Automatización de leyes de fusión de programas

Facundo Domínguez Alberto Pardo

Instituto de Computación, Universidad de la República, Montevideo, Uruguay
{fdomin,pardo}@fing.edu.uy

Resumen

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. Este tipo de diseño puede ser ineficiente debido a la generación de muchas estructuras de datos intermedias. 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 intermedias. Una serie importante de trabajos apuntan a automatizar estas técnicas para su inclusión en compiladores.

En el marco del desarrollo de un sistema que realiza automáticamente algunas de estas transformaciones sobre programas escritos en Haskell, nuestro objetivo es presentar una revisión de algunos de los algoritmos utilizados.

Palabras claves: Lenguajes de programación, especificación formal, transformación de programas, programación funcional.

1. Introducción

Consideremos un programa para contar la cantidad de palabras que ocurren en un texto y que terminan con un carácter dado. Este programa implementado en Haskell[1] [13] podría ser dado por la siguiente definición.

```
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''
```

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

filter :: (a->Bool) -> [a] -> [a]
filter p [] = []
filter p (a:as) =
     if p a then a : filter p as
       else filter p 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. El sufijo más grande de `str` que no empieza con espacios se corresponde con la expresión `dropWhile isSpace str`. 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'`.

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 del texto 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
               [] -> 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, esta solución es más difícil de escribir, reutiliza menos código, y es más difícil comprender qué hace. Muchos beneficios tiene el diseño modular de código como para dejarlo de lado demasiado rápido.

Si los compiladores fuesen capaces de derivar la versión eficiente a partir del programa modular el programador no tendría que preocuparse en absoluto por las estructuras de datos intermedias. Varios trabajos se han realizado con el objetivo de estudiar técnicas de transformación de programas que eliminen las estructuras intermedias. Debido a que estas estructuras intermedias son en general arborescentes, se dice que las técnicas son de *deforestación* [21]. La eliminación de estructuras intermedias implica el reemplazo de varias funciones por una equivalente que combine sus códigos[2]. Una línea importante de trabajos se ha desarrollado con el objetivo de automatizar algunas de estas técnicas, apuntando a su integración en compiladores.

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 fuente-a-fuente basadas en fusión sobre programas escritos en el lenguaje funcional Haskell. Una característica de este sistema es que se basa en un enfoque algebraico de la transformación de programas [1, 18, 19, 15], existiendo también otras propuestas alternativas [20, 2, 3, 12, 7]. En el enfoque que adoptamos las funciones se reescriben en términos de un esquema de recursión llamado *hilomorfismo* [15]. Los *hilomorfismos* cuentan con ciertas leyes de transformación, conocidas como *lluvia ácida* [19], mediante las cuales es posible fusionar programas. Para la construcción de este sistema contamos con la experiencia de implementaciones anteriores de características similares [16, 17]. Casi todos los algoritmos que integran nuestro sistema se encuentran descriptos en dichos trabajos.

Las principales contribuciones que hacemos son las siguientes.

- Revisamos y corregimos algunos de los algoritmos que forman el núcleo del sistema. En particular aquellos empleados por Onoue et al. [16] para derivar transformadores, que son funciones requeridas por las leyes de *lluvia ácida*.

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

[2] por lo que también se conoce a estas técnicas por el nombre de *fusión*.

- Ofrecemos una herramienta interactiva, llamada HFusion [8], que permite examinar el resultado de aplicar las leyes de fusión a composiciones de programas Haskell y examinar la función resultante en un formato comprensible, tanto como definición recursiva como en términos de su representación algebraica.
- Identificamos posibles extensiones prácticas y teóricas para ampliar el poder de la herramienta.

El resto del trabajo se organiza de la siguiente manera. En la Sección 2 se presenta una batería de conceptos teóricos sobre los cuales se apoya la herramienta. En el Sección 3 se presenta una visión global del sistema y los algoritmos en los que realizamos modificaciones importantes. Finalmente en el Sección 4 discutimos posibles extensiones que se pueden realizar a la herramienta.

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 esta sección es presentar brevemente estas herramientas e introducir la notación que emplearemos.

2.1. Functores

El instrumento básico que tenemos para hablar de la estructura recursiva de una definición es el functor, aunque pueda no parecer evidente hasta ver la definición de hilomorfismos.

