}\\ \quad \mathit{fmap}\ f\ N \qquad\quad = N\\ \quad \mathit{fmap}\ f\ (C\ \ a\ \ b) = C\ \ a\ (f\ \ b)\end{array}$$

Ejemplo 3.3 (Árboles Binarios)**.** Para los árboles binarios $\mathit{Bin}\ a$, su *signatura* queda determinada por el siguiente bifunctor.

$$\begin{array}{l}\mathbf{data}\ B\ \ a\ \ b = \mathit{BL} \mid \mathit{BN}\ \ b\ \ a\ \ b\\ \mathbf{instance}\ \mathit{Bifunctor}\ B\ \mathbf{where}\\ \quad \mathit{bimap}\ f\ \ g\ \mathit{BL} \qquad\qquad = \mathit{BL}\\ \quad \mathit{bimap}\ f\ \ g\ (\mathit{BN}\ \ l\ \ x\ \ r) = \mathit{BN}\ (g\ \ l)\ (f\ \ x)\ (g\ \ r)\end{array}$$

La instancia genérica para el functor parametrizado resulta:

$$\begin{array}{l}\mathbf{instance}\ \mathit{Functor}\ (B\ \ a)\ \mathbf{where}\\ \quad \mathit{fmap}\ f\ \mathit{BL} \qquad\qquad = \mathit{BL}\\ \quad \mathit{fmap}\ f\ (\mathit{BN}\ \ l\ \ x\ \ r) = \mathit{BN}\ (f\ \ l)\ x\ (f\ \ r)\end{array}$$

Ejemplo 3.4 (Tip-Trees)**.** Consideramos árboles binarios con valores en las hojas, o *Tip-Trees*, implementados por el siguiente tipo inductivo:

$$\mathbf{data}\ \mathit{Tip}\ \ a = \mathit{Tip}\ \ a \mid \mathit{Fork}\ (\mathit{Tip}\ \ a)\ (\mathit{Tip}\ \ a)$$

Su *signatura* queda determinada por el siguiente bifunctor.

data $T\ a\ b = TL\ a \mid TN\ b\ b$
instance *Bifunctor* T **where**
  $bimap\ f\ g\ (TL\ x) \quad\quad = TL\ (f\ x)$
  $bimap\ f\ g\ (TN\ m1\ m2) = TN\ (g\ m1)\ (g\ m2)$

La instancia genérica para el functor parametrizado resulta:

instance *Functor* $(T\ a)$ **where**
  $fmap\ f\ (TL\ x) = TL\ x$
  $fmap\ f\ (TN\ m1\ m2) = TN\ (f\ m1)\ (f\ m2)$

En este trabajo representaremos como puntos fijos a tipos de datos *regulares o inductivos*, éstos se caracterizan mediante declaraciones de tipo cuyos lados derechos no contienen espacios de funciones y las ocurrencias recursivas tienen los mismos argumentos que los lados izquierdos. Los *functores (y bifunctores)* que capturan las signaturas de éstos son construidos mediante productos, sumas, constantes y functores de tipo.

3.2.1. Fold

La existencia de F-álgebras iniciales para el functor F que modela la *signatura* de un tipo inductivo permite definir al operador que captura el esquema de recursión estructural sobre ese tipo:

Definición 3.9 (Fold)**.** Al único morfismo universal entre $(\mu\mathsf{F}, in_{\mathsf{F}})$ y cualquier F-álgebra (A, ϕ) lo denotaremos $(\!|\phi|\!)_{\mathsf{F}}$. Éste no es otro que el conocido operador *fold*, o *catamorfismo*, [20, 29, 43] asociado al functor F. El hecho de que por ser $(\!|\phi|\!)_{\mathsf{F}}$ una flecha (única y universal) de $\mathcal{A}lg_{\mathsf{F}(\mathcal{C})}$, resulta el (único y universal) homomorfismo entre $(\mu\mathsf{F}\ , in_{\mathsf{F}}\ )$ y cualquier $(A, \phi : \mathsf{F}\ A \to A)$ se expresa mediante el siguiente diagrama conmutativo:

$$\begin{tikzcd}
\mathsf{F}\ \mu F \arrow[r, "\mathsf{F}\ (\!|\phi|\!)_{\mathsf{F}}"] \arrow[d, "in_{\mathsf{F}}"'] & \mathsf{F}\ A \arrow[d, "\phi"] \\
\mu F \arrow[r, dotted, "(\!|\phi|\!)_{\mathsf{F}}"'] & A
\end{tikzcd}$$

Al diagrama anterior se lo conoce también como *Propiedad Universal de Fold* [30]. De esta propiedad se deriva la implementación del *fold* genérico:

$fold\ \ :: (Functor\ f) \Rightarrow (f\ x \to x) \to \mu\ f \to x$
$fold\ \phi = \phi \circ fmap\ (fold\ \phi) \circ unIn$

Ésta puede instanciarse para diferentes tipos de datos:

Ejemplo 3.5 (Listas)**.** Para listas, el *fold* genérico se instancia mediante el conocido operador *foldr* [8]:

$foldr \quad\quad\quad\quad :: (a \to b \to b) \to b \to [a] \to b$
$foldr\ f\ e\ [\,] \quad\quad = e$
$foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)$

Ejemplo 3.6 (*Tip-Trees*)**.** Para *Tip-Trees*, el operador *fold* genérico se instancia mediante el operador *foldTip*:

$$
\begin{aligned}
&\mathit{foldTip} :: (b \to b \to b) \to (a \to b) \to \mathit{Tip}\ a \to b\\
&\mathit{foldTip}\ g\ f\ (\mathit{Tip}\ x) = f\ x\\
&\mathit{foldTip}\ g\ f\ (\mathit{Fork}\ t1\ t2) = g\ (\mathit{foldTip}\ g\ f\ t1)\ (\mathit{foldTip}\ g\ f\ t2)
\end{aligned}
$$

Ejemplo 3.7 (Árboles Binarios)**.** Para los árboles binarios, el operador *fold* genérico se instancia mediante el operador *foldBin*:

$$
\begin{aligned}
&\mathit{foldBin} :: (b \to a \to b \to b) \to b \to \mathit{Bin}\ a \to b\\
&\mathit{foldBin}\ g\ f\ \mathit{Leaf} \qquad\qquad\quad = f\\
&\mathit{foldBin}\ g\ f\ (\mathit{Node}\ t1\ x\ t2) = g\ (\mathit{foldBin}\ g\ f\ t1)\ x\ (\mathit{foldBin}\ g\ f\ t2)
\end{aligned}
$$

3.2.2. Propiedades de Fold

El operador *fold* posee varias propiedades algebraicas que resultan útiles para calcular programas. Como corolario de la Propiedad Universal de Fold, se obtienen dos propiedades importantes:

Teorema 3.1 (Identidad de Fold)**.**

$$(\!|\mathit{in}_{\mathsf{F}}|\!)_{\mathsf{F}} = \ \mathsf{id}_{\mu\mathsf{F}} \qquad \textbf{(Fold.Id)}$$

Teorema 3.2 (Fusión Pura para Fold)**.** *Sean $h : A \to B$ un homomorfismo de F-álgebras entre dos F-álgebras $\phi : \mathsf{F}\ A \to A$ y $\varphi : \mathsf{F}\ B \to B$, entonces:*

$$h \circ (\!|\phi|\!)_{\mathsf{F}} \ = (\!|\varphi|\!)_{\mathsf{F}} \qquad \textbf{(Fold.Pure)}$$

Definición 3.10 (Transformador de F-álgebras)**.** Dados dos functores F y G, un *transformador de F-álgebras* es una función polimórfica $\mathsf{T}\ : \forall X.(\mathsf{F}\ X \to X) \to (\mathsf{G}\ X \to X)$ que convierte F-álgebras en G-álgebras.

Como T es polimórfica, para todo $h : A \to B$ *homomorfismo de F-álgebras* entre dos F-álgebras $\phi : \mathsf{F}\ A \to A$ y $\varphi : \mathsf{F}\ B \to B$, el siguiente diagrama conmmuta:

$$
\begin{tikzcd}
\mathsf{G}\ A \arrow[r, "\mathsf{G}\ f"] \arrow[d, "\mathsf{T}\ \phi"'] & \mathsf{G}\ B \arrow[d, "\mathsf{T}\ \varphi"] \\
A \arrow[r, "f"'] & B
\end{tikzcd}
$$

Intuitivamente, un *transformador* es una función polimórfica que construye cierta clase de álgebras, a partir de álgebras de otra clase. Usando este concepto, podemos estudiar la composición de funciones expresadas usando el operador *fold* para distintos tipos de datos, aplicando el siguiente resultado:

Teorema 3.3 (Acid Rain: Fusión Fold.Fold)**.** *Sea* $\mathsf{T} : \forall X.(\mathsf{F}\ X \to X) \to (\mathsf{G}\ X \to X)$ *un transformador de* F*-álgebras. Entonces:*

$$\begin{array}{ll} & \phi = \mathsf{T}\ in_{\mathsf{F}} \\ \Longrightarrow & \\ & (\!|\psi|\!)_{\mathsf{F}} \circ (\!|\phi|\!)_{\mathsf{G}} = (\!|\mathsf{T}\ \psi|\!)_{\mathsf{G}} \end{array} \qquad (\textbf{Fold.Fold})$$

Fokkinga [17] muestra como construir transformadores especializados con mayor estructura. Nos resultará util considerar la siguiente generalización de la propiedad anterior:

Teorema 3.4 (Acid Rain Generalizado: Fusión Fold - MapFold)**.** *Sea* H *un functor y* $\mathsf{T} : \forall X.(\mathsf{F}\ X \to X) \to (\mathsf{G}\ (\mathsf{H}\ X) \to \mathsf{H}\ X)$ *un transformador de* F*-álgebras. Entonces:*

$$\begin{array}{ll} & \phi = \mathsf{T}\ in_{\mathsf{F}} \\ \Longrightarrow & \\ & \mathsf{H}\ (\!|\psi|\!)_{\mathsf{F}} \circ (\!|\phi|\!)_{\mathsf{G}} = (\!|\mathsf{T}\ \psi|\!)_{\mathsf{G}} \end{array} \qquad (\textbf{Fold.MapFold})$$

3.2.3. Functores de Tipo

Definición 3.11 (Functor de Tipo)**.** Sea $\mathsf{D}\ A = \mu\mathsf{F}_{\mathsf{A}}$ el tipo inductivo parametrizado definido como el punto fijo del *functor* F_{A}. El constructor D es un functor $\mathsf{D}_{\mathsf{F}} : \mathcal{C} \to \mathcal{C}$, al que llamaremos *Functor de Tipo de F*. Sea $f : A \to B$, la acción sobre flechas de D_{F} se define como:

$$\mathsf{D}_{\mathsf{F}}\ f = (\!|in_{\mathsf{F}_{\mathsf{B}}} \circ \mathsf{F}\ (f, \mathsf{id}_{\mathsf{D}B})|\!)_{\mathsf{F}_{\mathsf{A}}} \qquad (\textbf{Type.Def})$$

Dejamos al lector verificar que esta definición cumple con las propiedades de functorialidad esperadas. En Haskell, el operador *map* genérico resulta:

$$\begin{array}{l} fmap\ \ :: (Bifunctor\ f) \Rightarrow (a \to b) \to \mu\ (f\ \ a) \to \mu\ (f\ \ b) \\ fmap\ f = fold\ (In \circ bimap\ f\ \ id) \end{array}$$

Los functores de tipo proveen además una ley de fusión con *fold* que se deriva de la propiedad de *Acid Rain para Fold* anterior. El teorema resulta:

Teorema 3.5 (Fusión Fold.Map)**.** *Sean* $f : A \to B$ *y* $\phi : \mathsf{F}_{\mathsf{B}}\ X \to X$ *un* F_{B}*-álgebra, entonces:*

$$(\!|\phi|\!)_{\mathsf{F}_{\mathsf{B}}} \circ \mathsf{D}_{\mathsf{F}_{\mathsf{A}}}\ f = (\!|\phi \circ \mathsf{F}\ (f, \mathsf{id}_{\mathsf{D}X})|\!)_{\mathsf{F}_{\mathsf{A}}} \qquad (\textbf{Fold.Type})$$

Capítulo 4

Recursión Aplicativa

> If the Grove were cut, all wizardry would fail. The roots of those trees are the roots of knowledge. The patterns the shadows of their leaves make in the sunlight write the words Segoy spoke in the Making
>
> Ursula K. Le Guin,
> *The Finder, Tales from Earthsea*

Nos proponemos caracterizar la recursión *estructural aplicativa*. Es decir, queremos analizar funciones recursivas estructurales que consumen valores de ciertos tipos de datos y producen valores embebidos en un efecto aplicativo para descubrir patrones comunes entre éstas. Particularmente, pretendemos identificar cuál es el rol de las operaciones de la interfaz aplicativa para un functor dado en estas definiciones, y cómo se consumen y producen los valores puros. Para ello, procedemos a definir algunos ejemplos con diferentes efectos aplicativos, analizando detalladamente estas características.

Ejemplo 4.1. El functor aplicativo *Maybe* modela las fallas como un efecto computacional. Queremos definir una función que suma los recíprocos de una lista, fallando en el caso de que haya algún valor nulo. Interpretaremos a la generación del recíproco de un número como una *acción aplicativa*: si el valor es no nulo retornamos una computación que produce su recíproco, caso contrario indicamos la falla con *Nothing*:

$$\begin{array}{l} \mathit{recip} \quad :: \mathit{Float} \to \mathit{Maybe\ Float} \\ \mathit{recip}\ x = \mathbf{if}\ (x \not\equiv 0)\ \mathbf{then}\ \mathit{pure}\ (1\ /\ x)\ \mathbf{else}\ \mathit{Nothing} \end{array}$$

Utilizamos esta *acción aplicativa* para definir la función *sumrecips* por recursión estructural:

$$\begin{array}{ll} \mathit{sumrecips} & :: [\mathit{Float}] \to \mathit{Maybe\ Float} \\ \mathit{sumrecips}\ [\,] & = \mathit{pure}\ 0 \\ \mathit{sumrecips}\ (x : \mathit{xs}) & = \mathit{pure}\ (+) \circledast \mathit{recip}\ x \circledast \mathit{sumrecips}\ \mathit{xs} \end{array}$$

Además de la aplicación de *recip* a los elementos de la lista, reconocemos la aplicación del $\mathsf{L}_{\mathit{Float}}$-álgebra pura $(+, 0)$ introducida en el functor aplicativo mediante *pure* . Este álgebra es el que usaríamos para definir la función *suma* usando el *fold* de listas, *foldr*, para una lista de valores puros.

Consideremos ahora realizar un *recorrido o traversal* sobre una lista con la acción aplicativa *recip* usando la instancia de *traverse* para [ ], presentada en el Ejemplo 2.8. Dicha función, aplicada a una lista de números da como resultado una lista de los recírprocos dentro del functor aplicativo, o *Nothing* en el caso de que uno de los valores sea nulo:

$$
\begin{array}{l}
\mathit{recips} :: [\mathit{Float}] \rightarrow \mathit{Maybe}\ [\mathit{Float}] \\
\mathit{recips} = \mathit{traverse}\ \mathit{recip}
\end{array}
$$

Si observamos las definiciones de *traverse*, tomando $f = \mathit{recip}$, y *sumrecips* vemos que éstas tienen la misma estructura, pero en la última se reemplaza el $\mathsf{L_A}$-álgebra inicial $((:), [\,])$ por $((+), 0)$ evitando reconstruir la lista. Podemos interpretar entonces a *sumrecips* como el resultado de combinar aplicativamente un *fold* puro con un *recorrido* es decir,

$$\mathit{sumrecips}\ \mathit{xs} = \mathit{pure}\ (\mathit{foldr}\ (+)\ 0) \circledast \mathit{traverse}\ \mathit{recip}\ \mathit{xs}$$

Reemplazando de acuerdo a la definición de *fmap* para un functor aplicativo (**App.Map**) presentada en el Capítulo 2, obtenemos la siguiente expresión:

$$\mathit{sumrecips} = \mathit{fmap}\ (\mathit{foldr}\ (+)\ 0) \circ \mathit{traverse}\ \mathit{recip}$$

Esta última definición nos refuerza la observación que hemos hecho sobre la primer definición de *sumrecips*, donde reconocimos una acción con efecto aplicativo y la aplicación de un álgebra pura. Además, nos permite distinguir de forma más clara el rol de las operaciones del functor aplicativo en ésa primer definición, observando que éstas intervienen con dos propósitos: *mapear* un *fold* puro a través del efecto, y realizar un *recorrido* de la lista, generando una lista de efectos. Por supuesto, esta última versión es menos eficiente que la primera, dado que se construye una lista intermedia con los valores recíprocos. No obstante nos permite interpretar composicionalmente el algoritmo.

