Proyecto de grado:
Implementación de un sistema de fusión

Informe final

Facundo Domínguez

Tutor: Dr. Alberto Pardo
Instituto de Computación

Carrera de Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República
Montevideo – Uruguay
Diciembre de 2004

Resumen

Desde que se empieza a escribir software a gran escala, la elaboración de metodologías que permitan escribir, verificar, mantener y reusar el código se han vuelto el centro de estudio del área que hoy se conoce como Ingeniería de Software. Con los años diversos paradigmas de programación y metodologías de desarrollo han fomentado intensamente la construcción y el uso de componentes modulares que son combinadas para obtener soluciones a problemas concretos.

El empleo de componentes modulares en el paradigma de programación funcional acarrea la necesidad de manipular estructuras de datos que sirvan como medio de comunicación entre unas y otras. Dependiendo de la situación el uso intenso de un diseño de estas características puede ser un factor de ineficiencia debido a la generación y consumo de las estructuras de datos intermedias. Muchas veces ocurre que si el diseño no se hubiese hecho modular el programa seguramente sería menos claro, pero más eficiente, dado que no tendría que computar el fardo de tales estructuras.

Existen técnicas de transformación de programas funcionales, que dado un programa escrito en forma modular, pueden combinar diferentes partes del mismo para construir un programa equivalente que no emplee estas estructuras de datos intermedias. Una serie importante de trabajos apuntan a automatizar estas técnicas para poder incluirlas dentro de las etapas de un compilador. Así los desarrolladores de software pueden beneficiarse del diseño modular sin preocuparse por cuestiones de eficiencia.

Nuestro objetivo es presentar una implementación de un sistema que realiza automáticamente algunas de estas transformaciones sobre programas escritos en Haskell. El presente trabajo se desarrolla en el marco de un proyecto CSIC[Par03].

Palabras clave: fusión de programas, deforestación, teoría de categorías, programación funcional.

Índice general

Capítulo 1

Introducción

En el diseño de aplicaciones se utiliza con frecuencia la estrategia de *divide y vencerás* para la organización del trabajo y del producto. La esencia de esta estrategia es la división del problema original en problemas más sencillos de resolver.

La estrategia de *divide y vencerás* se puede usar, por ejemplo, para escribir un programa que haga lo siguiente: Contar la cantidad de palabras que ocurren en una línea que terminan con un carácter dado. Podemos partir el problema de esta manera:

- Separar la línea en palabras.
- Seleccionar las palabras que terminan con el carácter dado.
- Contar la cantidad de palabras seleccionadas.

Respetando esta división, llegamos a la siguiente solución expresada en *Haskell*[1] como la función `count`.

```
count :: Char -> String -> Int
count c = length . filter ((==c).last) . words

words :: String -> [String]
words str = case dropWhile isSpace str of
             [] -> []
             str' -> let (w,str'') = break isSpace str'
                      in w : words str''

filter :: (a->Bool) -> [a] -> [a]
filter p [] = []
filter p (a:as) = if p a then a : filter p as
                         else filter p as

length :: [a] -> Int
length [] = 0
length (_:as) = 1 + length as
```

El predicado `(==c)::Char->Bool` retorna `True` cuando el argumento es igual al carácter `c`. La función `last` retorna el último elemento de una lista.

[1] Todos nuestros ejemplos estarán expresados en este lenguaje.

La expresión `dropWhile isSpace str` es el sufijo más grande de `str` que no empieza con espacios. La expresión `break isSpace str'` es un par cuya primer componente es el prefijo más grande de `str'` que no contiene espacios, y la segunda componente es el resto de `str'`.

Ahora que tenemos una solución, cabe preguntarse si hay alguna forma más eficiente de resolver el problema. La solución presentada crea primero una lista de palabras, la cual es consumida por `filter`, quien a su vez produce una nueva lista con las palabras que terminan con c. Por último `length` consume la lista generada por `filter`. De modo que, en la ejecución de este programa, se crean dos listas nuevas de tamaño proporcional al tamaño de la línea de entrada.

Podemos imaginar una solución que no adolezca este problema. La mejora es evidente si se observa que en esta nueva solución no se utiliza ninguna aplicación de los constructores de listas.

```
count c str =
    case dropWhile isSpace str of
      Nil -> 0
      str' -> let (w,str'') = break isSpace str' of
                in if ((==c).last) w
                     then  1 + count c str''
                     else count c str''
```

Si bien es más eficiente, dado que no se construyen listas intermedias, es bastante más difícil deducir que la solución es correcta.

Muchos beneficios tiene el diseño modular de código como para dejarlo de lado demasiado rápido. Entre ellos se cuenta el reuso de código empaquetado en componentes, la mantenibilidad del código que va asociado a la facilidad de entender, modificar, verificar y depurar tales programas.

Imagine ahora que se tiene un *software* que recibe como entrada un programa cualquiera expresado en forma modular como el primero y que retorna como salida el programa sintetizado en la forma del segundo, sin estructuras de datos intermedias. Teniendo un sistema con estas características sería posible, entonces, incentivar al programador a escribir sus programas de manera modular, asegurando que sean correctos, sin preocuparse por la eficiencia de los mismos. ¿Qué sentido tendría tratar de escribir un programa eficiente y oscuro, si el compilador fuese capaz de deducir tal programa a partir de la especificación modular?

Varios trabajos se han realizado con el objetivo de descubrir técnicas de transformación de programas que eliminen las estructuras intermedias. Como en lenguajes funcionales estas estructuras intermedias en general son arborescentes, se dice que las técnicas son de *deforestación* [Wad88]. Dado que la eliminación de estructuras intermedias implica el reemplazo de varias funciones por una equivalente que combine sus códigos, también se dice que las técnicas son de *fusión*. Otra línea importante de trabajos se ha desarrollado con el objetivo de automatizar algunas de las técnicas, apuntando a su integración dentro de un compilador.

Este trabajo trata sobre la construcción de un *sistema de fusión*. Con ello nos referimos a una herramienta que realiza automáticamente una serie de transformaciones basadas en fusión sobre programas escritos en un lenguaje funcional, y retorna programas escritos en el mismo lenguaje.

Una característica de este sistema es que se basa en un enfoque algebraico de la transformación de programas [BdM97, SF93, TM95, HIT96b, HIT96a], pero también existen otras propuestas [Wad88, Chi92, dMS99, JL98, GLPJ93].

En el enfoque que adoptamos las funciones se reescriben en términos de un esquema de recursión llamado *hilomorfismo*. Los *hilomorfismos* cuentan con ciertas leyes de transformación, conocidas como *Lluvia Ácida* [TM95], mediante las cuales es posible fusionar programas.

Para la construcción de este sistema contamos con la experiencia de implementaciones anteriores de características similares [OHIT97, Sch00]. Casi todos los algoritmos que integran nuestro sistema se encuentran descriptos en dichos trabajos.

Las principales contribuciones de nuestro trabajo son las siguientes.

- Revisamos y reescribimos la descripción de los algoritmos que forman el núcleo del sistema.
- Proponemos nuevos algoritmos para reestructurar la representación interna de una función. Ciertas situaciones donde originalmente las definiciones no se podían fusionar, ahora con las nuevas reestructuras a veces sí es posible.
- Proponemos una representación para la implementación de hilomorfismos alternativa a las planteadas hasta ahora en la literatura.
- Ofrecemos un sistema interactivo que permite examinar el resultado de aplicar las leyes de fusión en un formato comprensible (salvo algunas excepciones). Se puede observar como definición recursiva o en términos de su representación algebraica.
- Identificamos posibles extensiones prácticas y teóricas para ampliar el poder del sistema.

El resto del trabajo se organiza de la siguiente manera. En el Capítulo 2 se presenta la batería de conceptos teóricos sobre los cuales se apoya la implementación. En el Capítulo 3 se presentan los algoritmos implementados a nivel conceptual. En el Capítulo 4 se presentan las estructuras de datos y el comportamiento específico de los algoritmos más importantes. Finalmente en el Capítulo 5 realizamos una evaluación final de lo implementado y discutimos posibles extensiones que se le pueden realizar.

Capítulo 2

Fundamentos

Cuando un sistema de transformación de programas utiliza un enfoque algebraico, como es nuestro caso, tenemos a nuestra disposición herramientas teóricas que nos ayudan a explicar y verificar el funcionamiento del sistema. El objetivo de este capítulo es explicar algunas de esas herramientas.

Consideremos el siguiente programa que computa la suma de los cuadrados de los valores en las hojas de un árbol.

```
data BTree = Leaf Int | Join (BTree,BTree)

sumsqr :: BTree Int -> Int
sumsqr = sumBT . sqrLeaves

sumBT :: BTree -> Int
sumBT (Leaf i) = i
sumBT (Join (t1,t2)) = sumBT t1 + sumBT t2

sqrLeaves :: BTree -> BTree
sqrLeaves (Leaf i)  = Leaf (sqr i)
sqrLeaves (Join (t1,t2)) = Join (sqrLeaves t1,sqrLeaves t2)

sqr :: Int -> Int
sqr x = x*x
```

En este programa `sqrLeaves` calcula el cuadrado del valor de cada hoja en un árbol. La función `sumBT` se encarga de sumar los valores en las hojas resultantes del paso anterior. Si consideramos un término como el siguiente

```
Join (Join (Leaf 1,Leaf 9),Leaf 16)
```

la aplicación de `sumBT` sobre el término produce lo mismo que reemplazar todas las ocurrencias de `Join` por `(+)` y cada ocurrencia de `Leaf` por la función identidad `id`. Así obtenemos el término

```
(+) ((+) (id 1) (id 9)) (id 16)
```

Si `sumBT` reemplaza los constructores por las operaciones `(+)` e `id`, cabe cuestionarse por qué `squareLeafs` no coloca directamente estas operaciones en vez de los constructores. Así tendríamos una definición que computa lo mismo que `sumsqr` pero sin composiciones.

```
data Either a b = Left a | Right b
either :: (a->c) -> (b->c) -> Either a b -> c
either fa fb (Left a) = fa a
either fa fb (Right b) = fb b

type  F a = Either Int (a,a)

fmap :: (a->b) -> F a -> F b
fmap h = either Left (\(a1,a2) -> Right (h a1,h a2))

outF :: BTree -> F BTree
outF (Leaf i) = Left i
outF (Join (t1,t2)) = Right (t1,t2)

sumBT :: BTree -> Int
sumBT = either id (uncurry (+)) . fmap sumBT . outF

sqrLeaves :: BTree -> BTree
sqrLeaves = either Leaf Join . fmap sqrLeaves . c
  where c (Leaf i) = Left (sqr i)
        c (Join pair) = Right pair
```

Figura 2.1: Definiciones factorizadas

```
sqrLeaves' :: BTree Int -> BTree Int
sqrLeaves' (Leaf i) = id (sqr i)
sqrLeaves' (Join (t1,t2)) = sqrLeaves' t1 + sqrLeaves' t2
```

Esta sustitución de constructores por otras operaciones es uno de los pilares fundamentales de las técnicas de fusión que implementamos. Para poder automatizar esta sustitución aparece claramente la necesidad de reconocer automáticamente tanto los constructores que serán sustituidos, como las operaciones que serán puestas en su lugar. La estrategia será factorizar la definición de las funciones `sumBT` y `sqrLeaves` de modo que aparezcan explícitamente las partes sustitutas y las partes a sustituir. En la Figura 2.1 se presenta una forma factorizada de escribir las funciones `sumBT` y `sqrLeaves`. La función `uncurry` convierte una función binaria currificada en una que no lo es.

Para poder separar las definiciones nos hemos apoyado mucho en el tipo de datos `Either`, el cual funciona como estructura de datos intermedia entre las partes de cada definición. Esta forma de factorizar las definiciones de `sumBT` y `sqrLeaves` permite distinguir claramente 3 aspectos:

- Cómo se crean los argumentos de las llamadas recursivas. En el caso de `sumBT`, la función `outF` es la encargada de preparar los argumentos de las llamadas recursivas, así como también cualquier otro dato necesario para el resto de las computaciones que realiza `sumBT`. En el caso de `sqrLeaves` este trabajo lo realiza `c`.

- Sobre qué valores se realizan las llamadas recursivas. Las llamadas recursivas se aplican en una etapa exclusiva para ello. La función `fmap` es la que juega este papel de realizar las invocaciones.

- Cómo se opera con el resultado de las llamadas recursivas. En el caso de `sqrLeaves` se aplican los constructores que componen el árbol resultante a partir de las llamadas recursivas. En el caso de `sumBT` se suman los valores resultantes de las llamadas recursivas o simplemente se retorna el valor contenido en una hoja.

Con las funciones escritas de esta manera, podemos utilizar una ley enunciada informalmente como sigue:

> Dadas dos definiciones
>
> ```
> f1 = phi . fmap f1 . outF
> f2 = either Leaf Join . fmap f2 . psi'
> ```
>
> se puede definir una función `f3`
>
> ```
> f3 = phi . fmap f3 . psi'
> ```
>
> tal que `f1 . f2 = f3`.

En nuestro caso esta ley nos asegura que

```
sumBT . sqrLeaves = sqrLeaves'
```

donde `sqrLeaves'`, en forma factorizada, se define como

```
sqrLeaves' :: BTree -> BTree
sqrLeaves' = either id (uncurry (+)) . fmap sqrLeaves' . c
  where c (Leaf i) = Left (sqr i)
        c (Join pair) = Right pair
```

Dada una definición, es posible hasta cierto punto factorizarla como lo hicimos mediante la ejecución de un algoritmo. Mediante un proceso inverso también es posible juntar las partes del resultado de la ley para obtener la definición de `sqrLeaves'` que derivamos al principio.

2.1. Functores

En la formulación informal de la ley que usamos en el ejemplo anterior, aparece la noción de functor como la función `fmap` que organiza las llamadas recursivas. Damos la definición formal y más general.

Definición 2.1 (Functor) *Un functor* `F` *es un operador que actúa sobre tipos y funciones, y satisface las siguientes propiedades:*

- `F f :: F a ->F b  para toda f :: a ->b`
- `F id = id`
- `F (f . g) = F f . F g  ∀ f :: a ->b, g :: b ->c`

Un functor puede verse también como un constructor de tipo que recibe un tipo `X` y construye un tipo `F X`, más una función que describe la acción del functor sobre funciones.

Ejemplo 2.2 En el programa de la figura 2.1 el constructor de tipo `F` puede considerarse un functor si se define su acción sobre funciones como `F f = fmap f`. Es posible verificar que esta definición satisface la definición de functor. □

La definición de functor se puede generalizar para functores que reciben más de un argumento.

Definición 2.3 (Bifunctores) *Un functor binario* `F` *es un operador binario que actúa sobre tipos y funciones, y satisface las siguientes propiedades:*

- `F f g ::F a c ->F b d  para toda f::a->b, g::c->d`
- `F id id = id`
- `F (f . g) (h . k) = F f h . F g k  ∀ f::a->b, g:b->c, h:d->e, k:e->t`

Usualmente los functores específicos no se definen como hicimos en el Ejemplo 2.2 en términos de un programa funcional, sino mediante la composición de functores elementales. Así, las definiciones de functores son mucho más compactas. Esta es una necesidad fundamental del discurso teórico, el cuál sería más engorroso si no se apoyara en esta modularización de sus conceptos. Los functores elementales son los siguientes:

- El functor identidad se define como `I X = X, I f = f`
- Si `C` es un tipo, se define el functor constante $\overline{\texttt{C}}$ como $\overline{\texttt{C}}$ `X = C`, $\overline{\texttt{C}}$ `f = id`.
- Tenemos el bifunctor producto

  $$\begin{aligned}&\texttt{(X}\times\texttt{Y)} = \{\texttt{(x,y)} \mid \texttt{x} \in \texttt{X, y} \in \texttt{Y}\}\\ &\texttt{(f}\times\texttt{g) (x,y) = (f x,g y)}\end{aligned}$$

- El bifunctor producto se puede generalizar así:

  $$\begin{aligned}\texttt{X}_1 \times \cdots \times \texttt{X}_n &= \{(\texttt{x}_1,\ldots,\texttt{x}_n) | \texttt{x}_i \in \texttt{X}_i\}\\ (\texttt{f}_1 \times \cdots \times \texttt{f}_n)\ (\texttt{x}_1,\ldots,\texttt{x}_n) &= (\texttt{f}_1\ \texttt{x}_1,\ldots,\texttt{f}_n\ \texttt{x}_n)\end{aligned}$$

- El bifunctor suma disjunta:

  $$\begin{aligned}&\texttt{(X+Y) = \{1\}}\times\texttt{X} \cup \texttt{\{2\}}\times\texttt{Y} \cup \{\bot\}\\ &\texttt{(f+g) (1,x) = (1,f x)}\\ &\texttt{(f+g) (2,y) = (2,g y)}\\ &\texttt{(f+g)}\ \bot\ \texttt{=}\ \bot\end{aligned}$$

  En el tipo `X+Y` los elementos de `X` e `Y` se etiquetan con `1` y `2` para poder distinguir aquellos que se encuentran en la intersección de ambos tipos.

