Funciones de Plegado sobre listas

JOSÉ E. GALLARDO RUIZ

Lección Magistral

Índice General

Introducción

Muchos algoritmos sobre listas son expresados de un modo natural recursivamente y sus propiedades suelen ser demostradas mediante la técnica de inducción estructural. De hecho, estos algoritmos siguen un mismo patrón recursivo y las pruebas comparten un patrón inductivo común. Tener que repetir estos patrones varias veces es tedioso, consume demasiado tiempo y puede dar lugar a errores. Estos inconvenientes pueden ser evitados utilizando una *función de orden superior* y *un principio de prueba* que los capture, lo cual nos permitirá concentrarnos exclusivamente en las partes específicas de cada algoritmo. El principio de prueba servirá no solo como un método que simplifica la inducción, sino como un *principio de definición* que puede ser utilizado para derivar programas de un modo sistemático.

El objetivo de esta lección es presentar estas técnicas de generalización en el marco de un lenguaje funcional moderno como Haskell. Las técnicas pueden ser aplicadas a cualquier lenguaje funcional puro (que permita el razonamiento ecuacional) y pueden ser extendidas a otras estructuras de datos como los árboles, aunque no desarrollaremos esta posibilidad.

La organización de la lección es la siguiente: En la primera sección realizaremos una breve introducción al lenguaje de programación (Haskell), en especial las características de éste que serán necesarias en el resto de la lección. En la segunda sección presentaremos las funciones de plegado sobre listas $foldr$ y $foldl$. La tercera sección está dedicada a la propiedad universal de la función $foldr$ y sus aplicaciones. La cuarta sección estudia los teoremas de fusión correspondientes a cada una de las funciones de plegado. Por último, la sección quinta trata los teoremas de dualidad que establecen las relaciones entre las dos funciones de plegado.

1 Breve introducción a Haskell

Haskell (Peyton Jones 99; Thompson 99) es un lenguaje funcional fuertemente tipificado, puro y no estricto. La pureza del lenguaje indica que está libre de efectos laterales, lo cual permite razonar sobre los programas del mismo modo que se puede hacer en matemáticas, reemplazando expresiones por otras equivalentes. El carácter no estricto del lenguaje indica que los argumentos de las funciones no son evaluados antes de reemplazarlos en el cuerpo de ésta, es decir se usa un orden no aplicativo para implementar el lenguaje. Esta característica permite, entre otras cosas, trabajar con estructuras de datos infinitas en los programas. La tipificación estática del lenguaje hace que cada elemento de un programa tenga asignado un tipo. Los tipos se usan en tiempo de compilación para detectar errores en el programa.

1.1 Tipos

El tipo de una función se escribe separando mediante el constructor $\rightarrow$ los tipos de los distintos argumentos y el resultado. Por ejemplo, la función de enteros en enteros que incrementa su argumento puede ser definida del siguiente modo:

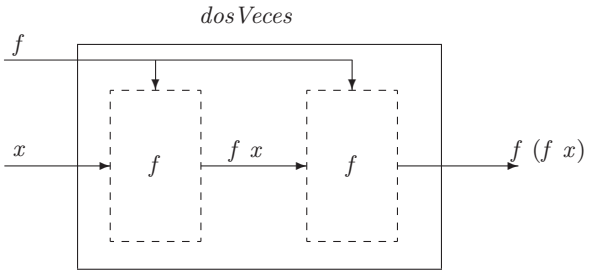

Figura 1. La función $dosVeces$

$$\begin{aligned} &inc \;\; :: Integer \rightarrow Integer \\ &inc\ x = x + 1 \end{aligned}$$

Las funciones de varios argumentos contienen varias flechas ($\rightarrow$) en su tipo. La siguiente función toma dos enteros y devuelve la suma de sus cuadrados:

$$\begin{aligned} &sumaCuadrados :: Integer \rightarrow Integer \rightarrow Integer \\ &sumaCuadrados\ x\ y = x\hat{}2 + y\hat{}2 \end{aligned}$$

Una vez definidas las funciones, puede usarse un evaluador (o intérprete) para ejecutarlas. Los siguientes *diálogos* muestran la notación que usaremos para representar una posible interacción con una implementación concreta:

```
? inc 10
11

? sumaCuadrados 2 3
13
```

1.2 Funciones de orden superior o combinadores

En Haskell las funciones son elementos del lenguaje de primera clase, en el sentido de que pueden formar parte de estructuras de datos, ser pasadas como argumentos a otras funciones o devolverse como resultados. A las funciones que actúan sobre funciones se las denomina *funciones de orden superior* o *combinadores* (tradicionalmente se usa también el término combinador para denotar funciones sin variables libres). Un ejemplo es la función *dosVeces* que toma dos argumentos, uno de los cuales es una función de enteros en enteros, que es aplicada dos veces a su segundo argumento:

$$\begin{aligned} &dosVeces :: (Integer \rightarrow Integer) \rightarrow Integer \rightarrow Integer \\ &dosVeces\ f\ x = f\ (f\ x) \end{aligned}$$

Así, el valor de la expresión *dosVeces inc* 10 es 12, lo cual podemos observar en la siguiente reducción o traza[1]:

$$
\begin{array}{ll}
 & \underline{dosVeces\ inc\ 10} \\
\Longrightarrow & \quad \{\text{Por definición de } dosVeces\} \\
 & inc\ (\underline{inc\ 10}) \\
\Longrightarrow & \quad \{\text{Por definición de } inc\} \\
 & \underline{inc\ 11} \\
\Longrightarrow & \quad \{\text{Por definición de } inc\} \\
 & 12
\end{array}
$$

1.3 Aplicación parcial

En Haskell es posible aplicar a una función menos argumentos de los que requiere. En dicho caso, el resultado que se obtiene es una nueva función. Esta característica se denomina aplicación parcial (o *currying* en los textos anglosajones). Sólo es necesario que el tipo del argumento aplicado a una función sea compatible con el tipo del primer argumento de ésta. Se obtiene como resultado una función con un argumento menos:

Tipificado de la aplicación de funciones

si $f :: t_1 \to t_2 \to \cdots \to t_n \to t_r$ y $v_1 :: t_1$
entonces $f\ v_1 :: t_2 \to \cdots \to t_n \to t_r$

Mediante sucesivas aplicaciones de la regla anterior se obtienen las demás parcializaciones. Si se aplican todos los argumentos se obtiene un valor con el tipo del resultado:

Si $v_1 :: t_1$ entonces $f\ v_1 :: t_2 \to \ldots t_n \to t_r$
Si $v_1 :: t_1,\ v_2 :: t_2$ entonces $f\ v_1\ v_2 :: t_3 \to \ldots t_n \to t_r$
...
Si $v_1 :: t_1,\ v_2 :: t_2, \ldots v_n :: t_n$ entonces $f\ v_1\ v_2 \ldots v_n :: t_r$

La aplicación parcial es útil porque permite definir nuevas funciones como concreciones de otras. Por ejemplo, si consideramos la función que comprueba si su segundo argumento es múltiplo del primero

$$
\begin{array}{l}
esMúltiploDe :: Integer \to Integer \to Bool \\
esMúltiploDe\ n\ m = (m\ `mod`\ n == 0)
\end{array}
$$

podemos obtener una función que compruebe la paridad de un número instanciando tan solo el primer argumento a 2:

[1] Denotaremos que una expresión e_1 se reduce a otra e_2 con $e_1 \Longrightarrow e_2$. Subrayaremos la parte de la expresión reducida cuando sea conveniente.

$$esPar :: Integer \rightarrow Bool$$
$$esPar = esMúltiploDe\ 2$$

y podríamos obtener el siguiente diálogo:

$$? \ \ esMúltiploDe\ 3\ 15$$
$$True$$

$$? \ \ esPar\ 13$$
$$False$$

Los operadores también pueden ser parcializados si no se aplica uno de los argumentos, obteniéndose una función de un argumento. Así, $(/2)$ es la función que divide por dos, mientras que $(1/)$ es la función inversa.