Ejemplo 4.2. Consideramos el functor aplicativo *Acumulador o Constante Monoidal*: *Acc a o*, introducido en la Sección 2.1.3, con la familia de monoides determinada por las listas $([\,a\,], (+\!\!+), [\,])$. Nos interesa recorrer un árbol binario y acumular aquellos valores almacenados que verifican una propiedad $p :: a \rightarrow \mathit{Bool}$ dada. Definimos la función recursiva estructural *take aplicativamente*:

$$
\begin{array}{lll}
\mathit{take} & :: (a \rightarrow \mathit{Bool}) \rightarrow \mathit{Bin}\ a \rightarrow \mathit{Acc}\ [\,a\,]\ (\mathit{Bin}\ a) & \\
\mathit{take}\ p\ \mathit{Leaf} & = \mathit{pure}\ \mathit{Leaf} & \\
\mathit{take}\ p\ (\mathit{Node}\ \mathit{tl}\ x\ \mathit{tr}) & = \mathit{pure}\ \mathit{Node} & \circledast\ \mathit{take}\ p\ \mathit{tl} \\
 & & \circledast\ (\mathbf{if}\ (p\ x)\ \mathbf{then}\ \mathit{Acc}\ [\,x\,]\ \mathbf{else}\ \mathit{Acc}\ [\,]) \\
 & & \circledast\ \mathit{take}\ p\ \mathit{tr}
\end{array}
$$

Como hemos visto en los capítulos anteriores, cualquier acumulación con esta familia de functores aplicativos significan en realidad un *recorrido* de la estructura de datos. En efecto, si consideramos la definición del operador *traverse* para el tipo de datos *Bin a* definido en el Ejemplo 2.9, observamos que la función *take* se describe como un recorrido cuya *acción aplicativa* es acumular el valor si se verifica la propiedad *p* o el *neutro* del monoide ([ ]), en caso contrario:

$$
\begin{array}{l}
\mathit{take}\ \ :: (a \rightarrow \mathit{Bool}) \rightarrow \mathit{Bin}\ a \rightarrow \mathit{Acc}\ [\,a\,]\ (\mathit{Bin}\ a) \\
\mathit{take}\ p = \mathit{traverse}\ (\lambda x \rightarrow \mathbf{if}\ (p\ x)\ \mathbf{then}\ \mathit{Acc}\ [\,x\,]\ \mathbf{else}\ \mathit{Acc}\ [\,])
\end{array}
$$

Ejemplo 4.3. Consideramos el functor aplicativo de *ziplists o listas vectorizadas* que presentamos en 2.6. Proponemos definir una función que dada una matriz –aproximada como una lista de ziplists– calcula la sumatoria de los cuadrados de los valores de las filas de la matriz transpuesta. Es decir, queremos definir una función con el siguiente tipo $sumSqrTrans :: [[Int]] \to [Int]$.

Proponemos diseñarla en forma modular, primero consideraremos la función *transpose* que calcula la matriz transpuesta *aplicativamente*, presentada en el Ejemplo 2.6. El cálculo de la suma de los cuadrados de una lista de valores es una acción pura fácilmente definible en función del *fold* de listas, *foldr*. Para terminar de definir *sumSqrTrans* podemos introducirlo en el functor aplicativo de los *ziplist* y combinarlo con el resultado de la transposición, *id est* mapear dicho *fold* sobre las filas de la matriz transpuesta:

$$
\begin{array}{l}
sumSqrTrans :: [[Int]] \to [Int] \\
sumSqrTrans = fmap\ (foldr\ (\lambda x\ sqs \to x \uparrow 2 + sqs)\ 0) \circ transpose
\end{array}
$$

En el Ejemplo 2.8 hemos observado que la función *transpose* es una instancia de la distribución—*dist* —de las listas con respecto al functor aplicativo de las *ziplists* , y por lo tanto se puede expresar como un *recorrido* con la función identidad, i.e. $dist = traverse\ id$. Aplicando este resultado, obtenemos la siguiente expresión:

$$
\begin{array}{l}
sumSqrTrans :: [[Int]] \to [Int] \\
sumSqrTrans = fmap\ (foldr\ (\lambda x\ sqs \to x \uparrow 2 + sqs)\ 0) \circ traverse\ id
\end{array}
$$

En estos ejemplos, reconocemos la incidencia de un patrón común. Todas estas funciones recursivas aplicativas pueden interpretarse como el resultado de *mapear* un *fold* puro luego de realizar un *recorrido* con una acción aplicativa. En el Ejemplo 4.2, donde la función *sumSqrTrans* es sólo un *recorrido*, podemos recuperar el esquema trivialmente considerando el álgebra inicial y las propiedades de identidad de Functores (**Functor.id**) y *fold* (**Fold.Id**). En el último ejemplo, Ejemplo 4.3, la acción aplicativa trivial en la definición de la transposición de matrices nos permite observar que este patrón también contempla el caso particular de la acción pura de un *fold* combinado aplicativamente con una distribución.

Proponemos estudiar este patrón como modelo de recursión estructural con efectos aplicativos. En este capítulo, derivaremos una definición genérica y eficiente del operador a partir de esta especificación, y presentaremos sus propiedades calculacionales.

4.1. El operador *ifold*

Para definir el patrón que capture la recursión estructural aplicativa vamos a trabajar en un mayor nivel de abstracción, puesto que queremos encontrar una definición genérica para dicho operador. Con ese propósito, representaremos los tipos de datos de forma genérica siguiendo, siguiendo los conceptos y definiciones introducidos en el Capítulo 3.

Primero, debemos generalizar los conceptos y definiciones relacionadas con *traverse*, presentadas en el Capítulo 2.

4.1.1. Bifunctores Bitraversables

Gibbons y de Oliveira [22] presentan una implementación genérica de *traverse*, utilizando bifunctores paramétricos para modelar la signatura de los tipos de datos y una noción de *bidistributividad* entre éstos y los functores aplicativos.

Definición 4.1 (Bifunctores Bitraversables)**.** Sea F un *bifunctor*. Decimos que F es bitraversable si para cualquier functor aplicativo C, existe una *transformación natural* $\delta_{\mathsf{F}}^{\mathsf{C}} : \mathsf{F}\ (\mathsf{C}\ A, \mathsf{C}\ X) \to \mathsf{C}\ \mathsf{F}\ (A, X)$ tal que el siguiente diagrama conmute:

$$
\begin{tikzcd}
\mathsf{F}\ (\mathsf{C}\ A, \mathsf{C}\ X) \arrow[r, "\delta^{\mathsf{C}}_{\mathsf{F}\ (A,X)}"] \arrow[d, "\mathsf{F}\ (\mathsf{C}\ f, \mathsf{C}\ g)"'] & \mathsf{C}\ \mathsf{F}\ (A, X) \arrow[d, "\mathsf{C}\ \mathsf{F}\ (f,g)"] \\
\mathsf{F}\ (\mathsf{C}\ B, \mathsf{C}\ Y) \arrow[r, "\delta^{\mathsf{C}}_{\mathsf{F}\ (B,Y)}"'] & \mathsf{C}\ \mathsf{F}\ (B, Y)
\end{tikzcd}
$$

En Haskell, representamos el hecho de que un bifunctor cumple con la propiedad de ser *bitraversable* mediante la siguiente clase:

class *Bifunctor s* $\Rightarrow$ *BiTraversable s* **where**
  $\delta :: (\mathit{Applicative}\ f) \Rightarrow s\ (f\ a)\ (f\ b) \rightarrow f\ (s\ a\ b)$

Esta ley distributiva se puede definir para todo bifunctor que captura la signatura de un tipo de datos *regular* y podría definirse politípicamente i.e. por inducción sobre la estructura del functor signatura [2, 42]. A modo de ejemplo, presentamos las *instancias* para algunos de los bifunctores que hemos utilizado en los ejemplos anteriores:

Ejemplo 4.4 (Listas y Árboles)**.** Presentamos los bifunctores que capturan la signatura de las listas L y de los árboles binarios B que introdujimos anteriormente, junto con la definición de la *bidistribución* para éstos.

data $L\ a\ b = N \mid C\ a\ b$
instance *BiTraversable L* **where**
  $\delta\ N \quad\quad\;\; = \mathit{pure}\ N$
  $\delta\ (C\ x\ xs) = \mathit{pure}\ C \circledast x \circledast xs$
data $B\ a\ b = BL \mid BN\ b\ a\ b$
instance *BiTraversable B* **where**
  $\delta\ BL \quad\quad\quad\;\; = \mathit{pure}\ BL$
  $\delta\ (BN\ tl\ x\ tr) = \mathit{pure}\ BN \circledast tl \circledast x \circledast tr$

Esta noción de bidistribución permite la definición de un *traverse* genérico [22]:

Definición 4.2 (*traverse*)**.** Sean F un *bifunctor bitraversable*, C un functor aplicativo, e $\iota : A \rightarrow \mathsf{C}\ B$ una acción aplicativa entonces, definimos al operador $\mathit{traverse}_{\mathsf{F}}\ \iota : \mu\mathsf{F}_{\mathsf{A}} \rightarrow \mathsf{C}\ \mu\mathsf{F}_{\mathsf{B}}$ como:

$$\mathit{traverse}_{\mathsf{F}}\ \iota = (\!|\mathsf{C}\ in_{\mathsf{F}_{\mathsf{B}}} \circ \delta^{\mathsf{C}}_{\mathsf{F}} \circ \mathsf{F}\ (\iota, id)|\!)_{\mathsf{F}_{\mathsf{A}}} \qquad \textbf{(Trv.Def)}$$

En Haskell la implementación resulta:

$\mathit{traverse} \quad :: (\mathit{Applicative}\ f, \mathit{BiTraversable}\ s) \Rightarrow (a \rightarrow f\ b) \rightarrow \mu\ (s\ a) \rightarrow f\ (\mu\ (s\ b))$
$\mathit{traverse}\ f = \mathit{fold}\ (\mathit{fmap}\ In \circ \delta \circ \mathit{bimap}\ f\ id)$

Otro concepto introducido por Gibbons y Oliveira en [22] que nos interesa rescatar es el de *transformación de functores aplicativos*. Preferimos el nombre de *morfismo de Functores Aplicativos* puesto que consideramos que este último refleja mejor su significado.

Definición 4.3 (Morfismo de Functores Aplicativos)**.** Sean C y D dos functores aplicativos y τ una *transformación natural* $\tau : \mathsf{C} \Rightarrow \mathsf{D}$ entre éstos, decimos que τ es un *morfismo de functores aplicativos* si se preserva por *pure* y $\circledast$, id est:

$$\tau \circ pure_{\mathsf{C}} = pure_{\mathsf{D}} \qquad (\textbf{Morph.}\mathit{pure})$$

$$\tau \circ \circledast_{\mathsf{C}} = \circledast_{\mathsf{D}} \circ (\tau \times \tau) \qquad (\textbf{Morph.}\circledast)$$

Estos morfismos estan estrechamente vinculados a la *bidistribución*, dado que requeriremos que ésta verifique la siguiente propiedad:

Teorema 4.1. *Sea* F *un bifunctor bitraversable, entonces para cualquier morfismo de functores aplicativos* $\tau : \mathsf{C} \Rightarrow \mathsf{D}$ *se debe verificar que:*

$$\tau \circ \delta_{\mathsf{F}}^{\mathsf{C}} = \delta_{\mathsf{F}}^{\mathsf{D}} \circ \mathsf{F}\ (\tau, \tau) \qquad (\textbf{Morph.}\delta)$$

Este requerimiento sobre δ_F resulta de formalizar el requerimiento de naturalidad para *dist* presentado en [22].

4.1.2. Derivación de *ifold*

Ahora que hemos generalizado *traverse* y los conceptos relacionados, disponemos de todas las herramientas necesarias para estudiar el patrón que hemos observado en los ejemplos introductorios a este capítulo: *mapear un fold puro sobre el resultado de realizar un recorrido con una acción aplicativa.* Proponemos el operador *ifold, fold aplicativo o idiomático*, como la forma de capturar este patrón y lo caracterizamos como sigue:

Definición 4.4 (Caracterización de Ifold)**.** Sean F un *bifunctor bitraversable*, C un functor aplicativo, $\iota : A \to \mathsf{C}\ B$ una acción aplicativa y $\phi : \mathsf{F_B}\ X \to X$ un $\mathsf{F_B}$-álgebra . Proponemos al operador $ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ \iota : \mu\mathsf{F_A} \to \mathsf{C}\ X$ mediante la siguiente especificación:

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ \iota = \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}} \circ traverse_{\mathsf{F}}\ \iota \qquad (\textbf{Ifold.Charn})$$

Podemos considerar que esta caracterización de $ifold^{\mathsf{C}}_{\mathsf{F_A}}$ es ineficiente puesto que supone la construcción de una estructura de datos intermedia por *traverse*, para luego ser inmediatamente consumida con *fold*. En los ejemplos introductorios al comienzo de este capítulo, hemos aventurado que esta especificación se corresponde con una función más eficiente donde se reemplazan los constructores de la estructura intermedia producida por *traverse* por el argumento de *fold*. Vamos a verificar formalmente dicha observación, derivando dicha definición *fusionada* desde la especificación de $ifold^{\mathsf{C}}_{\mathsf{F_A}}$:

$$\begin{aligned}
& \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}} \circ traverse_{\mathsf{F}}\ \iota \\
= & \qquad \{ \text{ Def. de } \mathit{traverse}, (\textbf{Trv.Def}) \ \} \\
& \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}} \circ (\!|\mathsf{C}\ in_{\mathsf{F_B}} \circ \delta_{\mathsf{F}}^{\mathsf{C}} \circ \mathsf{F}\ (\iota, id)|\!)_{\mathsf{F_A}} \\
= & \qquad \{ \ \textit{Acid Rain Generalizado}, (\textbf{Fold.MapFold}) \ \} \\
& (\!|\mathsf{C}\ \phi \circ \delta_{\mathsf{F}}^{\mathsf{C}} \circ \mathsf{F}\ (\iota, id)|\!)_{\mathsf{F_A}} \quad \textit{q.e.d.}
\end{aligned}$$

Para que el último paso de la derivación sea correcto, se debe poder expresar al $\mathsf{F_A}$-álgebra en la definición de *traverse* como la aplicación de un transformador de $\mathsf{F_B}$-álgebras $\mathsf{T_B}$ al álgebra inicial $in_{\mathsf{F_B}}$, *id est*:

$$\mathsf{C}\ in_{\mathsf{F_B}} \circ \delta_{\mathsf{F}}^{\mathsf{C}} \circ \mathsf{F}\ (\iota, id) = \mathsf{T_B}\ in_{\mathsf{F_B}}$$

De la ecuación anterior, la definición de T_B es inmediata. Sea $\xi : \mathsf{F}_\mathsf{B}\ X \to X$, entonces:

$$\mathsf{T}_\mathsf{B}\ \xi = \mathsf{C}\ \xi \circ \delta^\mathsf{C}_\mathsf{F} \circ \mathsf{F}\ (\iota, id)$$

Aplicando T_B a ϕ, se obtiene el A-álgebra expresada en la última ecuación de la derivación, obteniéndose el resultado esperado: se han reemplazado los constructores de la estructura intermedia (el D_B-álgebra inicial, $in_{\mathsf{F}_\mathsf{B}}$) por el F_B-álgebra pura.

En conclusión, habiendo partido de la espeficicación de $ifold^\mathsf{C}_{\mathsf{F}_\mathsf{A}}$ hemos derivado formalmente una función descripta como un fold del functor F_A sobre un F_A-álgebra con la siguiente estructura:

$$\mathsf{F}\ (A, \mathsf{C}\ X) \xrightarrow{\mathsf{F}\ (\iota, id)} \mathsf{F}\ (\mathsf{C}\ B, \mathsf{C}\ X) \xrightarrow{\delta} \mathsf{C}\ \mathsf{F}\ (B, X) \xrightarrow{\mathsf{C}\ \phi} \mathsf{C}\ X$$

donde $\iota : A \to \mathsf{C}\ B$ es una acción aplicativa, $\delta^\mathsf{C}_\mathsf{F} : \mathsf{F}\ (\mathsf{C}\ A, \mathsf{C}\ X) \to \mathsf{C}\ \mathsf{F}\ (A, X)$ es la bidistribución del bifunctor F respecto del functor aplicativo C y $\phi : \mathsf{F}\ (B, X) \to X$ el F_B-álgebra pura que mapeamos sobre el functor aplicativo. Procedemos a definir formalmente al operador $ifold^\mathsf{C}_{\mathsf{F}_\mathsf{A}}$ mediante esta expresión.

Definición 4.5 (Definición de Ifold)**.** Sean F un *bifunctor bitraversable*, C un functor aplicativo, $\iota : A \to \mathsf{C}\ B$ una acción aplicativa, y $\phi : \mathsf{F}_\mathsf{B}\ X \to X$ un F_B-álgebra pura. Entonces el operador $ifold^\mathsf{C}_{\mathsf{F}_\mathsf{A}} \phi\ \iota$ queda definido como:

$$ifold^\mathsf{C}_{\mathsf{F}_\mathsf{A}} \phi\ \iota = (\!|\mathsf{C}\ \phi \circ \delta^\mathsf{C}_\mathsf{F} \circ \mathsf{F}\ (\iota, id)|\!)_{\mathsf{F}_\mathsf{A}} \qquad \textbf{(Ifold.Def)}$$

La implementación genérica en Haskell resulta:

$$\begin{array}{l} \mathit{ifold} :: (\mathit{Applicative}\ c, \mathit{BiTraversable}\ s) \Rightarrow (s\ b\ x \to x) \to (a \to c\ b) \to \mu\ (s\ a) \to c\ x \\ \mathit{ifold}\ \phi\ \iota = \mathit{fold}\ (\mathit{fmap}\ \phi \circ \delta \circ \mathit{bimap}\ \iota\ \mathit{id}) \end{array}$$

Utilizamos esta definición para revisar los tres ejemplos introductorios que hemos presentado en este capítulo:

Ejemplo 4.5. Para las listas $[a]$, especializando la definición de *ifold*, obtenemos la siguiente expresión para *ifoldr*:

$$\begin{array}{ll} \mathit{ifoldr}\ :: (\mathit{Applicative}\ c) & \Rightarrow (b \to x \to x) \to x \to (a \to c\ b) \to [a] \to c\ x \\ \mathit{ifoldr}\ \phi\ \eta\ \iota\ [\,] & = \mathit{pure}\ \eta \\ \mathit{ifoldr}\ \phi\ \eta\ \iota\ (x : \mathit{xs}) & = \mathit{pure}\ \phi \circledast \iota\ x \circledast \mathit{ifoldr}\ \phi\ \eta\ \iota\ \mathit{xs} \end{array}$$

Recordando la especificación de *sumrecips*, $\mathit{sumrecips} = \mathit{fmap}\ (\mathit{foldr}\ (+)\ 0) \circ \mathit{traverse}\ \mathit{recip}$ obtenemos la expresión en función de *ifoldr*:

$$\mathit{sumrecips} = \mathit{ifoldr}\ (+)\ 0\ \mathit{recip}$$

Si expandimos la definición de *ifold* obtendremos la definición original que habíamos proporcionado para esta función.