- El bifunctor suma disjunta también se puede generalizar:

  $$\begin{aligned}\texttt{X}_1 + \cdots + \texttt{X}_n &= (\{\texttt{1}\} \times \texttt{X}_1) \cup \cdots \cup (\{\texttt{n}\} \times \texttt{X}_n) \cup \{\bot\}\\ (\texttt{f}_1 + \cdots + \texttt{f}_n)\ \texttt{(i,x)} &= \texttt{(i,f}_i\ \texttt{x)}\\ (\texttt{f}_1 + \cdots + \texttt{f}_n)\ \bot &= \bot\end{aligned}$$

Utilizaremos con frecuencia el tipo unitario de *Haskell*, que vamos a denotar como **1**=`{()}`, y su functor constante $\overline{\mathbf{1}}$. O sea, este es el tipo que contiene un único elemento que es `()`.

El símbolo $\bot$ representa la computación indefinida, y es lo que devuelven las funciones cuando se encuentran indefinidas para el argumento que reciben. Semánticamente, interpretamos los tipos mediante ordenes parciales completos

con elemento mínimo (los que comúnmente se conocen como *pointed CPOs*) [AJ94].

Asumimos de aquí en adelante que el producto tiene mayor precedencia que la suma y que ambos tienen menor precedencia que la aplicación.

Usando estas construcciones podemos expresar el functor de nuestro ejemplo de la siguiente manera:

```
F a = Int + a×a
F f = id + f×f
```

Nótese la diferencia de esta definición *matemática* de `F` con la anterior.

```
F a = Either Int (a,a)
F f = fmap f
```

Básicamente, lo que hicimos fue cambiar el constructor de tipo `Either` por la suma y el par `(a,a)` por el producto `a×a`. En general utilizaremos la suma en vez de `Either` porque la manipulación de la suma es homogénea para cualquier cantidad de argumentos, mientras que el constructor de tipo `Either` sólo puede ser usado para representar la suma binaria. Utilizar el functor suma nos obliga a reescribir como términos de un tipo suma los términos que antes eran de tipo `Either a b`. Donde estaba escrito `Left i` escribimos `(1,i)`, y donde estaba puesto `Right (t1,t2)` ponemos `(2,(t1,t2))`. Estos ya no son términos de *Haskell*, pero a efectos de las manipulaciones que vamos a realizar ello no nos molesta. Por ejemplo la definición de la función `either` nos quedaría:

```
either :: (a->c) ->(b->c) -> a+b -> c
either f g (1,a) = f a
either f g (2,b) = g b
```

Aún podemos abreviar más la definición de `F`. Esto es posible mediante la definición de operadores que toman functores y devuelven otros functores.

Siendo `F` y `G` functores:

- El producto de functores se define como:

$$\begin{aligned}(\mathtt{F} \times \mathtt{G})\ X &= \mathtt{F}\ X \times \mathtt{G}\ X \\ (\mathtt{F} \times \mathtt{G})\ f &= \mathtt{F}\ f \times \mathtt{G}\ f\end{aligned}$$

- La suma de functores se define como:

$$\begin{aligned}(\mathtt{F} + \mathtt{G})\ X &= \mathtt{F}\ X + \mathtt{G}\ X \\ (\mathtt{F} + \mathtt{G})\ f &= \mathtt{F}\ f + \mathtt{G}\ f\end{aligned}$$

Ahora podemos expresar el functor `F` del ejemplo de manera muy compacta.

$$\mathtt{F} = \overline{\mathtt{Int}} + \mathtt{I} \times \mathtt{I}$$

Dado un functor `F`, llamaremos **posiciones recursivas** de `F` a aquellas posiciones de la definición de `F` donde ocurre el functor `I`. En el caso de arriba `F` tiene dos posiciones recursivas en el segundo sumando.

2.2. Álgebras y coálgebras

Hasta ahora nos hemos manejado en el discurso con la noción de sustitución de constructores por operaciones. Esto quiere decir que hay un conjunto de constructores a sustituir y un conjunto de operaciones sustitutas. Esta noción de conjunto de operaciones y constructores se formaliza mediante el concepto de álgebra.

Definición 2.4 (F-álgebra) *Dado un functor* `F` *y un tipo* `A`*, una* `F`*-álgebra es una función* ϕ`::F A->A`*.*

En nuestro primer ejemplo teníamos dos `F`-álgebras.

```
either Leaf Join :: F Btree -> Btree
either id (uncurry (+)) :: F Int -> Int
```

Podemos decir que la definición de `sqrLeaves'` se obtuvo a partir de reemplazar en `sqrLeaves` el `F`-álgebra `either Leaf Join` por la `F`-álgebra `either id (uncurry (+))`.

En la manipulación de `F`-álgebras resulta práctica la definición del siguiente combinador.

Definición 2.5 (Análisis de casos) *Sean tipos* `T` *y* $\mathtt{X}_1, \ldots, \mathtt{X}_n$ *y funciones* $\mathtt{f}_1 : \mathtt{X}_1 \to \mathtt{T}, \ldots, \mathtt{f}_n : \mathtt{X}_n \to \mathtt{T}$*:*

$$\begin{array}{rcl} \mathtt{f}_1 \triangledown \cdots \triangledown \mathtt{f}_n & : & \mathtt{X}_1 + \cdots + \mathtt{X}_n \to \mathtt{T} \\ (\mathtt{f}_1 \triangledown \cdots \triangledown \mathtt{f}_n)\ \mathtt{(i,x)} & = & \mathtt{f}_i\ x \end{array}$$

La siguiente propiedad compartida por el análisis de casos y el functor suma nos será de utilidad más adelante:

$$(\mathtt{f}_1 \circ \mathtt{g}_1) \triangledown \cdots \triangledown (\mathtt{f}_n \circ \mathtt{g}_n) = (\mathtt{f}_1 \triangledown \cdots \triangledown \mathtt{f}_n) \circ (\mathtt{g}_1 + \cdots + \mathtt{g}_n)$$

Ahora podemos escribir las álgebras del ejemplo de la siguiente manera

```
Leaf▽Join :: F Btree -> Btree
id▽(uncurry (+)) :: F Int -> Int
```

El combinador de análisis de casos es una generalización de la función `either`. Sirve para construir `F`-álgebras cuando el functor `F` es una suma de una cantidad arbitraria de argumentos. La función `either`, por el contrario, sólo nos permite construir `F`-álgebras para functores con sumas binarias.

El functor `F` juega el papel de signatura de una `F`-álgebra, i.e. especifica la *aridad* de las operaciones del álgebra. Por lo tanto, al reemplazar una `F`-álgebra por otra estamos en realidad sustituyendo operaciones por otras de igual aridad.

Dada un `F`-álgebra de la forma

$\phi = \phi_1 \triangledown \cdots \triangledown \phi_n$, donde $\phi_i =$`(\(`$v_{i1}, \ldots, v_{in_i}$`)->`t_i`)`

llamaremos **variables recursivas** del álgebra a aquellas variables v_{ij} que correspondan a posiciones recursivas de `F`.

En las leyes de fusión que presentaremos en breve, también juega un papel importante un concepto dual al de `F`-álgebra.

Definición 2.6 (F-coálgebra) *Dado un functor* `F` *y un tipo* `A`*, una* `F`*-coálgebra es una función* ψ`::A->F A`*.*

Una F-coálgebra puede pensarse como la parte de una función que descompone la entrada en subpartes sobre las cuales luego aplica las llamadas recursivas. Para ello se aplica fundamentalmente operaciones selectoras sobre el tipo de entrada. En *Haskell* las operaciones selectoras se apoyan fuertemente en la comparación de patrones (*pattern matching*). Las funciones `outF` y `c` de la figura 2.1 son ejemplos de F-coálgebras.

```
outF :: BTree -> F BTree
outF (Leaf i) = Left i
outF (Join (t1,t2)) = Right (t1,t2)

c :: BTree -> F BTree
c (Leaf i) = Left (sqr i)
c (Join (t1,t2)) = Right (t1,t2)
```

Dada un F-coálgebra de la forma

$$\backslash v_0\texttt{->case } t_0 \texttt{ of } p_1\texttt{->(1,(}t_{11},\ldots,t_{1m}\texttt{))};\ldots;p_n\texttt{->(n,(}t_{n1},\ldots,t_{nm}\texttt{))}$$

llamaremos **términos recursivos** de la coálgebra a aquellos términos t_{ij} que correspondan a posiciones recursivas de F.

2.3. Tipos y Functores

Existe una relación estrecha entre los tipos de datos y los functores. Consideremos una definición genérica de un tipo de dato:

$$\begin{aligned}\texttt{data T a}_1\ldots\texttt{a}_r = \texttt{C}_1\ &(t_{11},\ldots,t_{1k_1})\\ &\vdots\\ \texttt{|}\ \texttt{C}_n\ &(t_{n1},\ldots,t_{nk_n})\end{aligned}$$

Restringimos un poco la forma de esta definición exigiendo que cada t_{ij} sea

- o bien un tipo cualquiera que no sea mutuamente recursivo con $\texttt{T a}_1\ldots\texttt{a}_r$,
- o bien $\texttt{T a}_1\ldots\texttt{a}_r$ (los argumentos del constructor de tipo T son los mismos que aquellos que aparecen a la izquierda del =),
- o bien una de las variables $\texttt{a}_1\ldots\texttt{a}_r$.

Utilizando los constructores del tipo de dato es posible construir la siguiente F-álgebra:

$$\texttt{c}_1\triangledown\cdots\triangledown\texttt{c}_n\texttt{:: F (T a}_1\ldots\texttt{a}_r\texttt{) -> T a}_1\ldots\texttt{a}_r$$

tal que, si el constructor $\texttt{C}_i$ tiene argumentos, entonces $\texttt{c}_i = \texttt{C}_i$, y si no $\texttt{c}_i = \backslash\texttt{()->C}_i$. El functor F es de la forma $\texttt{F}_1 + \cdots + \texttt{F}_n$, donde cada $\texttt{F}_i$ captura la signatura (la aridad) del constructor $\texttt{C}_i$, siendo la misma de la forma $\texttt{F}_{i1} \times \cdots \times \texttt{F}_{ik_i}$, donde

$$\texttt{F}_{ij} \quad = \quad \begin{array}{ll}\texttt{I} & \text{si } t_{ij} = \texttt{T a}_1\ldots\texttt{a}_r\\ \overline{t_{ij}} & \text{en otro caso}\end{array}$$

Aquí puede verse que las llamadas *posiciones recursivas* del functor F se corresponden con las posiciones recursivas de los constructores del tipo $\texttt{T a}_1\ldots\texttt{a}_r$. Si el constructor $\texttt{C}_i$ no tiene argumentos, i.e. si es una constante, entonces $\texttt{F}_i = \overline{\mathbf{1}}$.

Nótese que los constructores se definen sin currificar. La teoría obliga a manejar los constructores de esta forma[1], pero en la implementación no hu-

[1] A partir de aquí nos despegamos de la notación de *Haskell* para listas, dado que `(:)` es un constructor currificado.

bo inconvenientes al manipular constructores no currificados y por lo tanto el sistema no hace distinciones al respecto.

Ejemplo 2.7 Consideremos el tipo de las listas.

```
data List a = Nil | Cons (a,List a)
```

A partir de esta definición podemos construir la F-álgebra `(\()->Nil)▽Cons` cuyo functor es

$$\mathtt{F_a} = \overline{\mathtt{1}} + \overline{\mathtt{a}} \times \mathtt{I}$$

□

Usualmente se conoce a la F-álgebra de constructores como $in_{\mathtt{F}}$, y a su inversa como $out_{\mathtt{F}}$. Estas dos funciones definen una biyección entre los tipos

$$\mathtt{F\ (T\ a_1 \ldots a_r)} \underset{out_{\mathtt{F}}}{\overset{in_{\mathtt{F}}}{\rightleftarrows}} \mathtt{T\ a_1 \ldots a_r}$$

Estas álgebras y coálgebras específicas serán de interés al presentar formalmente las leyes de fusión.

Las restricciones impuestas sobre $\mathtt{T\ a_1 \ldots a_r}$ nos garantizan que el tipo tendrá un functor asociado compuesto por sumas y productos de functores `I` y $\overline{\mathtt{C}}$ para diversos tipos `C`. Estos functores son llamados *polinomiales*, y las estructuras de datos asociadas son las únicas que pueden ser eliminadas por el sistema de fusión.

2.4. Hilomorfismos

La forma en que hemos factorizado las funciones para luego combinarlas, lejos de ser arbitraria, es una manera de expresarlas como *hilomorfismos*.

Definición 2.8 (Hilomorfismo) *Sea un functor* `F`*, una* `F`*-álgebra* ϕ *y una* `F`*-coálgebra* ψ*. Un hilomorfismo es el mínimo punto fijo de la ecuación*

$$h = \phi \circ \mathtt{F}\ h \circ \psi$$

el cuál se denota $[\![\phi, \psi]\!]_{\mathtt{F}}$*.*

La practicidad de tener un hilomorfismo es la misma que argumentamos al factorizar las funciones. Tenemos bien separadas las etapas de preparación de los argumentos para las llamadas recursivas, las llamadas recursivas, y la combinación del resultado de estas llamadas.

Las definiciones del ejemplo

```
sumBT :: BTree -> Int
sumBT = either id (+) . g sumBT . outF

sqrLeaves :: BTree -> BTree
sqrLeaves = either Leaf Join . g sqrLeaves . c
  where c (Leaf i) = Left (sqr i)
        c (Join pair) = Right pair
```

serían escritas como hilomorfismos de la siguiente manera:

```
sumBT = ⟦id▽uncurry (+),outF⟧_{Int+I×I}

sqrLeaves = ⟦Leaf▽Join,c⟧_{Int+I×I}
  where c (Leaf i) = (1,sqr i)
        c (Join pair) = (2,pair)
```

Son usuales en la literatura algunas especializaciones del operador hilomorfismo. Tenemos, por ejemplo, los *catamorfismos*, que son hilomofismos de la forma $[\![\phi, out_{\mathtt{F}}]\!]_{\mathtt{F}}$, los cuales se denotan $(\!|\phi|\!)_{\mathtt{F}}$. Los catamorfismos representan funciones definidas por recursión estructural [BdM97]. Tenemos también los *anamorfismos*, que son hilomofismos de la forma $[\![in_{\mathtt{F}}, \psi]\!]_{\mathtt{F}}$, denotados como $[\!(\psi)\!]_{\mathtt{F}}$. En este caso representan funciones definidas por lo que se conoce como correcursión [GJ98, GH00].

Los hilomorfismos gozan de las siguientes leyes de fusión.

Teorema 2.9 (Lluvia ácida)

cata-ana:
$$[\![\phi, out_{\mathtt{F}}]\!]_{\mathtt{F}} \circ [\![in_{\mathtt{F}}, \psi]\!]_{\mathtt{F}} = [\![\phi, \psi]\!]_{\mathtt{F}}$$

cata-hilo:
$$\frac{\tau\texttt{:: (F a->a)->(G a->a)}}{[\![\phi, out_{\mathtt{F}}]\!]_{\mathtt{F}} \circ [\![\tau(in_{\mathtt{F}}), \psi]\!]_{\mathtt{G}} = [\![\tau(\phi), \psi]\!]_{\mathtt{G}}}$$

hilo-ana:
$$\frac{\sigma\texttt{:: (a->F a)->(a->G a)}}{[\![\phi, \sigma(out_{\mathtt{F}})]\!]_{\mathtt{G}} \circ [\![in_{\mathtt{F}}, \psi]\!]_{\mathtt{F}} = [\![\phi, \sigma(\psi)]\!]_{\mathtt{G}}}$$

El caso cata-ana es la formalización de la ley que aplicamos en el ejemplo. La `F`-álgebra $in_{\mathtt{F}}$ era `either Leaf Join` y su inversa $out_{\mathtt{F}}$ era la función `outF`.

La ley cata-hilo se basa en la misma idea de reemplazar los constructores del tipo intermedio (contenidos en la `F`-álgebra $in_{\mathtt{F}}$) por las operaciones de otra `F`-álgebra y por lo tanto se puede considerar una generalización de la primer ley. En este caso los constructores se encuentran dentro de un término descripto por τ.

La primer ley de fusión puede también verse como el proceso por el cual se reemplaza la coálgebra $out_{\mathtt{F}}$ por la coálgebra ψ. De acuerdo a esta interpretación, podría considerarse el caso hilo-ana también como una generalización de la primer ley para casos en que hay varias aplicaciones de $out_{\mathtt{F}}$ que deben ser reemplazadas.

Cuando una función tiene un tipo como el de τ o σ, se dice que la función es un transformador de álgebras/coálgebras de signatura `F` en álgebras/coálgebras de signatura `G` [Fok92].