$$? \ \ dosVeces\ (*2)\ 10$$
$$40$$

1.4 Listas

Las listas son un tipo de dato predefinido en Haskell. Las listas de Haskell son homogéneas (todos los elementos han de tener el mismo tipo). Solo hay dos constructores primitivos. El primero ($[\,]$) es utilizado para denotar *listas vacías* (listas que almacenan cero elementos). El segundo (:) permite añadir un nuevo elemento al principio de una lista. Al ser homogéneas, el tipo del nuevo elemento ha de coincidir con el de los demás. El tipo de la lista se denota escribiendo el tipo de los elementos entre corchetes. Un ejemplo es

$$l1 :: [Integer]$$
$$l1 = 1 : (2 : (3 : [\,]))$$

Esta notación es engorrosa, por lo que se permite otra notación más simple que consiste en enumerar los elementos entre corchetes (aunque siempre se traduce a la primitiva):

$$l2 :: [Integer]$$
$$l2 = [1, 2, 3]$$

$$l3 :: [Bool]$$
$$l2 = [True,\ False]$$

La funciones sobre listas se definen de un modo elegante en forma recursiva. El siguiente ejemplo define una función que calcula la longitud de una lista de enteros:

$$\begin{aligned} long\ \ &:: [Integer] \rightarrow Integer \\ long\ [\,] \ &= 0 \\ long\ (x : xs) \ &= 1 + long\ xs \end{aligned}$$

La primera ecuación es seleccionada para listas vacías, mientras que la segunda para listas no vacías. En el último caso, la variable x queda ligada a la cabeza (pri-

mer elemento) de la lista, mientras que *xs* queda ligada a la cola (la lista argumento sin la cabeza). Esto se llama una *definición por patrones*.

```
? long [2, 4, 6]
3
```

1.5 Polimorfismo

Haskell permite definir funciones polimórficas. Se trata de funciones que tienen sentido para más de un tipo. Por ejemplo, la siguiente función calcula la longitud de listas de cualquier tipo:

```
length          :: [a] → Integer
length []       = 0
length (x : xs) = 1 + length xs
```

El polimorfismo queda reflejado en el tipo de la función mediante la variable de tipo a, que denota un tipo arbitrario.

```
? length [True, False]
2

? length [2, 4, 6]
3
```

A veces, aparecen varias variables de tipo para denotar tipos arbitrarios pero posiblemente distintos. Por ejemplo la función de orden superior

```
map           :: (a → b) → [a] → [b]
map f []      = []
map f (x : xs) = (f x) : (map f xs)
```

permite aplicar una función a todos los elementos de una lista. Las dos variables de tipo a y b denotan que los tipo origen y destino de la función argumento pueden ser arbitrarios y distintos

```
? map ord ['h', 'o', 'l', 'a']
[104, 111, 108, 97]
```

aunque pueden ser iguales

```
? map inc [1, 2, 3]
[2, 3, 4]
```

Especialmente interesante es el operador que implementa la composición de funciones (ver Figura 2). Éste se encuentra definido del siguiente modo

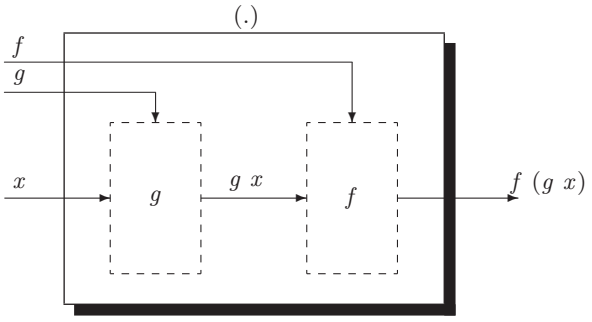

Figura 2. La composición de funciones

$$
\begin{array}{l}
(.) \;\; :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\
f \,.\, g \;= comp \\
\quad \textbf{where} \\
\qquad comp\; x \;=\; f\;(g\;x)
\end{array}
$$

El operador (.) permite construir una nueva función a partir de otras dos, componiéndolas. Por ejemplo, dado que $(*2)$ y $(+1)$ son las funciones que multiplican por dos y suman uno respectivamente, la función

$$
\begin{array}{l}
compuesta :: Integer \rightarrow Integer \\
compuesta \;= (+1) \,.\, (*2)
\end{array}
$$

toma un entero, lo multiplica por dos y lo incrementa.

```
? compuesta 10
21
```

1.6 Razonamiento ecuacional. Inducción

El principio de inducción estructural para listas finitas permite probar propiedades de funciones que actúan sobre esta estructura y puede enunciarse del siguiente modo:

Principio de inducción para listas finitas

$$
\forall xs :: [a] \;\centerdot\; P(xs) \Leftrightarrow \left\{ \begin{array}{l} P([\,]) \\ \wedge \\ \forall xs :: [a], \forall x :: a \;\centerdot\; (P(xs) \Rightarrow P(x : xs)) \end{array} \right.
$$

Aquí, P es la propiedad a demostrar. El punto (.) indica que los cuantificadores afectan a toda la expresión a la derecha de éste. La notación $\forall\, xs :: [a] \;\centerdot\; P(xs)$ es una abreviatura de la notación $\forall\, xs \;\centerdot\; xs :: [a] \;\centerdot\; P(xs)$ usada en (Dijkstra y Scholten 89). El caso base consiste en demostrar la propiedad para la lista vacía.

La lectura del paso inductivo es que deberemos probar que $P(x : xs)$ es cierta sea cual sea x y asumiendo que $P(xs)$ es cierta (*hipótesis de inducción*).

Éste y otros resultados pueden ser extendidos a listas infinitas, aunque no lo haremos aquí. Nos limitaremos en esta lección a listas finitas. El lector interesado puede consultar (Bird 98).

Usando el principio anterior, demostraremos la siguiente propiedad evidente para cualquier lista finita zs:

$$\forall\, k\ ::\ Integer, \forall\, zs\ ::\ [Integer]\ .\ sumar\ (map\ (k*)\ zs)\ =\ k\ *\ sumar\ zs$$

donde $(k*)$ es la notación usada en Haskell para denotar la función que multiplica por k y *sumar* es la función que suma los elementos de una lista de enteros:

$$\begin{array}{ll} sumar & ::\ [Integer]\ \rightarrow\ Integer \\ sumar\ [\,] & =0 \\ sumar\ (x : xs) & =x\ +\ sumar\ xs \end{array}$$

Para probar la propiedad anterior, utilizando el principio de inducción, basta con probar los dos casos:

- Caso Base:
  $\forall\, k\ ::\ Integer\ .\ sumar\ (map\ (k*)\ [\,])\ =\ k\ *\ sumar\ [\,]$
- Paso Inductivo:

$$\begin{array}{l} \forall\, k\ ::\ Integer, \forall\, xs\ ::\ [Integer], \forall\, x\ ::\ Integer\ . \\ \quad sumar\ (map\ (k*)\ xs)\ =\ k\ *\ sumar\ xs \\ \Rightarrow \\ \quad sumar\ (map\ (k*)\ (x : xs))\ =\ k\ *\ sumar\ (x : xs) \end{array}$$

Comenzamos por la parte izquierda del caso base:

$$\begin{array}{ll} & sumar\ (\underline{map\ (k*)\ [\,]}) \\ \equiv & \{\text{por definición de } map\} \\ & \underline{sumar\ [\,]} \\ \equiv & \{\text{por definición de } sumar\} \\ & 0 \end{array}$$

En cuanto a la parte derecha:

$$\begin{array}{ll} & k\ *\ \underline{sumar\ [\,]} \\ \equiv & \{\text{por definición de } sumar\} \\ & \underline{k\ *\ 0} \\ \equiv & \{\text{aritmética}\} \\ & 0 \end{array}$$

con lo que queda demostrado el caso base. Para el paso inductivo comenzamos por la parte izquierda:

$$sumar\ (\underline{map\ (k*)\ (x : xs)})$$

$$
\begin{array}{ll}
\equiv & \{\text{por definición de } map\} \\
& \underline{sumar\ (k * x\ :\ map\ (k*)\ xs)} \\
\equiv & \{\text{por definición de } sumar\} \\
& k * x\ +\ \underline{sumar\ (map\ (k*)\ xs)} \\
\equiv & \{\text{por Hipótesis de Inducción}\} \\
& k * x\ +\ k * sumar\ xs
\end{array}
$$

Para la parte derecha:

$$
\begin{array}{ll}
& k\ *\ \underline{sumar\ (x : xs)} \\
\equiv & \{\text{por definición de } sumar\} \\
& \underline{k\ *\ (x + sumar\ xs)} \\
\equiv & \{\text{distributiva de } (*) \text{ respecto } (+)\} \\
& k * x\ +\ k * sumar\ xs
\end{array}
$$

Con lo que queda demostrado el paso inductivo, que junto con la demostración del caso base establecen la propiedad original para cualquier lista finita.

2 Funciones de plegado

Para cada tipo recursivo es posible definir una función de orden superior que capture la recursión sobre argumentos de dicho tipo. Estas funciones se denominan *funciones de plegado* (*fold* en inglés) aunque también son conocidas como *catamorfismos* o *recursores*. En el caso de las listas, estas funciones capturan el comportamiento de las funciones recursivas que recorren todos los elementos de la lista para calcular un resultado. La lista puede recorrerse de derecha a izquierda o en el sentido inverso, por lo que existen distintas funciones para cada caso.