Ejemplo 4.6. Para los árboles binarios con valores en los nodos,*o tip-trees*, la definición se especializa a:

$$\begin{array}{ll} \mathit{ifoldB} :: (\mathit{Applicative}\ c) & \Rightarrow (x \to b \to x \to x) \to x \to (a \to c\ b) \to \mathit{Bin}\ a \to c\ x \\ \mathit{ifoldB}\ \phi\ \eta\ \iota\ \mathit{Leaf} & = \mathit{pure}\ \eta \\ \mathit{ifoldB}\ \phi\ \eta\ \iota\ (\mathit{Node}\ tl\ x\ tr) & = \mathit{pure}\ \phi \circledast \mathit{ifoldB}\ \phi\ \eta\ \iota\ tl \\ & \quad \circledast \iota\ x \\ & \quad \circledast \mathit{ifoldB}\ \phi\ \eta\ \iota\ tr \end{array}$$

Cuando dimos la especificación de *take*, observamos que ésta función era en realidad sólo un *recorrido*, puesto que no se consumía la estructura luego de la acumulación. Entonces indicamos eso usando como argumento el $\mathsf{B_A}$-álgebra inicial $(\mathit{Node}, \mathit{Leaf})$, resultando en la siguiente definición:

$$\mathit{take}\ p = \mathit{ifoldB}\ \mathit{Node}\ \mathit{Leaf}\ (\lambda x \to \mathbf{if}\ (p\ x)\ \mathbf{then}\ \mathit{Acc}\ [x]\ \mathbf{else}\ \mathit{Acc}\ [\,])$$

Observamos entonces que los *recorridos* son definibles usando *ifold*, identificables por tener como argumento el álgebra inicial del mismo tipo de datos que se consume. En la sección siguiente volveremos sobre este tema, cuando estudiemos las propiedades de *ifold*.

Ejemplo 4.7. Revisitamos finalmente la función $sqrModTrans$ y el functor aplicativo definido por las *ziplists*. Habíamos partido de la siguiente especificación:

$$\begin{array}{l} sumSqrTrans :: [[\mathit{Int}]] \to [\mathit{Int}] \\ sumSqrTrans = \mathit{fmap}\ (\mathit{foldr}\ (\lambda x\ sqs \to x \uparrow 2 + sqs)\ 0) \circ traverse\ id \end{array}$$

Tomando la definición de *ifoldr* anterior, completamos la nueva definición para $sqrModTrans$

$$sqrModTrans = \mathit{ifoldr}\ (\lambda x\ sqs \to x \uparrow 2 + sqs)\ [\,]\ id$$

Expandiendo la definición de *ifoldr* obtenemos una expresión recursiva estructural que se asemeja a la definicíon de $transpose$, pero los constructores de listas: (:) y [ ] han sido reemplazados por el álgebra $((\lambda x\ s \to x \uparrow 2 + s), 0)$:

$$\begin{array}{ll} sumSqrTrans & :: [[\mathit{Int}]] \to [\mathit{Int}] \\ sumSqrTrans\ [\,] & = \mathit{pure}\ 0 \\ sumSqrTrans\ (xs : xss) & = \mathit{pure}\ (\lambda x\ sqs \to x \uparrow 2 + sqs) \circledast xs \circledast sumSqrTrans\ xss \end{array}$$

Hemos presentado el operador $ifold$, derivado a partir de una especificación modular pero ineficiente. Postulamos que este operador captura la recursión aplicativa, *id est* consumir uniformemente un valor de un tipo de datos *traversable* obteniendo un valor puro encapsulado en un efecto aplicativo. Ésta se caracteriza por la distribución de una acción aplicativa paramétrica que introduce efectos independientes y genera valores que son consumidos por un $\mathsf{F_B}$-álgebra pura dentro del efecto. Nos proponemos estudiar las propiedades calculacionales de este operador y su utilidad para diseñar modular e incrementalmente algoritmos eficientes con efectos aplicativos.

4.2. Propiedades de *ifold*

Vamos a estudiar que propiedades calculacionales se pueden derivar de este operador, y como aplicarlas para obtener funciones aplicativas eficientes. Para esto introduciremos también algunas definiciones auxiliares sobre functores aplicativos y el modelo de semántica por álegebras iniciales, complementarias a las definiciones y propiedades presentadas en los Capítulos 2 y 3. Omitiremos las demostraciones de los teoremas de esta sección para concentrarnos en la aplicación de los mismas, la mayoría de éstas demostraciones se encuentran en el Apéndice A.

4.2.1. *Recorridos* via *ifold*

Hemos usado la definición genérica de *traverse* para derivar la definición del operador *ifold*. Nos interesa mostrar que ambos son interdefinibles, ya que lo único necesario para definir tanto a *traverse* como a *ifold* es la *bidistribución* del bifunctor δ_{F}. Entonces, dado el operador $ifold^{\mathsf{C}}_{\mathsf{F_A}}$, podemos definir $traverse_{\mathsf{F_A}}$ y por lo tanto también $dist^{\mathsf{C}}_{\mathsf{F}}$ como sigue:

Teorema 4.2 (Traverse como Ifold)**.** *Sea* F *un bifunctor bitraversable,* C *un functor aplicativo,* $\iota : A \to \mathsf{C}\ B$ *una acción aplicativa, entonces la siguiente también es una definición válida del operador* $traverse_{\mathsf{F_A}}\ \iota : \mu\mathsf{F_A} \to \mathsf{C}\ \mu\mathsf{F_B}$ *:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} in_{\mathsf{F_B}}\ \iota = traverse_{\mathsf{F_A}}\ \iota \qquad (\textbf{Ifold.}in_F)$$

La prueba de esta igualdad es trivial expandiendo las definiciones. Recordemos que si consideramos la identidad como acción aplicativa trivial, $dist = traverse\ id$.

Teorema 4.3 (Dist como Ifold)**.** *Sea* F *un bifunctor bitraversable y* C *un functor aplicativo, entonces las siguientes definiciones de* $dist^{\mathsf{C}}_{\mathsf{F_A}} : \mu\mathsf{F_{C\,A}} \to \mathsf{C}\ \mu\mathsf{F_A}$ *son equivalentes:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} in_{\mathsf{F_A}}\ \mathsf{id}_{\mathsf{C\,A}} = dist^{\mathsf{C}}_{\mathsf{F_A}} = traverse_{\mathsf{F_A}}\ \mathsf{id}_{\mathsf{C}\ A} \qquad (\textbf{Ifold.dist})$$

Estas dos propiedades, aunque bastante triviales nos permiten definir *recorridos* genéricos en función de *ifold*:

$$\begin{array}{l}
traverse \ :: (Applicative\ f, BiTraversable\ s) \Rightarrow (a \to f\ b) \to \mu\ (s\ a) \to f\ (\mu\ (s\ b)) \\
traverse\ f = ifold\ In\ f
\end{array}$$

Del mismo modo, podemos dar definiciones genéricas para *accumulate* y *reduce* para el caso particular de los Acumuladores de Monoides, generalizando las definiciones introducidas en la Sección 2.3 para tipos de datos expresados como puntos fijos de los *bifunctores* que capturan su signatura:

$$\begin{array}{ll}
accumulate & :: (Monoid\ b, BiTraversable\ s) \Rightarrow (a \to b) \to \mu\ (s\ a) \to b \\
accumulate\ p & = acc \circ ifold\ In\ (Acc \circ p) \\
reduce & :: (Monoid\ a, BiTraversable\ s) \Rightarrow \mu\ (s\ a) \to a \\
reduce & = acc \circ ifold\ In\ Acc
\end{array}$$

Si bien estas definiciones y propiedades por sí mismas no aportan mucho, nos van a permitir combinar las propiedades *propias* de *ifold* con los resultados ya conocidos de los *recorridos*, y considerando que éstos son funciones bastante ubicuas en la programación funcional, ampliar el espectro de algoritmos para los cuales estamos intentando desarrollar propiedades calculacionales.

4.2.2. Leyes de *ifold*

Teorema 4.4 (Identidad de Ifold)**.** *Sea* F *un bifunctor bitraversable y* C *un functor aplicativo entonces:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} in_{\mathsf{F_A}}\ pure_{\mathsf{C}} = pure_{\mathsf{C}} \qquad (\textbf{Ifold.Id})$$

Esta propiedad es una consecuencia directa de las propiedades de identidad de fold (**Fold.Id**) y *traverse* (**Traverse.pure**), y su demostración es trivial considerando estas propiedades más la caracterización de $ifold$, (**Ifold.Charn**). La podemos generalizar, reemplazando el álgebra inicial $in_{\mathsf{F_A}}$, por un $\mathsf{F_A}$-álgebra cualquiera, obteniendo la siguiente propiedad.

Teorema 4.5 (Acción Pura de Ifold)**.** *Sean* F *un bifunctor bitraversable,* C *un functor aplicativo y* $\phi : \mathsf{F_A}\ X \to X$ *un* $\mathsf{F_A}$*-álgebra pura, entonces:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ pure_{\mathsf{C}} = \ pure_{\mathsf{C}} \circ \llparenthesis \phi \rrparenthesis_{\mathsf{F_A}} \tag{\textbf{Pure.Fold}}$$

Expresada como diagrama conmutativo, la ecuación anterior resulta:

$$\begin{tikzcd}
\mu_{\mathsf{F_A}} \arrow[r, "\llparenthesis \phi \rrparenthesis_{\mathsf{F_A}}"] \arrow[dr, bend right, "ifold^{\mathsf{D}}_{\mathsf{F_A}} \phi\ pure"'] & X \arrow[d, "pure_{\mathsf{C}}"] \\
 & \mathsf{C}\ X
\end{tikzcd}$$

Este resultado nos indica que si la acción aplicativa no introduce efectos sino que simplemente produce el valor puro dentro del efecto, entonces, como el *recorrido* implícito en *ifold* es sólo una inyección— recordar la Propiedad de Identidad de Traverse (**Traverse.pure**) —, el resultado del *fold* puro puede ser obtenido fuera del efecto y luego introducido mediante *pure*.

Teorema 4.6 (Acción Pura Compuesta)**.** *Sean* F *un bifunctor bitraversable,* C *un functor aplicativo,* $f :\ A \to B$ *y* $\phi : \mathsf{F_A}\ X \to X$ *un* $\mathsf{F_A}$*-álgebra pura, entonces:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ (pure_{\mathsf{C}} \circ f) = \ pure_{\mathsf{C}} \circ \llparenthesis \phi \circ \mathsf{F}\ (f, \mathsf{id}) \rrparenthesis_{\mathsf{F_A}} \tag{\textbf{Pure.FoldMap}}$$

Expresada como diagrama conmutativo, la ecuación anterior resulta:

$$\begin{tikzcd}
\mu_{\mathsf{F_A}} \arrow[r, "\llparenthesis \phi \circ \mathsf{F}\ (f,\mathsf{id}) \rrparenthesis_{\mathsf{F_A}}"] \arrow[dr, bend right, "ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\, (pure_{\mathsf{C}} \circ f)"'] & X \arrow[d, "pure_{\mathsf{C}}"] \\
 & \mathsf{C}\ X
\end{tikzcd}$$

Teorema 4.7 (Fusión Ifold Morph)**.** *Sean* $\tau : \mathsf{C}\ \Rightarrow \mathsf{D}$ *un morfismo entre los functores aplicativos* C *y* D*,* $\iota : A \to \mathsf{C}\ B$ *una acción aplicativa y* $\phi : \mathsf{F_B}\ X \to X$ *un* $\mathsf{F_B}$*-álgebra pura, entonces:*

$$\tau \circ ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ \iota = ifold^{\mathsf{D}}_{\mathsf{F_A}} \phi\ (\tau \circ \iota) \tag{\textbf{Ifold.Morph}}$$

Expresada como diagrama conmutativo, la ecuación anterior resulta:

$$\begin{tikzcd}
\mu_{\mathsf{F_A}} \arrow[r, "ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ \iota"] \arrow[dr, bend right, "ifold^{\mathsf{D}}_{\mathsf{F_A}} \phi\ (\tau \circ \iota)"'] & \mathsf{C}\ X \arrow[d, "\tau_X"] \\
 & \mathsf{D}\ X
\end{tikzcd}$$

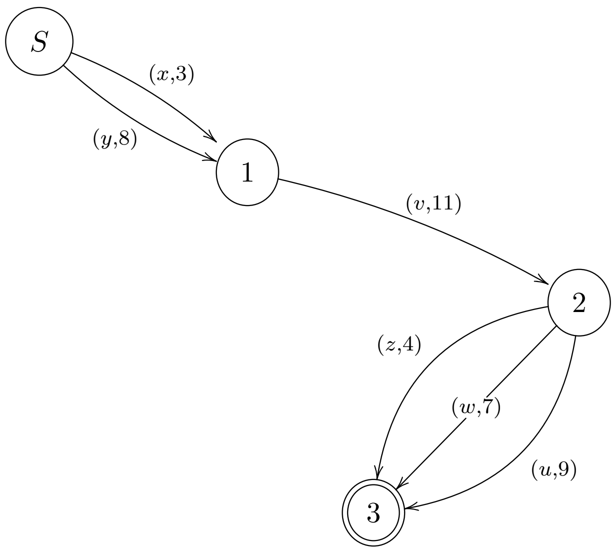
Si tenemos en cuenta la definición de *traverse* en funcíon de *ifold* (**Trv.Def**), obtenemos como corolario la siguiente propiedad para *traverse*:

Corolario 4.8. *Sean $\tau : \mathsf{C} \Rightarrow \mathsf{D}$ un momorfismo entre los functores aplicativos C y D, y $\iota : A \to \mathsf{C}\ B$ una acción aplicativa, entonces:*

$$\tau \circ traverse_{\mathsf{F_A}}\ \iota = traverse_{\mathsf{F_A}}\ (\tau \circ \iota) \qquad \textbf{(Trv.Morph)}$$

Ejemplo 4.8 (Caminos Maximales en Multiárboles)**.** Un vendedor ambulante debe elegir la ruta del día de acuerdo a la información estadística que posee sobre la ganancia que aportan los diferentes caminos disponibles en su área. Esta ruta comenzará en un punto fijo de partida, *el centro de distribución*, podra atravesar algunos puntos intermedios de control, *checkpoints*, y deberá terminar en alguno de los centros de devolución, *end-points.*

La topología del área asignada al vendedor se ha codificado en un grafo bastante particular, un *multigrafo acíclico dirgido*, id est un *bosque dirigido* con posiblemente más de un arco entre el mismo par de nodos. El nodo que representa al centro de distribución tendrá un sólo hijo. Para mayor simplicidad, el resto de los nodos intermedios será binario, es decir que tendran a lo sumo dos *nodos hijos* por cada *nodo padre.* Los arcos tiene un identificador, y pesos que representan la ganancia estimada para esa ruta. Veamos el siguiente ejemplo:



Queremos resolver aplicativamente el problema de encontrar la ruta que maximice las potenciales ganancias y calcular su potencial valor. Para esto consideremos el functor aplicativo *monádico* de las listas presentado en la Sección 2.5, que modela el efecto de *no determinismo o alternativa no determinista.* Usaremos los árboles binarios *Bin a* de ejemplos anteriores para modelar el grafo, siguiendo las siguientes pautas:

- Los nodos del árbol representaran los *checkpoints* o *endpoints.*
- No representaremos el punto de origen S esto implica que la cabeza del *Bin a* representa al primer nodo accesible desde el centro de distribución.
- El dato almacenado en los nodos serán los posibles caminos que conducen a él, junto con sus respectivos pesos. Es decir, un valor del siguiente tipo: $[(String, Int)]$.
- Un *endpoint* se representa por un nodo cuyos 2 hijos sean *Leafs*, e.g. *Node Leaf xs Leaf*.
- Un nodo cuyo dato almacenado sea la lista $[\,]$ representará a un nodo que no se puede acceder desde su padre, i.e. perteneciente a una diferente componente conexa. Representará un grafo mal formado y el algoritmo deberá fallar en este caso.