Ejemplo 2.10 (Fusión con τ) Para ver un caso de fusión con τ consideremos el siguiente programa, que cuenta la cantidad de elementos en una lista que satisfacen un predicado dado.

```
lf :: (a->Bool) -> List a -> Int
lf p = length . filter p
```

Primero escribamos como hilomorfismos las funciones `length` y `filter`.

```
filter :: (b->Bool) ->List b ->List b
filter p = [[(\()->Nil)▽φ2, ψ]]G
  where G = 1̄ + b̄ × I × I
        φ2 (b,v1,v2) = if p b then Cons (b,v1) else v2
        ψ Nil = (1,())
        ψ (Cons (b,ls)) = (2,(b,ls,ls))

length :: List b ->Int
length = [[(\()->0)▽φ2, outF]]F
  where F = 1̄ + b̄ × I
        φ2 (_,v1) = 1+v1
```

Ahora escribimos el álgebra de `filter` como $\tau(in_{\mathtt{F}})$.

```
filter p = [[(τ(inF), ψ]]G
  where F = 1̄ + b̄ × I
        G = 1̄ + b̄ × I × I
        inF = (\()->Nil)▽Cons
        τ :: (F a->a) ->(G a->a)
        τ(α) = τ1(α)▽τ2(α)
        τ1(α1▽α2) = α1
        τ2(α1▽α2) (a,v1,v2) = if p a then α2 (a,v1)
                                     else v2
```

Si aplicamos la ley cata-hilo tenemos que podemos reescribir el programa `lf` como

```
lf :: (a->Bool) ->List a ->Int
lf p = [[τ((\()->0)▽φ2), ψ]]G
  where φ2 (_,v1) = 1+v1
```

La definición en *Haskell*:

```
lf :: (a->Bool)->List a ->Int
lf p Nil = 0
lf p (Cons (b,ls)) = if p b then 1+lf ls else lf ls
```

□

Ejemplo 2.11 (Fusión con σ) Para ver un caso de fusión con σ consideremos la composición `impares.upto n`. La función `impares` construye una lista con los elementos en posiciones impares de otra lista. La función `upto` construye la lista de enteros entre dos números que se le dan como argumentos.

```
impares :: List b -> List b
impares Cons (b,Cons (_,bs)) = Cons (b,impares bs)
impares Cons (a,Nil) = Cons (a,Nil)
impares Nil = Nil

upto :: Int -> Int -> List Int
upto n m = if m>n then Nil
                  else Cons (m,upto n (m+1))
```

Así, la composición `impares.upto n` calcula la lista de números impares hasta `n` cuando se aplica sobre el valor `1`. Escribimos las funciones como hilomorfismos.

```
impares = ⟦Cons▽id,ψ⟧G
  where G = b̄ × I + List b
        ψ (Cons (b,Cons(_,bs))) = (1,(b,bs))
        ψ (Cons (b,Nil)) = (2,Cons (b,Nil))
        ψ Nil = (2,Nil)

upto n = ⟦inF,c⟧Int+I×I
  where F = 1̄ + Int × I
        c m = if m>n  then (1,())
                      else (2,(m,m+1))
```

Ahora escribimos la coálgebra de `impares` como $\sigma(out_{\mathtt{F}})$.

```
impares = ⟦uncurry (:)▽id,σ(outF)⟧G
  where G = b̄ × I + List b
        F = Int + I × I
        outF l = case l of
                   Nil -> (1,())
                   Cons (i,is) -> (2,(i,is))
        σ :: (a->F a)->(a->G a)
        σ(β) l = case (β l) of
                   (1,()) ->(2,Nil)
                   (2,(i,is)) -> case (β is)
                                   (1,()) ->(2,Cons (i,Nil))
                                   (2,(_,is')) ->(1,(i,is'))
```

Si aplicamos la ley para el caso hilo-ana tenemos que `impares.upto n` es lo mismo que

```
iu = ⟦Cons▽id,σ(c)⟧G
  where c :: Int ->F Int
        c m = if m>n  then (1,())
                      else (2,(m,m+1))
```

El resultado de la fusión corresponde a la siguiente definición en *Haskell*.

```
iu :: Int -> Int -> List Int
iu n m = if m>n then Nil
           else if m+1>n then Cons (m,Nil)
                  else Cons (m,iu n ((m+1)+1))
```

□

2.5. Transformaciones naturales

Por último, necesitamos introducir la noción de transformación natural. Esta noción será necesaria en la Sección 3.2 cuando hablemos de procesamientos previos a la fusión.

Informalmente, una transformación natural es una función que recibe una estructura de datos con elementos de un tipo A cualquiera, y retorna otra estructura conteniendo elementos del mismo tipo.

Definición 2.12 (Transformación natural) *Sean dos functores* `F` *y* `G`. *Una transformación natural* η : `F` $\Rightarrow$ `G` *es una función polimórfica* η`::F a->G a`.

El tipo polimórfico de la función η asegura la siguiente propiedad:

$$\eta \circ \texttt{F f} = \texttt{G f} \circ \eta$$

para toda `f::A->B`. Esto es consecuencia de las propiedades de parametricidad (más conocidas como los *free theorems* [Wad89]) asociadas a toda función polimórfica.

Ejemplo 2.13

```
      η  ::  (ℕ,a)->a+(ℕ,a,a)
η (0,a)  =   (1,a)
η (n,a)  =   (2,(n-1,a,a))
```

Podríamos escribir el tipo de esta función como

```
η::(ℕ̄×I) a -> (I+ℕ̄×I×I) a
```

Se ve aquí que el valor específico del elemento de tipo `a` no tiene influencia en el comportamiento de la función, se utiliza para construir una terna simplemente copiándolo a las posiciones en la estructura de salida. □

Enumeramos algunas propiedades importantes de las transformaciones naturales. Sean $\eta_1 : \texttt{F} \Rightarrow \texttt{G}$ y $\eta_2 : \texttt{F}' \Rightarrow \texttt{G}'$:

- $\eta_1 \times \eta_2 : \texttt{F} \times \texttt{G} \Rightarrow \texttt{F}' \times \texttt{G}'$
- $\eta_1 + \eta_2 : \texttt{F} + \texttt{G} \Rightarrow \texttt{F}' + \texttt{G}'$
- si $\texttt{F} = \texttt{G}'$ entonces $\eta_1 \circ \eta_2 : \texttt{F}' \Rightarrow \texttt{G}$

Existe también una propiedad conocida como *hylo-shift* que relaciona las transformaciones naturales y los hilomorfismos.

$$\frac{\eta : \texttt{F} \Rightarrow \texttt{G}}{[\![\phi \circ \eta, \psi]\!]_{\texttt{F}} = [\![\phi, \eta \circ \psi]\!]_{\texttt{G}}}$$

Capítulo 3

Algoritmos

El sistema desarrollado en este proyecto tiene un ciclo de ejecución compuesto por varias etapas. A lo largo de este capítulo presentaremos los algoritmos que utiliza el sistema y que conforman las distintas etapas. Otras descripciones de estos algoritmos pueden encontrarse en [OHIT97, Sch00].

1. **Construcción de una representación interna de las funciones recursivas.** Este paso podría considerarse el puente de entrada al sistema. Aquí las funciones cuyas definiciones están expresadas en un lenguaje ajeno al sistema, en nuestro caso *Haskell*, son traducidas a una forma en que resulte sencillo realizar los análisis posteriores. Concretamente, se utiliza una representación interna en términos de hilomorfismos.

2. **Reconocimiento de la forma de los hilomorfismos.** Una vez que una definición ha sido incorporada al sistema, se le realiza un análisis para determinar si sus componentes satisfacen las hipótesis de alguna ley de fusión. En esta etapa se responden preguntas como: ¿ Este hilomorfismo es un catamorfismo? ¿ Es un anamorfismo? ¿ Su álgebra es de la forma $\tau(in_{\mathtt{F}})$? ¿ Su álgebra es de la forma $\sigma(out_{\mathtt{F}})$? ¿ Hay alguna transformación que se le pueda hacer a este hilomorfismo para obtener otro equivalente que se pueda fusionar? Los algoritmos de esta etapa son algoritmos que reestructuran los hilomorfismos, algoritmos que reconocen la forma de álgebras y coálgebras, y algoritmos que derivan transformadores a partir de ellas. Algunos de los algoritmos de reestructura que se presentan en la Sección 3.2 han sido propuestos durante este proyecto.

3. **Aplicación de las leyes de fusión.** Cada ley puede considerarse como una función (parcial) que recibe dos definiciones de funciones `f` y `g`, y retorna la definición de la función resultante de fusionar `f . g`. Dadas dos funciones que se quieran fusionar, si sus partes satisfacen las hipótesis de alguna ley de fusión entonces se les aplica la ley. En la práctica, en esta etapa debiera haber un algoritmo que buscase dentro de un programa composiciones de funciones fusionables. Dicho algoritmo no forma parte de la versión actual. Su estudio e implementación figura en la planificación de extensiones futuras a la herramienta. En la implementación actual el usuario del sistema es quien indica que composiciones deben ser fusionadas.

4. **Escritura del resultado de la fusión en *Haskell*.** Esta etapa es el puente de salida del sistema. Aquí las funciones expresadas como hilomor-

$$
\begin{array}{lcll}
\textit{program} & ::= & v{=}t & \\
 & & \vdots & \\
 & & v{=}t & \\
t & ::= & v & \text{(variables)} \\
 & | & l & \text{(literales)} \\
 & | & (t, \ldots, t) & \text{(tuplas)} \\
 & | & \backslash b\texttt{->}t & \text{(lambda expresiones)} \\
 & | & \texttt{let } v{=}t \texttt{ in } t & \text{(expresiones let)} \\
 & | & \texttt{case } t \texttt{ of} & \text{(expresiones case)} \\
 & & \quad p\texttt{->}t & \\
 & & \quad \vdots & \\
 & & \quad p\texttt{->}t & \\
 & | & v\ t \ldots t & \text{(aplicación de funciones)} \\
 & | & c\ t \ldots t & \text{(aplicación de constructores)} \\
 & | & t\ t & \text{(aplicación general de términos)} \\
 & | & \bot & \text{(el término indefinido)} \\
b & ::= & v & \text{(variables)} \\
 & | & (b, \ldots, b) & \text{(tuplas de variables)} \\
p & ::= & v & \text{(variables)} \\
 & | & (p, \ldots, p) & \text{(tuplas de patrones)} \\
 & | & c(p, \ldots, p) & \text{(aplicación de constructores)} \\
 & | & l & \text{(literales)}
\end{array}
$$

Figura 3.1: Gramatica de términos

fismos son traducidas de nuevo al lenguaje original.

3.1. Derivación de hilomorfismos

Siendo la primer etapa la que recibe las definiciones de funciones, se hace necesario definir la gramática en la cual se hayan expresadas estas definiciones. En la figura 3.1 presentamos nuestra gramática de términos.

La gramática es un subconjunto de *Haskell* lo bastante amplio como para que todas las definiciones del lenguaje se puedan escribir de alguna manera con esta sintaxis. Las definiciones son tan sólo un identificador que tiene asociado un término. Para nuestra implementación no hemos requerido ninguna información de tipos.

Las aplicaciones de funciones y constructores se distingue de la aplicación general. Esto es porque resulta práctico para los algoritmos tener en una lista todos los argumentos de un constructor o una llamada recursiva.

En la figura 3.2 presentamos el algoritmo de derivación de hilomorfismos de Onoue et al. Las definiciones que acepta el algoritmo son de la siguiente forma

$$\texttt{f = } \backslash v_1 \ldots v_m \texttt{ ->case } t_0 \texttt{ of } p_1 \texttt{ ->} t_1 \texttt{;} \ldots \texttt{;} p_n \texttt{->} t_n$$

Se asume que la función hace recursión sobre un único argumento y que ese argumento es el último (v_m). Esto quiere decir que todos los argumentos $v_1, \ldots, v_{m-1}$ son constantes, i.e. que las llamadas recursivas tienen los mismos argumentos en las primeras $m - 1$ posiciones. También se asume que el término está bien tipado.

$$\mathcal{H}(\texttt{f,}\backslash v_1 \ldots v_m \texttt{ ->case } t_0 \texttt{ of } p_1 \texttt{ ->}t_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}t_n) =$$
$$\backslash v_1 \ldots v_{m-1} \texttt{ ->}[\![\phi_1 \triangledown \cdots \triangledown \phi_n, \psi]\!]_{\texttt{F}}$$

where $\psi = \backslash v_m$ `->` `case` t_0 `of`

$$p_1 \texttt{ ->(1,(}v_{11}, \ldots, v_{1l_1}, t'_{11}, \ldots, t'_{1s_1}\texttt{))}$$
$$\vdots$$
$$p_n \texttt{ ->(n,(}v_{n1}, \ldots, v_{nl_n}, t'_{n1}, \ldots, t'_{ns_n}\texttt{))}$$

$\phi_i = \backslash\texttt{(}v_{i1}, \ldots, v_{il_i}, u_1, \ldots, u_{s_i}\texttt{)->}t''_i$

$\texttt{F} = \texttt{F}_1 + \cdots + \texttt{F}_n$

$\texttt{F}_i = \Gamma(v_{i1}) \times \cdots \times \Gamma(v_{il_i}) \times \texttt{I}_1 \times \cdots \times \texttt{I}_{s_i}$

$(\{v_{i1}, \ldots, v_{il_i}\}, \{(u_1, t'_{i1}), \ldots, (u_{s_i}, t'_{is_i})\}, t''_i) = \mathcal{D}(p_i, t_i)$

$$\mathcal{D}(p_i\texttt{,v}) = (\{\texttt{v}\}, \emptyset\texttt{,v}) \quad \text{si } \texttt{v} \in \mathit{vars}(p_i) \cup \{v_m\}$$
$$(\{\}\texttt{,}\emptyset\texttt{,v}) \quad \text{en otro caso}$$

$$\mathcal{D}(p_i\texttt{,(t1,}\ldots\texttt{,tn)}) = (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n\texttt{,(t1',}\ldots\texttt{,tn')})$$
where $(c_i, c'_i\texttt{,ti'}) = \mathcal{D}(p_i\texttt{,ti})$

$$\mathcal{D}(p_i, C_j \texttt{ (t1,}\ldots\texttt{,tn)}) = (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n, C_j \texttt{ (t1',}\ldots\texttt{,tn')})$$
where $(c_i, c'_i\texttt{,ti'}) = \mathcal{D}(p_i\texttt{,ti})$

$$\mathcal{D}(p_i\texttt{,g t1}\ldots\texttt{tm)}) = (\{\}, \{\texttt{(u,tm)}\}\texttt{,u}) \quad \text{si } \texttt{g=f y } v_i = \texttt{ti para } i < m$$
$$(c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n\texttt{,g t1'}\ldots\texttt{tn')} \quad \text{si no}$$
where $(c_i, c'_i\texttt{,ti'}) = \mathcal{D}(p_i\texttt{,ti})$

$$\mathcal{D}(p_i\texttt{,let v=t0 in t1}) = (c_0 \cup c_1, c'_0 \cup c'_1\texttt{,let v=t0' in t1'})$$
where $(c_i, c'_i\texttt{,ti'}) = \mathcal{D}(p_i\texttt{,ti})$

$$\mathcal{D}(p_i\texttt{,}\backslash\texttt{v ->t}) = (c, c'\texttt{,}\backslash\texttt{v ->t'})$$
where $(c, c'\texttt{,t'}) = \mathcal{D}(p_i\texttt{,t})$

$$\mathcal{D}(p_i\texttt{,case } t_0 \texttt{ in } p_1\texttt{->}t_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}t_n) =$$
$$(c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n\texttt{,case } t_0 \texttt{ of } p_1\texttt{->}t'_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}t'_n)$$
where $(c_i, c'_i, t'_i) = \mathcal{D}(p_i, t_i)$

Figura 3.2: Algoritmo de derivación de hilomorfismos

La definición que devuelve el algoritmo es de la forma

$$\texttt{f = } \backslash v_1 \ldots v_{m-1} \texttt{ ->}[\![\phi_1 \triangledown \cdots \triangledown \phi_n, \psi]\!]$$

Cada componente ϕ_i se construye a partir de cada alternativa t_i. En t_i se reemplazan las llamadas recursivas por variables frescas que representan el resultado de esas llamadas recursivas. El algoritmo $\mathcal{D}$ es el que se encarga de obtener una variable fresca `u` distinta para cada ocurrencia de una llamada recursiva. En la coálgebra se retornan todas los variables ligadas por los patrones p_i que se usan en cada t_i y los argumentos de las llamadas recursivas.

El algoritmo tiene algunos detalles técnicos que están implícitos. Ninguna de las variables de entrada $v_1 \ldots v_{m-1}$ es capturada dentro de los términos. Esto quiere decir que ninguna construcción `case` contiene en sus patrones tales variables, ninguna lambda abstracción las usa como variables de entrada, y ninguna construcción `let` las redeclara. Esta condición es necesaria para que el control que hace el algoritmo $\mathcal{D}$ antes de abstraer una llamada recursiva tenga sentido. Una manera de esquivar el problema es renombrar todas las variables con nombres únicos en una etapa previa, pero este enfoque produce programas difíciles de leer dado que todas sus variables serían generadas. La solución que

hemos adoptado en la implementación verifica si los argumentos constantes de las llamadas recursivas son las variables de entrada o variables capturadas, en el segundo caso el algoritmo falla.