2.1 foldr

Muchas de las funciones que se definen sobre listas siguen el siguiente patrón recursivo:

- Si el argumento es la lista vacía, se devuelve cierto valor base (correspondiente al *caso base* de la definición).
- En otro caso, se opera, mediante cierta función u operador (*plegador*), la cabeza de la lista y una llamada recursiva con la cola de la lista.

Por ejemplo, una función que sume los elementos de una lista de enteros puede ser definida del siguiente modo:

$$
\begin{array}{ll}
sumar & :: [Integer] \rightarrow Integer \\
sumar\ [\,] & = 0 \\
sumar\ (x : xs) & = x\ +\ sumar\ xs
\end{array}
$$

```
? sumar [1,2,3]
6
```

En esta definición el valor base es 0 y el plegador es $(+)$.

Otro ejemplo es la función *concat*, que concatena las sublistas de una lista de listas:

$$
\begin{array}{l}
concat \quad :: [[a]] \to [a] \\
concat\ [\,] \quad = [\,] \\
concat\ (xs : xss) = xs +\!\!+ concat\ xss
\end{array}
$$

$$
\begin{array}{l}
?\ \ concat\ [\,[1,2],\ [3],\ [4,5,6]] \\
[1,2,3,4,5,6]
\end{array}
$$

Aquí, el valor base es la lista vacía y el plegador es la concatenación de listas $(+\!\!+)$.

Ambas definiciones siguen el siguiente esquema o plantilla:

$$
\begin{array}{l}
fun\ [\,] \quad = \boxed{\ z\ } \\
fun\ (x : xs) = x \boxed{\ \oplus\ } (fun\ xs)
\end{array}
$$

donde z representa el valor base y $(\oplus)$ el plegador. Abstrayendo sobre estos valores, podemos definir un combinador que tome como argumentos ambos y capture el comportamiento común[2]:

$$
\begin{array}{l}
foldr \quad :: (a \to b \to b) \to b \to [a] \to b \\
foldr\ (\oplus)\ z = fun \\
\quad \textbf{where} \\
\quad\ fun\ [\,] \quad = z \\
\quad\ fun\ (x : xs) = x \oplus (fun\ xs)
\end{array}
$$

Los ejemplos anteriores pueden ser definidos de un modo mucho más breve utilizando *foldr*:

$$
\begin{array}{l}
sumar :: [Integer] \to Integer \\
sumar = foldr\ (+)\ 0
\end{array}
$$

$$
\begin{array}{l}
concat :: [[a]] \to [a] \\
concat = foldr\ (+\!\!+)\ [\,]
\end{array}
$$

Vemos que *foldr* es la función de *plegado* de las listas y, como tal, simplemente reemplaza constructores por funciones. Así, dado un operador $(\oplus)$ con tipo $a \to b \to b$, y un valor z con tipo b, la expresión $foldr\ (\oplus)\ z$ procesa una lista de tipo $[a]$ para producir un resultado con tipo b, reemplazando para ello el constructor final $[\,]$ de la lista por z, y cada ocurrencia del constructor (:) en la lista por el operador $(\oplus)$. Podemos entonces definir el comportamiento de *foldr* en la forma:

[2] *foldr* es un operador predefinido cuyo nombre viene del término inglés *fold* (plegar) y *right* (a la derecha).

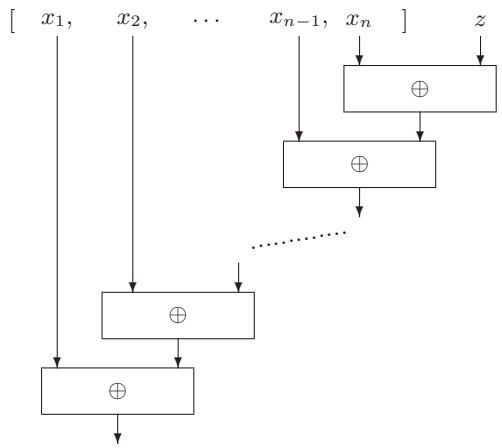

Figura 3. Reducción de $foldr\ \oplus\ z\ [x_1,\ \ldots,\ x_{n-1},\ x_n]$

Comportamiento de foldr

$$foldr(\oplus)\ z\ (x_1 : (x_2 : (\ldots\ : (x_n : [\,]) \ldots)))$$
$$\Longrightarrow$$
$$x_1\ \oplus\ (x_2\ \oplus\ (\ldots\ \oplus\ (x_n\ \oplus\ z) \ldots))$$

Por ejemplo, la evaluación de la expresión $foldr\ (+)\ 0\ [1,2,3]$ es la siguiente[3]:

$$\begin{array}{ll}
 & foldr\ (+)\ 0\ [1,2,3] \\
\rightsquigarrow & \quad \{\text{sintaxis de listas}\} \\
 & \underline{foldr\ (+)\ 0\ (1 : (2 : (3 : [\,])))} \\
\Longrightarrow & \quad \{\text{primera ecuación de } foldr\} \\
 & 1\ +\ (\underline{foldr\ (+)\ 0\ (2 : (3 : [\,]))}) \\
\Longrightarrow & \quad \{\text{primera ecuación de } foldr\} \\
 & 1\ +\ (2\ +\ (\underline{foldr\ (+)\ 0\ (3 : [\,])})) \\
\Longrightarrow & \quad \{\text{primera ecuación de } foldr\} \\
 & 1\ +\ (2\ +\ (3\ +\ (\underline{foldr\ (+)\ 0\ [\,]}))) \\
\Longrightarrow & \quad \{\text{segunda ecuación de } foldr\} \\
 & 1\ +\ (2\ +\ (3\ +\ 0)) \\
\Longrightarrow & \quad \{\text{definición de } (+) \text{ tres veces}\} \\
 & 6
\end{array}$$

Como muestra el ejemplo, las distintas llamadas a *foldr* simplemente han reemplazado el constructor (:) por el operador (+) y [ ] por la constante 0. La Figura 3 muestra gráficamente la función de plegado.

[3] Notaremos que una expresión e_1 es interpretada como otra e_2 escribiendo $e_1 \rightsquigarrow e_2$.

Una definición alternativa de *foldr* que aparece en la mayoría de los textos es la siguiente, aunque el lector puede comprobar que es idéntica a la presentada previamente:

```
foldr                :: (a → b → b) → b → [a] → b
foldr (⊕) z []       = z
foldr (⊕) z (x : xs) = x ⊕ (foldr (⊕) z xs)
```

Son muchas las funciones sobre listas que pueden ser definidas usando *foldr*. Por ejemplo:

```
-- Producto de una lista de enteros
multiplicar :: [Integer] → Integer
multiplicar = foldr (∗) 1

-- Conjunción de una lista de valores lógicos
and :: [Bool] → Bool
and = foldr (&&) True

-- Disyunción de una lista de valores lógicos
or :: [Bool] → Bool
or = foldr (||) False

-- La longitud de una lista
length :: [a] → Int
length = foldr unoMás 0
  where
    unoMás x n = 1 + n
-- Invierte una lista
reverse :: [a] → [a]
reverse = foldr alFinal []
  where
    alFinal x xs = xs ++ [x]
```

En principio, los programas escritos usando *foldr* pueden parecer menos claros que los escritos usando recursividad explícita. Sin embargo, es conveniente utilizar este combinador por los siguientes motivos:

- El programador solo tiene que proporcionar los *ingredientes* de *foldr* para el problema concreto. No es necesario prestar atención a la recursividad, ya que ésta se encuentra encapsulada en el combinador. Esto mejora el desarrollo de los programas y se evitan errores.
- Las funciones definidas usando *foldr* son más apropiadas para razonar ecuacionalmente, lo cual simplificará el proceso de establecer propiedades y transformar programas.
- Algunos compiladores pueden optimizar automáticamente expresiones definidas mediante *foldr* y combinadores similares.