Dadas estas pautas, la implementación del ejemplo resulta:

$$
\begin{array}{l}
eg1 :: Bin\ [(String, Int)] \\
eg1 = Node\ Leaf \\
\qquad\qquad [(\texttt{"x"}, 3), (\texttt{"y"}, 8)] \\
\qquad\qquad (Node\ (Node\ Leaf \\
\qquad\qquad\qquad\qquad [(\texttt{"z"}, 4), (\texttt{"w"}, 7), (\texttt{"u"}, 9)] \\
\qquad\qquad\qquad\qquad Leaf) \\
\qquad\qquad\quad [(\texttt{"v"}, 11)] \\
\qquad\qquad\quad Leaf)
\end{array}
$$

Describiremos el algoritmo aplicativo como una composición de etapas modulares y luego utilizaremos las propiedades de ifold para calcular una versión eficiente del algoritmo. Las etapas resultan:

1. Transformar el *multiárbol binario* en una lista con todas las combinaciones posibles de árboles binarios.
2. Calcular para cada uno de ellos, el camino máximal, retornando un par $(String, Int)$ donde el valor de tipo $String$ representa al camino maximal y el valor entero su ganancia.
3. Elegir el mejor de todos ellos mediante una función $max :: [(t, t1)] \rightarrow Maybe\ (t, t1)$ que nos devuelve finalmente la mejor ruta, si esta existe.

La primer etapa se implementa con la distribución para Bin y el functor aplicativo $[\,]$, $dist^{[\,]}_{Bin}$. Esta distribución resulta un *recorrido* con la función identidad:

$$
\begin{array}{l}
stage1 :: Bin\ [(String, Int)] \rightarrow [Bin\ (String, Int)] \\
stage1 = traverse\ id
\end{array}
$$

Por la definición del functor aplicativo *monádico* $[\,]$, ver pág. 6, en el caso de representar el árbol binario a un grafo mal formado, esto es con una lista vacía almacenada en algún nodo, la distribución dará como resultado la lista vacía. Es importante destacar que esta etapa hace al algoritmo muy ineficiente dado que generar todas las posibles combinaciones es muy costoso, tanto espacial como temporalmente.

La segunda etapa consiste en calcular para cada árbol de la lista el el camino maximal. Consideramos la función $maxPath :: Bin\ (String, Int) \rightarrow (String, Int)$ que encuentra el camino máximal, la misma puede ser descripta fácilmente con $foldB$. Entonces la segunda etapa es un map de $maxPath$ sobre una lista de árboles.

$$
\begin{array}{l}
maxPath :: Bin\ (String, Int) \rightarrow (String, Int) \\
maxPath = foldBin\ (\lambda(xl, nl)\ (x, n)\ (xr, nr) \rightarrow \mathbf{if}\ (nl \geqslant nr) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ (x +\!\!+\ xl, nl + n) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ (x +\!\!+\ xr, nr + n)) \\
\qquad\qquad\qquad (\texttt{""}, 0) \\
stage2 :: [Bin\ (String, Int)] \rightarrow [(String, Int)] \\
stage2 = map\ maxPath
\end{array}
$$

Si consideramos la composición de las 2 primeras etapas encontramos el patrón que caracteriza a un ifold. Reemplazar a esta composición por un ifold eliminará la necesidad de generar la lista con todos los posibles árboles, mejorando la eficiencia del algoritmo considerablemente:

$$\begin{array}{l}
\mathit{stage12} :: \mathit{Bin}\ [(\mathit{String}, \mathit{Int})] \to [(\mathit{String}, \mathit{Int})] \\
\mathit{stage12} = \mathit{ifoldB}\ (\lambda(\mathit{xl}, \mathit{nl})\ (x, n)\ (\mathit{xr}, \mathit{nr}) \to \mathbf{if}\ (\mathit{nl} \geqslant \mathit{nr}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ (x +\!\!+\ \mathit{xl}, \mathit{nl} + n) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ (x +\!\!+\ \mathit{xr}, \mathit{nr} + n)) \\
\qquad\qquad\qquad\quad (\texttt{""}, 0) \\
\qquad\qquad\qquad\quad \mathit{id}
\end{array}$$

Para la tercera etapa, consideramos la función *max* que nos retorna el mejor camino, entre los caminos maximales, si estos existen.

$$\begin{array}{ll}
\mathit{max} & :: [(\mathit{String}, \mathit{Int})] \to \mathit{Maybe}\ (\mathit{String}, \mathit{Int}) \\
\mathit{max}\ [\,] & = \mathit{Nothing} \\
\mathit{max}\ (p@(x, n) : \mathit{ps}) & = \mathbf{case}\ (\mathit{max}\ \mathit{ps})\ \mathbf{of} \\
 & \qquad \mathit{Just}\ \mathit{p2}@(y, z) \to \mathbf{if}\ (n > z)\ \mathbf{then}\ (\mathit{Just}\ p)\ \mathbf{else}\ \mathit{Just}\ \mathit{p2} \\
 & \qquad \mathit{Nothing} \qquad\quad\ \to \mathit{Just}\ p
\end{array}$$

Componiendo esta última etapa con la función *stage21* completaremos finalmente el algoritmo:

$$\begin{array}{ll}
\mathit{maxRputeB1} & :: \mathit{Bin}\ [(\mathit{String}, \mathit{Int})] \to \mathit{Maybe}\ (\mathit{String}, \mathit{Int}) \\
\mathit{maxRoute} & = \mathit{max} \circ \mathit{stage12}
\end{array}$$

De la definición de *max* se puede comprobar que esta función es un *morfismo* entre los functores aplicativos [ ] y *Maybe*, verificándose las condiciones impuestas por las ecuaciones (**Morph.***pure*) y (**Morph.**$\circledast$). Aplicando la propiedad **Ifold.Morph** a *maxRoute*, obtenemos la expresión eficiente en función de *ifoldB*:

$$\begin{array}{l}
\mathit{maxRoute} = \mathit{ifoldB}\ (\lambda(\mathit{xl}, \mathit{nl})\ (x, n)\ (\mathit{xr}, \mathit{nr}) \to \mathbf{if}\ (\mathit{nl} \geqslant \mathit{nr}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ (x +\!\!+\ \mathit{xl}, \mathit{nl} + n) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ (x +\!\!+\ \mathit{xr}, \mathit{nr} + n)) \\
\qquad\qquad\qquad\quad (\texttt{""}, 0) \\
\qquad\qquad\qquad\quad \mathit{max}
\end{array}$$

Teorema 4.9 (*Acid Rain*: Fusión Map Fold.Ifold)**.** *Sean* F, G *dos bifunctores bitraversables,* C *un functor aplicativo, y* $\mathsf{T_B} : \forall X.(\mathsf{F_B}\ X \to X) \to (\mathsf{G_B}\ X \to X)$ *un transformador de* $\mathsf{F_B}$*-álgebras, entonces:*

$$\begin{array}{l}
\qquad \psi = \mathsf{T_B}\ \mathit{in}_{\mathsf{F_B}} \\
\Longrightarrow \\
\qquad \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}} \circ \mathit{ifold}^{\mathsf{C}}_{\mathsf{G_A}} \psi\ \iota = \mathit{ifold}^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ \phi)\ \iota
\end{array} \qquad (\textbf{Map Fold.Ifold})$$

Es decir, el siguiente diagrama conmuta:

$$\begin{tikzcd}
\mu_{\mathsf{G_A}} \arrow[r, "\mathit{ifold}^{\mathsf{C}}_{\mathsf{G_A}} \psi\ \iota"] \arrow[dr, bend right, "\mathit{ifold}^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ \phi)\ \iota"'] & \mathsf{C}\ \mu_{\mathsf{F_B}} \arrow[d, "\mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}}"] \\
 & \mathsf{C}\ X
\end{tikzcd}$$

Esta propiedad resulta una generalización de la *Caracterización de Ifold*, (**Ifold.Charn**). Ilustraremos su uso con algunos ejemplos.

Para que sea más claro inferir las definiciones de los *transformadores de* $\mathsf{F}_{(\_)}$*-álgebras*, escribiremos los ejemplos usando la representación genérica de los tipos de datos y las implementaciones genéricas de *ifold, traverse y map.*

Ejemplo 4.9. Consideremos la función $checkBin :: (a \to Bool) \to \mu\ (B\ a) \to Maybe\ (\mu\ (B\ a))$ que verifica que un árbol binario cumpla con una propiedad esperada. Esta puede expresarse fácilmente en función del *traverse* genérico.

$$\begin{array}{l} checkBin\quad :: (a \to Bool) \to \mu\ (B\ a) \to Maybe\ (\mu\ (B\ a)) \\ checkBin\ p = traverse\ (\lambda x \to \mathbf{if}\ (p\ x)\ \mathbf{then}\ (Just\ x)\ \mathbf{else}\ Nothing) \end{array}$$

Del resultado de este recorrido nos interesa quedarnos con algunos de los valores del árbol, e.g. los hijos derechos de cada nodo, para almacenarlos en una lista. Definimos esta función, a la que llamaremos *rightProp*, como el *map* de un *fold* puro compuesto con *checkBin*.

$$\begin{array}{l} rightProp\quad :: (a \to Bool) \to \mu\ (B\ a) \to Maybe\ (\mu\ (L\ a)) \\ rightProp\ p = fmap\ (fold\ \psi) \circ checkBin\ p \\ \quad \mathbf{where}\ \psi\ BL \qquad\quad\ = In\ N \\ \qquad\qquad \psi\ (BN\ t1\ x\ t2) = In\ (C\ x\ t2) \end{array}$$

Caracterizando a esta función como un *ifold*, obtenemos la expresión:

$$\begin{array}{l} rightProp\quad :: (a \to Bool) \to \mu\ (B\ a) \to Maybe\ (\mu\ (L\ a)) \\ rightProp\ p = ifold\ \psi\ (\lambda x \to \mathbf{if}\ (p\ x)\ \mathbf{then}\ (Just\ x)\ \mathbf{else}\ Nothing) \\ \quad \mathbf{where}\ \psi\ BL \qquad\quad\ = In\ N \\ \qquad\qquad \psi\ (BN\ t1\ x\ t2) = In\ (C\ x\ t2) \end{array}$$

Finalmente, consideramos que el árbol binario tiene valores enteros y que nos interesa sumar aquellos valores almacenados en la lista que genera *rightProp* dentro del efecto. Expresada composicionalmente, resulta:

$$\begin{array}{l} sumRightProp :: (Int \to Bool) \to \mu\ (B\ Int) \to Maybe\ Int \\ sumRightProp\ p = fmap\ (fold\ \zeta) \circ rightProp\ p \\ \quad \mathbf{where}\ \zeta\ N \qquad\ = 0 \\ \qquad\qquad \zeta\ (C\ x\ xs) = x + xs \end{array}$$

Aplicando la propiedad (**Map Fold.Ifold**), se obtiene la siguiente expresión deforestada, evitando la generación de la lista intermedia:

$$\begin{array}{l} sumRightProp :: (Int \to Bool) \to \mu\ (B\ Int) \to Maybe\ Int \\ sumRightProp\ p = ifold\ \phi\ (\lambda x \to \mathbf{if}\ (p\ x)\ \mathbf{then}\ (Just\ x)\ \mathbf{else}\ Nothing) \\ \quad \mathbf{where}\ \phi\ BL \qquad\quad\ = 0 \\ \qquad\qquad \phi\ (BN\ t1\ x\ t2) = x + t2 \end{array}$$

Este último paso es correcto si se verifica la precondición del teorema i.e. si podemos expresar al $\mathsf{F_A}$-álgebra ψ como la aplicación de un *transformador de* L_{Int}*-álgebras* W_{Int} a $in_{\mathsf{L}_{Int}}$ de modo tal que también resulte $\mathsf{W}_{Int}\ \zeta\ =\ \phi$. Derivamos la definición de W_{Int} a partir de la definición de ψ:

$$\begin{array}{l} \psi\ (BN\ t1\ x\ t2) = In\ (C\ x\ t2) \\ \psi\ BL \qquad\qquad\ = In\ N \end{array}$$

$$
\begin{array}{l}
\equiv \{\ \eta\text{-conversión}\ \} \\
\quad \psi\ (BN\ t1\ x\ t2) = (\lambda\quad (h_1, h_2) \to h_1\ x\ t2)\ (In\ (\lambda y\ ys \to C\ y\ ys), In\ N) \\
\quad \psi\ BL \qquad\qquad\quad = (\lambda\quad (h_1, h_2) \to h_2) \qquad\quad (In\ (\lambda y\ ys \to C\ y\ ys), In\ N) \\
\equiv \{\ \textit{Eureka: } \psi = \mathsf{W}_{Int}\ in_{\mathsf{L}_{Int}}\ \} \\
\quad \psi\ (BN\ t1\ x\ t2) = \mathsf{W}_{Int}\ (In\ (\lambda y\ ys \to C\ y\ ys), In\ N)\ (BN\ t1\ x\ t2) \\
\quad \psi\ BL \qquad\qquad\quad = \mathsf{W}_{Int}\ (In\ (\lambda y\ ys \to C\ y\ ys), In\ N)\ BL \\
\Leftarrow \{\ \text{Definición de } \mathsf{W}_{Int}\ \} \\
\quad \mathsf{W}_{Int}\ (f, e) = \lambda t \to \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad BN\ t1\ x\ t2 \to f\ x\ t2 \\
\qquad\qquad\qquad\qquad\qquad\qquad BL \qquad\qquad \to e \quad {\scriptstyle q.e.d.}
\end{array}
$$

Para finalizar, nos resta verificar que con este transformador se obtiene ϕ:

$$
\begin{array}{l}
\quad \mathsf{W}_{Int}\ \zeta \\
\equiv \{\ \text{Definiciones de } \mathsf{W}_{Int} \text{ y } \zeta\ \} \\
\quad \lambda t \to \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad\qquad BN\ t1\ x\ t2 \to (\lambda x\ xs \to x + xs)\ x\ t2 \\
\qquad\qquad\qquad BL \qquad\qquad \to 0 \\
\equiv \{\ \beta\text{-conversión}\ \} \\
\quad \lambda t \to \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad\qquad BN\ t1\ x\ t2 \to x + t2 \\
\qquad\qquad\qquad BL \qquad\qquad \to 0 \\
\equiv \{\ \text{Definición de } \phi\ ,\ \textit{eliminación} \text{ de } \textbf{case}\ \} \\
\quad \phi \quad \mathsf{Q.E.D.}
\end{array}
$$

En las últimas propiedades presentadas, hemos fusionado funciones aplicadas al resultado de un *ifold*. Para el caso dual, nos interesa estudiar alguna propiedad de *Acid Rain* para *Ifold*, similar a la presentada en el Capítulo 3 para *Fold*.

Sin embargo, la presencia de efectos hace que ésta no sea inmediata, dado que debemos asegurarnos que los efectos paramétricos producidos sean los mismos tanto para la expresión composicional como para la expresión *deforestada*. El hecho de que los efectos sean *paramétricos* hace que esto ocurra si la *cantidad de parámetros* se preserva con la aplicación de la primer función.

La primer opción es considerar que la función que se aplica primero es un *map* de un *functor de tipo*. La acción de flechas de un functor preservan la estructura, por lo que la *cantidad de parámetros* se mantiene.

Teorema 4.10 (Fusión Ifold Map)**.** *Sean* F *un bifunctor bitraversable y* $\mathsf{D_F}$ *el functor de tipo asociado a éste. Sea* $f : X \to A$*, entonces:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\, \iota \circ \mathsf{D_F}\ f = ifold^{\mathsf{C}}_{\mathsf{F_X}} \phi\, (\iota \circ f) \qquad (\textbf{Ifold.Type})$$

Nos interesa generalizar *map* a un *fold* aleatorio, y además obtener algún resultado que nos permita preservar el efecto paramétrico, alterando sólamente el cálculo del valor puro. Por lo tanto, debemos caracterizar bao qué condiciones condiciones resulta el álgebra *indpendiente* de la acción aplicativa:

Definición 4.6 ($\mathsf{G_A}$-álgebra Independiente de la Acción Aplicativa)**.** Sea ϕ un $\mathsf{G_A}$-álgebra y $\iota : A \to \mathsf{C}\ B$ una acción aplicativa. Entonces, si ϕ resulta $\phi = \mathsf{T_A}\ in_{\mathsf{F_A}}$ donde $\mathsf{T_A}$ es un transformador de $\mathsf{F_A}$-álgebras en $\mathsf{G_A}$-álgebras y además se cumple que:

$$\mathsf{T_A}\ (\mathsf{C}\ in_{\mathsf{F_B}} \circ \delta^{\mathsf{C}}_{\mathsf{F}} \circ \mathsf{F}\ (\iota, id)) = \mathsf{C}\ (\mathsf{T_B}\ in_{\mathsf{F_B}}) \circ \delta^{\mathsf{C}}_{\mathsf{G}} \circ \mathsf{G}\ (\iota, id) \qquad (\textbf{Alg.Ind})$$

decimos que ϕ es *independiente de los efectos* de ι.