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

$\texttt{F}\ f : \texttt{F}\ a \to \texttt{F}\ b \quad \forall f : a \to b$

$\texttt{F}\ id_a = id_{\texttt{F}\,a}$

$\texttt{F}\ (f.g) = \texttt{F}\ f\ .\ \texttt{F}\ g \quad \forall f : a \to b, g : b \to c$

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. La definición de functor se puede generalizar para functores que reciben más de un argumento. Un **functor binario** (o bifunctor) `F` es un operador binario que actúa sobre tipos y funciones, y satisface las siguientes propiedades:

$\texttt{F}\ fg : \texttt{F}\ a\ c \to \texttt{F}\ b\ d$

$\texttt{F}\ id\ id = id$

$\texttt{F}\ (f.g)\ (h.k) = \texttt{F}\ f\ h\ .\ \texttt{F}\ g\ k$

Tenemos los siguientes functores elementales. 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 los bifunctores producto y suma:

```
(X×Y) = {(x,y) | x ∈ X, y ∈ Y}   (X+Y) = {1}×X ∪ {2}×Y ∪ {⊥}
(f×g) (x,y) = (f x,g y)          (f+g) (1,x) = (1,f x)
                                 (f+g) (2,y) = (2,g y)
                                 (f+g) ⊥ = ⊥
```

Ambos functores se pueden generalizar para mayor cantidad de argumentos. 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.

Utilizaremos con frecuencia el tipo unitario de Haskell, que vamos a denotar como $\mathbf{1}$=`{()}`, y su functor constante $\overline{\mathbf{1}}$. O sea, éste es el tipo que contiene un único elemento que es `()`. Asumimos de aquí en adelante que el producto tiene mayor precedencia que la suma y que ambos tienen menor precedencia que la aplicación. Siendo `F` y `G` functores, se puede definir el producto de functores como: $(\texttt{F} \times \texttt{G})\ X = \texttt{F}\ X \times \texttt{G}\ X$, $(\texttt{F} \times \texttt{G})\ f = \texttt{F}\ f \times \texttt{G}\ f$. La suma de functores se define de manera análoga.

2.2. Álgebras y Coálgebras

La fusión de funciones implica el reemplazo de ciertas operaciones por otras en una expresión dada. Los conceptos de estas operaciones que manipulamos se formalizan como álgebras y coálgebras.

Sea un functor `F` y un tipo a, una **F-álgebra** es una función $\phi : \texttt{F}\ a \to a$.

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

Sean tipos `T` y $\texttt{X}_1, \ldots, \texttt{X}_n$ y funciones $\texttt{f}_1 : \texttt{X}_1 \to \texttt{T}, \ldots, \texttt{f}_n : \texttt{X}_n \to \texttt{T}$. Definimos el **análisis de casos** $\texttt{f}_1 \triangledown \cdots \triangledown \texttt{f}_n : \texttt{X}_1 + \cdots + \texttt{X}_n \to \texttt{T}$ como $(\texttt{f}_1 \triangledown \cdots \triangledown \texttt{f}_n)\ \texttt{(i,x)} = \texttt{f}_i\ x$.

Dada un F-álgebra de la forma $\phi = \phi_1 \triangledown \cdots \triangledown \phi_n$, donde $\phi_i = (\backslash(v_{i1}, \ldots, v_{in_i})\texttt{->}t_i)$, 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.

Sea un functor F y un tipo a, una **F-coálgebra** es una función $\psi : a \to \texttt{F}\ a$.

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. Dada un F-coálgebra de la forma

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

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. Utilizando los constructores de un tipo de dato $\texttt{T}\ a_1 \ldots a_r$ es posible construir la siguiente F-álgebra:

$\texttt{c}_1 \triangledown \cdots \triangledown \texttt{c}_n\texttt{: F (T}\ a_1 \ldots a_r\texttt{) -> T}\ a_1 \ldots a_r$

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 $\texttt{F}_{ij} = \texttt{I}$ si $t_{ij} = \texttt{T}\ a_1 \ldots a_r$, o $\texttt{F}_{ij} = \overline{t_{ij}}$ en otro caso.