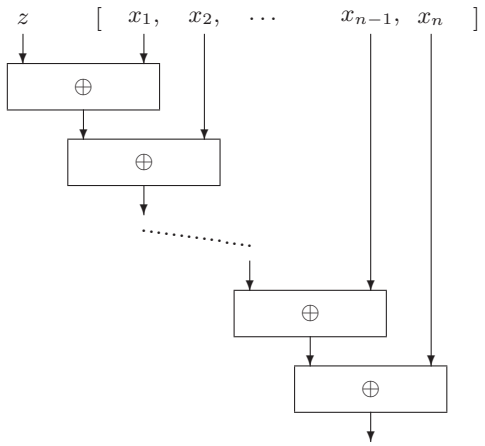

Figura 4. Reducción de $foldl\ (\oplus)\ z\ [x_1,\ \ldots,\ x_{n-1},\ x_n]$

2.2 foldl

Existe una función similar a *foldr* que realiza el plegado de la lista de izquierda a derecha. Esta función se denomina *foldl* y se define del siguiente modo:

$$
\begin{array}{l}
foldl \quad :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\
foldl\ (\oplus)\ z\ = fun\ z \\
\quad \textbf{where} \\
\quad\ fun\ acc\ [\,] \quad\ = acc \\
\quad\ fun\ acc\ (x : xs)\ = fun\ (acc\ \oplus\ x)\ xs
\end{array}
$$

El comportamiento de la función *foldl*, como muestra la Figura 4, es el siguiente:

$$
\begin{array}{c}
\textit{Comportamiento de foldl} \\
foldl(\oplus)\ z\ [x_1, x_2, \ldots, x_n] \\
\Longrightarrow \\
((\ldots((z\ \oplus\ x_1)\ \oplus\ x_2)\ldots\ \oplus\ x_{n-1})\ \oplus\ x_n)
\end{array}
$$

Una definición alternativa de *foldl* que aparece en la mayoría de los textos es la siguiente, aunque el lector puede comprobar que es idéntica a la presentada previamente:

$$
\begin{array}{l}
foldl \quad :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\
foldl\ (\oplus)\ z\ [\,] \quad = z \\
foldl\ (\oplus)\ z\ (x : xs)\ = foldl\ (\oplus)\ (z\ \oplus\ x)\ xs
\end{array}
$$

Como ejemplo, definiremos una función que calcule el valor decimal de una lista de dígitos usando la regla de Horner:

$$decimal\ [x_1,\ x_2,\ x_3]\ =\ 10\ *\ (10\ *\ (10\ *\ (0\ +\ x_1))\ +\ x_2)\ +\ x_3$$

Si definimos el operador $(\times)$ como $a\ \times\ b\ =\ 10\ *\ a\ +\ b$ entonces la regla de Horner puede expresarse:

$$decimal\ [x_1,\ x_2,\ x_3]\ =\ ((0\ \times\ x_1))\ \times\ x_2)\ \times\ x_3$$

Esta expresión se corresponde con el patrón de cómputo realizado por *foldl* por lo que podemos definir la función *decimal* como:

```
decimal :: [Integer] → Integer
decimal = foldl (×) 0
  where
    a × b = 10 * a + b
```

o usando la composición

```
decimal :: [Integer] → Integer
decimal = foldl ((+) . (10*)) 0

? decimal [1,2,3]
123
```

Aunque los resultados de *foldl* y *foldr* coinciden a veces (veremos en qué condiciones posteriormente) una de las dos definiciones puede ser más eficiente. Por ejemplo, la definición de *reverse* con *foldl* es

```
reverse' :: [a] → [a]
reverse' = foldl alInicio []
  where
    alInicio xs x = x : xs
```

Recordemos que la que usa *foldr* es

```
reverse :: [a] → [a]
reverse = foldr alFinal []
  where
    alFinal x xs = xs ++ [x]
```

Ambas funciones realizan m llamadas recursivas al invertir una lista de longitud m. Sin embargo, el coste de cada una de las llamadas es distinto en cada caso, lo cual influye en la complejidad final de las funciones. En efecto, la función *alInicio* tiene complejidad $O(1)$ mientras que *alFinal* es $O(n)$ siendo n la longitud de su segundo argumento. Por ello, la definición *reverse'* es lineal en la longitud de la lista a invertir mientras que *reverse* es cuadrática sobre el mismo parámetro.

3 La propiedad universal de foldr

La propiedad universal de *foldr* puede ser enunciada como la siguiente equivalencia entre dos definiciones de una función g que procese listas finitas[4].

Para cualquier $z :: b$, cualquier operador $(\oplus) :: a \to b \to b$ y cualquier función $g :: [a] \to b$, se tiene la siguiente equivalencia

Propiedad universal de foldr

$$\forall x :: a, xs :: [a] \,.\, \left\{ \begin{array}{lcl} g\ [\,] & = & z \\ g\ (x : xs) & = & x \oplus (g\ xs) \end{array} \right\} \quad \Leftrightarrow \quad g = \mathit{foldr}\ (\oplus)\ z$$

Llamemos $(\mathcal{I})$ a la parte izquierda de la equivalencia. Sean, además, z, $(\oplus)$ y g tres valores de tipos adecuados. El sentido $(\Leftarrow)$ de la propiedad es obvio, ya que sustituyendo g por $\mathit{foldr}\ (\oplus)\ z$ en las dos ecuaciones de $(\mathcal{I})$ se obtiene

$$\begin{array}{ll} \mathit{foldr}\ (\oplus)\ z\ [\,] & = z \\ \mathit{foldr}\ (\oplus)\ z\ (x : xs) & = x\ \oplus\ (\mathit{foldr}\ (\oplus)\ z\ xs) \end{array}$$

que es la definición de *foldr*.

El sentido $(\Rightarrow)$ puede ser demostrado por inducción sobre la lista. Supongamos cierta $(\mathcal{I})$. Vamos a probar

$$\forall\ xs\ ::\ [a]\ .\ g\ xs\ =\ \mathit{foldr}\ (\oplus)\ z\ xs$$

por inducción sobre la lista xs.

- Caso Base: Hay que demostrar

  $$g\ [\,]\ =\ \mathit{foldr}\ (\oplus)\ z\ [\,]$$

  Para la parte izquierda tenemos:

  $$\begin{array}{ll} & g\ [\,] \\ \equiv & \{\text{por } (\mathcal{I})\} \\ & z \end{array}$$

  Para la parte derecha tenemos:

  $$\begin{array}{ll} & \mathit{foldr}\ (\oplus)\ z\ [\,] \\ \equiv & \{\text{por definición de } \mathit{foldr}\} \\ & z \end{array}$$

  con lo que queda demostrado el caso base.

- Paso Inductivo: Hay que probar

[4] Para listas infinitas véase (Bird 98)

$$
\begin{array}{l}
\quad g\ xs\ =\ foldr\ (\oplus)\ z\ xs \qquad \{\text{Hipótesis de Inducción}\} \\
\Rightarrow \\
\quad g\ (x : xs)\ =\ foldr\ (\oplus)\ z\ (x : xs)
\end{array}
$$

Para la parte izquierda tenemos:

$$
\begin{array}{ll}
 & g\ (x : xs) \\
\equiv & \{\text{por } (\mathcal{I})\} \\
 & x\ \oplus\ (g\ xs) \\
\equiv & \{\text{por Hipótesis de Inducción}\} \\
 & x\ \oplus\ (foldr\ (\oplus)\ z\ xs)
\end{array}
$$

Para la parte derecha tenemos:

$$
\begin{array}{ll}
 & foldr\ (\oplus)\ z\ (x : xs) \\
\equiv & \{\text{por definición de } foldr\} \\
 & x\ \oplus\ (foldr\ (\oplus)\ z\ xs)
\end{array}
$$

con lo que queda demostrado el paso inductivo y con ello la propiedad para cualquier lista finita.

3.1 La propiedad universal como principio de prueba

El atractivo principal de la propiedad universal es que hace explícitas las dos condiciones requeridas por la mayoría de demostraciones inductivas. Así, para demostrar una propiedad específica que tome la forma de la parte derecha de la propiedad universal, basta con demostrar las dos condiciones de la izquierda, las cuales se suelen poder verificar sin necesidad de utilizar la inducción. Estas dos condiciones forman el *principio de prueba*. Es decir, la propiedad universal encapsula un patrón simple de pruebas inductivas sobre listas al igual que la función *foldr* encapsula un patrón de definición recursivo.