Una consecuencia de esta noción de independencia de los efectos, es que se puede alterar el orden entre la generación de efectos paramétricos mediante *traverse* y la generación de valores puros mediante el $\mathsf{G_A}$-álgebra. Esto se traduce al siguiente *Lemma*:

Lemma 4.11. *Sean $\iota : A \to \mathsf{C}\ B$ una acción aplicativa y $\phi : \mathsf{G}\ (A, \mu\mathsf{F_A}) \to \mu\mathsf{F_A}$ un $\mathsf{G_A}$-álgebra independiente de los efectos de ι. Entonces, el siguiente diagrama conmuta:*

$$\begin{tikzcd}
\mu\mathsf{G_A} \arrow[r, "(\!|\phi|\!)_{\mathsf{G_A}}"] \arrow[d, "traverse_{\mathsf{G_A}}\ \iota"'] & \mu\mathsf{F_A} \arrow[d, "traverse_{\mathsf{F_A}}\ \iota"] \\
\mathsf{C}\ \mu\mathsf{G_B} \arrow[r, "\mathsf{C}\ (\!|\mathsf{T_B}\ in_{\mathsf{F_B}}|\!)_{\mathsf{G_B}}"'] & \mathsf{C}\ \mu\mathsf{F_B}
\end{tikzcd}$$

Ahora, podemos caracterizar las condiciones necesarias para enunciar una propiedad de *Acid Rain*:

Teorema 4.12 (Acid Rain: Fusión Ifold.Fold)**.** *Dados F y G dos bifunctores bitraversables, y C un functor aplicativo y $\iota : A \to \mathsf{C}\ B$ una acción aplicativa. Si phi es un $\mathsf{G_A}$-álgebra independiente de los efectos de ι, entonces:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} \psi\ \iota \circ (\!|\phi|\!)_{\mathsf{G_A}} = ifold^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ \psi)\ \iota \qquad \textbf{(Ifold.Fold)}$$

Ésta se puede expresar de forma equivalente, mediante el siguiente diagrama:

$$\begin{tikzcd}
\mu\mathsf{G_A} \arrow[r, "(\!|\phi|\!)_{\mathsf{G_A}}"] \arrow[dr, bend right, "ifold^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ \psi)\ \iota"'] & \mu\mathsf{F_A} \arrow[d, "ifold^{\mathsf{C}}_{\mathsf{F_A}} \psi\ \iota"] \\
 & \mathsf{C}\ X
\end{tikzcd}$$

. Ilustramos el uso de esta propiedad con algunos ejemplos:

Ejemplo 4.10. Consideramos la función *sumRecips* definida en el Ejemplo 4.6, expresada en función del *ifold* general. Vamos a componer esta función con un *map* que genere los cuadrados de una lista dada, obteniendo entonces la siguiente función:

$$\begin{aligned}
&sumSqrRecips :: [Float] \to Maybe\ Float \\
&sumSqrRecips = sumrecips \circ map\ (\uparrow 2)
\end{aligned}$$

Aplicando la propiedad de Acid Rain, (**Ifold.Fold**), se obtiene la siguiente expresión

$$\begin{aligned}
&sumSqrRecips :: [Float] \to Maybe\ Float \\
&sumSqrRecips = ifoldr\ (\lambda x\ y \to x \uparrow 2 + y)\ 0\ recip
\end{aligned}$$

Para probar la independencia del álgebra de *map* $(\uparrow 2)$ con respecto a *recip*, resulta conveniente expresar los álgebras y el transformador en función del bifunctor L. La derivación del transformador T es inmediata inspeccionando la definición genérica de *map* definida en la página 18.

$\mathsf{T_B} = (\lambda h \to h \circ bimap\ (\uparrow 2)\ id)$
$\Rightarrow$
$\mathsf{T_B}\ (fmap\ In \circ \delta \circ bimap\ recip\ id)$
$\equiv$ { Definición de $\mathsf{T_B}$ y *recip* }
$fmap\ In \circ \delta \circ bimap\ (\lambda x \to \mathbf{if}\ x \not\equiv 0\ \mathbf{then}\ Just\ 1\ /\ x\ \mathbf{else}\ Nothing)\ id \circ bimap\ (\uparrow 2)\ id$
$\equiv$ { Comp. Bifunctores, identidad }
$fmap\ In \circ \delta \circ bimap\ ((\lambda x \to \mathbf{if}\ x \not\equiv 0\ \mathbf{then}\ Just\ 1\ /\ x\ \mathbf{else}\ Nothing) \circ (\uparrow 2))\ id$
$\equiv$ { Álgebra, Propiedades *Maybe* }
$fmap\ In \circ \delta \circ bimap\ (fmap\ (\uparrow 2) \circ (\lambda x \to \mathbf{if}\ x \not\equiv 0\ \mathbf{then}\ (Just\ 1\ /\ x)\ \mathbf{else}\ Nothing))\ id$
$\equiv$ { Comp. Bifunctores, Def. *recip* }
$fmap\ In \circ \delta \circ bimap\ (fmap\ (\uparrow 2))\ id \circ bimap\ recip\ id$
$\equiv$ { Naturalidad de δ }
$fmap\ In \circ fmap\ (bimap\ (\uparrow 2)\ id) \circ \delta \circ bimap\ recip\ id$
$\equiv$ { Comp. Functores }
$fmap\ (In \circ bimap\ (\uparrow 2)\ id) \circ \delta \circ bimap\ recip\ id$
$\equiv$ { Definición de TB }
$fmap\ (TB\ In) \circ \delta \circ bimap\ recip\ id$ Q.E.D.

En las propiedades de *ifold* que hemos visto hasta ahora, se usaba la *composición* $\circ$ para combinar funciones definidas con los operadores *ifold, map y fold.* Si Consideramos los combinadores de acciones aplicativas $\odot$ y $\otimes$ presentados en el Capítulo 2, se obtienen las siguientes propiedades.

Teorema 4.13 (Acción Paralela)**.** *Sean* F, G *dos bifunctores bitraversables,* $\mathsf{C_1}, \mathsf{C_2}$ *dos functores aplicativos y* $\iota : A \to \mathsf{C_2}\ B,\ \kappa : A \to \mathsf{C_2}\ B$ *dos acciones aplicativas, entonces:*

$$ifold^{\mathsf{C_1}}_{\mathsf{F_A}} \phi\ \iota \otimes ifold^{\mathsf{C_2}}_{\mathsf{F_A}} \phi\ \kappa = ifold^{\mathsf{C_1} \times \mathsf{C_2}}_{\mathsf{F_A}} \phi\ (\iota \otimes \kappa) \qquad \textbf{(Ifold.}\otimes\textbf{)}$$

Teorema 4.14 (Composición Secuencial)**.** *Sean* F*y* G *dos bifunctores bitraversables,* $\mathsf{C_1}$ *y* $\mathsf{C_2}$ *dos functores aplicativos, y* $\iota : A \to \mathsf{C_2}\ B$ *y* $\kappa : B \to \mathsf{C_1}\ C$ *dos acciones aplicativas. Entonces, si* $\psi : \mathsf{F}\ (B, \mu\mathsf{G_B}) \to \mu\mathsf{G_B}$ *es un* $\mathsf{F_B}$*-álgebra independiente de los efectos de* κ *resulta:*

$$\psi = \mathsf{T_E}\ in_{\mathsf{G_E}}$$
$$\Longrightarrow$$
$$ifold^{\mathsf{C_1}}_{\mathsf{G_B}} \phi\ \kappa \odot ifold^{\mathsf{C_2}}_{\mathsf{F_A}} \psi\ \iota = ifold^{\mathsf{C_2} \bullet \mathsf{C_1}}_{\mathsf{F_A}} (\mathsf{T_E}\ \phi)\ (\kappa \odot \iota) \qquad \textbf{(Ifold.}\odot\textbf{)}$$

Si consideramos el caso particular donde el *ifold* que se aplica primero es un *recorrido*, se obtiene la siguiente propiedad como corolario:

Corolario 4.15 (Composición Secuencial Ifold.Traverse)**.** *Sea* F *un bifunctor bitraversable,* $\mathsf{C_1}, \mathsf{C_2}$ *dos functores aplicativos y* $\iota : A \to \mathsf{C_2}\, B,\ \kappa : B \to \mathsf{C_1}\, C$ *dos acciones aplicativas. Entonces:*

$$ifold^{\mathsf{C_1}}_{\mathsf{F_B}} \phi\, \kappa \odot traverse_{\mathsf{F_A}}\ \iota = ifold^{\mathsf{C_2} \bullet \mathsf{C_1}}_{\mathsf{F_A}} \phi\, (\kappa \odot \iota) \qquad \textbf{(Ifold}\odot\textbf{Trv)}$$

Capítulo 5

ifold para Efectos Incrementales

> Seeing, contrary to popular wisdom, isn't believing. It's where belief stops, because it isn't needed any more.
>
> Terry Pratchett, *Pyramids*

Vamos a ilustrar la utilidad del operador *ifold* mediante la construcción de un evaluador de expresiones aritméticas incremental. Comenzaremos con un evaluador simple y agregaremos nuevos efectos aprovechando la composición ($\bullet$) y combinación paralela ($\times$) de functores aplicativos.

5.1. Un evaluador simple

El punto de partida es un ejemplo presentado por McBride y Paterson [41]. Se construye un evaluador para un lenguaje de expresiones aritméticas con variables, donde se entreteje un entorno para asignarle valores a las variables libres. Introducimos el tipo de datos para las expresiones aritméticas y definimos al entorno como una lista de pares: identificador , valor entero:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Exp}\ a = \mathit{Var}\ a \mid \mathit{Val}\ \mathit{Int} \mid (\mathit{Exp}\ a) :+: (\mathit{Exp}\ a)\\
&\quad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}, \mathit{Ord})\\
&\textbf{type}\ \mathit{Env}\ a = [(a, \mathit{Int})]
\end{aligned}
$$

Procedemos a definir el evaluador, por recursión estructural sobre el árbol de expresiones aritméticas:

$$
\begin{array}{lll}
\mathit{eval} & & :: \mathit{Exp}\ a \to \mathit{Env}\ a \to \mathit{Int}\\
\mathit{eval}\ (\mathit{Var}\ x) & \gamma & = \mathit{fetch}\ x\ \gamma\\
\mathit{eval}\ (\mathit{Val}\ i) & \gamma & = i\\
\mathit{eval}\ (e1 :+: e2) & \gamma & = \mathit{eval}\ e1\ \gamma + \mathit{eval}\ e2\ \gamma
\end{array}
$$

Donde, *fetch* es la búsqueda en el entorno, definida como sigue:

$$
\begin{array}{ll}
\mathit{fetch} & :: (\mathit{Eq}\ a) \Rightarrow a \to \mathit{Env}\ a \to \mathit{Int}\\
\mathit{fetch}\ s\ ((t, x) : \mathit{xss}) & = \textbf{if}\ (s \equiv t)\ \textbf{then}\ x\ \textbf{else}\ \mathit{fetch}\ s\ \mathit{xss}
\end{array}
$$

Vamos a reescribir las ecuaciones de *eval*, para que tengan el siguiente tipo : $Env\ a \to Int$, abstrayendo el entorno. Para eso recurrimos a los combinadores $\mathbb{K}$ y $\mathbb{S}$:

$$\begin{array}{ll} \mathbb{K} & :: a \to env \to a \\ \mathbb{K}\ x\ \gamma = x & \\ \mathbb{S} & :: (env \to a \to b) \to (env \to a) \to (env \to b) \\ \mathbb{S}\ ef\ \ es\ \gamma = (ef\ \gamma)\ (es\ \gamma) & \end{array}$$

$$\begin{array}{ll} eval & :: Exp\ a \to Env\ a \to Int \\ eval\ (Var\ x) & = fetch\ x \\ eval\ (Val\ i) & = \mathbb{K}\ i \\ eval\ (e1 :+: e2) & = \mathbb{K}\ (+)\ `\mathbb{S}`\ eval\ e1\ `\mathbb{S}`\ eval\ e2 \end{array}$$

El tipo de datos $(\to)\ (Env\ a)$ es un functor aplicativo, donde $\mathbb{K}$ y $\mathbb{S}$ son respectivamente *pure* y $\circledast$. Esto no debería sorprendernos, dado que $(\to)\ (Env\ a)$ es una mónada de entorno i.e. *enviroment monad o reader monad.* Damos la instancia respectiva, dejando pendiente probar las leyes asociadas al functor aplicativo.

$$\begin{array}{l} \textbf{instance}\ Functor\ ((\to)\ (Env\ a))\ \textbf{where} \\ \quad fmap\ g\ fs = (g \circ fs) \\ \textbf{instance}\ Applicative\ ((\to)\ (Env\ a))\ \textbf{where} \\ \quad pure = \mathbb{K} \\ \quad (\circledast)\ \ = \mathbb{S} \end{array}$$

Reescribimos *eval* reflejando la interfaz aplicativa:

$$\begin{array}{ll} eval\ :: (Eq\ a) \Rightarrow Exp\ a \to Env\ a \to Int & \\ eval\ (Var\ x) & = fetch\ x \\ eval\ (Val\ i) & = pure\ i \\ eval\ (e1 :+: e2) & = pure\ (+) \circledast eval\ e1 \circledast eval\ e2 \end{array}$$

Nos proponemos mostrar que *eval* verifica el patrón de recursión aplicativa que captura *ifold*. Para derivar *ifold*, necesitamos interpretar el tipo de datos de las expresiones aritméticas como punto fijo de su signatura. Ésta esta dada por el siguiente *bifunctor paramétrico*:

$$\begin{array}{l} \textbf{data}\ E\ a\ b = B\ a \mid V\ Int \mid A\ b\ b \\ \textbf{instance}\ Bifunctor\ E\ \textbf{where} \\ \quad bimap\ f\ g\ (A\ x\ y) = A\ (g\ x)\ (g\ y) \\ \quad bimap\ f\ g\ (B\ v)\ \ \ = B\ (f\ v) \\ \quad bimap\ f\ g\ (V\ i)\ \ \ = V\ i \end{array}$$

Para definir *traverse*, necesitamos que el bifunctor sea *bitraversable*:

$$\begin{array}{l} \textbf{instance}\ BiTraversable\ E\ \textbf{where} \\ \quad \delta\ (V\ i)\ \ \ = pure\ (V\ i) \\ \quad \delta\ (B\ x)\ \ \ = pure\ B \circledast x \\ \quad \delta\ (A\ x\ y) = pure\ A \circledast x \circledast y \end{array}$$

Observamos que en la definición de *eval* la *acción aplicativa* es la búsqueda del identificador x en el entorno, siendo efectivamente su tipo $a \to (Env\ a \to Int) \equiv a \to C\ Int$. Si consideramos un *traversal* con ésta, el resultado sería una función que dado un entorno, reemplazaría en el árbol los identificadores de variables por los valores de sus contextos.

Finalmente, podríamos combinar con ese traversal un E_{Int}-álgebra pura $\eta : \mathsf{E}_{Int}\ X \to X$ que evalúa un árbol de expresiones aritméticas en el que el dato parámetrico no son los identificadores, sino los valores que éstos tenían asignados en el entorno. Dicho $\mathsf{E}_{\mathsf{Int}}$-álgebra se expresa como:

$$
\begin{aligned}
&\eta && :: E\ Int\ Int \to Int \\
&\eta\ (V\ i) && = i \\
&\eta\ (B\ x) && = x \\
&\eta\ (A\ x\ y) && = x + y
\end{aligned}
$$

Obtenemos entonces la siguiente especificación para el evaluador:

$$eval = fmap\ (fold\ \eta) \circ traverse\ fetch$$

Partiendo de esta especificación, utilizamos la propiedad de caracterización **Ifold.Charn** para obtener la definición final de *eval* expresado en términos de *ifold*:

$$
\begin{aligned}
&eval :: (Eq\ a) \Rightarrow \mu\ (E\ a) \to Env\ a \to Int \\
&eval = ifold\ \eta\ fetch
\end{aligned}
$$

5.2. Un evaluador con soporte de fallas

La versión anterior de *eval* resulta impráctica puesto que no está definido que ocurre si la variable no se encuentra en el entorno. Introduciremos la posibilidad de falla mediante el uso del functor aplicativo *Maybe*. El functor aplicativo entonces será la la composición de $(Env\ a \to)$ y *Maybe*:

$$\textbf{type}\ ME\ a\ x = (((\to)\ (Env\ a)) \bullet Maybe)\ x$$

Debemos redefinir la búsqueda en el contexto, incorporando la posibilidad de falla: si la variable buscada no se encuentra en el entorno el valor será *Nothing*.

$$
\begin{aligned}
&lookup && :: (Eq\ a) \Rightarrow a \to ME\ a\ Int \\
&lookup\ x && = Comp\ (foldr\ (\lambda(y,z)\ mz \to \textbf{if}\ (x \equiv y)\ \textbf{then}\ Just\ z\ \textbf{else}\ mz)\ Nothing)
\end{aligned}
$$

Ahora debemos dar la definición para el E_{Int} -álgebra. Observamos que esta es independiente del efecto aplicativo, y entonces no cambia. La definición del evaluador resulta entonces:

$$
\begin{aligned}
&wrappedEval :: (Eq\ a) \Rightarrow \mu\ (E\ a) \to ME\ a\ Int \\
&wrappedEval = ifold\ \phi\ lookup \\
&\quad \textbf{where}\ \phi\ (V\ i) \quad = i \\
&\qquad\qquad\ \ \phi\ (B\ x) \quad = x \\
&\qquad\qquad\ \ \phi\ (A\ x\ y) = x + y \\
&eval :: (Eq\ a) \Rightarrow \mu\ (E\ a) \to Env\ a \to Maybe\ Int \\
&eval = \quad (unComp \circ wrappedEval)
\end{aligned}
$$

Dejando de lado la necesidad de desempaquetar el evaluador, como consecuencia de la implementación de la composición de functores, observamos que el único cambio con respecto a la definición anterior es adaptar la *acción aplicativa* de *búsqueda en el contexto* para que soporte fallas.