Aquí puede verse que las posiciones del functor F donde ocurre el functor I, se corresponden con las posiciones recursivas de los constructores del tipo $\texttt{T}\ a_1 \ldots a_r$. Por eso llamamos **recursivas** a estas posiciones del functor. Si el constructor $\texttt{C}_i$ no tiene argumentos, i.e. si es una constante, entonces $\texttt{F}_i = \overline{\texttt{1}}$. Nótese que los constructores se consideran no currificados. La teoría obliga a manejar los constructores de esta forma[3], pero en la implementación no hubo inconvenientes al manipular constructores no currificados y por lo tanto el sistema no hace distinciones al respecto.

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

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

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

2.4. Hilomorfismos

Finalmente, las álgebras y coálgebras pueden combinarse para formar definiciones recursivas utilizando el siguiente operador recursivo.

Sea un functor F, una F-álgebra ϕ y una F-coálgebra ψ. Un **hilomorfismo** es el mínimo punto fijo de la ecuación $h = \phi \circ \texttt{F}\ h \circ \psi$ el cual se denota $[\![\phi, \psi]\!]_{\texttt{F}}$.

La practicidad de tener un hilomorfismo consiste en tener bien separadas las etapas de preparación de los argumentos para las llamadas recursivas, las llamadas recursivas, y la combinación del resultado de estas llamadas. Son usuales en la literatura algunas especializaciones del operador hilomorfismo. Tenemos, por ejemplo, los *catamorfismos*, que son hilomorfismos de la forma $[\![\phi, out_{\texttt{F}}]\!]_{\texttt{F}}$, los cuales se denotan $(\!|\phi|\!)_{\texttt{F}}$. Los catamorfismos representan funciones definidas por recursión estructural [1]. Tenemos también los *anamorfismos*, que son hilomorfismos de la forma $[\![in_{\texttt{F}}, \psi]\!]_{\texttt{F}}$, denotados como $[\!(\psi)\!]_{\texttt{F}}$. En este caso representan funciones definidas por lo que se conoce como correcursión [6].

Los hilomorfismos gozan de las siguientes leyes de fusión, y son las que se pretende automatizar.

$$[\![\phi, out_{\texttt{F}}]\!]_{\texttt{F}} \circ [\![in_{\texttt{F}}, \psi]\!]_{\texttt{F}} = [\![\phi, \psi]\!]_{\texttt{F}} \quad \text{(cata-ana)}$$

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

[3]Desde aquí nos apartamos de la notación de Haskell para listas, dado que (:) es un constructor currificado.

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

Puede apreciarse en estas leyes la intención de eliminar la estructura intermedia entre dos hilomorfismos tan sólo sustituyendo unos términos por otros en las definiciones. Cuando una función tiene un tipo como el de $\tau \circ \sigma$, se dice que la función es un *transformador* de álgebras/coálgebras de signatura `F` en álgebras/coálgebras de signatura `G` [5].

Ejemplo 2.1 (Fusión con τ) Para ver un caso de fusión 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`.

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

```
filter :: (b->Bool) -> List b -> List b
filter p = [(\()->Nil)▽φ2, ψ]G
  where G = 1 + b × 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))
```

El álgebra de `filter` se puede escribir como $\tau(in_{\mathtt{F}})$, donde

```
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 obtenemos: `lf p =` $[\![\tau((\backslash()\text{->}0)\triangledown\phi_2), \psi]\!]_{\mathtt{G}}$
que corresponde a la siguiente definición recursiva:

```
lf p Nil = 0
lf p (Cons (b,ls)) = if p b then 1+lf ls else lf ls
```

□

3. Algoritmos

La herramienta desarrollada tiene un ciclo de ejecución compuesto por varias etapas. A lo largo de este capítulo presentaremos fundamentalmente los algoritmos de la segunda etapa, que son aquellos en los que hemos realizado cambios importantes a los presentados en [16, 17]. Otras descripciones de estos y los otros algoritmos pueden encontrarse en esos mismos trabajos.

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.

t	:: =	v	(variables)	b	:: =	v	(variables)
	|	l	(literales)		|	$(b, \ldots, b)$	(tuplas de variables)
	|	$(t, \ldots, t)$	(tuplas)	p	:: =	v	(variables)
	|	$\backslash b$->t	(lambda expresiones)		|	$(p, \ldots, p)$	(tuplas de patrones)
	|	case t of p->t; $\cdots$; p->t	(expresiones case)		|	$c(p, \ldots, p)$	(aplicación de constructores)
	|	$v\ t \ldots t$	(ap. de funciones)		|	l	(literales)
	|	$c\ (t, \ldots, t)$	(ap. de constructores)				
	|	$t\ t$	(ap. general de términos)				

Figura 1: Gramática de términos

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 qué 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 hilomorfismos 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. Nuestra gramática de términos se presenta en la figura 1. Permite construir un subconjunto de Haskell lo bastante amplio como para que las definiciones de funciones se puedan ajustar a ella con un esfuerzo razonable. Las definiciones son tan sólo una variable 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.

Tenemos, entonces, un algoritmo de derivación que dada una función de la siguiente forma

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

retorna un hilomorfismo equivalente. Se asume que la función hace recursión sobre un único argumento. Esto quiere decir que todos los otros argumentos que tenga la función deben ser constantes (i.e. dichos argumentos en las llamadas recursivas deben coincidir con los correspondientes de la invocación original).

La definición que devuelve el algoritmo es de la forma: $\texttt{f = } \backslash v_1 \ldots v_{m-1} \texttt{ ->} [\![\phi_1 \triangledown \cdots \triangledown \phi_n, \psi]\!]$ No incluimos la descripción del algoritmo, la cuál puede consultarse en [16, 17].

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. Cuando tenemos la composición $[\![\phi, \psi]\!]_{\mathtt{F}} \circ [\![\phi', \psi']\!]_{\mathtt{G}}$ 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

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

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 encontrar un hilomorfismo equivalente de la forma $[\![in_{\mathtt{G}}, \eta \circ \psi]\!]_{\mathtt{G}}$, donde η es una función derivada por un algoritmo de reestructura. De manera dual podemos intentar encontrar un hilomorfismo $[\![\phi \circ \eta', out_{\mathtt{G}}]\!]_{\mathtt{G}}$ donde η' también es una función derivada por un algoritmo de reestructura.

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

Nos enfrentamos ahora a la pregunta ¿$\phi' = in_{\mathtt{F}}$? La forma trivial que puede tener $\phi' : \mathtt{G}\ A \to A$ que nos asegura que es $in_{\mathtt{F}}$ 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 $in_{\mathtt{F}}$ 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[17]. Onoue et al.[16] proponen aplicar ϕ e $in_{\mathtt{F}}$ a un cierto conjunto de valores para comprobar que den los mismo resultados. Cómo 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 $in_{\mathtt{F}}$, hay varias reestructuras que se pueden realizar para obtener un hilomorfismo con álgebra $in_{\mathtt{F}}$. Onoue et al. presentan un ejemplo de reestructura posible junto con el algoritmo que la realiza.