3.2. Reconocer esquemas de producción y consumo de datos

Para poder aplicar el teorema de la lluvia ácida sobre una composición de hilomorfismos, es necesario reconocer cuál es el tipo de álgebra y coálgebra que estos hilomorfismos tienen. En la situación

$$[\![\phi, \psi]\!]_{\mathtt{F}} \circ [\![\phi', \psi']\!]_{\mathtt{G}}$$

queremos conocer la forma de ψ y ϕ', para saber si estamos en alguna de las tres situaciones en que podemos aplicar el Teorema de Lluvia Ácida. El título de esta sección se debe a que ψ indica cómo se consume la estructura de datos intermediaria de los hilomorfismos y ϕ' indica cómo se produce esa estructura. En las siguientes sub-secciones estaremos intentando responder las siguientes preguntas

- ¿$\phi' = in_{\mathtt{G}}$?
- ¿$\psi = out_{\mathtt{F}}$?
- ¿$\phi' = \tau(in_{\mathtt{F}})$?
- ¿$\psi = \sigma(out_{\mathtt{G}})$?

El reconocimiento de esquemas de producción y consumo de datos va muy de la mano con técnicas de reestructuración que facilitan el reconocimiento. Dado un hilomorfismo $[\![\phi, \psi]\!]_{\mathtt{F}}$ la esencia de las reestructuras es descomponer ϕ en $\phi'' \circ \eta$, siendo una transformación natural $\eta : \mathtt{F} \Rightarrow \mathtt{G}$. Luego podemos aplicar la propiedad de *hylo-shift* vista en la Sección 2.5, para llegar a

$$[\![\phi, \psi]\!]_{\mathtt{F}} = [\![\phi'', \eta \circ \psi]\!]_{\mathtt{G}}$$

donde ϕ'' sería posiblemente una función más simple de analizar que ϕ.

De manera análoga podemos realizar la descomposición dual ψ en $\eta \circ \psi''$, siendo $\eta : \mathtt{G} \Rightarrow \mathtt{F}$. Así tendremos que

$$[\![\phi, \psi]\!]_{\mathtt{F}} = [\![\phi \circ \eta, \psi'']\!]_{\mathtt{G}}$$

3.2.1. Reconocer $in_{\mathtt{F}}$

Nos enfrentamos ahora a la pregunta ¿$\phi' = in_{\mathtt{G}}$? La forma trivial que puede tener $\phi' : \mathtt{G}\ A \to A$ que nos asegura que es $in_{\mathtt{G}}$ es $C_1 \triangledown \cdots \triangledown C_n$ donde $C_1, \ldots, C_n$ son todos los constructores del tipo A (o `(\()->`C_i`)` en el caso de que C_i no tenga argumentos).

Es claro que hay muchos términos que pueden ser equivalentes a un término de la forma que acabamos de describir. Pero nosotros sólo reconoceremos $in_{\mathtt{G}}$ si se encuentra en esta forma con la esperanza de que sea suficiente para un número considerable de casos. Este también ha sido en esencia el enfoque de Schwartz[Sch00]. Onoue et al.[OHIT97] proponen aplicar ϕ e $in_{\mathtt{F}}$ a un cierto conjunto de valores para comprobar que den los mismo resultados. Como se

elige el conjunto de valores de prueba y qué certeza dan de que las funciones sean equivalentes no queda claro en la presentación que hacen de la técnica.

Cuando el álgebra no está en forma $in_{\mathtt{G}}$, hay varias reestructuras que se pueden realizar para obtener un hilomorfismo con álgebra $in_{\mathtt{G}}$. De las reestructuras que presentamos a continuación, la primera es una síntesis de las reestructuras propuestas por Onoue et al. y Schwartz, las dos reestructuras que se presentan luego fueron identificadas y descritas en el transcurso del proyecto.

Moviendo argumentos

Dada un álgebra $\phi = \phi_1 \triangledown \cdots \triangledown \phi_n$ donde cada ϕ_i es de la forma

$$\phi_i\texttt{=\textbackslash(}v_{i1},\ldots,v_{ik_i}\texttt{)->}C_i\ \ \texttt{(}t_{i1},\ldots,t_{il_i}\texttt{)}$$

Podemos reestructurar cada ϕ_i como $C_i \circ \eta_i$ donde

$$\eta_i\texttt{=\textbackslash(}v_{i1},\ldots,v_{ik_i}\texttt{)->(}t_{i1},\ldots,t_{il_i}\texttt{)}$$

Luego, por propiedad del análisis de casos, podemos descomponer el álgebra de esta manera:

$$\phi = (C_1 \circ \eta_1) \triangledown \cdots \triangledown (C_n \circ \eta_n) = (C_1 \triangledown \cdots \triangledown C_n) \circ (\eta_1 + \cdots + \eta_n)$$

Si cada η_i es una transformación natural, entonces $(\eta_1 + \cdots + \eta_n)$ sería una transformación natural que podríamos mover a la coálgebra. La condición que imponemos para que cada η_i sea una transformación natural es que cada t_{ij} sea, o bien una variable, o bien un término que no referencia variables recursivas.

Ejemplo 3.1 (Reestructura de `mirror`) Consideremos, por ejemplo, la función `mirror`.

```
data Tree a = Empty | Node (a,Tree a,Tree a)

mirror : Tree a -> Tree a
mirror (Node (a,t1,t2)) = Node (a,mirror t1,mirror t2)
mirror Empty = Empty
```

Vista como hilomorfismo tiene la siguiente forma:

$$\begin{aligned}
\texttt{mirror} &= [\![\phi, out_{\mathtt{F}}]\!]_{\mathtt{F}} \\
\texttt{F} &= \overline{\texttt{1}} + \overline{\texttt{a}} \times \texttt{I} \times \texttt{I} \\
\phi &\texttt{ :: F (Tree a) -> Tree a} \\
\phi &\texttt{ = (\textbackslash()->Empty)}\triangledown\texttt{(\textbackslash(a,v1,v2)->Node (a,v2,v1))} \\
out_{\mathtt{F}} &\texttt{ = \textbackslash l-> case l of} \\
&\qquad\texttt{Empty -> (1,())} \\
&\qquad\texttt{Node (a,t1,t2) -> (2,(a,t1,t2))}
\end{aligned}$$

Si descomponemos el álgebra de la siguiente manera

$$\phi\texttt{ = (\textbackslash()->Empty)}\triangledown\texttt{Node} \circ \texttt{(id+(\textbackslash(a,v1,v2)->(a,v2,v1)))}$$

siendo una transformación natural $\texttt{(id+(\textbackslash(a,v1,v2)->(a,v2,v1))):F} \Rightarrow \texttt{F}$, podemos reestructurar el hilomorfismo obteniendo

$$[\![\phi, out_{\mathtt{F}}]\!]_{\mathtt{F}}\texttt{=}[\![in_{\mathtt{F}}\texttt{,(id+(\textbackslash(a,v1,v2)->(a,v2,v1)))} \circ out_{\mathtt{F}}]\!]_{\mathtt{F}}$$

□

Ejemplo 3.2 (Reestructura de `map`) Pensemos ahora en la función `map`

```
map : (a->b) -> List a -> List b
map f (Cons (a,as)) = Cons (f a,map as)
map f Nil = Nil
```

que vista como hilomorfismo tiene la siguiente forma

$$\begin{aligned}
\texttt{map f} &= [\![\phi, out_{\texttt{F}}]\!]_{\texttt{F}} \\
\texttt{F} &= \overline{\texttt{1}} + \overline{\texttt{a}} \times \texttt{I} \\
\phi &: \texttt{F (List a)->List a} \\
\phi &= \texttt{(\textbackslash()->Nil)} \triangledown \texttt{(\textbackslash(a,v)->Cons (f a,v))} \\
out_{\texttt{F}} &= \texttt{\textbackslash l-> case l of} \\
&\qquad \texttt{Nil -> (1,())} \\
&\qquad \texttt{Cons (i,is) -> (2,(i,is))}
\end{aligned}$$

El álgebra se descompone de la siguiente manera

$$\phi = \texttt{(\textbackslash()->Nil)} \triangledown \texttt{Cons} \circ \texttt{(id+(\textbackslash(a,v)->(f a,v)))}$$

siendo una transformación natural $\texttt{(id+(\textbackslash(a,v)->(f a,v))):F} \Rightarrow \overline{\texttt{1}} + \overline{\texttt{b}} \times \texttt{I}$. Reestructurando tenemos

$$\begin{aligned}
[\![\phi, out_{\texttt{F}}]\!]_{\texttt{F}} &= [\![in_{\texttt{G}}, \texttt{(id+(\textbackslash(a,v)->(f a,v)))} \circ out_{\texttt{F}}]\!]_{\texttt{G}} \\
\texttt{G} &= \overline{\texttt{1}} + \overline{\texttt{b}} \times \texttt{I}
\end{aligned}$$

□

Agregando constructores

Sea un functor $\texttt{F} = \texttt{F}_1 + \cdots + \texttt{F}_k$. Dada un álgebra

$$\phi = (C_1 \triangledown \cdots \triangledown C_k) \texttt{ :: F A -> A}$$

donde hay constructores de `A` que no aparecen entre C_1 y C_k. Podemos reestructurar el álgebra como

$$\phi = (C_1 \triangledown \cdots \triangledown C_k \triangledown \cdots \triangledown C_n) \circ g$$

donde g se define

$$\begin{aligned}
\texttt{g} &:: \texttt{F A->F' A} \\
\texttt{g} &= \texttt{(\textbackslash vs->(1,vs))} \triangledown \cdots \triangledown \texttt{(\textbackslash vs->(k,vs))} \\
\texttt{F}' &= \texttt{F}_1 + \cdots + \texttt{F}_k + \texttt{F}_{k+1} + \cdots + \texttt{F}_n
\end{aligned}$$

g juega el papel de conversor del tipo `F A` en el tipo `F' A` que lo contiene. Entran valores de una suma de k sumandos y retorna una suma de n sumandos tal que los únicos sumandos que devuelve son los k primeros. Esto hace que ϕ preserve su comportamiento. Nótese que g es una transformación natural entre `F` y $\texttt{F}'$.

Ejemplo 3.3 (Reestructura de `repeat`) Sea la función

```
repeat : a -> List a
repeat a = Cons (a,repeat a)
```

Si derivamos su forma de hilomorfismo obtenemos

$$\begin{aligned}
\texttt{repeat} &= [\![\phi, \psi]\!]_{\texttt{F}} \\
\texttt{F} &= \overline{\texttt{a}} \times \texttt{I} \\
\phi &: \texttt{F (List a)->List a} \\
\phi &= \texttt{Cons} \\
\psi &= \texttt{\textbackslash a->(a,a)}
\end{aligned}$$

Podemos descomponer el álgebra en

$$\phi \texttt{ = (Cons}\triangledown\texttt{(\textbackslash()->Nil)) } \circ \texttt{ (\textbackslash vs->(1,vs))}$$

siendo $\texttt{(\textbackslash vs->(1,vs))}:\overline{\texttt{a}} \times \texttt{I} \Rightarrow \overline{\texttt{a}} \times \texttt{I} + \overline{\mathbf{1}}$. Reestructurando tenemos

$$\begin{aligned} [\![\phi, \psi]\!]_{\texttt{F}} &= [\![in_{\texttt{G}}, \texttt{(\textbackslash vs->(1,vs))} \circ \psi]\!]_{\texttt{G}} \\ \texttt{G} &= \overline{\texttt{a}} \times \texttt{I} + \overline{\mathbf{1}} \\ in_{\texttt{G}} &= \texttt{Cons}\triangledown\texttt{(\textbackslash()->Nil)} \end{aligned}$$

□

Moviendo `cases`

Dado un término ϕ de la siguiente forma

$$\begin{aligned} \phi\texttt{=\textbackslash(}v_1, \ldots, v_k\texttt{)->case } t_0 \texttt{ of} \\ p_1\texttt{->}t_1 \\ \vdots \\ p_m\texttt{->}t_m \end{aligned}$$

podemos descomponer el término como

$$\begin{aligned} &\phi = (g_1 \triangledown \cdots \triangledown g_m) \circ \eta \\ &\textit{where } \eta \texttt{ = \textbackslash(}v_1, \ldots, v_k\texttt{)->case } t_0 \texttt{ of} \\ &\qquad\qquad p_1\texttt{->(1,}bv(p_1, t_1)\texttt{)} \\ &\qquad\qquad \vdots \\ &\qquad\qquad p_m\texttt{->(m,}bv(p_m, t_m)\texttt{)} \end{aligned}$$

donde $g_i\texttt{=\textbackslash}bv(p_i, t_i)\texttt{->}t_i$ y donde $bv(p_i, t_i)$ denota las variables que aparecen en t_i y que ocurren en el patrón p_i o entre las variables de entrada $\{v_1, \ldots, v_k\}$.

Si $\phi\texttt{::Fa->a}$ y t_0 no referencia variables recursivas, entonces podemos asegurar que η es una transformación natural.

Utilizando esta estrategia, podemos factorizar un álgebra $\phi_1 \triangledown \cdots \triangledown \phi_n$ como

$$\phi_1 \triangledown \cdots \triangledown \phi_n = (\phi'_1 \triangledown \cdots \triangledown \phi'_n) \circ (\eta_1 + \cdots + \eta_n)$$

donde:

- η_i acarrea una estructura `case` cuando la misma se encontrase originalmente en ϕ_i. El correspondiente ϕ'_i sería un análisis de casos de la forma $g_{i_1} \triangledown \cdots \triangledown g_{i_{m_i}}$.
- $\eta_i = id$ si ϕ_i no tiene ninguna estructura `case`.

En caso de que alguna ϕ_i tenga más de una estructura `case` se puede volver a factorizar ϕ'_i de la misma manera.

Ejemplo 3.4 (reestructura de `upto`) Consideremos la función `upto` que computa la lista de enteros entre dos enteros dados.

```
upto : Int -> Int -> List Int
upto n m = case m>n of
            True -> Nil
            False -> Cons (m,upto n (m+1))
```

Nótese que hemos escrito un `if` utilizando una construcción `case` para poder aplicar la reestructura.

Derivando el hilomorfismo tenemos

```
upto n = [[φ,\m->(m,m+1)]]_F
     F = Int × I
     φ :: Int×(List Int) -> List Int
     φ = \(m,v) -> case m>n of
                     True ->Nil
                     False ->Cons (m,v)
```

Podemos descomponer el álgebra de la siguiente forma

```
φ = in_G ∘ η
 where   G = 1 + Int × I
       in_G = (\()->Nil)▽Cons
         η :: F a -> G a
         η = \(m,v) -> case m>n of
                         True ->(1,())
                         False ->(2,(m,v))
```

Reestructurando tenemos

$$[\![\phi,\backslash\texttt{m->(m,m+1)}]\!]_{\texttt{F}} = [\![in_{\texttt{G}}, \eta \circ (\backslash\texttt{m->(m,m+1)})]\!]_{\texttt{G}}$$

□

3.2.2. Reconocer $out_{\texttt{F}}$

Nos enfrentamos ahora a la pregunta ¿$\psi = out_{\texttt{F}}$? siendo

$$\begin{aligned}\psi = \ & \backslash\texttt{l->case } t_0 \texttt{ of}\\ & \quad p_1\texttt{->(1,}(t_{11},\ldots,t_{1k_1})\texttt{)}\\ & \qquad\vdots\\ & \quad p_n\texttt{->(n,}(t_{n1},\ldots,t_{nk_n})\texttt{)}\end{aligned}$$

La forma trivial que puede tener ψ que nos asegura que es $out_{\texttt{F}}$ es la siguiente:

1. Cada patrón p_i debe ser una aplicación de constructor a variables.
2. Las variables en p_i deben ser devueltas en la correspondiente tupla de salida en el mismo orden. No se puede devolver otra cosa que estas variables.
3. Hay un único patrón para cada constructor del tipo de entrada.

Como en el caso de $in_{\texttt{F}}$, pueden haber términos equivalentes a $out_{\texttt{F}}$ en una forma distinta a la mencionada que no tendremos en cuenta. Presentamos a continuación dos técnicas de reestructuras de coálgebras para obtener $out_{\texttt{F}}$ a partir de coálgebras que no lo son. La primera de ellas es una síntesis de la reestructura propuesta por Onoue et al.[OHIT97] para coálgebras. La segunda es una reestructura identificada y descrita durante el proyecto.

Moviendo términos

Dada una coálgebra

$$\begin{aligned}\psi = \ & \backslash\texttt{l->case } t_0 \texttt{ of}\\ & \quad p_1\texttt{->(1,}(t_{11},\ldots,t_{1k_1})\texttt{)}\\ & \qquad\vdots\\ & \quad p_n\texttt{->(n,}(t_{n1},\ldots,t_{nk_n})\texttt{)}\end{aligned}$$

podemos reestructurarla como

$$\begin{aligned}
\psi &= (\eta_1 + \cdots + \eta_n) \circ \psi' \\
\psi' &= \ \texttt{\textbackslash l->case}\ t_0\ \texttt{of} \\
&\qquad\qquad p_1\ \texttt{->}bv(p_1) \\
&\qquad\qquad\quad \vdots \\
&\qquad\qquad p_n\ \texttt{->}bv(p_n) \\
\eta_i &= \backslash bv(p_i)\ \texttt{->(i,(}t_{i1}, \ldots, t_{ik_i}\texttt{))}
\end{aligned}$$

siendo $bv(p_i)$ las variables que ocurren en p_i. La condición suficiente que usaremos para asegurar que $(\eta_1 + \cdots + \eta_n)$ sea una transformación natural, y por lo tanto tenga sentido hacer esta reestructura, es que cada t_{ij} debe ser

- o bien una variable que siempre que ocurre lo hace como término recursivo,
- o bien una variable que siempre que ocurre no lo hace como término recursivo,
- o bien no es un término recursivo y no contiene variables que ocurren en términos recursivos.