5.3. Acumuladores: Contando usos de variables

Vamos a combinar los functores aplicativos de *entorno*, *Maybe* y un acumulador de listas de pares, donde acumularemos las variables que fueron leídas del entorno, y la cantidad de veces que fue usada. Debemos tener en cuenta que si se usa la composición de acumuladores con otro functor aplicativo no obtendremos nada útil, dado que no se puede examinar el *phantom type*. Entonces, debemos usar el producto entre el acumulador y Maybe, *id est* el functor aplicativo será:

$$\textbf{type}\ CE\ a\ x = (((\to)\ (Env\ a)) \bullet (Acc\ (MonP\ a) \times Maybe))\ x$$

Debemos mostrarle al sistema de clases de Haskell que las listas de pares definen efectivamente un monoide:

$$
\begin{array}{l}
\textbf{newtype}\ MonP\ a = MonP\ \{monp :: [(a, Int)]\} \\
\quad \textbf{deriving}\ Show \\
\textbf{instance}\ (Eq\ a) \Rightarrow Monoid\ (MonP\ a)\ \textbf{where} \\
\quad \varnothing \qquad\qquad\qquad\qquad\quad = MonP\ [\,] \\
\quad (MonP\ xs) \oplus (MonP\ ys) = MonP\ (foldr\ insert\ xs\ ys) \\
\qquad \textbf{where}\ insert\ p\ [\,] \qquad\qquad = [p] \\
\qquad\quad insert\ (x,c)\ (p@(y,n) : ys) = \textbf{if}\ (x \equiv y)\ \textbf{then}\ (x, c+n) : ys \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ p : insert\ (x,c)\ ys
\end{array}
$$

Procedemos entonces a definir entonces la acción aplicativa para este efecto. Para ello modificaremos la búsqueda en el entorno para que cuando se encuentra el identificador buscado, obtenemos el valor e incrementamos en uno la incidencia del mismo:

$$
\begin{array}{l}
lookup \quad\ :: (Eq\ a) \Rightarrow a \to CE\ a\ Int \\
lookup\ x\ \ = Comp\ (foldr\ (\lambda(y,z)\ mz \to \textbf{if}\ (x \equiv y) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (Prod\ (Acc\ (MonP\ [(x,1)]))\ (Just\ z)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ mz) \\
\qquad\qquad\qquad\qquad (Prod\ (Acc\ \varnothing)\ Nothing)) \\
wrappedEval :: (Eq\ a) \Rightarrow \mu\ (E\ a) \to CE\ a\ Int \\
wrappedEval = ifold\ \phi\ lookup \\
\quad \textbf{where}\ \phi\ \ (V\ i) \quad = i \\
\qquad\qquad \phi\ \ (B\ x) \quad = x \\
\qquad\qquad \phi\ \ (A\ x\ y) = x + y \\
eval \qquad\ :: (Eq\ a) \Rightarrow \mu\ (E\ a) \to Env\ a \to (MonP\ a, Maybe\ Int) \\
eval\ t\ env = \langle acc \circ \pi_1, \pi_2 \rangle\ ((unComp\ (wrappedEval\ t))\ env)
\end{array}
$$

Hemos comprobado que el uso del operador *ifold* permite un diseño modular e incremental de algoritmos con efectos aplicativos, donde a partir de una función con efectos simples, se pueden combinar efectos aprovechando diferentes formas de combinar functores aplicativos, obteniendo funciones con efectos más complejos. Particularmente, destacamos como este desarrollo incremental no afecta el cálculo del valor puro: en las diferentes versiones de *eval* que hemos implementado, el álgebra pura es invariante, sólo la acción aplicativa ι cambia en cada diseño, a medida que cambia el efecto aplicativo.

5.4. Combinadores de acciones aplicativas

En el desarrollo incremental del evaluador, hemos utilizado la composición y producto de functores aplicativos para obtener functores aplicativos que modelen efectos más complejos.

Pero, en cada instancia hemos redefinido las acciones aplicativas en forma monolítica. Vamos a explorar el uso de los combinadores de *acciones aplicativas* $\odot, \otimes$ presentados en la Sección 2.2.

Utilizaremos el functor aplicativo *identidad* para recuperar el valor almacenado en el entorno. Entonces, procedemos a descomponer el efecto aplicativo de la siguiente forma: un efecto computacional de *búsqueda en un entorno con soporte de fallas*, modelado por el functor aplicativo $(\rightarrow)$ $Env\ a \bullet Maybe$, compuesto con el producto del acumulador del monoide de pares que hemos utilizado anteriormente con el functor identidad.

El functor aplicativo queda entonces definido por el siguiente tipo de datos:

$$\mathbf{type}\ CX\ a\ x = (((\rightarrow)\ (Env\ a) \bullet Maybe) \bullet (Acc\ (MonP\ a) \times Identity))\ Int$$

Nos proponemos escribir la acción aplicativa para el evaluador combinando acciones aplicativas que representen a cada efecto por separado. Consideramos la búsqueda en el entorno con fallas con una acción monolítica que nos devuelve el par identificador - valor y la acción que cuenta el uso de un identificador.

$$\begin{array}{ll}
lookup & :: (Eq\ a) \Rightarrow a \rightarrow Env\ a \rightarrow Maybe\ (a, Int) \\
lookup\ str\ [\,] & = Nothing \\
lookup\ str\ ((x, y) : xs) & = \mathbf{if}\ (x \equiv str)\ \mathbf{then}\ Just\ (x, y)\ \mathbf{else}\ lookup\ str\ xs
\end{array}$$

$$lookupMP = Comp \circ lookup$$

$$\begin{array}{l}
accBinder :: a \rightarrow Acc\ (MonP\ a)\ x \\
accBinder\ x = (Acc\ (MonP\ [(x, 1)]))
\end{array}$$

Combinando estas acciones, obtenemos la siguiente definición para el evaluador:

$$\begin{array}{l}
wrappedEval :: (Eq\ a) \Rightarrow \mu\ (E\ a) \rightarrow CX\ a\ Int \\
wrappedEval = ifold\ \phi\ (((accBinder \circ fst) \otimes (Id \circ snd)) \odot lookupMP) \\
\quad \mathbf{where}\ \phi\ (V\ i) \quad = i \\
\qquad\qquad \phi\ (B\ x) \quad = x \\
\qquad\qquad \phi\ (A\ x\ y) = x + y \\
eval \qquad :: (Eq\ a) \Rightarrow \mu\ (E\ a) \rightarrow Env\ a \rightarrow Maybe\ (MonP\ a, Int) \\
eval\ t\ env = fmap\ \langle acc \circ \pi_1, unId \circ \pi_2 \rangle \\
\qquad\qquad\qquad ((unComp \circ unComp \circ wrappedEval)\ t\ env)
\end{array}$$

Vemos entonces que el uso del operador *ifold* no solo nos permite un diseño incremental de un evaluador con diferentes efectos aplicativos, sino que además, mediante el uso de combinadores de acciones aplicativas podemos expresar las acciones aplicativas de efectos complejos en función de acciones aplicativas más simples, favoreciendo diseños modulares y flexibles.

Capítulo 6

Conclusiones y Trabajos Futuros

En este trabajo hemos estudiado las características de la recursión estructural en presencia de efectos aplicativos, presentado el operador genérico *ifold* que captura dicha recursión estructural sobre un tipo de datos recursivo cuya signatura está determinada por un *bifunctor bitraversable*.

Este operador, se deriva de una especificación composicional en la que intervienen un *fold* puro y un *recorrido* que introduce efectos aplicativos paramétricos mediante la función *traverse*. Este operador implementa eficientemente dicho patrón y además resulta una generalización del operador *traverse*, ya que permite definirlo como un caso particular de recursión aplicativa. Adicionalmente, hemos estudiado diversas propiedades *calculaciones* de *ifold*, enunciando y demostrando diferentes leyes de fusión para el operador que permiten *deforestar* expresiones composicionales aplicativas. Finalmente, hemos mostrado como como el operador *ifold* puede utilizarse para diseñar modularmente algoritmos con *efectos aplicativos* incrementales, agregando funcionalidad en cada etapa de forma sencilla y flexible.

Postulamos que el operador *ifold* captura la esencia de la *recursión aplicativa*. Cuando se interpreta intuitivamente a las operaciones del functor aplicativo como un mecanismo para aplicar una función pura a valores embebidos en un contexto con *efectos computacionales*, resulta natural que la *recursión aplicativa* se caracterice mediante un álgebra pura combinada *aplicativamente* con los efectos paramétricos independientes introducidos por el *recorrido* con una acción aplicativa.

Gibbons y Oliveira [22] sostienen que el operador *traverse* "captura la esencia del patrón *iterador*". Nuestro trabajo demuestra que los *traversals o recorridos* con este operador resultan fundamentales para la definición de la *recursión aplicativa*.

6.1. Trabajos Futuros

El trabajo realizado abre el camino a analizar las siguientes extensiones o trabajos futuros:

Prueba de la Inicialidad de *ifold* El operador *ifold*, tal cual lo hemos presentado en este trabajo, es definido en términos del *fold genérico*. Pero, para poder asegurar que este efectivamente es el *fold aplicativo* debemos modelar la categoría de los *álgebras aplicativos* y probar que *ifold* es el morfismo único y universal entre el *álgebra aplicativa inicial* y cualquier otro *álgebra aplicativa*.

Extensión de la noción de *Bifunctor Bitraversable* El operador introduce efectos mediante una acción aplicativa paramétrica que *mapeada* por sobre el functor bitraversable. Esto resulta en una limitación si pretendemos obtener diferentes efectos para un mismo parámetro, en el caso de consumir una estructura de datos que tiene diferentes constructores con valores del tipo del parámetro e.g. *red-black trees*. Si bien se podría evitar este problema trabajando con tipos de datos *isomorfos* con un único constructor paramétrico y una suma de los parámetros; o utilizar sistemas con tipos de datos más expresivos, como es el caso de los *GADTs* [32], o *tipos de datos algebraicos generalizados*, éstas no resolverían el problema de fondo. Para esto, proponemos resolver este problema extendiendo la noción de *bifunctor bitraversable* generalizando la cantidad de parámetros i.e. estudiar un concepto de *n-traversabilidad*.

***Unfold* e *Hylos* Aplicativos** Pardo [46, 47] presenta un esquema de correcursión monádica *unfold monádico*, dual al *fold monádico*, y un esquema de recursión general monádica o *hylomorfismo monádico*, junto a las propiedades algebraicas de estos esquemas. Siguiendo esta línea, nos interesa formular esquemas de correcursión y recursión general con functores aplicativos.

Fusión *Short Cut* Aplicativa La técnica de *Short Cut Fusion* [23, 24, 51], permite eliminar estructuras intermedias en programas funcionales modulares definidos como composición de *folds* con *buenos productores* de estructuras, definidos mediante el operador *build*. Recientemente, Ghani y Johann [18, 19, 33] y Manzino y Pardo [40] extendieron esta técnica para contextos con efectos *monádicos*. Nuestra intención es desarrollar una noción de *build aplicativo*, estudiar su comportamiento cuando es combinado con el operador *ifold* y derivar propiedades de fusión.

Apéndice A

Demostraciones de las propiedades de *ifold*

There Be Dragons!

A continuación, completamos las demostraciones de las propiedades de *Ifold* presentadas en el Capítulo 4.

Teorema A.1 (Acción Pura de Ifold)**.** *Sea* F *un bifunctor bitraversable,* C *un functor aplicativo y* $\phi : \mathsf{F_A}\ X \to X$ *entonces:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ pure_{\mathsf{C}} = \ pure_{\mathsf{C}} \circ (\!|\phi|\!)_{\mathsf{F_A}} \qquad \textbf{(Pure.Fold)}$$

Demostración:
Expandiendo el diagrama con la *Caracterización de Ifold* (**Ifold.Charn**), este resulta:

$$\begin{tikzcd}
\mu_{\mathsf{F_A}} \arrow[r, "(\!|\phi|\!)_{\mathsf{F_A}}"] \arrow[d, "traverse_{\mathsf{F}}\ pure_{\mathsf{C}}"'] \arrow[dr, bend right, "ifold^{\mathsf{D}}_{\mathsf{F_A}} \phi\ pure"] & X \arrow[d, "pure_{\mathsf{C}}"] \\
\mathsf{C}\ \mu_{\mathsf{F_A}} \arrow[r, "\mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_A}}"'] & \mathsf{C}\ X
\end{tikzcd}$$

$$\begin{array}{ll}
& ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ pure_{\mathsf{C}} \\
= & \qquad \{ \text{ Caracterización de } ifold^{\mathsf{C}}_{\mathsf{F_A}}\text{: } (\textbf{Ifold.Charn}) \ \} \\
& \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_A}} \circ traverse_{\mathsf{F}}\ pure_{\mathsf{C}} \\
= & \qquad \{ \text{ Identidad para } traverse[22] \ \} \\
& \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_A}} \circ pure_{\mathsf{C}} \\
= & \qquad \{ \text{ Propiedades } (\textbf{App.Map}),(\textbf{App.Homo})\} \\
& pure_{\mathsf{C}} \circ (\!|\phi|\!)_{\mathsf{F_A}} \quad \text{Q.E.D.}
\end{array}$$

La justificación del último paso se obtiene luego de interpretar en forma *point free* las siguientes propiedades conocidas de los functores aplicativos:

$$\begin{aligned} &\mathit{fmap}\ f\ x = \mathit{pure}\ f \circledast x && (\textbf{App.Map}) \\ &\mathit{pure}\ f \circledast \mathit{pure}\ x = \mathit{pure}\ (f\ x) && (\textbf{App.Homo}) \end{aligned}$$

Alternativamente, también se puede justificar este paso por la *naturalidad* de *pure*, si consideramos esta como una transformación natural entre el functor identidad y el functor aplicativo C i.e. $\mathit{pure}_{\mathsf{C}} : \mathbb{I} \Rightarrow \mathsf{C}$.

Teorema A.2 (Acción Pura Compuesta)**.** *Sean* F *un bifunctor bitraversable,* C *un functor aplicativo,* $f:\ A \to B$ *y* $\phi : \mathsf{F_A}\ X \to X$ *un* $\mathsf{F_A}$*-álgebra pura, entonces:*

$$\mathit{ifold}^{\mathsf{C}}_{\mathsf{F_A}} \phi\, (\mathit{pure}_{\mathsf{C}} \circ f) = \ \mathit{pure}_{\mathsf{C}} \circ (\!| \phi \circ \mathsf{F}\ (f, \mathsf{id}) |\!)_{\mathsf{F_A}} \qquad ((\textbf{Pure.FoldMap}))$$

Demostración:
Debemos probar que el siguiente diagrama conmuta:

$$\begin{tikzcd} \mu_{\mathsf{F_A}} \arrow[rr, "(\!|\phi \circ \mathsf{F}\ (f, \mathsf{id})|\!)_{\mathsf{F_A}}"] \arrow[drr, bend right, "\mathit{ifold}^{\mathsf{C}}_{\mathsf{F_A}} \phi\, (\mathit{pure}_{\mathsf{C}} \circ f)"'] & & X \arrow[d, "\mathit{pure}_{\mathsf{C}}"] \\ & & \mathsf{C}\ X \end{tikzcd}$$

$$\begin{aligned}
& \mathit{ifold}^{\mathsf{C}}_{\mathsf{F_A}} \phi\, (\mathit{pure}_{\mathsf{C}} \circ f) \\
= \ & \qquad \{ \text{ Caracterización de } \mathit{ifold}^{\mathsf{C}}_{\mathsf{F_A}}, (\textbf{Ifold.Charn}) \} \\
& \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}} \ \circ\ \mathit{traverse}_{\mathsf{F}}\ (\mathit{pure}_{\mathsf{C}} \circ f) \\
= \ & \qquad \{ \textit{ Free Theorem} \text{ de } \textit{traverse}, (\ \textbf{Traverse.Map}). \} \\
& \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}} \ \circ\ \mathit{traverse}_{\mathsf{F}}\ \mathit{pure}_{\mathsf{C}} \circ \mathsf{D_F}\ f \\
= \ & \qquad \{ \text{ Caracterización de } \mathit{ifold}^{\mathsf{C}}_{\mathsf{F_B}}, (\textbf{Ifold.Charn}) \} \\
& \mathit{ifold}^{\mathsf{C}}_{\mathsf{F_B}} \phi\, \mathit{pure}_{\mathsf{C}} \circ \mathsf{D_F}\ f \\
= \ & \qquad \{ \text{ Acción Pura de } \textit{ifold}, (\textbf{Pure.Fold}) \} \\
& \mathit{pure}_{\mathsf{C}} \circ (\!|\phi|\!)_{\mathsf{F_B}} \ \circ \mathsf{D_F}\ f \\
= \ & \qquad \{ \text{ Fusión } \textit{Fold - Map}, (\textbf{Fold.Type}) \} \\
& \mathit{pure}_{\mathsf{C}} \circ (\!|\phi \circ \mathsf{F}\ (f, \mathsf{id})|\!)_{\mathsf{F_A}} \quad \mathsf{Q.E.D.}
\end{aligned}$$

Teorema A.3 (Fusión Pura para Ifold)**.** *Sean* $\tau : \mathsf{C} \ \Rightarrow \mathsf{D}$ *un homomorfismo entre los functores aplicativos* C *y* D*,* $\iota : A \to \mathsf{C}\ B$ *una acción aplicativa y* $\phi : \mathsf{F_B}\ X \to X$ *un* $\mathsf{F_B}$*-álgebra pura, entonces:*

$$\tau \circ \mathit{ifold}^{\mathsf{C}}_{\mathsf{F_A}} \phi\ \iota = \mathit{ifold}^{\mathsf{D}}_{\mathsf{F_A}} \phi\ (\tau \circ \iota) \qquad (\textbf{Ifold.Morph})$$