3.2.2. *Reconocer* $out_{\mathtt{F}}$

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

$\psi =$ `\l->case` t_0 `of` p_1`->(1,(`$t_{11}, \ldots, t_{1k_1}$`));...;`p_n`->(n,(`$t_{n1}, \ldots, t_{nk_n}$`))`

La forma trivial que puede tener ψ que nos asegura que es $out_{\mathtt{F}}$ 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 $in_{\mathtt{F}}$, pueden haber términos equivalentes a $out_{\mathtt{F}}$ en una forma distinta a la mencionada que no tendremos en cuenta. De manera dual al caso de las álgebras, es posible reestructurar el hilomorfismo para obtener uno equivalente que tenga $out_{\mathtt{F}}$ como coálgebra. Onoue et al. presentan un ejemplo reestructura que se puede efectuar con estos fines, junto con el algoritmo que la realiza.

3.2.3. *Reconocer* τ

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

Como no es posible derivar τ a partir de cualquier álgebra y functor, supondremos que el álgebra ϕ dada como entrada tiene una cierta forma restrictiva que depende del functor `F`. Debe ser un análisis de casos $\phi_1 \triangledown \cdots \triangledown \phi_n$, donde cada ϕ_i es de la forma `(\(`$v_1, \ldots, v_{k_i}$`)->`t_i`)` y t_i está en una forma normal con respecto a `F`. La forma normal con respecto a `F` puede ser:

1. una variable recursiva (resultado de una llamada recursiva).
2. una aplicación de constructor C_j `(`$t'_1, \ldots, t'_m$`)` donde cada t'_j que de acuerdo a `F` se encuentra en una posición recursiva del constructor C_j está en forma normal con respecto a `F`. Si t'_j no se encuentra en una posición recursiva entonces puede ser cualquier término que no referencie variables recursivas.
3. un término cualquiera que no referencie variables recursivas.

Cuando el álgebra no está en esta forma normal, a veces es posible reestructurar el hilomorfismo para obtener uno equivalente que sí tenga un álgebra en esta forma [16].

En la Figura 2 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 $out_{\mathtt{F}}$. Es seguro que esa coálgebra está disponible pues no estaríamos derivando τ si el hilomorfismo no estuviera en una composición a la derecha de un catamorfismo.

Ejemplo de derivación de τ Consideremos la siguiente función

$$\mathcal{T}(\texttt{F}, \phi\texttt{::G a->a}) \texttt{ :: (F a->a)->(G a->a)}$$
$$\mathcal{T}(\texttt{F}_1 + \cdots + \texttt{F}_m, \phi_1 \triangledown \cdots \triangledown \phi_n) = \backslash(\alpha_1 \triangledown \cdots \triangledown \alpha_m)\texttt{->}\mathcal{T}'(\phi_1) \triangledown \cdots \triangledown \mathcal{T}'(\phi_n)$$

where

$$\mathcal{T}'(\backslash\texttt{bvs->t}) = \backslash\texttt{bvs->}\mathcal{A}\ \texttt{t}$$
$$\mathcal{A}(\texttt{v}) = \texttt{v} \quad \text{(Si v es una variable recursiva)}$$
$$\mathcal{A}(C_j\ \ (t_1, \ldots, t_k)) = \alpha_j\ \ (\texttt{F}_j\ \ \mathcal{A}\ \ (t_1, \ldots, t_k))$$
$$\mathcal{A}(C_j) = \alpha_j\ \ ()$$
$$\mathcal{A}(\texttt{t}) = (\!|\alpha|\!)_{\texttt{F}}\ \texttt{t} \quad \text{(Todos los otros casos)}$$

Figura 2: Algoritmo de derivación de τ

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

como hilomorfismo $\texttt{appendll = }\backslash\texttt{a->}[\![\phi_1 \triangledown \phi 2, out_{\texttt{F}}]\!]_{\texttt{F}}$

$\texttt{where}\quad \texttt{F = } \mathbf{1} + \overline{\texttt{List a}} \times \texttt{I}$

$\phi_1 = \backslash\texttt{() -> Nil}$

$\phi_2 = \backslash\texttt{(as,ass)-> Cons (Cons (a,as),ass)}$

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

$\tau\ \texttt{::(F b->b)->F b->b}$

$\tau = \backslash\alpha_1 \triangledown \alpha_2\texttt{->}\phi_1 \triangledown \phi_2\ \ \texttt{where}\ \ \phi_1 = \backslash\texttt{()->}\ \mathcal{A}(\texttt{Nil})$