Ejemplo 3.5 (Reestructura de `filter`) Consideremos, por ejemplo, la función `filter`.

```
filter : (a->Bool) -> List a -> List a
filter p (Cons (a,as)) = if p a
                         then Cons (a,filter p as)
                         else filter p as
filter p Nil = Nil
```

Derivando el hilomorfismo para `filter` tenemos

$$\begin{aligned}
\texttt{filter p} &= [\![(\texttt{\textbackslash()->Nil}) \triangledown \phi_2, \psi]\!]_{\texttt{F}} \\
&\quad \texttt{where}\quad \texttt{F} = \texttt{I} + \overline{\texttt{a}} \times \texttt{I} \times \texttt{I} \\
&\qquad\quad \phi_2 = \backslash\texttt{(a,v1,v2)-> if p a then Cons (a,v1)} \\
&\qquad\qquad\qquad\qquad \texttt{else v2} \\
&\qquad\quad \psi = \backslash\texttt{l-> case l of} \\
&\qquad\qquad\qquad\quad \texttt{Cons (a,ls) ->(2,(a,ls,ls))} \\
&\qquad\qquad\qquad\quad \texttt{Nil ->(1,())}
\end{aligned}$$

Podemos descomponer la coálgebra como

$$\begin{aligned}
\psi &= (\texttt{id} + \eta_2) \circ out_{\texttt{G}} \\
&\quad \texttt{where}\quad \texttt{G} = \texttt{I} + \overline{\texttt{a}} \times \texttt{I} \\
&\qquad\quad \eta_2 = \backslash\texttt{(a,ls)-> (a,ls,ls)} \\
&\qquad\quad out_{\texttt{G}} = \backslash\texttt{l-> case l of} \\
&\qquad\qquad\qquad\quad \texttt{Nil -> (1,())} \\
&\qquad\qquad\qquad\quad \texttt{Cons (i,is) -> (2,(i,is))}
\end{aligned}$$

Si ahora aplicamos la propiedad de *hilo-shift* obtenemos

$$\texttt{filter p} = [\![((\texttt{\textbackslash()->Nil}) \triangledown \phi_2) \circ (id + \eta_2), out_{\texttt{G}}]\!]_{\texttt{G}}$$

Este es un caso donde se introduce una transformación natural η que duplica variables. La reestructura se pudo hacer porque `ls` es una variable que ocurre siempre como término recursivo y `a` es una variable que nunca ocurre como término recursivo.

Por el contrario, si `ls` hubiese ocurrido una vez como término recursivo y otra vez como no recursivo, la reestructura no hubiese sido posible. □

Agregando constructores

Dada una coálgebra

$$\begin{array}{l}\psi = \ \texttt{\textbackslash l->case } t_0 \texttt{ of}\\ \qquad\qquad C_1\ \texttt{(}p_{11},\ldots,p_{1r_1}\texttt{)->(1,}t_1\texttt{)}\\ \qquad\qquad\quad\vdots\\ \qquad\qquad C_n\ \texttt{(}p_{n1},\ldots,p_{nr_n}\texttt{)->(n,}t_n\texttt{)}\end{array}$$

donde existe un constructor C_{n+1} correspondiente al tipo de entrada que no ocurre en los patrones del `case`, podemos descomponer la coálgebra como

$$\begin{array}{l}\psi = (\eta_1 \triangledown \cdots \triangledown \eta_n \triangledown \bot) \circ \psi'\\ \psi' = \ \texttt{\textbackslash l->case } t_0 \texttt{ of}\\ \qquad\qquad C_1\ \texttt{(}p_{11},\ldots,p_{1r_1}\texttt{)->(1,}t_1\texttt{)}\\ \qquad\qquad\quad\vdots\\ \qquad\qquad C_n\ \texttt{(}p_{n1},\ldots,p_{nr_n}\texttt{)->(n,}t_n\texttt{)}\\ \qquad\qquad C_{n+1}\ \texttt{(}u_1,\ldots,u_{r_{n+1}}\texttt{) ->(n+1,(}u_1,\ldots,u_{r_{n+1}}\texttt{))}\\ \eta_i = \texttt{\textbackslash vs->(i,vs)}\end{array}$$

Hemos agregado en ψ' una alternativa artificial para el caso del constructor C_{n+1}, y ahora podemos aspirar a interpretarla como $out_{\mathtt{F}}$ para el tipo de entrada. El término

$$\eta = (\eta_1 \triangledown \cdots \triangledown \eta_n \triangledown \bot)$$

es una transformación natural

$$\eta : \mathtt{F}_1 + \cdots + \mathtt{F}_n + \mathtt{F}_{n+1} \Rightarrow \mathtt{F}_1 + \cdots + \mathtt{F}_n$$

Cada $\mathtt{F}_i$ es el functor que describe la signatura del correspondiente constructor C_i. El cometido de η es imitar el comportamiento de ψ en el caso en que difiere de ψ'. El resultado de aplicar ψ a un término de la forma $C_{n+1}\ \ (t_1,\ldots,t_{r_{n+1}})$ es un valor indefinido, por tanto cuando η recibe el valor $\texttt{(n+1,(}t_1,\ldots,t_{r_{n+1}}\texttt{))}$ el resultado debería ser indefinido. Para eso se utiliza

$$\bot\texttt{::F}_{n+1}\ \texttt{a -> F}_1 + \cdots + \mathtt{F}_n$$

un función indefinida para cualquier valor de entrada.

Así podemos hacer fusión con funciones definidas parcialmente.

Ejemplo 3.6 Por ejemplo

```
mapP :: (a->b) -> List a -> List b
mapP f (Cons (a,as)) = Cons (f a,mapP f as)
```

Derivando el hilomorfismo tenemos

$$\begin{array}{l}\texttt{mapP f = } [\![\phi,\psi]\!]_{\mathtt{F}}\\ \quad\texttt{where}\ \ \mathtt{F} = \overline{\mathtt{a}} \times \mathtt{I}\\ \qquad\phi = \texttt{\textbackslash(1,(a,v))-> Cons (f a,v)}\\ \qquad\psi = \texttt{\textbackslash l-> case l of}\\ \qquad\qquad\qquad\texttt{Cons (a,as) ->(1,(a,as))}\end{array}$$

Podemos descomponer la coálgebra ψ como

$$\begin{array}{l}\psi = (\texttt{\textbackslash vs->(1,vs)}) \triangledown \bot \circ \psi'\\ \mathtt{F} = \overline{\mathtt{a}} \times \mathtt{I}\\ \psi' = \texttt{\textbackslash l-> case l of}\\ \qquad\qquad\texttt{Cons (a,as) ->(1,(a,as))}\\ \qquad\qquad\texttt{Nil ->(2,())}\end{array}$$

Reestructurando:

$$\begin{aligned}
\texttt{mapP f} &= [\![\phi \circ \texttt{(\vs->(1,vs))} \triangledown \bot, \psi']\!]_{\texttt{G}} \\
\texttt{F} &= \overline{\texttt{a}} \times \texttt{I} \\
\texttt{G} &= \overline{\texttt{a}} \times \texttt{I} + \overline{\texttt{1}} \\
\phi &= \texttt{\textbackslash(1,(a,v))-> Cons (f a,v)} \\
\psi' &= \texttt{\textbackslash l-> case l of} \\
&\qquad \texttt{Cons (a,as) ->(1,(a,as))} \\
&\qquad \texttt{Nil ->(2,())}
\end{aligned}$$

De manera similar a como ocurría con las álgebras, aquí tenemos un caso de coálgebra que es casi $out_{\texttt{G}}$ pero los sumandos del tipo devuelto están al revés. De la misma forma podemos reescribir el hilomorfismo para corregir este detalle, por tanto en la práctica el orden no nos preocupa.

$$\begin{aligned}
\texttt{mapP f} &= [\![\phi \circ \bot \triangledown \texttt{(\vs->(1,vs))}, \psi']\!]_{\texttt{G}} \\
\texttt{F} &= \overline{\texttt{a}} \times \texttt{I} \\
\texttt{G} &= \overline{\texttt{a}} \times \texttt{I} + \overline{\texttt{1}} \\
\phi &= \texttt{\textbackslash(2,(a,v))-> Cons (f a,v)} \\
\psi' &= \texttt{\textbackslash l-> case l of} \\
&\qquad \texttt{Cons (a,as) ->(2,(a,as))} \\
&\qquad \texttt{Nil ->(1,())}
\end{aligned}$$

□

3.2.3. Reconocer τ

Queremos saber ahora si $\phi = \tau(in_{\texttt{F}})$, para τ`:: (F a->a)->(G a->a)`. Para ello tenemos un algoritmo que dado un álgebra ϕ retorna su descomposición en $\tau(in_{\texttt{F}})$.

En nuestra presentación del algoritmo supondremos que el término ϕ que se recibe como entrada es un análisis de casos $\phi_1 \triangledown \cdots \triangledown \phi_n$, donde cada ϕ_i es de la forma $\texttt{(\textbackslash(}v_1, \ldots, v_{k_i}\texttt{)->}t_i\texttt{)}$ y t_i está en una forma normal. La forma normal puede ser una de las siguientes:

1. una variable recursiva.
2. una aplicación de constructor C_j $(t'_1, \ldots, t'_m)$ donde cada t'_j que se encuentra en una posición recursiva del constructor C_j está en forma normal. Si t'_j no se encuentra en una posición recursiva entonces puede ser cualquier término que no referencie variables recursivas. Decimos que un t'_j está en una posición recursiva si el functor `F` indica que t'_j ocupa una posición recursiva del constructor C_j.
3. un término cualquiera que no referencie variables recursivas.

Para obtener nuestra forma normal en los casos en que ocurren `cases` en el álgebra podemos usar la reestructura que planteamos para el reconocimiento de $in_{\texttt{F}}$.

En la Figura 3.3 presentamos el algoritmo de derivación de τ. El objetivo del algoritmo $\mathcal{A}$ es abstraer los constructores, sustituyéndolos por las correspondientes operaciones de la `F`-álgebra α. Este algoritmo de abstracción se aplica recursivamente sólo en los argumentos recursivos, los cuales vienen indicados por el functor `F`. Desde luego, necesitamos conocer el functor para poder ejecutar el algoritmo. El functor se puede derivar de una coálgebra que se encuentre en forma $out_{\texttt{F}}$. Es seguro que esa coálgebra está disponible pues no estaríamos

$$\mathcal{T}(\mathtt{F}, \phi\mathtt{::G\ a->a})\ \mathtt{::\ (F\ a->a)->(G\ a->a)}$$
$$\mathcal{T}(\mathtt{F}_1 + \cdots + \mathtt{F}_m, \phi_1 \triangledown \cdots \triangledown \phi_n) = \backslash(\alpha_1 \triangledown \cdots \triangledown \alpha_m)\mathtt{->}\mathcal{T}'(\phi_1) \triangledown \cdots \triangledown \mathcal{T}'(\phi_n)$$
where
$$\mathcal{T}'(\backslash\mathtt{bvs->t}) = \backslash\mathtt{bvs->}\mathcal{A}\ \mathtt{t}$$
$$\mathcal{A}(\mathtt{v}) = \mathtt{v} \quad \text{(Si v es una variable recursiva)}$$
$$\mathcal{A}(C_j\ \ (t_1, \ldots, t_k)) = \alpha_j\ \ (\mathtt{F}_j\ \ \mathcal{A}\ \ (t_1, \ldots, t_k))$$
$$\mathcal{A}(C_j) = \alpha_j\ \ ()$$
$$\mathcal{A}(\mathtt{t}) = (\!|\alpha|\!)_{\mathtt{F}}\ \ \mathtt{t} \quad \text{(Todos los otros casos)}$$

Figura 3.3: Algoritmo de derivación de τ

derivando τ si el hilomorfismo no estuviera en una composición a la derecha de un catamorfismo.

Dada $\phi : \mathtt{G}\ A \to A$, las dos propiedades que garantizan la corrección del algoritmo son

- $\mathcal{T}(\mathtt{F}, \phi)\ \mathtt{::\ (F\ a->a)->(G\ a->a)}$
- $\mathcal{T}(\mathtt{F}, \phi)\ in_{\mathtt{F}} = \phi$

Onoue et al.[OHIT97] dan un boceto de la prueba de estas propiedades que no incluiremos aquí.

Ejemplo 3.7 (Derivación de τ) Consideremos la siguiente función

```
appendll :: a -> List (List a) -> List (List a)
appendll a Nil = Nil
appendll a (Cons (as,ass)) = Cons (Cons (a,as),appendll ass)
```

que como hilomorfismo tiene la forma

$$\mathtt{appendll\ =\ \backslash a->}[\![\phi_1 \triangledown \phi 2, out_{\mathtt{F}}]\!]_{\mathtt{F}}$$
$$\mathtt{where\ \ F = 1 + \overline{List\ a} \times I}$$
$$\phi_1 = \backslash\mathtt{() -> Nil}$$
$$\phi_2 = \backslash\mathtt{(as,ass)-> Cons\ (Cons\ (a,as),ass)}$$

Si aplicamos $\mathcal{T}(\mathtt{F}, \phi_1 \triangledown \phi_2)$ obtenemos el transformador

$$\tau\ \mathtt{::(F\ b->b)->F\ b->b}$$
$$\tau = \backslash\alpha_1 \triangledown \alpha_2\mathtt{->}\phi_1 \triangledown \phi_2$$
$$\mathtt{where}\ \phi_1 = \backslash\mathtt{()->}\ \mathcal{A}(\mathtt{Nil})$$
$$\phi_2 = \backslash\mathtt{(as,ass)->}\ \mathcal{A}(\mathtt{Cons\ (Cons\ (a,as),ass)})$$

que es lo mismo que

$$\tau\ \mathtt{::(F\ b->b)->G\ b->b}$$
$$\tau = \backslash\alpha_1 \triangledown \alpha_2\mathtt{->}\phi_1 \triangledown \phi_2$$
$$\mathtt{where}\ \phi_1 = \backslash\mathtt{()->}\ \alpha_1\ \mathtt{()}$$
$$\phi_2 = \backslash\mathtt{(as,ass)->}\ \alpha_2\ \mathtt{(Cons\ (a,as)},\mathcal{A}(\mathtt{ass}))$$

Observar que no hemos abstraido en ϕ_2 una de las ocurrencias del constructor `Cons`, debido a que ocurre en una posición no recursiva según el functor `F`, y por tanto sólo se vuelve a aplicar el algoritmo $\mathcal{A}$ sobre el argumento `ass`.

Finalmente, obtenemos

```
τ ::(F b->b)->G b->b
τ = \α1▽α2->φ1▽φ2
    where φ1 = \()-> α1 ()
          φ2 = \(as,ass)-> α2 (Cons (a,as),ass)
```

□

Onoue et al.[OHIT97] proponen una forma normal algo distinta para los términos dentro de cada ϕ_i:

1. el término es una variable de entrada de ϕ_i;
2. es una aplicación de constructor a términos en forma normal;
3. es una aplicación de hilomorfismo $[\![\phi'_1 \triangledown \cdots \triangledown \phi'_n, out_{\mathtt{F}}]\!]_{\mathtt{F}}\ v'$ donde cada ϕ'_i está en la misma forma restrictiva que ϕ_i, y v es una variable recursiva.
4. es de la forma $f\ t_1 \ldots t_n$ donde f es una función global y los términos $t_1, \ldots, t_n$ no referencian variables recursivas.

Las diferencias fundamentales se encuentran en la segunda y tercer cláusula. De hecho estas cláusulas conducen a errores al derivar τ a partir de términos que las satisfacen (a continuación presentamos contraejemplos). Nuestras cláusulas son un intento por corregir estas situaciones, pese a que no hemos realizado una prueba formal.

Contraejemplo 3.8 (para la segunda cláusula) Consideremos las funciones

```
mapl :: (a->a) -> Tree a -> Tree a
mapl f Empty = Empty
mapl f (Node (a,t1,t2)) = Node (f a,mapl f t1,t2)

prunel :: (a -> Bool) -> Tree a -> Tree a
prunel p Empty = Empty
prunel p (Node (a,t1,t2)) =
     case p a of
       True  -> prunel p t2
       False -> Node (a,prunel p t1,prunel p t2)
```

Si derivamos los hilomorfismos para estas funciones tenemos

```
mapl = \f->[[φ, out_F]]_F
       where F = 1 + Int × I × Tree a
             φ = (\()->Empty)▽(\(i,t1,t2)->Node (f i,t1,t2))

prunel = \p->[[φ1▽φ2, out_H]]_H
 where     H = 1 + a × I × I
          φ1 = \() -> Empty
          φ2 = \(a,v1,v2) -> case p a of
                               True ->v2
                               False ->Node (a,v1,v2)
```

Si deseamos fusionar `mapl (+1)`$\circ$`prunel (==8)` necesitamos reestructurar `prunel`, dado que no es un anamorfismo y su álgebra tampoco esta en la forma que admite el algoritmo de derivación de τ (pues ϕ_2 contiene un `case` cuyas ramas referencian variables recursivas).