Por ejemplo, para demostrar la siguiente igualdad

$$(1+)\ .\ sumar\ =\ foldr\ (+)\ 1$$

que establece que es lo mismo sumar los elementos de una lista e incrementar el resultado que plegar la lista con $(+)$ partiendo del valor 1, dado que la igualdad es una instancia de la consecuencia de la propiedad universal

$$\underbrace{(1+)\ .\ sumar}_{g}\ =\ foldr\ \underbrace{(+)}_{(\oplus)}\ \underbrace{1}_{z}$$

basta con demostrar las precondiciones de ésta, o sea,

$$
\begin{array}{l}
\forall\, x\ ::\ Integer, \forall\, xs\ ::\ [Integer]\,\bullet \\
\quad ((1+)\ .\ sumar)\ [\,] \qquad\ = 1 \\
\quad ((1+)\ .\ sumar)\ (x : xs)\ = x\ +\ (((1+)\ .\ sumar)\ xs)
\end{array}
$$

Eliminando el operador de composición obtenemos las siguientes ecuaciones

$$\begin{aligned} 1 \;+\; &sumar\,[\,] &&= 1 \\ 1 \;+\; &sumar\,(x : xs) &&= x \;+\; (1 \;+\; sumar\; xs) \end{aligned}$$

que pueden ser verificadas fácilmente

$$\begin{aligned} & 1 \;+\; sumar\,[\,] \\ \equiv\;& \{\text{por definición de } sumar\} \\ & 1 \;+\; 0 \\ \equiv\;& \{\text{aritmética}\} \\ & 1 \end{aligned}$$

$$\begin{aligned} & 1 \;+\; sumar\,(x : xs) \\ \equiv\;& \{\text{por definición de } sumar\} \\ & 1 \;+\; (x \;+\; sumar\; xs) \\ \equiv\;& \{\text{asociatividad y conmutatividad de } (+)\} \\ & x \;+\; (1 \;+\; sumar\; xs) \end{aligned}$$

con lo que quedan demostradas las dos precondiciones. Aplicando la propiedad universal queda demostrada la propiedad original sin necesidad de usar inducción. En general, es posible demostrar la equivalencia de dos funciones sobre listas usando la propiedad universal si una de ellas puede ser expresada usando *foldr*.

3.2 Síntesis de programas vía la propiedad universal

Además de como un *principio de prueba*, la propiedad universal puede ser utilizada para calcular la definición de funciones recursivas sobre listas en base a *foldr*. Como primer ejemplo simple, veamos como se puede sintetizar la definición basada en *foldr* de *sumar* a partir de su definición recursiva:

$$\begin{aligned} &sumar &&:: [Integer] \to Integer \\ &sumar\,[\,] &&= 0 \\ &sumar\,(x : xs) &&= x \;+\; sumar\; xs \end{aligned}$$

Se trata de calcular los valores de $(\oplus)$ y z que verifiquen la ecuación $sumar = foldr\,(\oplus)\,z$. Por la propiedad universal, esta ecuación es equivalente a las siguientes:

$$\begin{aligned} &sumar\,[\,] &&= z \\ &sumar\,(x : xs) &&= x \;\oplus\; (sumar\; xs) \end{aligned}$$

y de la primera ecuación podemos calcular el valor de z:

$$\begin{aligned} & sumar\,[\,] \;=\; z \\ \Leftrightarrow\;& \{\text{por definición de } sumar\} \\ & 0 \;=\; z \end{aligned}$$

De la segunda calculamos $(\oplus)$:

$$
\begin{array}{ll}
 & sumar\ (x : xs)\ =\ x\ \oplus\ (sumar\ xs) \\
\Leftrightarrow & \{\text{por definición de } sumar\} \\
 & x\ +\ sumar\ xs\ =\ x\ \oplus\ (sumar\ xs) \\
\Leftarrow & \{\text{generalizando } sumar\ xs \text{ a } y\} \\
 & \forall\, y\ .\ x\ +\ y\ =\ x\ \oplus\ y \\
\Leftrightarrow & \{\text{extensionalidad de funciones}\} \\
 & (+)\ =\ (\oplus)
\end{array}
$$

Es decir, usando la propiedad universal hemos calculado la siguiente definición de *sumar*:

$$sumar\ =\ foldr\ (+)\ 0$$

La generalización de la expresión *sumar xs* a y es la clave del cálculo. De hecho, este paso no es específico del ejemplo y aparecerá en la síntesis en base a *foldr* de cualquier función recursiva con esta técnica. Hay que enfatizar que todo el proceso es posible gracias a la *transparencia referencial* del lenguaje, que permite sustituir una expresión por otra equivalente.

El ejemplo desarrollado es artificial, ya que la definición de sumar en base a *foldr* es inmediata. Sin embargo, la técnica es útil en otros casos menos evidentes. Veamos como podemos expresar la función

$$
\begin{array}{ll}
map & ::\ (a\ \rightarrow\ b)\ \rightarrow\ [a]\ \rightarrow\ [b] \\
map\ f\ [\,] & =\,[\,] \\
map\ f\ (x : xs) & =\, f\ x\ :\ map\ f\ xs
\end{array}
$$

usando *foldr*. Queremos resolver la ecuación $map\ f\ =\ foldr\ (\oplus)\ z$. Por la propiedad universal, esta ecuación es equivalente a las dos siguientes:

$$
\begin{array}{ll}
map\ f\ [\,] & =\ z \\
map\ f\ (x : xs) & =\ x\ \oplus\ (map\ f\ xs)
\end{array}
$$

Calculamos z de la primera:

$$
\begin{array}{ll}
 & map\ f\ [\,]\ =\ z \\
\Leftrightarrow & \{\text{por definición de } map\} \\
 & [\,]\ =\ z
\end{array}
$$

De la segunda calculamos $(\oplus)$:

$$
\begin{array}{ll}
 & map\ f\ (x : xs)\ =\ x\ \oplus\ (map\ f\ xs) \\
\Leftrightarrow & \{\text{por definición de } map\} \\
 & f\ x\ :\ map\ f\ xs\ =\ x\ \oplus\ (map\ f\ xs) \\
\Leftarrow & \{\text{generalizando } map\ f\ xs \text{ a } ys\} \\
 & \forall\, ys\ .\ f\ x\ :\ ys\ =\ x\ \oplus\ ys \\
\Leftrightarrow & \{\text{introduciendo } aplica\} \\
 & \forall\, ys\ .\ aplica\ x\ ys\ =\ x\ \oplus\ ys\ \textbf{where}\ aplica\ z\ zs\ =\ f\ z\ :\ zs
\end{array}
$$

$\Leftrightarrow$ {extensionalidad de funciones}
$aplica\ =\ (\oplus)$ **where** $aplica\ z\ zs\ =\ f\ z\ :\ zs$

Por lo que hemos calculado la siguiente definición de *map*:

$$\begin{array}{l} map\ f\ =\ foldr\ aplica\ [\,] \\ \quad \textbf{where} \\ \quad\quad aplica\ z\ zs\ =\ f\ z\ :\ zs \end{array}$$

En general, cualquier función recursiva que pueda ser expresada usando *foldr* puede ser calculada utilizando esta técnica.

Ejercicio 1. Considérese la siguiente función que calcula los prefijos de una lista

$$\begin{array}{ll} prefijos & :: [a] \rightarrow [[a]] \\ prefijos\ [\,] & = [[\,]] \\ prefijos\ (x : xs) & = [\,]\ :\ map\ (x :)\ (prefijos\ xs) \end{array}$$

Por ejemplo

$$\begin{array}{l} ?\ \ prefijos\ [1,2,3] \\ [[\,],\ [1],\ [1,2],\ [1,2,3]] \end{array}$$

Sintetiza una definición de *prefijos* usando *foldr*. □

4 Fusión de plegados

En esta sección presentamos los teoremas de fusión correspondientes a las dos funciones de plegado.

4.1 Un teorema de fusión para foldr

La siguiente ecuación aparece a menudo al razonar con funciones sobres listas que pueden ser expresadas usando *foldr*

$$h\ .\ foldr\ (\otimes)\ w\ =\ foldr\ (\oplus)\ z$$

La parte izquierda de la ecuación representa la función que primero pliega con $(\otimes)$ y un valor base w y al resultado de este plegado le aplica la función h.

Evidentemente, la propiedad no es siempre cierta, pero podemos usar la propiedad universal para calcular las condiciones que deben cumplir h, $(\oplus)$, w, $(\otimes)$ y z para que la propiedad se verifique.