Demostración:
Expandiendo las definiciones con (**Ifold.Charn**), obtenemos siguiente diagrama:

$$\begin{tikzcd}
 & & \mathsf{C}\ X \arrow[dddd, "\tau_X"] \\
 & \mathsf{C}\ (\mu_{\mathsf{F_B}}) \arrow[ur, bend right, "\mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}}"'] \arrow[dd, "\tau_{\mu_{\mathsf{F_B}}}"] & \\
\mu_{\mathsf{F_A}} \arrow[uurr, bend left, "ifold^{\mathsf{C}}_{\mathsf{F_A}}\phi\ \iota"] \arrow[ur, "traverse_{\mathsf{F}}\ \iota"'] \arrow[dr, "traverse_{\mathsf{F}}\ (\tau\circ\iota)"] \arrow[ddrr, bend right, "ifold^{\mathsf{D}}_{\mathsf{F_A}}\phi\ (\tau\circ\iota)"'] & & \\
 & \mathsf{D}\ (\mu_{\mathsf{F_B}}) \arrow[dr, bend left, "\mathsf{D}\ (\!|\phi|\!)_{\mathsf{F_B}}"] & \\
 & & \mathsf{D}\ X
\end{tikzcd}$$

Procedemos a probar que las definiciones conmutan:

$\tau \circ ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\ \iota$

$=$ { Caracterización de $ifold^{\mathsf{C}}_{\mathsf{F_A}}$, (**Ifold.Charn**) }

$\tau \circ \mathsf{C}\ (\!|\phi|\!)_{F_B} \circ traverse_{\mathsf{F}}\ \iota$

$=$ { *Naturalidad* de τ }

$\mathsf{D}\ (\!|\phi|\!)_{F_B} \circ \tau \circ traverse_{\mathsf{F}}\ \iota$

$=$ { Definición de $traverse_{\mathsf{F}}$ (**Trv.Def**) }

$\mathsf{D}\ (\!|\phi|\!)_{F_B} \circ \tau \circ (\!|\mathsf{C}\ in_{\mathsf{F_B}} \circ \delta^{\mathsf{C}}_{\mathsf{F}} \circ \mathsf{F}\ (\iota, \mathsf{id})|\!)_{\mathsf{F_A}}$

$=$ { Fusión pura para Fold, **Fold.Pure** }

$\mathsf{D}\ (\!|\phi|\!)_{F_B} \circ (\!|\mathsf{D}\ in_{\mathsf{F_B}} \circ \delta^{\mathsf{D}}_{\mathsf{F}} \circ \mathsf{F}\ (\tau \circ \iota, \mathsf{id})|\!)_{\mathsf{F_A}}$

$=$ { Definición de $traverse_{\mathsf{F}}$ (**Trv.Def**) }

$\mathsf{D}\ (\!|\phi|\!)_{F_B} \circ traverse_{\mathsf{F}}\ (\tau \circ \iota)$

$=$ { Caracterización de $ifold^{\mathsf{D}}_{\mathsf{F_A}}$, (**Ifold.Charn**) }

$ifold^{\mathsf{D}}_{\mathsf{F_A}} \phi\ (\tau \circ \iota)$ Q.E.D.

Para justificar el uso correcto de la propiedad de Fusión pura para Fold, nos resta verificar que τ es un homomorfismo de $\mathsf{F_A}$-álgebras, *id est* verificar que se cumple la siguiente ecuación:

$$\tau \circ \mathsf{C}\ in_{\mathsf{F_B}} \circ \delta^{\mathsf{C}}_{\mathsf{F}} \circ \mathsf{F}\ (\iota, id) = \mathsf{D}\ in_{\mathsf{F_B}} \circ \delta^{\mathsf{D}}_{\mathsf{F}} \circ \mathsf{F}\ (\tau \circ \iota, \mathsf{id}) \circ \mathsf{F_A}\ \tau_X$$

Esta queda probada por la conmutatividad del siguiente diagrama:

$$
\begin{tikzcd}
\mathsf{F}\ (A, \mathsf{C}\ X) \arrow[r, "\mathsf{F_A}\ \tau_X"] \arrow[d, "\mathsf{F}\ (\iota, \mathsf{id})"'] & \mathsf{F}\ (A, \mathsf{D}\ X) \arrow[d, "\mathsf{F}\ (\tau \circ \iota, \mathsf{id})"] \\
\mathsf{F}\ (\mathsf{C}\ B, \mathsf{C}\ X) \arrow[r, "\mathsf{F}\ (\tau_B, \tau_X)"] \arrow[d, "\delta^{\mathsf{C}}_{\mathsf{F}}"'] & \mathsf{F}\ (\mathsf{D}\ B, \mathsf{D}\ X) \arrow[d, "\delta^{\mathsf{D}}_{\mathsf{F}}"] \\
\mathsf{C}\ \mathsf{F}\ (B, X) \arrow[r, "\tau_{\mathsf{F}\ (B,X)}"] \arrow[d, "\mathsf{C}\ in_{\mathsf{F_B}}"'] & \mathsf{D}\ \mathsf{F}\ (B, X) \arrow[d, "\mathsf{D}\ in_{\mathsf{F_B}}"] \\
\mathsf{C}\ X \arrow[r, "\tau_X"'] & \mathsf{D}\ X
\end{tikzcd}
$$

El rectángulo inferior conmuta por la *naturalidad* de τ, el central por la propiedad de preservación de la distribución por homomorfismo de functores aplicativos, (**Morph.**δ), y el superior por la *bifunctorialidad* de F Q.E.D.

Teorema A.4 (Ley de Fusión Map Fold - Ifold)**.** *Sean* F, G *dos bifunctores bitraversables,* C *un functor aplicativo, y* $\mathsf{T}\ : \forall A\ X.(\mathsf{F_A}\ X \to X) \to (\mathsf{G_A}\ X \to X)$ *un transformador de* $\mathsf{F}_{(\_)}$*-álgebras, entonces:*

$$
\begin{aligned}
& \psi = \mathsf{T_B}\ in_{\mathsf{F_B}} \\
\Longrightarrow \\
& \mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}} \circ ifold^{\mathsf{C}}_{\mathsf{G_A}} \psi\ \iota = ifold^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ \phi)\ \iota
\end{aligned}
\qquad (\textbf{Map Fold.Ifold})
$$

Demostración:
Tenemos que verificar la conmutatividad del siguiente diagrama:

$$
\begin{tikzcd}
\mu_{\mathsf{G_A}} \arrow[dr, "ifold^{\mathsf{C}}_{\mathsf{G_A}} \psi\ \iota"] \arrow[drr, bend left, "traverse_{\mathsf{G}}\ \iota"] \arrow[dd, "ifold^{\mathsf{C}}_{\mathsf{G_A}}\ (\mathsf{T}\ \phi)\ \iota"'] & & \\
& \mathsf{C}\ \mu_{\mathsf{F_B}} \arrow[dl, "\mathsf{C}\ (\!|\phi|\!)_{\mathsf{F_B}}"] & \mathsf{C}\ \mu_{\mathsf{G_B}} \arrow[l, "\mathsf{C}\ (\!|\psi|\!)_{\mathsf{G_B}}"] \arrow[dll, bend left, "\mathsf{C}\ (\!|\mathsf{T_B}\ \phi|\!)_{\mathsf{G_B}}"] \\
\mathsf{C}\ X & &
\end{tikzcd}
$$

$$
\begin{array}{ll}
 & \mathsf{C}\ \llparenthesis \phi \rrparenthesis_{\mathsf{F_B}} \circ\ ifold^{\mathsf{C}}_{\mathsf{G_A}} \psi\ \iota \\
= & \qquad \{\ \text{Caracterización de } ifold^{\mathsf{C}}_{\mathsf{G_A}} \text{: } (\textbf{Ifold.Charn})\ \} \\
 & \mathsf{C}\ \llparenthesis \phi \rrparenthesis_{\mathsf{F_B}} \circ\ \mathsf{C}\ \llparenthesis \psi \rrparenthesis_{\mathsf{G_B}} \circ\ traverse_{\mathsf{G}}\ \iota \\
= & \qquad \{\ \text{Functorialidad de } \mathsf{C}\ \} \\
 & \mathsf{C}\ (\llparenthesis \phi \rrparenthesis_{\mathsf{F_B}} \circ\ \llparenthesis \psi \rrparenthesis_{\mathsf{G_B}}) \circ\ traverse_{\mathsf{G}}\ \iota \\
= & \qquad \{\ \textit{Hypothesis}\ \} \\
 & \mathsf{C}\ (\llparenthesis \phi \rrparenthesis_{\mathsf{F_B}} \circ\ \llparenthesis \mathsf{T_B}\ in_{\mathsf{F_B}} \rrparenthesis_{\mathsf{G_B}}) \circ\ traverse_{\mathsf{G}}\ \iota \\
= & \qquad \{\ \textit{Acid Rain para Fold}\ (\textbf{Fold.Fold})\ \} \\
 & \mathsf{C}\ \llparenthesis \mathsf{T_B}\ \phi \rrparenthesis_{\mathsf{G_B}} \circ\ traverse_{\mathsf{G}}\ \iota \\
= & \qquad \{\ \text{Caracterización de } ifold^{\mathsf{C}}_{\mathsf{G_A}} \text{: } (\textbf{Ifold.Charn})\ \} \\
 & ifold^{\mathsf{C}}_{\mathsf{G_A}}\ (\mathsf{T}\ \phi)\ \iota \quad \text{Q.E.D.}
\end{array}
$$

Teorema A.5 (Fusión Ifold Map)**.** *Sean* F *un bifunctor bitraversable y* $\mathsf{D_F}$ *el functor de tipo asociado a éste. Sea* $f: X \to A$*, entonces:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\, \iota \circ \mathsf{D_F}\ f = ifold^{\mathsf{C}}_{\mathsf{F_X}} \phi\, (\iota \circ f) \qquad (\textbf{Ifold.Type})$$

Demostración:

$$
\begin{array}{ll}
 & ifold^{\mathsf{C}}_{\mathsf{F_A}} \phi\, \iota \circ \mathsf{D_F}\ f \\
= & \qquad \{\ \text{Caracterización de } ifold^{\mathsf{C}}_{\mathsf{F_A}},\ (\textbf{Ifold.Charn})\ \} \\
 & \mathsf{C}\ \llparenthesis \phi \rrparenthesis_{\mathsf{F_B}} \circ\ traverse_{\mathsf{F}}\ \iota \circ \mathsf{D_F}\ f \\
= & \qquad \{\ \textit{Free Theorem} \text{ de } \textit{traverse},\ (\ \textbf{Traverse.Map}).\ \} \\
 & \mathsf{C}\ \llparenthesis \phi \rrparenthesis_{\mathsf{F_B}} \circ\ traverse_{\mathsf{F}}\ (\iota \circ f) \\
= & \qquad \{\ \text{Caracterización de } ifold^{\mathsf{C}}_{\mathsf{F_X}},\ (\textbf{Ifold.Charn})\ \} \\
 & ifold^{\mathsf{C}}_{\mathsf{F_X}} \phi\, (\iota \circ f) \quad \text{Q.E.D.}
\end{array}
$$

Lemma A.6. *Sean* $\iota : A \to \mathsf{C}\ B$ *una acción aplicativa y* $\phi : \mathsf{G}\ (A, \mu\mathsf{F_A}) \to \mu\mathsf{F_A}$ *un* $\mathsf{G_A}$*-álgebra independiente de los efectos de* ι*. Entonces, el siguiente diagrama conmuta:*

$$
\begin{tikzcd}
\mu_{\mathsf{G_A}} \arrow[r, "\llparenthesis \phi \rrparenthesis_{\mathsf{G_A}}"] \arrow[d, "traverse_{\mathsf{G}}\ \iota"'] \arrow[dr, bend left=10, "ifold^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ in_{\mathsf{F_B}})\ \iota"] & \mu_{\mathsf{F_A}} \arrow[d, "traverse_{\mathsf{F}}\ \iota"] \\
\mathsf{C}\ \mu_{\mathsf{G_B}} \arrow[r, "\mathsf{C}\ \llparenthesis \mathsf{T_B}\ in_{\mathsf{F_B}} \rrparenthesis_{\mathsf{G_B}}"'] & \mathsf{C}\ \mu_{\mathsf{F_B}}
\end{tikzcd}
$$

Demostración:

$$
\begin{array}{ll}
& traverse_{\mathsf{F}}\ \iota \circ \llparenthesis \phi \rrparenthesis_{\mathsf{G_A}} \\
= & \hfill \{\ \textit{Hypothesis}\ \} \\
& traverse_{\mathsf{F}}\ \iota \circ \llparenthesis (\mathsf{T_A}\ in_{\mathsf{F_A}}) \rrparenthesis_{\mathsf{G_A}} \\
= & \hfill \{\ \text{Definición de } traverse_{\mathsf{F}}\ ,\ (\textbf{Trv.Def})\ \} \\
& \llparenthesis \mathsf{C}\ in_{\mathsf{F_B}} \circ \delta^{\mathsf{C}}_{\mathsf{F}} \circ \mathsf{F}\ (\iota, id) \rrparenthesis_{\mathsf{F_A}} \circ \llparenthesis (\mathsf{T_A}\ in_{\mathsf{F_A}}) \rrparenthesis_{\mathsf{G_A}} \\
= & \hfill \{\ \textit{Acid Rain para Fold}\ (\textbf{Fold.Fold})\ \} \\
& \llparenthesis \mathsf{T_A}\ (\mathsf{C}\ in_{\mathsf{F_B}} \circ \delta^{\mathsf{C}}_{\mathsf{F}} \circ \mathsf{F}\ (\iota, id)) \rrparenthesis_{\mathsf{G_A}} \\
= & \hfill \{\ \phi\ \textit{independiente} \text{ de } \iota,\ (\textbf{Alg.Ind})\ \} \\
& \llparenthesis \mathsf{C}\ (\mathsf{T_B}\ in_{\mathsf{F_B}}) \circ \delta^{\mathsf{C}}_{\mathsf{G}} \circ \mathsf{G}\ (\iota, id) \rrparenthesis_{\mathsf{G_A}} \\
= & \hfill \{\ \text{Definición de } ifold^{\mathsf{C}}_{\mathsf{G_A}},\ (\textbf{Ifold.Def})\ \} \\
& ifold^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ in_{\mathsf{F_B}})\ \iota \\
= & \hfill \{\ \text{Caracterización de } ifold^{\mathsf{C}}_{\mathsf{G_A}},\ (\textbf{Ifold.Charn})\ \} \\
& \mathsf{C}\ \llparenthesis (\mathsf{T_B}\ in_{\mathsf{F_B}}) \rrparenthesis_{\mathsf{G_A}} \circ traverse_{\mathsf{G}}\ \iota \quad \text{Q.E.D.}
\end{array}
$$

Teorema A.7 (Acid Rain: Ley de Fusión Fold - Ifold)**.** *Dados* F *y* G *dos bifunctores bitraversables, y* C *un functor aplicativo y* $\iota : A \to \mathsf{C}\ B$ *una acción aplicativa. Si* ϕ *es un* $\mathsf{G_A}$*-álgebra independiente de los efectos de* ι*, entonces:*

$$ifold^{\mathsf{C}}_{\mathsf{F_A}} \psi\ \iota \circ \llparenthesis \phi \rrparenthesis_{\mathsf{G_A}} = ifold^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ \psi)\ \iota \qquad (\textbf{Ifold.Fold})$$

Demostración:
Tenemos que probar que el siguiente diagrama conmuta:

$$
\begin{tikzcd}
\mu_{\mathsf{G_A}} \arrow[r, "\llparenthesis \phi \rrparenthesis_{\mathsf{G_A}}"] \arrow[d, "traverse_{\mathsf{G}}\ \iota"'] \arrow[rrd, bend left, "ifold^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ \psi)\ \iota"] & \mu_{\mathsf{F_A}} \arrow[d, "traverse_{\mathsf{F}}\ \iota"] \arrow[rd, "ifold^{\mathsf{C}}_{\mathsf{F_A}} \psi\ \iota"] & \\
 & & \mathsf{C}\ X \\
\mathsf{C}\ \mu_{\mathsf{G_B}} \arrow[r, "\mathsf{C}\ \llparenthesis \mathsf{T_B}\ in_{\mathsf{F_B}} \rrparenthesis_{\mathsf{G_B}}"'] \arrow[rru, bend right, "\mathsf{C}\ \llparenthesis \mathsf{T_B}\ \psi \rrparenthesis_{\mathsf{G_B}}"'] & \mathsf{C}\ \mu_{\mathsf{F_B}} \arrow[ru, "\mathsf{C}\ \llparenthesis \psi \rrparenthesis_{F_B}"'] &
\end{tikzcd}
$$

$ifold^{\mathsf{C}}_{\mathsf{F_A}} \psi\ \iota \circ (\!|\phi|\!)_{\mathsf{G_A}}$

$= \quad \{ \text{ Caracterización de } ifold^{\mathsf{C}}_{\mathsf{F_A}}, \textbf{(Ifold.Charn)} \}$

$\mathsf{C}\ (\!|\psi|\!)_{F_B} \circ traverse_{\mathsf{F}}\ \iota \circ (\!|\phi|\!)_{\mathsf{G_A}}$

$= \quad \{ \textit{ Hypothesis: } \phi = \mathsf{T_A}\ in_{\mathsf{G_A}} \}$

$\mathsf{C}\ (\!|\psi|\!)_{F_B} \circ traverse_{\mathsf{F}}\ \iota \circ (\!|\mathsf{T_A}\ in_{\mathsf{F_A}}|\!)_{\mathsf{G_A}}$

$= \quad \{ \text{ Lemma 4.11 } \}$

$\mathsf{C}\ (\!|\psi|\!)_{F_B} \circ \mathsf{C}\ (\!|\mathsf{T_B}\ in_{\mathsf{F_B}}|\!)_{\mathsf{G_B}} \circ traverse_{\mathsf{G}}\ \iota$

$= \quad \{ \mathsf{C}\ \textit{functor}, \textbf{(Functor.}\circ) \}$

$\mathsf{C}\ ((\!|\psi|\!)_{F_B} \circ (\!|\mathsf{T_B}\ in_{\mathsf{F_B}}|\!)_{\mathsf{G_B}}) \circ traverse_{\mathsf{G}}\ \iota$

$= \quad \{ \textit{ Acid Rain para Fold}, \textbf{(Fold.Fold)} \}$

$\mathsf{C}\ (\!|\mathsf{T_B}\ \psi|\!)_{\mathsf{G_B}} \circ traverse_{\mathsf{G}}\ \iota$

$= \quad \{ \text{ Caracterización de } ifold^{\mathsf{C}}_{\mathsf{G_A}}, \textbf{(Ifold.Charn)} \}$

$ifold^{\mathsf{C}}_{\mathsf{G_A}} (\mathsf{T_B}\ \psi)\ \iota$ Q.E.D.