El resultado de reestructurar `prunel` es

```
prunel = \p ->⟦φ1▽(id▽φ3), η ∘ ψ⟧G
        where  H = 1̄ + ā × I × I
               G = 1̄ + (I + ā × I × I)
               ψ =  \l->case l of
                          Empty ->(1,())
                          (Node a t1 t2) ->(2,(a,t1,t2))
               η :: H b->G b
               η = id + η2
               η2 = \(a,v1,v2) ->case p a of
                                   True ->(2,(1,v2))
                                   False ->(2,(2,(a,v1,v2)))
              φ1 = \() -> Empty
              φ3 = \(a,v1,v2) ->Node (a,v1,v2)
```

Como el álgebra de `prunel` luego de la reestructura tampoco está en forma $in_{\mathtt{F}}$, tendremos que derivar τ a partir del functor `F` de `mapl`. Nótese que en este caso los términos en el álgebra de `prunel` satisfacen la forma normal de Onoue et al. pero no la nuestra (no se satisface ninguna de nuestras cláusulas, en especial no se satisface la segunda porque en ϕ_3 el argumento v_2 del constructor `Node` es no recursivo según el functor `F` pero referencia a la variable recursiva v_2). Si seguimos con el contraejemplo, dado que se satisfacen las cláusulas de Onoue et al. no tendríamos objeciones para proceder a evaluar $\mathcal{T}(\mathtt{F}, \phi_1 \triangledown \mathtt{id} \triangledown \phi_3)$.

```
τ(α1▽α2) = (\()->α1 ())▽id▽(\(a,v1,v2)->α2 (a,v1,v2))
```

Nótese que el algoritmo $\mathcal{T}$ pretende que el tipo de la expresión derivada sea

```
τ :: (F b->b)->(G b->b)
```

sin embargo, este no es el tipo. La expresión derivada tiene tipo

```
τ :: (H b->b)->(G b->b)
```

En este punto ya no hay una ley de fusión que podamos aplicar, pero resulta instructivo observar cómo al pasarnos por alto este error la fusión cata-hilo retorna un resultado erróneo:

```
  mpl = ⟦φ, η ∘ ψ⟧G
where   H = 1̄ + Int̄ × I × I
        G = 1̄ + I + Int̄ × I × I
        ψ =  \l->case l of
                   Empty ->(1,())
                   (Node a t1 t2) ->(2,(a,t1,t2))
        η :: H b->G b
        η = id + η2
        η2 = \(a,v1,v2) -> case (==8) a of
                             True ->(2,v2)
                             False ->(3,(a,v1,v2))
        τ(α1▽α2) = α1▽id▽α2
        φ = τ((\()->Empty)▽(\(i,t1,t2)->Node ((+1) i,t1,t2)))
```

Hemos abreviado el cuerpo de τ para mejorar su legibilidad. Si expresamos el hilomorfismo anterior como una definición recursiva tenemos:

```
mpl :: Tree Int -> Tree Int
mpl Empty = Empty
mpl (Node (a,t1,t2)) =
     case (==8) a of
       True  -> mpl t2
       False -> Node ((+1) a,mpl t1,mpl t2)
```

Puede verificarse que

```
= mpl (Node (1,Node (1,Empty,Empty),Node (1,Empty,Empty)))
  (Node (2,Node (2,Empty,Empty),Node (2,Empty,Empty)))
```

cuando la composición original retornaba algo distinto

```
  (mapl (+1).prunel (==8))
      (Node (1,Node (1,Empty,Empty),Node (1,Empty,Empty)))
= (Node (2,Node (2,Empty,Empty),Node (1,Empty,Empty)))
```

Una diferencia fundamental entre ambas definiciones, que puede ayudar a explicar la discordancia de los resultados, estriba en que `mpl` tiene dos llamadas recursivas mientras que `mapl` sólo tiene una.

Resumiendo, el algoritmo de derivación de τ puede fallar si se aplica sobre álgebras cuyos términos son admitidos por la segunda cláusula de la forma normal de Onoue et al. □

Contraejemplo 3.9 (para la tercer cláusula) Para poder presentar el contraejemplo necesitamos extender primero el algoritmo $\mathcal{T}$ que hemos presentado, para que considere el caso de esta tercer cláusula que no aparece en nuestra forma normal.

Específicamente Onoue et al. presentan la siguiente ecuación en la definición del algoritmo $\mathcal{A}$:

$$\mathcal{A}(\llbracket \phi, out_{\mathtt{F}} \rrbracket_{\mathtt{F}}\ v') = \llbracket \mathcal{T}(\mathtt{F}, \phi)(\alpha_1 \triangledown \cdots \triangledown \alpha_m), out_{\mathtt{F}} \rrbracket_{\mathtt{F}}\ v'$$

siendo v' una variable recursiva.

Consideremos las siguientes funciones

```
concatr :: List (List a) -> List a
concatr Nil = Nil
concatr (Cons (a,as)) = append a (concatr as)

append :: List a -> List a -> List a
append l Nil = l
append l (Cons (a,as)) = Cons (a,append l as)
```

La función `append` concatena sus dos argumentos en una única lista. Para este contraejemplo hemos seleccionado para los argumentos de `append` un orden distinto del usual sólo para que el argumento recursivo sea el último. La función `concatr` hace lo mismo que `concat` ∘ `reverse`, i.e. dada una lista de listas se la da vuelta y luego se concatenan todas las listas en ella.

Si derivamos hilomorfismos de estas definiciones tenemos

$$\texttt{concatr} = \llbracket (\backslash() \texttt{->Nil}) \triangledown (\backslash(\texttt{a,v}) \texttt{->append a v}), out_{\mathtt{F}'} \rrbracket_{\mathtt{F}'}$$
$$\quad \texttt{where F}' = \overline{\mathbf{1}} + \overline{\texttt{List a}} \times \mathtt{I}$$
$$\texttt{append} = \backslash \texttt{l->} \llbracket (\backslash() \texttt{->l}) \triangledown (\backslash(\texttt{a,v}) \texttt{->Cons (a,v)}), out_{\mathtt{F}} \rrbracket_{\mathtt{F}}$$
$$\quad \texttt{where F} = \overline{\mathbf{1}} + \overline{\texttt{a}} \times \mathtt{I}$$

Si ahora queremos fusionar alguna función, digamos `reverse`, con `concatr`, tendremos que derivar necesariamente τ a partir del álgebra de `concatr`. Según nuestra forma normal, no podríamos aplicar el algoritmo debido a que los términos del álgebra no satisfacen ninguna cláusula. Pero de vuelta, los términos sí satisfacen la forma normal de Onoue et al. Según ellos el τ derivado sería:

τ `:: (F b ->b)->(F' b->b)`
τ `=` $\backslash(\alpha_1 \triangledown \alpha_2)$`->` $\alpha_1 \triangledown$`(\(a,v)->`$[\![\tau'(\alpha_1 \triangledown \alpha_2), out_{\mathtt{F}}]\!]_{\mathtt{F}}$ `v)`
  `where` τ' `=` $\backslash(\alpha_1 \triangledown \alpha_2)$`-> (\()->a)`$\triangledown \alpha_2$

El tipo de este τ tampoco es correcto, pues el tipo de la variable `v` a la que se aplica $[\![\tau'(\alpha_1 \triangledown \alpha_2), out_{\mathtt{F}}]\!]_{\mathtt{F}}$ debería ser del tipo del dominio de $out_{\mathtt{F}}$ (i.e. `List a`), pero la signatura de τ dice que debe ser de tipo `b`, para `b` arbitrario. De ignorar este error y continuar con la fusión, se obtendría una definición mal tipada.

Resumiendo, el algoritmo de derivación de τ puede fallar si se aplica sobre álgebras cuyos términos son admitidos por la tercer cláusula de la forma normal de Onoue et al. □

3.2.4. Reconocer σ

Para concluir con el examen de álgebras y coálgebras, nos preguntamos por último si $\psi = \sigma(out_{\mathtt{G}})$, para σ `:: (a->G a)->(a->F a)`. Para ello tenemos un algoritmo que dada una coálgebra ψ retorna su descomposición en $\sigma(out_{\mathtt{F}})$.

Las coálgebras que acepta el algoritmo de Onoue et al.[OHIT97] son de la siguiente forma

$$\backslash v_0\texttt{->case } v_0 \texttt{ of } p_1\texttt{->(1,(}t_{11},\ldots,t_{1m}\texttt{))};\ldots;p_n\texttt{->(n,(}t_{n1},\ldots,t_{nm}\texttt{))}$$

Es decir, el case se debe evaluar sobre la variable de entrada. También se requieren las siguientes restricciones adicionales:

- Los términos recursivos deben ser variables, y en los términos no recursivos no deben referenciarse dichas variables.
- Los patrones p_i satisfacen la siguiente forma normal:
  1. son una variable; o
  2. son de la forma C_i $(p'_1 \ldots p'_{k_i})$ tal que cada p'_j en una posición recursiva del constructor C_i está en forma normal. Los p'_j en posiciones no recursivas pueden ser cualquier patrón que no referencie variables que aparezcan en algún término recursivo. Decimos que un p'_j está en una posición recursiva si el functor `G` indica que p'_j ocupa una posición recursiva del constructor C_j.

De manera dual al algoritmo de derivación de τ, en este algoritmo se intentan abstraer constructores de los patrones de la coálgebra. Antes de pasar a esta tarea, realizamos un preprocesamiento de los patrones para eliminar anidamientos. Considérese, por ejemplo, la función

```
impares :: List a -> List a
impares = \v-> case v of
                Nil -> Nil
                Cons (a,Nil) -> Cons (a,Nil)
                Cons (b,Cons (_,l)) -> Cons (b,impares l)
```

que extrae las posiciones impares de una lista.

Si derivamos el hilomorfismo para esta función obtenemos una coálgebra

```
ψ :: List a -> F (List a)
ψ = \v->case v of
          Nil ->(1,())
          Cons (a,Nil) ->(2,a)
          Cons (b,Cons (_,l)) ->(3,(b,l))
```

donde `F` es el functor $\overline{\mathtt{1}} + \overline{\mathtt{a}} + \overline{\mathtt{a}} \times \mathtt{I}$. Eliminando los anidamientos de patrones obtenemos la siguiente expresión equivalente:

```
ψ = \v-> case v of
           Nil ->(1,())
           _ -> case v of
                  Cons (a,u1) ->
                    case u1 of
                      Nil ->(2,a)
                      _ -> case v of
                             Cons (b,u2) ->
                               case u2 of
                                 Cons (_,l) ->(3,(b,l))
                  _ -> case v of
                         Cons (b,u3) ->
                           case u3 of
                             Cons (_,l) ->(3,(b,l))
```

Esta expresión se obtiene casi aplicando mecánicamente algunas de las identidades que definen la semántica del chequeo de patrones para *Haskell*[Jon03]. En particular consideramos las dos reglas siguientes:

- `case v of {`p_1 `->` e_1`;...;`p_n `->` e_n`} =`
  ```
  case v of
   {p1-> e1;
    _ ->
           ...
                case v of
                 {pn-> en}...}
  ```
- `case v of {C` $p_1 \ldots p_n$ `->` e`;_ ->` e'`} =`
  ```
  case v of
   {C v1...vn -> case v1 of
                   {p1->
                           ...
                                case vn of
                                 {pn-> e;
                                  _ -> e' };
                             ...
                    _ -> e' }
    _ -> e' }
  ```
  al menos un patrón p_i no es una variable; y $v_1..v_n$ son variables frescas.

En breve presentaremos el algoritmo exacto que utilizamos para obtener la expresión de arriba. La coálgebra se ve algo más confusa de la nueva manera. Sin

embargo, desde el punto de vista algorítmico nos interesa esta forma pues no contiene anidamientos sobre las posiciones recursivas de los constructores que ocurren en los patrones.

Siendo $out_{\mathtt{G}}$

```
out_G :: List a -> G (List a)
out_G = \l-> case l of
              Nil ->(1,())
              Cons (a,as) ->(2,(a,as))
```

donde $\mathtt{G} = \overline{\mathtt{1}} + \overline{\mathtt{a}} \times \mathtt{I}$ es el functor de las listas, podemos reescribir la coálgebra de la siguiente forma:

```
ψ = \v-> case out_G v of
           (1,()) ->(1,())
           _ -> case out_G v of
                  (2,(a,u1)) ->
                    case out_G u1 of
                      (1,()) ->(2,a)
                      _ -> case out_G v of
                             (2,(b,u2)) ->
                               case out_G u2 of
                                 (2,(_,l)) ->(3,(b,l))
                  _ -> case out_G v of
                         (2,(b,u3)) ->
                           case out_G u3 of
                             (2,(_,l)) ->(3,(b,l))
```

Si en esta última expresión abstraemos $out_{\mathtt{G}}$ obtendremos la expresión

```
σ :: (b->G b) -> (b->F b)
σ = \β v ->
    case β v of
      (1,()) ->(1,())
      _ ->case β v of
            (2,(a,u1)) -> case β u1 of
                            (1,()) ->(2,a)
                            _ -> case β v of
                                   (2,(b,u2)) ->
                                     case β u2 of
                                       (2,(_,l)) ->(3,(b,l))
           _ -> case β v of
                  (2,(b,u3)) ->
                    case β u3 of
                      (2,(_,l)) ->(3,(b,l))
```

Dada una coálgebra, el objetivo de nuestro algoritmo de derivación de σ es entonces obtener otra coálgebra sin patrones anidados en posiciones recursivas, donde las aplicaciones de $out_{\mathtt{G}}$ serán finalmente abstraídas. Nuestro algoritmo se basa fuertemente en el dado por Onoue et al.[OHIT97], pero reestructura los análisis de casos de forma diferente. El resultado de la fusión con σ propuesta por Onoue et al. era muy difícil de codificar como un programa recursivo. La eliminación previa de anidamientos en los patrones es el paso fundamental que nos permite superar este problema.

$$
\begin{array}{l}
\mathcal{F}(\texttt{G,case}\ v_0\ \texttt{of}\ p_1\ \texttt{->}t_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}t_n) = \\
\quad \mathcal{G}(\ \texttt{case}\ v_0\ \texttt{of} \\
\qquad p_1\ \texttt{->}\ \ t_1 \\
\qquad \_\ \ \texttt{->} \\
\qquad\qquad \ddots \\
\qquad\qquad\quad p_{n-1}\texttt{->}\ t_{n-1} \\
\qquad\qquad\quad \_\quad \texttt{->}\ \texttt{case}\ v_0\ \texttt{of} \\
\qquad\qquad\qquad\quad p_n\ \texttt{->}t_n \qquad)
\end{array}
$$

$$\texttt{G}_1 + \cdots + \texttt{G}_m = \texttt{G}$$

$$
\begin{array}{l}
\mathcal{G}(\texttt{case}\ v_0\ \texttt{of}\ C_j\ \ (p_1,\ldots,p_k)\ \texttt{->}t_1\texttt{;}\_\ \texttt{->}t_2)\ \texttt{=} \\
\ \ \texttt{case}\ v_0\ \texttt{of} \\
\ \ C_j\ \ (p'_1,\ldots,p'_k)\ \texttt{->}\ (f_1 \circ \cdots \circ f_k)\ \ t_1 \\
\ \ \_\ \texttt{->}\ t'_2 \\
\ \ \textit{where}\quad (f_1,\ldots,f_k) = \texttt{G}_j\ \ (\mathcal{G}\circ)\ \ (g\ \ p'_1\ \ p_1,\ldots,g\ \ p'_k\ \ p_k) \\
\qquad\quad (p'_1,\ldots,p'_k) = \texttt{G}_j\ \ (\mathit{const}(u))\ \ (p_1,\ldots,p_k)\ \ (u \text{ es una variable fresca}) \\
\qquad\quad t'_2 = \mathcal{G}(t_2) \\
\qquad\quad g\ \ p'_i\ \ p_i\ \ t = \textbf{si}\ p'_i \neq p_i\ \ \textbf{entonces}\ \ \texttt{case}\ u\ \texttt{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p_i\ \texttt{->}\ t \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \_\ \texttt{->}\ t'_2 \\
\qquad\qquad\qquad\qquad\quad \textbf{sino}\quad t
\end{array}
$$

$$\mathcal{G}(\texttt{case}\ v_0\ \texttt{of}\ u_0\ \texttt{->}t_1\texttt{;}\_\ \texttt{->}t_2) = \texttt{case}\ v_0\ \texttt{of}\ u_0\ \texttt{->}t_1\texttt{;}\_\ \texttt{->}t_2$$

$$\mathcal{G}(t) = t$$

Figura 3.4: Algoritmo de eliminación de anidamientos

Nuestra versión del algoritmo de derivación de σ adolece de un problema. En el término σ derivado arriba, puede observarse que en el caso que β `v≠(1,())` y β `u1≠(1,())` tenemos que `u1=u2`. Así se deduce que β `u1=`β `u2`. Esto pone de manifiesto que la evaluación de $\sigma(\beta)$ puede llegar a repetir cálculos lo cuál no es deseable.