La forma de la ecuación coincide con la propiedad universal (ya que la parte derecha es un plegado) por lo que la ecuación es equivalente a las dos siguientes:

$$\begin{array}{l} \forall\, x\ ::\ a,\ \forall\, xs\ ::\ [a]\ \centerdot \\ \quad (h\ .\ foldr\ (\otimes)\ w)\ [\,] \qquad\quad = z \\ \quad (h\ .\ foldr\ (\otimes)\ w)\ (x : xs)\ = x\ \oplus\ ((h\ .\ foldr\ (\otimes)\ w)\ xs) \end{array}$$

que simplificando quedan como

$$\begin{array}{l}\forall\, x :: a,\ \forall\, xs :: [a] \,\centerdot \\ \quad h\ (\mathit{foldr}\ (\otimes)\ w\ [\,]) \qquad\quad = z \\ \quad h\ (\mathit{foldr}\ (\otimes)\ w\ (x : xs)) = x \oplus (h\ (\mathit{foldr}\ (\otimes)\ w\ xs))\end{array}$$

De la primera ecuación obtenemos la siguiente condición:

$$\begin{array}{ll} & h\ (\mathit{foldr}\ (\otimes)\ w\ [\,]) = z \\ \Leftrightarrow & \{\text{por definición de } \mathit{foldr}\} \\ & h\ w = z\end{array}$$

Mientras que de la segunda:

$$\begin{array}{ll} & h\ (\mathit{foldr}\ (\otimes)\ w\ (x : xs)) = x \oplus (h\ (\mathit{foldr}\ (\otimes)\ w\ xs)) \\ \Leftrightarrow & \{\text{por definición de } \mathit{foldr}\} \\ & h\ (x \otimes (\mathit{foldr}\ (\otimes)\ w\ xs)) = x \oplus (h\ (\mathit{foldr}\ (\otimes)\ w\ xs)) \\ \Leftarrow & \{\text{generalizando } (\mathit{foldr} \otimes w\ xs) \text{ a } ys\} \\ & \forall\, ys \centerdot h\ (x \otimes ys) = x \oplus (h\ ys)\end{array}$$

Es decir, hemos calculado usando la propiedad universal dos condiciones simples que garantizan la igualdad anterior para listas finitas:

Teorema de fusión de foldr

$$\left.\begin{array}{lcl} h\ w & = & z \\ \forall x, y \centerdot h\ (x \otimes y) & = & x \oplus (h\ y)\end{array}\right\} \Rightarrow \quad h\ .\ \mathit{foldr}\ (\otimes)\ w = \mathit{foldr}\ (\oplus)\ z$$

El teorema se denomina de fusión porque se fusiona el plegado $\mathit{foldr}\ (\otimes)\ w$ seguido de la función h en un único plegado $(\mathit{foldr}\ (\oplus)\ z)$.

Podemos ilustrar el teorema con la siguiente transformación

$$\begin{array}{ll} & (h\ .\ \mathit{foldr}\ (\otimes)\ w)\ [a_1, a_2, a_3] \\ \equiv & \{\text{por definición de } (.)\} \\ & h\ (\mathit{foldr}\ (\otimes)\ w\ [a_1, a_2, a_3]) \\ \equiv & \{\text{por definición de } \mathit{foldr}\} \\ & h\ (a_1 \otimes (a_2 \otimes (a_3 \otimes w))) \\ \equiv & \{\text{ya que } h\ (x \otimes y) = x \oplus (h\ y)\} \\ & a_1 \oplus h\ (a_2 \otimes (a_3 \otimes w)) \\ \equiv & \{\text{ya que } h\ (x \otimes y) = x \oplus (h\ y)\} \\ & a_1 \oplus (a_2 \oplus h\ (a_3 \otimes w)) \\ \equiv & \{\text{ya que } h\ (x \otimes y) = x \oplus (h\ y)\} \\ & a_1 \oplus (a_2 \oplus (a_3 \oplus (h\ w))) \\ \equiv & \{\text{ya que } h\ w = z\} \\ & a_1 \oplus (a_2 \oplus (a_3 \oplus z)) \\ \equiv & \{\text{por definición de } \mathit{foldr}\} \\ & \mathit{foldr}\ (\oplus)\ z\ [a_1, a_2, a_3]\end{array}$$

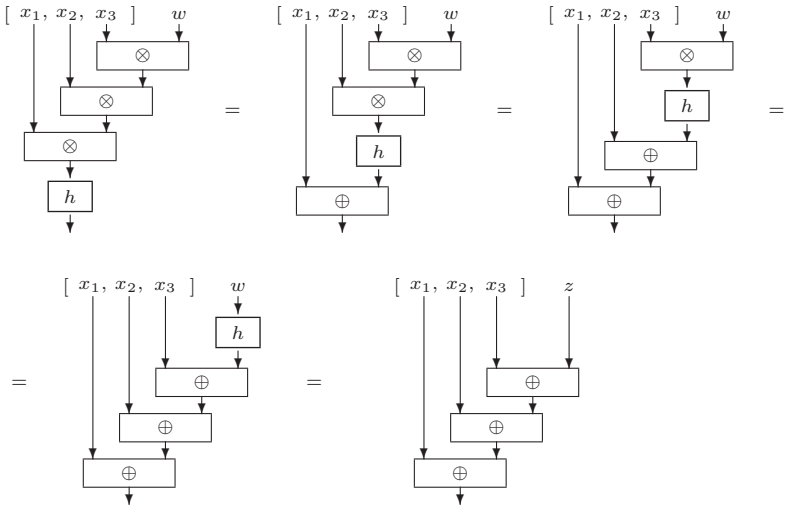

Figura 5. El teorema de fusión de $foldr$

Gráficamente (ver Figura 5), el bloque constituido por la función h tras el operador $(\otimes)$ puede ser sustituido por un bloque con la salida de la función h como segundo argumento del operador $(\oplus)$. De este modo la función h asciende a través del plegado hasta alcanzar el valor base w.

4.1.1 Fusión de map y foldr

Como aplicación del teorema veamos cuando puede fusionarse una aplicación de *map* seguida de un plegado en un único plegado:

$$foldr\ f\ u\ .\ map\ g\ =\ foldr\ (\oplus)\ z$$

Es decir, se trata de calcular las incógnitas $(\oplus)$ y z para que la propiedad sea cierta. Dado que *map* puede ser expresada usando *foldr*, la propiedad anterior puede ser escrita como

$$(foldr\ f\ u\ .\ foldr\ aplica\ [\,]\ \textbf{where}\ aplica\ z\ zs\ =\ g\ z\ :\ zs)\ =\ foldr\ (\oplus)\ z$$

Esta expresión es un caso particular de la consecuencia del teorema de fusión

$$\underbrace{foldr f\ u}_{h}\ .\ foldr\ \underbrace{aplica}_{(\otimes)}\ \underbrace{[\,]}_{w}\ =\ foldr\ (\oplus)\ z$$

Por el teorema de fusión basta que se cumplan las siguientes dos condiciones:

$$\begin{array}{ll} foldr\ f\ u\ [\,] & =z \\ (foldr\ f\ u\ (aplica\ x\ ys)\ \textbf{where}\ aplica\ z\ zs\ =\ g\ z\ :\ zs) & =x\ \oplus\ (foldr\ f\ u\ ys) \end{array}$$

De la primera ecuación obtenemos que los valores de u y z han de coincidir:

$$
\begin{array}{ll}
 & \mathit{foldr}\ f\ u\ [\,]\ =\ z \\
\Leftrightarrow & \{\text{por definición de } \mathit{foldr}\} \\
 & u\ =\ z
\end{array}
$$

La segunda puede transformarse en otra condición más restrictiva

$$
\begin{array}{ll}
 & \mathit{foldr}\ f\ u\ (\mathit{aplica}\ x\ ys)\ =\ x\ \oplus\ (\mathit{foldr}\ f\ u\ ys) \\
\Leftrightarrow & \{\text{por definición de } \mathit{aplica}\} \\
 & \mathit{foldr}\ f\ u\ (g\ x\ :\ ys)\ =\ x\ \oplus\ (\mathit{foldr}\ f\ u\ ys) \\
\Leftrightarrow & \{\text{por definición de } \mathit{foldr}\} \\
 & f\ (g\ x)\ (\mathit{foldr}\ f\ u\ ys)\ =\ x\ \oplus\ (\mathit{foldr}\ f\ u\ ys) \\
\Leftarrow & \{\text{generalizando } (\mathit{foldr}\ f\ u\ ys) \text{ u } xs\} \\
 & \forall\ xs\ \centerdot\ f\ (g\ x)\ xs\ =\ x\ \oplus\ xs \\
\Leftrightarrow & \{\text{por extensionalidad de funciones}\} \\
 & f\ (g\ x)\ =\ (\oplus)\ x \\
\Leftrightarrow & \{\text{por composición de funciones}\} \\
 & (f\ .\ g)\ x\ =\ (\oplus)\ x \\
\Leftrightarrow & \{\text{por extensionalidad de funciones}\} \\
 & f\ .\ g\ =\ (\oplus)
\end{array}
$$

Aplicando las dos condiciones a la ecuación original obtenemos la propiedad de fusión de *map* y *foldr*:

Propiedad de fusión de map y foldr
$\mathit{foldr}\ f\ u\ .\ \mathit{map}\ g\ =\ \mathit{foldr}\ (f.g)\ u$

que dice que un plegado tras una aplicación de *map* siempre pueden ser simplificados a un único plegado. La interpretación gráfica de la propiedad puede verse en la Figura 6.

La propiedad puede ser utilizada para optimizar programas que generen una lista intermedia. Mediante el esquema anterior, lo transformamos en otro programa que no genere la lista intermedia y dé una única pasada a la lista original.