$\phi_2 = \backslash\texttt{(as,ass)->}\ \mathcal{A}(\texttt{Cons (Cons (a,as),ass)})$

lo mismo que $\tau\ \texttt{::(F b->b)->G b->b}$

$\tau = \backslash\alpha_1 \triangledown \alpha_2\texttt{->}\phi_1 \triangledown \phi_2\ \ \texttt{where}\ \ \phi_1 = \backslash\texttt{()->}\ \alpha_1\ \texttt{()}$

$\phi_2 = \backslash\texttt{(as,ass)->}\ \alpha_2\ \texttt{(Cons (a,as),}\mathcal{A}\texttt{(ass))}$

Obsérvese que no hemos abstraído 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`. $\tau\ \texttt{::(F b->b)->G b->b}$

$\tau = \backslash\alpha_1 \triangledown \alpha_2\texttt{->}\phi_1 \triangledown \phi_2\ \ \texttt{where}\ \ \phi_1 = \backslash\texttt{()->}\ \alpha_1\ \texttt{()}$

$\phi_2 = \backslash\texttt{(as,ass)->}\ \alpha_2\ \texttt{(Cons (a,as),ass)}$

□

Onoue et al.[16] proponen un forma normal un poco más amplia para los términos dentro de cada ϕ_i. Ellos agregan la siguiente cláusula:

4. es una aplicación de hilomorfismo $[\![\phi'_1 \triangledown \cdots \triangledown \phi'_n, out_{\texttt{F}}]\!]_{\texttt{F}}\ v'$ donde cada ϕ'_i está en la misma forma restrictiva que ϕ_i, y v es una variable recursiva.

y extienden el algoritmo con esta ecuación:$\mathcal{A}([\![\phi'_1 \triangledown \cdots \triangledown \phi'_n, out_{\texttt{F}}]\!]_{\texttt{F}}\ v') = [\![\mathcal{T}(\phi'_1 \triangledown \cdots \triangledown \phi'_n)(\alpha_1 \triangledown \cdots \alpha_m), out_{\texttt{F}}]\!]_{\texttt{F}}\ v'$ Si v' es una variable recursiva, el tipo del τ derivado debería ser polimórfico en el tipo de v'. Sin embargo, nótese que el hilomorfismo $[\![\mathcal{T}(\phi'_1 \triangledown \cdots \triangledown \phi'_n)(\alpha_1 \triangledown \cdots \alpha_m), out_{\texttt{F}}]\!]_{\texttt{F}}$ requiere que el tipo de su entrada sea μF. De modo que la salida del algoritmo de [16] no es correcta en este caso. Nuestras cláusulas son un intento por corregir esta situación.

En [16] la definición de la forma normal no es recursiva, lo cuál no es correcto dado que el algoritmo recorre recursivamente los subtérminos guiándose por el functor `F` dado como argumento. Siendo que las recursión del algoritmo es guiada por el functor `F`, decidimos expresar la forma normal en función de este functor.

3.2.4. Reconocer σ

Para concluir con el examen de álgebras y coálgebras, nos preguntamos por último si $\psi = \sigma(out_{\texttt{F}})$, para σ `:: (a->F a)->(a->G a)`. Para ello tenemos un algoritmo que dada una coálgebra ψ y un functor `F` retorna su descomposición en $\sigma(out_{\texttt{F}})$. Como no es posible derivar σ a partir de cualquier coálgebra y functor, supondremos que la coálgebra ψ dada como entrada tiene una cierta forma restrictiva que depende del functor `F`. Las coálgebras que acepta nuestro algoritmo deben ser de la siguiente forma:

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

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 con respecto al functor `F`:
  1. son una variable; o
  2. son de la forma $C_i\ (p'_1 \ldots p'_{k_i})$ tal que cada p'_j que de acuerdo al functor `F` esté en una posición recursiva del constructor C_i está en forma normal con respecto al functor `F`. Los p'_j en posiciones no recursivas pueden ser cualquier patrón que no referencie variables que aparezcan en algún término recursivo.

Cuando la coálgebra no está en esta forma normal, a veces es posible reestructurar el hilomorfismo para obtener uno equivalente que sí tenga una coálgebra en esta forma [16].

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 [13]. 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_F :: List a -> F (List a)
out_F = \l-> case l of Nil ->(1,())
                       Cons (a,as) ->(2,(a,as))
```