El problema se origina por la forma en que hemos eliminado los anidamientos de patrones. Como trabajo futuro proponemos estudiar la definición de un mejor algoritmo de eliminación de anidamientos que utilice una estrategia diferente para construir la estructura de `cases`.

En la Figura 3.4 presentamos los algoritmos que eliminan los anidamientos de patrones. La función $\mathcal{F}$ se encarga darle a $\mathcal{G}$ una estructura de `cases` que tienen sólo dos alternativas. El functor `G` que se le proporciona a $\mathcal{F}$ debe ser el functor del anamorfismo involucrado (no estaríamos intentando derivar σ si no tuviésemos una composición con un anamorfismo a la izquierda). La función $\mathit{const}(u)$ siempre retorna u no importa el argumento al cuál se aplique. Cuando se indica que u es una variable fresca, esto quiere decir que cada aplicación de $\mathit{const}(u)$ devuelve una variable fresca distinta.

En la Figura 3.5 presentamos el algoritmo de derivación de σ que identifica y abstrae las ocurrencias de $\mathit{out}_{\texttt{G}}$ en un término resultado del procesamiento anterior.

El algoritmo de abstracción para τ se invocaba sobre las posiciones recursivas de una aplicación de constructor. De manera dual, el algoritmo de derivación de σ abstrae sobre `cases` correspondientes a posiciones recursivas (según el functor `F`) de los constructores C_j que ocurren en los patrones. Nótese que debido al preprocesamiento realizado por el algoritmo $\mathcal{F}$, todos los `cases` encontrados por

$\mathcal{S}$(G,\v->t) =\β->\v->$\mathcal{B}(t)$
$\mathcal{B}$(case u of C_j $(p_1,\ldots,p_k)$ ->t_1;_ ->t_2) =
  case β u of
    (j,($p_1,\ldots,p_k$)) ->$\mathcal{B}(t_1)$
    _ ->$\mathcal{B}(t_2)$
$\mathcal{B}$(case u of p -> t_1;_ -> t_2) =
  si p **es un término recursivo entonces**
    $\mathcal{B}(t_1)[u/p]$
  sino case $[\![\beta]\!]_G$ u of p ->t_1;_ ->t_2)
$\mathcal{B}(t) = t$

$t[u/p]$ denota la sustitución de p por u en el término t.

Figura 3.5: Algoritmo de derivación de σ

el algoritmo $\mathcal{B}$ se evalúan sobre posiciones recursivas de los constructores.

Se inserta un anamorfismo $[\![\beta]\!]_G$ cuando el patrón no es un término recursivo ni una aplicación de constructor. Vale la pena recordar que si p es un término recursivo las restricciones sobre la entrada del algoritmo aseguran que es una variable.

Las restricciones adicionales sobre la forma de las coálgebras son necesarias a nuestro entender, si bien no son mencionadas por Onoue et al. [OHIT97] en su presentación del algoritmo.

Contraejemplo 3.10 (Los términos recursivos deben ser variables) Consideremos la siguiente función

```
mapT :: (a->b) -> Tree a -> Tree b
mapT f Empty = Empty
mapT f (Node (a,t1,t2)) = Node (f a,mapT f t1,mapT f t2)

addDepth :: Tree Int -> Tree Int
addDepth Empty = Empty
addDepth (Node (a,t1,t2)) = Node (a,addDepth (mapT (+1) t1),
                                    addDepth (mapT (+1) t2))
```

La función `addDepth` recibe un árbol de enteros, y suma a cada nodo la profundidad a la que se encuentra dentro del árbol.

Derivando hilomorfismos de estas definiciones tenemos

```
mapT = \f->[[ (\()->Empty)▽(\(i,t1,t2)->Node (f i,t1,t2)), out_F ]]_F
       where F = 1 + Int × I × I

addDepth = \p->[[φ1▽φ2, ψ]]_G
   where    G = 1 + Int × I × I
            ψ :: Tree Int -> G (Tree Int)
            ψ = \t -> case t of
                        Empty ->(1,())
                        Node (a,t1,t2) ->
                           (2,(a,mapT (+1) t1,mapT (+1) t2))
            φ1 = \() -> Empty
            φ2 = \(a,v1,v2) -> Node (a,v1,v2)
```

Si quisiéramos fusionar `addDepth` ∘ `mapT f`, dado que `addDepth` no es un catamorfismo y no se puede reestructurar para que lo sea, tenemos que derivar σ a partir de ψ. Así obtenemos

```
σ :: (F b->b) -> (G b->b)
σ(β) = case β t of
         (1,()) -> (1,())
         (2,(a,t1,t2)) -> (2,(a,mapT (+1) t1,mapT (+1) t2))
```

Sin embargo el tipo del σ derivado no es correcto. Según el functor `G t1` es de tipo `b` arbitrario, pero a `t1` se le aplica `mapT (+1)` que espera un argumento de tipo `Tree Int`. Así llegamos a que el algoritmo de derivación de σ puede fallar si no se exige que los términos recursivos de la coálgebra de entrada sean variables. □

La restricción sobre la forma que pueden tener los patrones de la coálgebra sirve para simplificar el algoritmo. Como trabajo futuro se podría intentar ampliar el conjunto de patrones manipulables.

Capítulo 4

Implementación del sistema

Un sistema de fusión requiere una manipulación intensa de términos de sintaxis abstracta. Ello es uno de los incentivos principales para usar *Haskell* como lenguaje de desarrollo. Lo que sigue es una presentación de las principales estructuras de datos y funciones que componen la implementación realizada.

4.1. Representación de programas

La representación de la sintaxis abstracta mediante tipos de datos es directa.

```
type Definiciones = [(Variable,Term)]

data Term = Tvar Variable               -- Variable
          | Tlit Literal                -- Literal
          | Ttuple [Term]               -- Tupla
          | Tlamb Boundvar Term         -- Lambda expresio'n
          | Tlet Variable Term Term     -- let
          | Tcase Term [Pattern] [Term] -- case
          | Tfapp Variable [Term]       -- Aplicacio'n de funcio'n
          | Tcapp Constructor [Term]    -- Aplicacio'n de constructor
          | Tapp Term Term              -- Aplicacio'n
          | Tbottom                     -- Te'rmino indefinido

data Variable = Vuserdef String -- Variable definida por el usuario
              | Vgen Int        -- Variable generada

type Constructor = String

data Boundvar = Bvar Variable      -- Variable
              | Bvtuple [Boundvar] -- Tupla de variables

data Pattern = Pvar Variable               -- Variable
             | Ptuple [Pattern]            -- Tupla de patrones
             | Pcons Constructor [Pattern] -- Aplicacio'n de constructor
             | Plit Literal                -- Literal
```

Un programa es dado por una lista de definiciones de funciones. Una vez que dichas definiciones son representadas en términos de la sintaxis abstracta, procedemos a procesar cada una con el algoritmo de derivación de hilomorfismos.

4.2. Representación de hilomorfismos

Un hilomorfismo es representado como una tupla con las siguientes componentes

- un álgebra
- una coálgebra
- un functor
- un nombre para el hilomorfismo
- argumentos constantes

Concretamente,

```
type Hylo a ca = (Algebra a,
                  Coalgebra ca,
                  Functor,
                  String,
                  [Boundvar])
```

Por ejemplo,

h=$\backslash v_1 \ldots v_k$ ->$[\![\phi, \psi]\!]_{\mathtt{F}}$

se almacenan como

(ϕ,ψ,F,‘‘h’’,[$v_1, \ldots, v_k$])

4.3. Representación de functores

Las operaciones básicas relacionados a un functor son:

- Dado un valor retornado por una coálgebra, determinar si es argumento de una invocación recursiva.
- Dado un valor de entrada de un álgebra, determinar si es el resultado de una invocación recursiva.

Para resolverlos, asociamos un identificador interno a cada posición que retorna una coálgebra, o que recibe un álgebra. El `Functor` especifica para cada alternativa del hilomorfismo la lista de identificadores de posiciones recursivas en esa alternativa. Las preguntas anteriores se contestan, pues, preguntando si el identificador de una posición pertenece a la lista correspondiente.

```
type Functor=[[PosID]]
```

En la práctica usamos como identificadores internos los nombres de las variables de entrada del álgebra. Así tenemos, por ejemplo, que para el caso de la función

map f = $[\![\phi, out_{\mathtt{F}}]\!]_{\mathtt{F}}$
F = $\overline{1} + \overline{\mathtt{a}} \times \mathtt{I}$
ϕ : F (List a)->List a
ϕ = ($\backslash$()->Nil)$\triangledown$($\backslash$(a,v)->Cons (f a,v))

el functor se representa como [[],[v]].

4.4. Representación de álgebras

Las álgebras siempre concuerdan con el esquema $\phi_1 \triangledown \cdots \triangledown \phi_n$, donde cada ϕ_i puede ser:

- una lambda expresión cualquiera (en el caso de que el álgebra se encuentre en una forma general)
- un constructor (en el caso de que el álgebra sea $in_{\mathtt{F}}$)
- un término de la forma $\tau_i(\phi)$ (en el caso de que el álgebra se encuentre en una forma $\tau(\phi)$)

```
type Albegra a = [Acomponent a]
type Acomponent a = ([Boundvar],TermWrapper a)
```

Cada valor de tipo `Acomponent a` corresponde a un término ϕ_i y es la representación de una lambda expresión donde la primer componente son las variables de entrada y la segunda componente es una representación del cuerpo. Así, tenemos que un álgebra como la de `map`:

```
(\()->Nil)▽(\(a,vs)->Cons (f a,vs))
```

se representa:

```
[([],Nil),([a,vs],Cons (f a,vs))]
```

Una estructura de tipo `TermWrapper a` representa un término como los de tipo `Term`, con el agregado de que se marcan algunas partes del término que se pueden reestructurar. Es decir, se marcan partes del término que pueden ser "movidas" a la coálgebra mediante los algoritmos de reestructura presentados en la Sección 3.2.1.

```
data TermWrapper a = TWcase Term [Pattern] [TermWrapper a]
                   | TWeta (TermWrapper a) Term
                   | TWsimple a
                   | TWacomp (Acomponent a)
```

Por ejemplo el álgebra que presentamos arriba podría haberse guardado como

```
[([],TWsimple Nil),
 ([a,vs],TWeta (TWsimple Cons) (\(a,vs)->(f a,vs)))]
```

Valores de la forma `TWsimple t` representan términos sin marcas. En el ejemplo, `TWsimple Nil` simplemente representa el mismo término que `Nil`[1]. Además el constructor `TWsimple` es usado para indicar que no se puede reestructurar ninguna parte del término que contiene.

`TWeta` representa la composición entre un término con marcas y otro sin ellas. En el ejemplo,

```
TWeta (TWsimple Cons) (\(a,vs)->(f a,vs))
```

es una representación del término

```
Cons . (\(a,vs)->(f a,vs))
```

[1] Por razones de legibilidad, integramos en una misma expresión representaciones y elementos del lenguaje objeto. Así, debe entenderse la expresión `TWsimple Nil` como el constructor `TWsimple` aplicado alguna representación del término `Nil`.

Al mismo tiempo el constructor `TWeta` señala que su segundo argumento puede ser movido a la coálgebra.

Consideremos ahora el caso del álgebra de `filter`.

```
(\()->Nil)▽(\(a,v1,v2) -> case p a of
                            True ->Cons (a,v1)
                            False ->v2)
```

Podríamos usar la siguiente representación

```
[((),TWsimple Nil),
 ([a,v1,v2], TWcase (p a)  [True,False]
             [TWsimple (Cons (a,v1)), TWsimple v2])]
```

Aquí el constructor `TWcase` indica que el `case` sobre `p a` puede ser movido a la coálgebra según la reestructura vista en la Sección 3.2.1.

Un valor `TWacomp ac` indica que la componente de álgebra `ac` no puede ser movida a la transformación natural.

El argumento `a` del tipo `TermWrapper a` permite variar la representación del cuerpo de las componentes ϕ_i de un álgebra, según esta sea de la forma $in_{\mathtt{F}}$, $\tau(\phi)$ o general. Puede pensarse que `TermWrapper` es una estructura para representar de manera implícita transformaciones naturales dentro de un álgebra, y que tiene agujeros de tipo `a` donde se colocan los *términos propios* del álgebra. Estos términos propios quedan definidos por su condición de no poder ser movidos a la coálgebra. En caso de que el álgebra esté en forma general, sus términos propios son del tipo `Term`. En caso de que el álgebra esté en forma $in_{\mathtt{F}}$, sus términos propios son del tipo `InFComp`.

```
type InFComp = Constructor
```

El tipo `InFComp` representa una componente de álgebra en forma $in_{\mathtt{F}}$. El constructor es el correspondiente a esa componente de $in_{\mathtt{F}}$. para que un álgebra se encuentre en forma $in_{\mathtt{F}}$, además de tener tipo `Algebra InFComp`, es necesario que haya una y sólo una ocurrencia de una expresión de tipo `InFComp` para cada constructor del tipo producido.

Por último, la representación de los términos propios en caso de que el álgebra esté en la forma $\tau(\phi)$ es la siguiente:

```
data TauComp = Tauphi (TauTerm Term)
             | TauinF (TauTerm InFComp)
             | Tautau (TauTerm TauComp)
type Tau' a = TauTerm (Acomponent a)
```

Esta representación de términos propios es la unión de 3 casos:

- Un caso para cuando la forma del álgebra es $\tau(\phi)$, siendo ϕ un álgebra del tipo general `Algebra Term`;
- otro caso para cuando la forma es $\tau(in_{\mathtt{F}})$;
- y otro para la forma $\tau(\tau'(\phi))$, siendo ϕ de cualquier tipo.

En el sistema, siempre que un álgebra está en forma $\tau(\phi)$ es porque es el resultado de la fusión de otros hilomorfismos.

$$[\![\phi, out_{\mathtt{F}}]\!]_{\mathtt{F}} \circ [\![\tau(in_{\mathtt{F}}), \psi]\!]_{\mathtt{G}}$$

Así es que el parámetro del tipo `TauTerm` denota el tipo de las componentes del álgebra ϕ que se utilizan dentro de $\tau(\phi)$.

```
data TauTerm a = Taucons a [TauTerm a]
               | Tausimple Term
               | Taucata (Hylo a OutFComp) (TauTerm a)
```

El tipo `TauTerm` es una imitación de la estructura de los términos que devuelve el algoritmo de derivación de τ (Sección 3.2.3). `Taucons a ts` representa la aplicación de la componente de álgebra `a` a los argumentos `ts`. `Tausimple t` representa lo mismo que el término `t`. `Taucata h t` representa la aplicación del catamorfismo `h` al término `t`. Nótese que, debido al algoritmo de derivación de τ, `h` es $[\![\phi, out_{\mathtt{F}}]\!]_{\mathtt{F}}$, o sea, el hilomorfismo que se encontraba a la izquierda en la composición que dio origen a la fusión.

4.5. Representación de coálgebras

Una coálgebra es un término de la forma

$$\psi = \texttt{\textbackslash l->case } t_0 \texttt{ of}\quad \begin{array}{l} p_1\texttt{->(1,(}t_{11},\ldots,t_{1k_1}\texttt{))} \\ \quad\vdots \\ p_n\texttt{->(n,(}t_{n1},\ldots,t_{nk_n}\texttt{))} \end{array}$$

Por tanto representamos las coálgebras como

```
type Coalbegra ca = (Boundvar,Term,ca)
```

La primer componente de la tupla es la variable de entrada. La segunda componente es el término sobre el cuál se evalúa el case. La última componente son las alternativas de la coálgebra. El parámetro `ca` nos permite variar la representación de las alternativas de la coálgeba según esta sea de la forma $out_{\mathtt{F}}$, $\sigma(\psi)$ o una forma más general. Así tenemos que una coálgebra como la de arriba se almacena así:

$$\texttt{(l,}t_0\texttt{,[}p_1\texttt{->(1,(}t_{11},\ldots,t_{1k_1}\texttt{)),}\ldots\texttt{,}p_n\texttt{->(n,(}t_{n1},\ldots,t_{nk_n}\texttt{))])}$$

Las alternativas de una coálgebra en la forma más general se definen como

```
type PsiAlts = [(Pattern,(Int,[TupleTerm]))]
type TupleTerm = (PosID,Term)
```

`PosID` es el identificador de posición que referencia la representación de functor (Sección 4.3). La coálgebra de arriba tiene tipo `Coalgeba PsiAlts`, y la lista de alternativas se representa (omitiendo los identificadores de posición) como

$$\texttt{[(}p_1\texttt{,(1,[}t_{11},\ldots,t_{1k_1}\texttt{])),}\ldots\texttt{,(}p_n\texttt{,(n,[}t_{n1},\ldots,t_{nk_n}\texttt{]))]}$$

Cuando la coálgebra se puede llevar a la forma $out_{\mathtt{F}}$, las alternativas se representan así:

```
type OutFAlts = [(Constructor,[Variable])]
```

Por ejemplo, si la coálgebra fuese:

```
\l -> case l of
        Nil -> (1,())
        Cons (a,as) -> (2,(a,as))
```

entonces se podría almacenar como

```
(l,l,[(Nil,[]),(Cons,[a,as])])
```

Estrictamente hablando, para que una coálgeba de tipo `Coalgebra OutFAlts` realmente se encuentre en forma $out_{\mathtt{F}}$, debe haber una alternativa por cada constructor del tipo de entrada. En la práctica se puede esquivar esta restricción haciendo una manipulación cuidadosa durante la fusión la cual se explica mediante la reestructura vista en la Sección 3.2.2.