Por ejemplo, la siguiente definición de la función *sumarCuadrados* calcula la suma de los cuadrados de los elementos de una lista:

```
sumarCuadrados :: [Int] → Int
sumarCuadrados = sumar . map alCuadrado
  where
    alCuadrado x = x * x
```

Aunque la definición es muy clara, es poco eficiente: se genera una lista intermedia con los cuadrados de los elementos. Como $\mathit{sumar}\ =\ \mathit{foldr}\ (+)\ 0$, la definición anterior es equivalente a

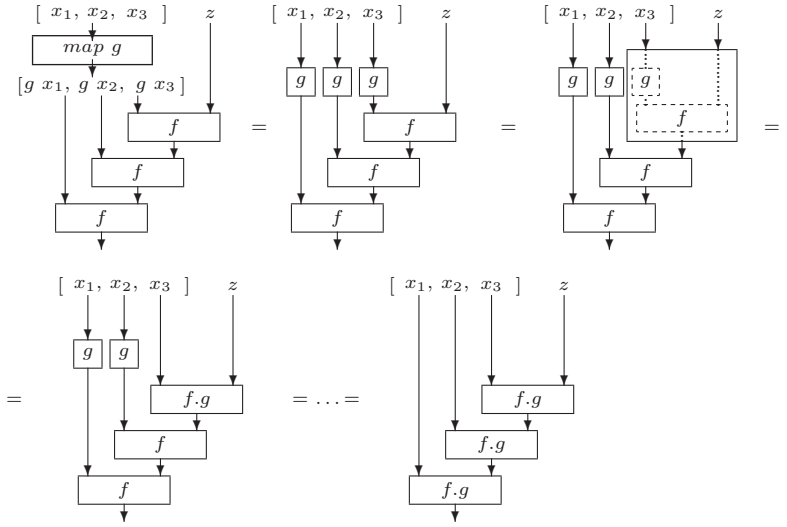

Figura 6. Fusión de *map* y *foldr*

$$
\begin{array}{l}
sumarCuadrados \;::\; [Int] \;\rightarrow\; Int \\
sumarCuadrados \;=\; foldr\;(+)\;0\;.\;map\;alCuadrado \\
\quad \textbf{where} \\
\qquad alCuadrado\;x \;=\; x * x
\end{array}
$$

Es posible utilizar la propiedad de fusión de *map* y *foldr* para sintetizar una función más eficiente que no genere la lista intermedia:

$$
\begin{array}{l}
sumarCuadrados \;::\; [Int] \;\rightarrow\; Int \\
sumarCuadrados \;=\; foldr\;(\;(+)\;.\;alCuadrado\;)\;0 \\
\quad \textbf{where} \\
\qquad alCuadrado\;x \;=\; x * x
\end{array}
$$

4.2 Un teorema de fusión para foldl

También podemos enunciar la correspondiente propiedad para la función de plegado a la izquierda:

Teorema de fusión de foldl

$$
\left.
\begin{array}{lcl}
h\;w & = & z \\
\forall x, y \,.\, h\;(x\;\otimes\;y) & = & (h\;x)\;\oplus\;y
\end{array}
\right\}
\;\Rightarrow\;
h\;.\;foldl\;(\otimes)\;w = foldl\;(\oplus)\;z
$$

La demostración de este teorema es un poco más complicada y requiere que demostremos previamente que la hipótesis del teorema, que llamaremos ($\mathcal{I}$), implica el siguiente lema ($\mathcal{L}$) para todo x, y y as con los tipos adecuados:

$$(\mathcal{L}) \quad \forall\, x,\ y,\ as \ .\ h\,(foldl\,(\otimes)\,(x\ \otimes\ y)\,as) \ =\ foldl\,(\oplus)\,((h\,x)\ \oplus\ y)\,as$$

Es decir, hemos de demostrar la siguiente implicación:

$$(\mathcal{I}) \quad \Rightarrow \quad \left\{ \begin{array}{l} \forall\, x,\ y,\ as \,. \\ \quad h\ (foldl\ (\otimes)\ (x\ \otimes\ y)\ as) \\ = \\ \quad foldl\ (\oplus)\ ((h\,x)\ \oplus\ y)\ as \end{array} \right.$$

Supongamos cierto $(\mathcal{I})$. Procedemos por inducción sobre la lista as:

- Caso Base. Hemos de probar:

$$\begin{array}{l} \forall\, x,\ y \,. \\ \quad h\ (foldl\ (\otimes)\ (x\ \otimes\ y)\ [\,]) \\ = \\ \quad foldl\ (\oplus)\ ((h\,x)\ \oplus\ y)\ [\,] \end{array}$$

Transformamos la parte izquierda:

$$\begin{array}{ll} & h\,(foldl\,(\otimes)\,(x\ \otimes\ y)\,[\,]) \\ \equiv & \{\text{por definición de } foldl\} \\ & h\,(x\ \otimes\ y) \end{array}$$

Y ahora la derecha:

$$\begin{array}{ll} & foldl\,(\oplus)\,((h\,x)\ \oplus\ y)\,[\,] \\ \equiv & \{\text{por definición de } foldl\} \\ & ((h\,x)\ \oplus\ y) \\ \equiv & \{\text{por } (\mathcal{I})\} \\ & h\,(x\ \otimes\ y) \end{array}$$

con lo que queda demostrado el caso base.

- Paso Inductivo. Tenemos la siguiente hipótesis de inducción:

$$\begin{array}{l} \forall\, x',\ y' \,. \\ \quad h\ (foldl\ (\otimes)\ (x'\ \otimes\ y')\ as) \\ = \\ \quad foldl\ (\oplus)\ ((h\,x')\ \oplus\ y')\ as \end{array}$$

y hemos de probar

$$\begin{array}{l} \forall\, x,\ y,\ a \,. \\ \quad h\ (foldl\ (\otimes)\ (x\ \otimes\ y)\ (a : as)) \\ = \\ \quad foldl\ (\oplus)\ ((h\,x)\ \oplus\ y)\ (a : as) \end{array}$$

Transformamos la parte izquierda:

$$
\begin{array}{ll}
& h\ (\mathit{foldl}\ (\otimes)\ (x\ \otimes\ y)\ (a : as)) \\
\equiv & \{\text{por definición de } \mathit{foldl}\} \\
& h\ (\mathit{foldl}\ (\otimes)\ ((x\ \otimes\ y)\ \otimes\ a)\ as)
\end{array}
$$

Y ahora la derecha:

$$
\begin{array}{ll}
& \mathit{foldl}\ (\oplus)\ ((h\ x)\ \oplus\ y)\ (a : as) \\
\equiv & \{\text{por definición de } \mathit{foldl}\} \\
& \mathit{foldl}\ (\oplus)\ (((h\ x)\ \oplus\ y)\ \oplus\ a)\ as \\
\equiv & \{\text{por } (\mathcal{I})\} \\
& \mathit{foldl}\ (\oplus)\ (h\ (x\ \otimes\ y)\ \oplus\ a)\ as \\
\equiv & \{\text{por Hipótesis de Inducción, tomando } x' = x \otimes y,\ y' = a\} \\
& h\ (\mathit{foldl}\ (\otimes)\ ((x\ \otimes\ y)\ \otimes\ a)\ as)
\end{array}
$$

con lo que queda demostrado el paso inductivo y con ello el lema $(\mathcal{L})$.

Pasamos ahora a probar el teorema de fusión. Hay que probar

$$(\mathcal{I})\ \Rightarrow\ \ \forall\ as\ ::\ [a]\ \centerdot\ (h\ .\ \mathit{foldl}\ (\otimes)\ w)\ as\ =\ \mathit{foldl}\ (\oplus)\ z\ as$$

Consideremos pues $(\mathcal{I})$ cierta. Puesto que $(\mathcal{I})\ \Rightarrow\ (\mathcal{L})$, también podemos considerar $(\mathcal{L})$ cierta. Ahora probaremos la implicación anterior por inducción sobre as.

- Caso Base. Hay que probar

$$(h\ .\ \mathit{foldl}\ (\otimes)\ w)\ [\,]\ =\ \mathit{foldl}\ (\oplus)\ z\ [\,]$$

Transformamos la parte izquierda:

$$
\begin{array}{ll}
& (h\ .\mathit{foldl}\ (\otimes)\ w)\ [\,] \\
\equiv & \{\text{por definición de } (.)\} \\
& h\ (\mathit{foldl}\ (\otimes)\ w\ [\,]) \\
\equiv & \{\text{por definición de } \mathit{foldl}\} \\
& h\ w
\end{array}
$$

Y ahora la derecha:

$$
\begin{array}{ll}
& \mathit{foldl}\ (\oplus)\ z\ [\,] \\
\equiv & \{\text{por definición de } \mathit{foldl}\} \\
& z \\
\equiv & \{\text{por } (\mathcal{I})\} \\
& h\ w
\end{array}
$$

con lo que queda demostrado el caso base del teorema.