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

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

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

$\mathcal{S}$(F,\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**
    $\mathcal{B}(t_1)[u/p]$
  si no case $[\![\beta]\!]_{\mathtt{F}}$ 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: Algoritmo de derivación de σ

```
σ :: (b->F b) -> (b->G 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,(_,1)) -> (3,(b,1))
                                                 _ -> case β v of (2,(b,u3)) ->
                                                        case β u3 of (2,(_,1)) ->(3,(b,1))
```

Dada una coálgebra y un functor F, el objetivo de nuestro algoritmo de derivación de σ es obtener otra coálgebra sin patrones anidados en posiciones recursivas, donde las aplicaciones de $out_{\mathtt{F}}$ serán finalmente abstraídas. Nuestro algoritmo es una variación del sugerido por Onoue et al.[16], pues el algoritmo de Onoue es incorrecto. Si vemos los patrones como las estructuras arborescentes que son, el chequeo de un término contra un patrón se debe hacer nodo a nodo del árbol en una recorrida en profundidad, de acuerdo a la semántica de Haskell [13]. Sin embargo el algoritmo de [16] produce un σ que realiza los chequeos de los nodos en una recorrida por amplitud. La forma en que nosotros eliminamos los anidamientos en los patrones pretende preservar el orden en que se chequeaban los nodos en los patrones originales. Por otra parte el algoritmo de Onoue et al. [16] tampoco establece restricciones sobre la forma que pueden tener los patrones, lo cuál hace que produzca algunas salidas incorrectas (aún pasando por alto el orden incorrecto de verificación de los patrones) cuando estas restricciones no se cumplen.

En la Figura 3 presentamos el algoritmo de derivación de σ que identifica y abstrae las ocurrencias de $out_{\mathtt{F}}$ 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 el algoritmo $\mathcal{B}$ se evalúan sobre posiciones recursivas de los constructores. Se inserta un anamorfismo $[\![\beta]\!]_{\mathtt{F}}$ 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.

Duplicación de computaciones En el término σ derivado arriba, puede observarse que en el caso que β v$\neq$(1,()) y β u1$\neq$(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. Para eliminar estas duplicaciones de computaciones utilizamos una serie de transformaciones de las estructuras `case` que preservan la semántica de los términos. Dichas reglas se aplicarían luego de haber derivado σ. La siguiente sería una de tales reglas:

```
case β u of                                          = case β u of
   ···; (i,(v1,...,vn)) ->                                 ···; (i,(v1,...,vn)) ->
         C[ case β u of                                        C[ei[v1,...,vn/v'1,...,v'n]]; ···
              ···; (i,(v'1,...,v'n)) -> ei; ··· ]; ···
```

En esta regla $\texttt{t[}(v_1,\ldots,v_n)\texttt{/}(v'_1,\ldots,v'_n)\texttt{]}$ es la sustitución simultánea en el término `t` de las variables v'_i por las variables v_i. `C[t]` denota `C[t/`v`]` donde v representa algún espacio en `C` a rellenar con el término `t`. Mediante esta y otras reglas similares sería posible transformar el σ derivado

```
σ = \β v -> case β v of (1,()) -> (1,())
                        (2,(a,u1)) -> case β u1 of (1,()) ->(2,a)
                                                   (2,(_,1)) ->(3,(a,1))
```

En este término ya no hay computaciones duplicadas, y así alcanzamos nuestro objetivo.

4. Trabajos futuros

El sistema que hemos implementado es interactivo, el usuario ingresa nuevas definiciones en forma incremental y solicita la fusión de composiciones de ellas. El sistema además puede desplegar la definición recursiva de cualquier función presente en el ambiente, y la representación interna en términos algebraicos.

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.

También existe un conjunto considerable de extensiones que se podrían incorporar.

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 [14]. Por ejemplo, un caso de fusión de paramorfismos con hilomorfismos sería: `dropWhile p` $\circ$ `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 [] = []
```

Leyes de fusión que abarcan estos casos pueden encontrarse en [4].

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 [9, 11].

Otras transformaciones Existen técnicas de transformación de programas complementarias a la deforestación que permiten optimizar un programa. Sería posible incorporar técnicas para *tupling* [10], una táctica para obtener funciones recursivas eficientes agrupando funciones recursivas en una tupla.

Referencias

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

[2] 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.

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

[4] Facundo Domínguez and Alberto Pardo. Program fusion with paramorphisms. In *Workshop on Mathematically Structured Functional Programming (MSFP 2006)*, electronic workshops in computing (eWIC). BCS, July 2006.

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

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

[7] 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.

[8] Hfusion: A fusion tool for haskell programs.
URL: http://www.fing.edu.uy/inco/proyectos/fusion/tool.

[9] 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.

[10] 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.

[11] 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.

[12] 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 Graduate Institute in USA, 1998.

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

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

[15] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of Functional Programming Languages and Computer Architecture'91,* LNCS 523. Springer-Verlag, August 1991.

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

[17] J. Schwartz. Eliminating Intermediate Lists in pH. Master's thesis, Massachusetts Institute of Technology, USA, May 2000.

[18] 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.

[19] 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.

[20] 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.

[21] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.