Una coálgebra en forma $\sigma(out_{\mathtt{F}})$ es más compleja que las otras, y esta complejidad impregna la representación. Para empezar, una coálgebra así no se puede ajustar al esquema que propusimos en un principio:

$$\begin{aligned}\psi = \ &\backslash\mathtt{l->case}\ t_0\ \mathtt{of}\\ &\quad p_1\mathtt{->(1,(}t_{11},\ldots,t_{1k_1}\mathtt{))}\\ &\qquad\vdots\\ &\quad p_n\mathtt{->(n,(}t_{n1},\ldots,t_{nk_n}\mathtt{))}\end{aligned}$$

No se ajusta porque, de acuerdo al algoritmo de derivación de σ (Sección 3.2.4), las alternativas de la coálgebra no devuelven sólo tuplas, sino que pueden ser estructuras `case` anidadas. En vez de representar las alternativas como una lista, las representamos con un único término de tipo `TermS`, específico para representar el término resultante del algoritmo de derivación de σ.

Para coálgebras en forma $\sigma(\psi)$, tenemos esta representación:

```
type SigmaAlts = (TermS,WrappedCA)
```

La primer componente es el término que representa la estructura de σ, y la segunda componente es una representación de la coálgebra ψ argumento de σ.

El tipo `TermS` es el dual del tipo `Tau` y en esencia son similares.

```
data TermS = TcaseR Term (Int,[Pattern]) TermS TermS
           | TcaseSana Term Pattern TermS TermS
           | TtermS (Int,[TupleTerm])
```

El tipo `TermS` es una imitación de la estructura de los términos que devuelve el algoritmo de derivación de σ (sección 3.2.4).

- `TtermS (i,`$(t_{i1},\ldots,t_{in})$`)` indica el final de la estructura de `cases` anidados y `(i,`$(t_{i1},\ldots,t_{in})$`)` es el valor que la coálgebra retorna en uno de sus casos.

- `TcaseR t0 (i,`$[p_1,\ldots,p_n]$`) t1 t2` representa un término de la forma

  ```
  case β t0 of
    (i,(p1,...,pn)) -> t1
    _ -> t2
  ```

  donde β es la coálgebra dada como parámetro a σ.

- `TcaseSana t0 p t1 t2` representa un término de la forma

  ```
  case ⟦β⟧F t0 of
    p -> t1
    _ -> t2
  ```

Similar al caso de τ, si tenemos una fusión como la siguiente

$$[\![\phi, \sigma(out_{\mathtt{F}})]\!]_{\mathtt{G}} \circ [\![in_{\mathtt{F}}, \beta]\!]_{\mathtt{F}} = [\![\phi, \sigma(\beta)]\!]_{\mathtt{G}}$$

el anamorfismo $[\![\beta]\!]_{\mathtt{F}}$ es el mismo que aparecía a la derecha en la composición.

4.6. Fusiones

Para poder fusionar los hilomorfismos tenemos una función implementada por cada ley.

```
fusionarCataAna :: Hylo a OutFAlts -> Hylo InFComp ca
                   -> Maybe (Hylo a ca)

fusionarCataHylo :: Hylo a' OutFAlts -> Hylo Term ca
                    -> Maybe (Hylo TauComp ca)

fusionarHyloAna :: Hylo a PsiAlts -> Hylo InFComp ca'
                   -> Maybe (Hylo SigmaAlts ca)
```

El tipo `Maybe` se define como

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

En general se devuelve `Just h` donde `h` es el resultado de la fusión.

La función `fusionarCataAna`, es una función que sustituye en el álgebra del hilomorfismo derecho las componentes de tipo `InFComp` por las componentes de álgebra del hilomorfismo a la izquierda. Esto implica que `fusionarCataAna` debe recorrer cada estructura de tipo `TermWrapper` para encontrar los términos propios. La función `fusionarCataHylo` opera de manera similar con la diferencia de que durante el proceso se debe derivar τ, y lo que se sustituye son ocurrencias de constructores.

La función `fusionarCataHylo` intenta derivar τ a partir del álgebra del segundo hilomorfismo y en caso de conseguirlo realiza la fusión. La derivación de τ no se coloca en una función independiente dado que requiere del functor `F` del primer hilomorfismo. Así, para cada instancia de fusión se obtiene un τ específico, cuya forma dependiende de ese functor `F`. Teniendo que derivar un τ distinto para cada instancia de fusión, no tiene mucho sentido tratar de calcular τ por adelantado. De manera dual `fusionarHyloAna` intenta derivar σ a partir de la coálgebra del primer hilomorfismo.

Aquí es donde se ve la ventaja de tener el tipo `TermWrapper`. Al marcar los términos que se pueden mover a la coálgebra, no es necesario moverlos explícitamente para poder distinguirlos de los términos propios. Así la fusión se puede realizar sin una reestructura explícita, que en el caso de tener que mover una estructura `case` podría ser algo engorroso. Varias reformulaciones del proceso de reestructura fueron necesarias antes de poder sintetizar una solución como esta.

la función `fusionarHyloAna`, previo a la derivación de σ y la fusión propiamente dicha, realiza todo el posprocesamiento necesario para eliminar de los patrones las aplicaciones anidadas de constructores.

Decimos que las funciones fallan si los hilomorfismos no pueden ser fusionados. En estos casos se retorna `Nothing`. Los casos de falla son los siguientes:

- `fusionarCataAna` falla si los hilomorfismos tienen distintos functores.

- `fusionarCataHylo` falla si no es posible derivar τ sobre el álgebra del segundo hilomorfismo.
- `fusionarHyloAna` falla si no es posible derivar σ sobre la coálgebra del primer hilomorfismo.

Cuando se derivan los hilomorfismos a partir de la sintaxis abstracta, estos tienen el tipo `Hylo Term PsiAlts`. Para aplicar alguna de las leyes, debemos refinar el tipo de los hilomorfismos mediante la aplicación de las funciones que implementan las reestructuras.

```
getCata :: Hylo a ca -> Maybe (Hylo a OutFAlts)
getAna :: Hylo a ca -> Maybe (Hylo InFComp ca)
```

La función `getCata` intenta aplicar todas las reestructuras vistas en la sección 3.2.2 para convertir el hilomorfismo en un catamorfismo. Así, si las reestructuras son exitosas se devuelve `Just h`, donde `h` es el hilomorfismo reestructurado. Si las reestructuras no pueden obtener un catamorfismo se retorna `Nothing`.

Puede verse la reestructura como el proceso de transformar alternativas de coálgebra de tipo `PsiAlts` en alternativas de tipo `OutFAlts`. La función `getCata` falla cuando una alternativa de tipo `PsiAlts` no satisface las hipótesis de ningún algoritmo de reestructura.

La función `getAna` realiza el proceso de convertir cada componente de álgebra de tipo `Algebra Term` en una componente de tipo `Algebra InFComp`. Recordamos que la reestructura no es explícita. Lo único que se hace es marcar los términos que se podrían mover a la coálgebra según los algoritmos de la sección 3.2.1.

Capítulo 5

Conclusiones

En este proyecto hemos presentado el diseño e implementación de una herramienta capaz de fusionar automáticamente composiciones de funciones, eliminando de esta forma las estructuras intermedias. En su estado actual, el sistema es interactivo, permitiendo que el usuario ingrese nuevas definiciones en forma incremental y que solicite la fusión de composiciones de ellas. El sistema además puede desplegar tanto la definición recursiva de cualquier función presente en el ambiente, como la representación interna en términos de hilomorfismos de cualquier función.

Fue una preocupación del diseño que la salida fuese lo más comprensible posible. Sin embargo, la salida de las fusiones que utilizan la ley hilo-ana no es la ideal. Esto podría deberse en parte a la complejidad de la semántica asociada a los patrones donde se encuentran los constructores que se abstraen. La parte más problemática de esta fusión es la que traduce la representación interna del resultado de la fusión en el lenguaje de salida del sistema. Se necesitaría más trabajo para lograr una traducción óptima.

Para poder hablar de la efectividad del sistema como optimizador de programas, es necesario incorporarlo en un compilador para realizar pruebas que permitan la comparación contra otros sistemas. Para esto habría que adaptar la implementación actual al lenguaje interno del compilador. También habría que implementar la búsqueda dentro de un programa de aquellas composiciones potencialmente fusionables.

Existe un conjunto considerable de extensiones que podrían incorporarse para ampliar las posibilidades del sistema.

Paramorfismos Hasta ahora todas las implementaciones de sistemas de fusión que se basan en el mismo enfoque que nosotros fusionan catamorfismos y anamorfismos con otros hilomorfismos. Ninguna de estas implementaciones es capaz de fusionar paramorfismos con hilomorfismos, siendo un paramorfismo una representación de una función recursiva primitiva [Mee92]. Por ejemplo, un caso de fusión de paramorfismos con hilomorfismos sería:

```
dropWhile p ∘ filter q
```

donde `dropWhile` se define como

```
dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile p (a:as) = if p a then dropWhile p as else (a:as)
dropWhile p [] = []
```

Esta función es un paramorfismo porque en un caso copia directamente en la salida la cola de la lista de entrada. Se puede derivar un hilomorfismo a partir de una función recursiva primitiva, pero no es posible aplicar ninguno de los algoritmos de reestructura o derivación de σ para poder fusionarlo cuando aparece a la izquierda de la composición. Sin embargo, existe una forma de escribir una definición recursiva equivalente a la composición particular `dropWhile p` $\circ$ `filter q`:

```
dWfil :: (a->Bool) -> (a->Bool) -> [a] -> [a]
dWfil p q (a:as) = if q a then if p a then dWfil p q as
                                      else a : filter q as
                          else dWfil p q as
dWfil p q [] = []
```

Para poder manipular automáticamente composiciones de este tipo es necesario analizar leyes de fusión que combinen paramorfismos e hilomorfismos.

Deforestación parcial Las reestructuras propuestas sólo mueven términos entre el álgebra y la coálgebra de un hilomorfismo. Sería posible reestructurar hilomorfismos de otras maneras, ampliando así las situaciones sobre las que se pueden aplicar las leyes de fusión. Consideremos las siguientes funciones sobre árboles binarios:

```
prune :: (a->Bool) -> Tree a -> Tree a
prune p Empty = Empty
prune p (Node (a,t1,t2)) =
          case p a of
            True -> Empty
             False -> Node (a,prune p t1,prune p t2)

mapL :: (a->a) -> Tree a -> Tree a
mapL f Empty = Empty
mapL f (Node (a,t1,t2)) = Node (f a,mapL t1,t2)
```

Como antes, ninguna de las implementaciones similares a la nuestra puede fusionar la composición:

```
mapL f ∘ prune p
```

Derivemos los hilomorfismos.

```
prune p = ⟦(\()->Empty)▽(\(a,v1,v2)->Node (a,v1,v2)), ψ⟧F
            where F = 1̄ + ā × I × I
                  ψ Empty = (1,())
                  ψ (Node (a,t1,t2)) =
                       case p a of
                         True -> (1,())
                         False -> (2,(a,t1,t2))

mapL f = ⟦(\()->Empty)▽(\(a,v1,t2)->Node (f a,v1,t2)), out_G⟧G
           where G = 1̄ + ā × I × (Tree a)‾
```

El hilomorfismo para `prune` está presentado como se vería luego de reestructurarlo moviendo el `case` sobre `p a` del álgebra a la coálgebra. Nótese que la ley cata-ana no se puede aplicar porque `F` y `G` son distintos. Luego tendríamos que derivar τ a partir del álgebra de `prune`, pero al intentar evaluar $\mathcal{T}$ utilizando

el functor `G` y el álgebra de `prune` tenemos que la entrada no satisface la forma normal del algoritmo.

Si tuviéramos una reestructura que tomara el hilomorfismo para `prune` y nos devolviera el que escribimos a continuación, podríamos aplicar la ley cata-ana:

```
prune' p = ⟦(\()->Empty)▽(\(a,v1,v2)->Node (a,v1,v2)), ψ⟧F'
             where F' = 1̄ + ā × I × Tree a
                   ψ Empty = (1,())
                   ψ (Node (a,t1,t2)) =
                       case p a of
                         True -> (1,())
                         False -> (2,(a,t1,prune p t2))
```

Nótese que esta versión del hilomorfismo deja sin abstraer la llamada recursiva `prune p t2` de la definición original de `prune`. La ley cata-ana se puede aplicar porque ahora sí coinciden los functores `G` y `F'`. El resultado de la fusión sería:

```
prmL :: (a->Bool) -> (a->a) -> Tree a -> Tree a
prmL p f Empty = Empty
prmL p f (Node (a,t1,t2)) =
           case p (f a) of
             True -> Empty
             False -> Node (f a,prmL p t1,prune p t2)
```

En este caso no se ha eliminado completamente la estructura intermedia inherente a la composición, por eso llamamos a este un caso de *deforestación parcial*. Para poder manipular automáticamente estos casos parece posible desarrollar transformaciones de hilomorfismos que permitan determinar selectivamente qué llamadas recursivas se desean abstraer y cuáles no para lograr que los functores concuerden.

Recursión mutua y recursión en más de un argumento Se pueden incorporar técnicas de fusión para definiciones mutuamente recursivas o con recursión en más de un argumento. La función `zip`, la igualdad estructural, y los analizadores sintácticos recursivos descendentes son ejemplos que serían abarcados por estas técnicas. Utilizando un enfoque algebraico pueden encontrarse en la literatura algunas propuestas al respecto [HIT97, IHT98].

Otras transformaciones Existen técnicas de transformación de programas complementarias a la deforestación que permiten optimizar un programa. Sería posible incorporar técnicas para *tupling* [HITT96], una táctica para obtener funciones recursivas eficientes agrupando funciones recursivas en una tupla.

La construcción de una herramienta de transformación de programas como la implementada, es apenas un primer paso hacia la investigación y la explotación de las posibilidades que da el enfoque algebraico para el tratamiento automático de programas funcionales. Al intentar incorporar extensiones podría constatarse que el diseño necesita muchos cambios, y entonces habría que escribir el código nuevamente. Nuestras aspiraciones al respecto de este trabajo son que sirva como introducción a los problemas asociados a la implementación de un sistema de fusión *sencillo*, y que pueda considerarse como el resultado de pulir y complementar los algoritmos planteados en trabajos precedentes.

Bibliografía

[AJ94] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[BdM97] R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, UK, 1997.

[Chi92] W.N. Chin. Safe fusion of functional expressions. In *In Proc. Conference on Lisp and Functional Programming, San Francisco, California*, pages 11–20, June 1992.

[dMS99] O. de Moor and G. Sittampalam. Generic program transformation. In *Advanced Functional Programming*, Lecture Notes in Computer Science vol. 1608. Springer-Verlag, 1999.

[Fok92] M.M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, Universiteit Twente, The Netherlands, 1992.

[GH00] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. Submitted for publication, August 2000.

[GJ98] J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *Proc. 3rd. ACM SIGPLAN International Conference on Functional Programming.* ACM, September 1998.

[GLPJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

[HIT96a] Z. Hu, H. Iwasaki, and M. Takeichi. Calculating Accumulations. Technical Report METR 96-03, Faculty of Engineering, University of Tokyo, March 1996.

[HIT96b] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings 1st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'96, Philadelphia, PA, USA, 24–26 May 1996*, volume 31(6), pages 73–82. ACM Press, New York, 1996.

[HIT97] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An extension of the acid rain theorem. In T. Ida, A. Ohori, and M. Takeichi, editors, *Proceedings 2nd Fuji Int. Workshop on Functional and Logic Programming, Shonan Village Center, Japan, 1–4 Nov. 1996*, pages 91–105. World Scientific, Singapore, 1997.

[HITT96] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings 2nd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'97, Amsterdam, The Netherlands, 9–11 June 1997*, volume 32(8), pages 164–175. ACM Press, New York, 1996.

[IHT98] Hideya Iwasaki, Zhenjiang Hu, and Masato Takeichi. Towards manipulation of mutually recursive functions. In *Fuji International Symposium on Functional and Logic Programming*, pages 61–79, 1998.

[JL98] Patricia Johann and John Launchbury. Warm fusion for the masses: Detailing virtual data structure elimination in fully recursive languages. In *SDRR Project Phase II, Final Report*, Computer Science and Engineering Department,Oregon Gradute Institute in USA, 1998.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[Mee92] Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[OHIT97] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In *Algorithmic Languages and Calculi*, pages 76–106, 1997.

[Par03] Alberto Pardo. Fusión de programas recursivos con efectos. Proyecto CSIC I+D, 2003.
URL: http://www.fing.edu.uy/inco/proyectos/fusion.

[Sch00] J. Schwartz. Eliminating intermediate lists in ph. Master's thesis, Massachusetts Institute of Technology, USA, May 2000.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, New York, 1993.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 306–313. ACM Press, New York, 1995.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*, pages 344–358. Berlin: Springer-Verlag, 1988.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York, 1989.