Teorema A.8 (Acción Paralela)**.** *Sean* F, G *dos bifunctores bitraversables,* $\mathsf{C_1}, \mathsf{C_2}$ *dos functores aplicativos y* $\iota : A \to \mathsf{C_2}\ B,\ \kappa : A \to \mathsf{C_2}\ B$ *dos acciones aplicativas.*

$$ifold^{\mathsf{C_1}}_{\mathsf{F_A}} \phi\ \iota \otimes ifold^{\mathsf{C_2}}_{\mathsf{F_A}} \phi\ \kappa = ifold^{\mathsf{C_1} \times \mathsf{C_2}}_{\mathsf{F_A}} \phi\ (\iota \otimes \kappa) \qquad \textbf{(Ifold.}\otimes)$$

Demostración:

$ifold^{\mathsf{C_1} \times \mathsf{C_2}}_{\mathsf{F_A}} \phi\ (\iota \otimes \kappa)$

$= \quad \{ \textbf{(Ifold.Charn)} \}$

$\mathsf{C_1} \times \mathsf{C_2}\ (\!|\phi|\!)_{\mathsf{F}} \circ traverse\ (\iota \otimes \kappa)$

$= \quad \{ \text{ Prop.de } traverse, (\textbf{Traverse}.\otimes) \}$

$\mathsf{C_1} \times \mathsf{C_2}\ (\!|\phi|\!)_{\mathsf{F}} \circ (traverse\ \iota \otimes traverse\ \kappa)$

$= \quad \{ \text{ Functorialidad de } \times \}$

$\mathsf{C_1} \times \mathsf{C_2}\ (\!|\phi|\!)_{\mathsf{F}} \circ traverse\ \iota \otimes \mathsf{C_1} \times \mathsf{C_2}\ (\!|\phi|\!)_{\mathsf{F}} \circ traverse\ \kappa$

$= \quad \{ \textbf{(Ifold.Charn)} \}$

$ifold^{\mathsf{C_1}}_{\mathsf{F_A}} \phi\ \iota \otimes ifold^{\mathsf{C_2}}_{\mathsf{F_A}} \phi\ \kappa$ Q.E.D.

Teorema A.9 (Composición Secuencial)**.** *Sean* F*y* G *dos bifunctores bitraversables,* $\mathsf{C_1}$ *y* $\mathsf{C_2}$ *dos functores aplicativos, y* $\iota : A \to \mathsf{C_2}\ B$ *y* $\kappa : B \to \mathsf{C_1}\ C$ *dos acciones aplicativas. Entonces, si* $\psi : \mathsf{F}\ (B, \mu\mathsf{G_B}) \to \mu\mathsf{G_B}$ *es un* $\mathsf{F_B}$*-álgebra independiente de los efectos de* κ *resulta:*

$$\psi = \mathsf{T_E}\ in_{\mathsf{G_E}}$$
$$\Longrightarrow$$
$$ifold^{\mathsf{C_1}}_{\mathsf{G_B}} \phi\ \kappa \odot ifold^{\mathsf{C_2}}_{\mathsf{F_A}} \psi\ \iota = ifold^{\mathsf{C_2} \bullet \mathsf{C_1}}_{\mathsf{F_A}} (\mathsf{T_E}\ \phi)\ (\kappa \odot \iota) \qquad \textbf{(Ifold.}\odot)$$

Demostración:

$$
\begin{array}{ll}
& ifold^{\mathsf{C_1}}_{\mathsf{G_B}} \phi\ \kappa \odot ifold^{\mathsf{C_2}}_{\mathsf{F_A}} \psi\ \iota \\
= & \hfill \{ \text{ Definición de } \odot \ \} \\
& \mathsf{C_2}\ (ifold^{\mathsf{C_1}}_{\mathsf{G_B}} \phi\ \kappa) \circ ifold^{\mathsf{C_2}}_{\mathsf{F_A}} \psi\ \iota \\
= & \hfill \{ \text{ Caracterización de } ifold^{\mathsf{C_2}}_{\mathsf{F_A}} \text{: } (\textbf{Ifold.Charn}) \ \} \\
& \mathsf{C_2}\ (ifold^{\mathsf{C_1}}_{\mathsf{G_B}} \phi\ \kappa) \circ \mathsf{C_2}\ (\!| \psi |\!)_{F_B} \circ \mathit{traverse}\ \iota \\
= & \hfill \{ \ \mathsf{C_2}\ \mathit{functor}, (\textbf{Functor.}\circ) \ \} \\
& \mathsf{C_2}\ (ifold^{\mathsf{C_1}}_{\mathsf{G_B}} \phi\ \kappa \circ (\!| \psi |\!)_{F_B}) \circ \mathit{traverse}\ \iota \\
= & \hfill \{ \ \mathit{Hypothesis} \ \} \\
& \mathsf{C_2}\ (ifold^{\mathsf{C_1}}_{\mathsf{G_B}} \phi\ \kappa \circ (\!| \mathsf{T_E}\ in_{\mathsf{G_E}}\ |\!)_{F_B}) \circ \mathit{traverse}\ \iota \\
= & \hfill \{ \text{ Acid Rain para Ifold, } (\textbf{Ifold.Fold}) \ \} \\
& \mathsf{C_2}\ (ifold^{\mathsf{C_1}}_{\mathsf{F_B}} (\mathsf{T_E}\ \phi)\ \kappa) \circ \mathit{traverse}\ \iota \\
= & \hfill \{ \ (\textbf{Ifold.Charn}) \text{ para } ifold^{\mathsf{C_1}}_{\mathsf{F_B}} \ \} \\
& \mathsf{C_2}\ (\mathsf{C_1}\ (\!| \mathsf{T_E}\ \phi |\!)_{F_E} \circ \mathit{traverse}\ \kappa) \circ \mathit{traverse}\ \iota \\
= & \hfill \{ \ \mathsf{C_2}\ \mathit{functor}, (\textbf{Functor.}\circ) \ \} \\
& \mathsf{C_2} \bullet \mathsf{C_1}\ (\!| \mathsf{T_E}\ \phi |\!)_{F_E} \circ \mathsf{C_2}\ \mathit{traverse}\ \kappa \circ \mathit{traverse}\ \iota \\
= & \hfill \{ \text{ Definición de } \odot \ \} \\
& \mathsf{C_2} \bullet \mathsf{C_1}\ (\!| \mathsf{T_E}\ \phi |\!)_{F_E} \circ (\mathit{traverse}\ \kappa \odot \mathit{traverse}\ \iota) \\
= & \hfill \{ \text{ Prop. de } \mathit{traverse}, (\ \textbf{Traverse}.\odot) \ \} \\
& \mathsf{C_2} \bullet \mathsf{C_1}\ (\!| \mathsf{T_E}\ \phi |\!)_{F_E} \circ \mathit{traverse}\ (\kappa \odot \iota) \\
= & \hfill \{ \ (\textbf{Ifold.Charn}) \text{ para } ifold^{\mathsf{C_2 \bullet C_1}}_{\mathsf{F_A}} \ \} \\
& ifold^{\mathsf{C_2 \bullet C_1}}_{\mathsf{F_A}} (\mathsf{T_E}\ \phi)\ (\kappa \odot \iota) \quad \text{Q.E.D.}
\end{array}
$$

Bibliografía

[1] Abramsky, S., and Jung, A. Domain Theory. In *Handbook of Logic in Computer Science Volume 3*, S. Abramsky, D. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, 1994, pp. 1–168.

[2] Backhouse, R., Jansson, P., Jeuring, J., and Meertens, L. Generic programming — an introduction. In *LNCS* (1999), vol. 1608, Springer-Verlag, pp. 28–115. Revised version of lecture notes for AFP'98.

[3] Backhouse, R. C., Bijsterveld, M., Geldrop, R. v., and Woude, J. v. d. Categorical fixed point calculus. In *CTCS '95: Proceedings of the 6th International Conference on Category Theory and Computer Science* (London, UK, 1995), Springer-Verlag, pp. 159–179.

[4] Barr, M., and Wells, C. *Toposes, Triples and Theories.* Springer-Verlag, 1983.

[5] Barr, M., and Wells, C. *Category Theory for Computing Science*, third ed. Centre de recherches mathématiques, Montreal, Canada, August 1999.

[6] Benton, N., Hughes, J., and Moggi, E. Monads and effects. In *APPSEM* (2000), G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, Eds., vol. 2395 of *Lecture Notes in Computer Science*, Springer, pp. 42–122.

[7] Bird, R., and de Moor, O. *Algebra of programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[8] Bird, R. S. *Introduction to Functional Programming Using Haskell.* Prentice-Hall, 1998.

[9] Brady, E. Idris - systems programming meets full dependent types. In *To appear in PLPV 2011.* (2011).

[10] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. Links: Web programming without tiers. In *FMCO* (2006), F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4709 of *Lecture Notes in Computer Science*, Springer, pp. 266–296.

[11] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. The essence of form abstraction. In *Programming Languages and Systems*, G. Ramalingam, Ed., vol. 5356 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 205–220.

[12] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. An idiom's guide to formlets, Jun 2008.

[13] Dybjer, P. Category theory and programming language semantics: An overview. In *Category Theory and Computer Programming*, D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, Eds., vol. 240 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1986, pp. 163–181.

[14] Fokkinga, M. Calculate Categorically! *Formal Aspects of Computing 4*, 4 (1992), 673–692.

[15] Fokkinga, M. M. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Enschede, February 1992. Imported from EWI/DB PMS [db-utwente:phdt:0000003413].

[16] Fokkinga, M. M. Monadic Maps and Folds for Arbitrary Datatypes. Tech. Rep. Memoranda Inf 94-28, University of Twente, Enschede, Netherlands, June 1994.

[17] Fokkinga, M. M. Datatype laws without signatures. *Mathematical Structures in Computer Science 6*, 1 (February 1996), 1–32. Imported from EWI/DB PMS [db-utwente:arti:0000003409].

[18] Ghani, N., and Johann, P. Monadic augment and generalised short cut fusion. *Journal of Functional Programming 17*, 6 (2007), 731–776.

[19] Ghani, N., and Johann, P. Short cut fusion of recursive programs with computational effects. In *Trends in Functional Programming* (2009), P. Achten, P. Koopman, and M. Morazán, Eds., vol. 9 of *Trends in Functional Programming*, Intellect, pp. 113–128. ISBN 978-1-84150-277-9.

[20] Gibbons, J. Origami programming. In *The Fun of Programming*, J. Gibbons and O. de Moor, Eds., Cornerstones in Computing. Palgrave, 2003, pp. 41–60.

[21] Gibbons, J. Datatype-generic programming. In *Datatype-Generic Programming*, R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, Eds., vol. 4719 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2007, ch. 1, pp. 1–71–71.

[22] Gibbons, J., and d. S. Oliveira, B. C. The essence of the iterator pattern. *J. Funct. Program. 19*, 3-4 (2009), 377–402.

[23] Gill, A. *Cheap deforestation for non-strict functional languages*. PhD thesis, The University of Glasgow, January 1996.

[24] Gill, A., Launchbury, J., and Peyton Jones, S. A short cut to deforestation. In *FPCA '93: Proceedings of the conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1993), ACM Press, pp. 223–232.

[25] Goguen, J. A. A Categorical Manifesto. *Mathematical Structures in Computer Science 1*, 1 (1991), 49–67.

[26] Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. Initial algebra semantics and continuous algebras. *J. ACM 24*, 1 (January 1977), 68–95.

[27] Hughes, J. Why functional programming matters. *Comput. J. 32*, 2 (1989), 98–107.

[28] Hughes, J. Generalising monads to arrows. *Sci. Comput. Program. 37*, 1-3 (2000), 67–111.

[29] Hutton, G. Fold and unfold for program semantics. In *ICFP* (1998), pp. 280–288.

[30] Hutton, G. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming 9*, 4 (1999), 355–372.

[31] Jaskelioff, M., and Moggi, E. Monad transformers as monoid transformers. *Theoretical Computer Science 411*, 51-52 (2010), 4441 – 4466. European Symposium on Programming 2009 - ESOP 2009.

[32] Johann, P., and Ghani, N. Foundations for structured programming with gadts. In *POPL* (2008), G. C. Necula and P. Wadler, Eds., ACM, pp. 297–308.

[33] Johann, P., and Ghani, N. Monadic fold, monadic build, monadic short cut fusion. In *Proceedings of the 10th Symposium on Trends in Functional Programming (TFP'09)* (2009), pp. 9 – 23.

[34] Lehman, D. J., and Smyth, M. B. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory 14*, 2 (1981), 97–140.

[35] Liang, S., Hudak, P., and Jones, M. Monad Transformers and Modular Interpreters. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995* (New York, NY, USA, 1995), ACM Press, pp. 333–343.

[36] Lindley, S., Wadler, P., and Yallop, J. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In *Mathematical Structures in Functional Programming* (2008), V. Capreta and C. McBride, Eds., vol. 5520 of *Electronic Notes in Theoretical Computer Science*, Elsevier.

[37] Lindley, S., Wadler, P., and Yallop, J. The arrow calculus. *J. Funct. Program. 20*, 1 (2010), 51–69.

[38] MacLane, S. *Categories for the Working Mathematician*, second ed., vol. 5 of *Graduate Texts in Mathematics*. Springer, New York, September 1998.

[39] Manes, E. G., and Arbib, M. A. *Algebraic approaches to program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[40] Manzino, C., and Pardo, A. Shortcut fusion of monadic programs. *Journal of Universal Computer Science 14*, 21 (2008), 3431–3446.

[41] McBride, C., and Paterson, R. Applicative programming with effects. *Journal of Functional Programming 18*, 01 (2008), 1–13.

[42] Meertens, L. Functor pulling. In *Proc. Workshop on Generic Programming* (1998), R. Backhouse and T. Sheard, Eds.

[43] Meijer, E., Fokkinga, M. M., and Paterson, R. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture* (London, UK, 1991), Springer-Verlag, pp. 124–144.

[44] Moggi, E. Computational lambda-calculus and monads. In *LICS* (1989), IEEE Computer Society, pp. 14–23.

[45] Moggi, E. Notions of computation and monads. *Information and Computation 93* (1989), 55–92.

[46] Pardo, A. Fusion of recursive programs with computational effects. *Theoretical Computer Science. 260*, 1-2 (2001), 165–207.

[47] Pardo, A. Combining datatypes and effects. In *Advanced Functional Programming* (2004), V. Vene and T. Uustalu, Eds., vol. 3622 of *Lecture Notes in Computer Science*, Springer, pp. 171–209.

[48] PEYTON JONES, S. L. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering Theories of Software Construction* (2001), T. Hoare, M. Broy, and R. Steinbruggen, Eds., IOS Press, pp. 47–96.

[49] PIERCE, B. C. *Basic Category Theory for Computer Scientists (Foundations of Computing)*, 1 ed. The MIT Press, August 1991.

[50] SWIERSTRA, S. D. Combinator parsing: A short tutorial. In *LerNet ALFA Summer School* (2008), A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, Eds., vol. 5520 of *Lecture Notes in Computer Science*, Springer, pp. 252–300.

[51] TAKANO, A., AND MEIJER, E. Shortcut deforestation in calculational form. In *In Proc. Conference on Functional Programming Languages and Computer Architecture* (1995), ACM Press, pp. 306–313.

[52] WADLER, P. Theorems for Free! In *Proceedings of the 4th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'89* (New York, 1989), ACM Press, pp. 347–359.

[53] WADLER, P. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci. 73*, 2 (1990), 231–248.

[54] WADLER, P. The essence of functional programming. In *POPL* (1992), pp. 1–14.

[55] WADLER, P. Monads for functional programming. In *Advanced Functional Programming* (1995), J. Jeuring and E. Meijer, Eds., vol. 925 of *Lecture Notes in Computer Science*, Springer, pp. 24–52.