- Paso Inductivo. Tenemos la siguiente hipótesis de inducción:

$$(h\ .\ \mathit{foldl}\ (\otimes)\ w)\ as\ =\ \mathit{foldl}\ (\oplus)\ z\ as$$

y hemos de probar

$$\forall\, a \;\centerdot\; (h \;.\; foldl\ (\otimes)\ w)\ (a : as) \;=\; foldl\ (\oplus)\ z\ (a : as)$$

Transformamos la parte izquierda:

$$\begin{array}{ll} & (h \;.foldl\ (\otimes)\ w)\ (a : as) \\ \equiv & \{\text{por definición de } (.)\} \\ & h\ (foldl\ (\otimes)\ w\ (a : as)) \\ \equiv & \{\text{por definición de } foldl\} \\ & h\ (foldl\ (\otimes)\ (w\ \otimes\ a)\ as) \end{array}$$

Y ahora la derecha:

$$\begin{array}{ll} & foldl\ (\oplus)\ z\ (a : as) \\ \equiv & \{\text{por definición de } foldl\} \\ & foldl\ (\oplus)\ (z\ \oplus\ a)\ as \\ \equiv & \{\text{por } (\mathcal{I})\} \\ & foldl\ (\oplus)\ ((h\ w)\ \oplus\ a)\ as \\ \equiv & \{\text{por } (\mathcal{L})\} \\ & h\ (foldl\ (\otimes)\ (w\ \otimes\ a)\ as) \end{array}$$

con lo que queda demostrado el paso inductivo y con ello el teorema de fusión.

Podemos ilustrar el teorema con la siguiente transformación

$$\begin{array}{ll} & (h\ .\ foldl\ (\otimes)\ w)\ [a_1, a_2, a_3] \\ \equiv & \{\text{por definición de } (.)\} \\ & h\ (foldl\ (\otimes)\ w\ [a_1, a_2, a_3]) \\ \equiv & \{\text{por definición de } foldl\} \\ & h\ (((w\ \otimes\ a_1)\ \otimes\ a_2)\ \otimes\ a_3) \\ \equiv & \{\text{ya que } h\ (x\ \otimes\ y)\ =\ (h\ x)\ \oplus\ y\} \\ & h\ ((w\ \otimes\ a_1)\ \otimes\ a_2)\ \oplus\ a_3 \\ \equiv & \{\text{ya que } h\ (x\ \otimes\ y)\ =\ (h\ x)\ \oplus\ y\} \\ & (h\ (w\ \otimes\ a_1)\ \oplus\ a_2)\ \oplus\ a_3 \\ \equiv & \{\text{ya que } h\ (x\ \otimes\ y)\ =\ (h\ x)\ \oplus\ y\} \\ & ((h\ w\ \oplus\ a_1)\ \oplus\ a_2)\ \oplus\ a_3 \\ \equiv & \{\text{ya que } h\ w\ =\ z\} \\ & ((z\ \oplus\ a_1)\ \oplus\ a_2)\ \oplus\ a_3 \\ \equiv & \{\text{por definición de } foldl\} \\ & foldl\ (\oplus)\ z\ [a_1, a_2, a_3] \end{array}$$

Gráficamente (ver Figura 7), el bloque constituido por la función h tras el operador $(\otimes)$ puede ser sustituido por un bloque con la salida de la función h como primer argumento del operador $(\oplus)$. De este modo la función h asciende a través del plegado hasta alcanzar el valor base w.

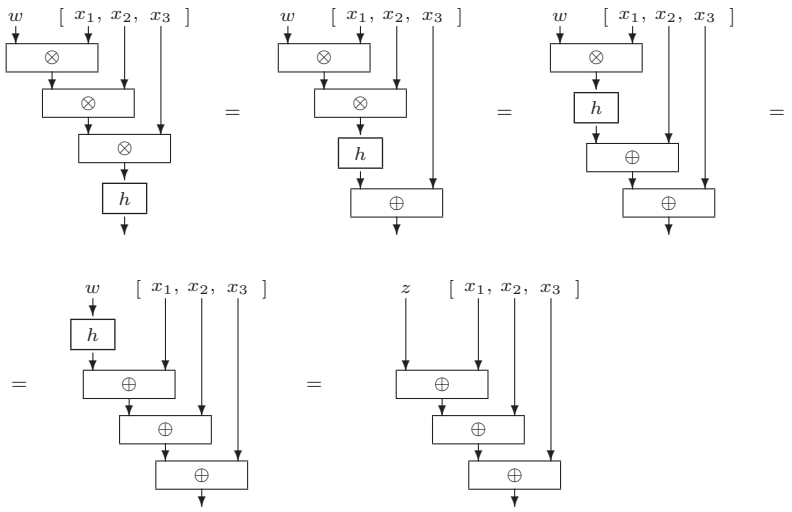

Figura 7. El teorema de fusión de $foldl$

5 Relación entre foldr y foldl

Existen una serie importante de leyes que relacionan *foldr* con *foldl*. Éstas se conocen como los teoremas de dualidad y los exponemos a continuación. El lector interesado en las demostraciones de estos teoremas puede consultar (Bird 98).

5.1 Primer Teorema de dualidad

Si $(\oplus)$ es asociativo y z conmuta con $(\oplus)$ entonces los plegados a la izquierda y derecha computan el mismo resultado para cualquier lista finita xs.

Primer Teorema de dualidad

$$\left.\begin{array}{c} \forall a, b, c \ . \ (a \oplus b) \oplus c = a \oplus (b \oplus c) \\ \forall a \ . \ z \oplus a = a \oplus z \end{array}\right\} \Rightarrow foldr\ (\oplus)\ z\ xs \ = \ foldl\ (\oplus)\ z\ xs$$

Por lo que la funciones

$$\begin{array}{l} sumar\ ::\ [Integer] \ \rightarrow \ Integer \\ sumar\ = foldr\ (+)\ 0 \end{array}$$

$$\begin{array}{l} multiplicar\ ::\ [Integer] \ \rightarrow \ Integer \\ multiplicar\ = foldr\ (*)\ 1 \end{array}$$

también pueden ser definidas como

$$\begin{array}{l} sumar'\ ::\ [Integer] \ \rightarrow \ Integer \\ sumar'\ = foldl\ (+)\ 0 \end{array}$$

$$\begin{aligned} &multiplicar' :: [Integer] \to Integer \\ &multiplicar' = foldl\ (*)\ 1 \end{aligned}$$

ya que los argumentos cumplen las condiciones del teorema.

Ejercicio 2. ¿Se puede usar este teorema para demostrar la equivalencia de las siguientes definiciones?

$$\begin{aligned} &concat :: [[a]] \to [a] \\ &concat = foldr\ (+\!\!+)\ [\,] \end{aligned}$$

$$\begin{aligned} &concat' :: [[a]] \to [a] \\ &concat' = foldl\ (+\!\!+)\ [\,] \end{aligned}$$

En caso afirmativo, ¿cuál de las dos definiciones es más eficiente? □

El teorema puede ser justificado del siguiente modo. Consideremos los siguientes plegados

$$\begin{aligned} &foldr\ (\oplus)\ z\ [x_1,\ x_2,\ x_3] = x_1 \oplus (x_2 \oplus (x_3 \oplus z)) \\ &foldl\ (\oplus)\ z\ [x_1,\ x_2,\ x_3] = ((z \oplus x_1) \oplus x_2) \oplus x_3 \end{aligned}$$

La primera expresión puede ser transformada del siguiente modo si se cumplen las condiciones del teorema:

$$\begin{aligned} & x_1 \oplus (x_2 \oplus (x_3 \oplus z)) \\ \equiv\ & \{z \text{ conmuta}\} \\ & x_1 \oplus (x_2 \oplus (z \oplus x_3)) \\ \equiv\ & \{(\oplus) \text{ es asociativo}\} \\ & x_1 \oplus ((x_2 \oplus z) \oplus x_3) \\ \equiv\ & \{z \text{ conmuta}\} \\ & x_1 \oplus ((z \oplus x_2) \oplus x_3) \\ \equiv\ & \{(\oplus) \text{ es asociativo}\} \\ & (x_1 \oplus (z \oplus x_2)) \oplus x_3 \\ \equiv\ & \{(\oplus) \text{ es asociativo}\} \\ & ((x_1 \oplus z) \oplus x_2) \oplus x_3 \\ \equiv\ & \{z \text{ conmuta}\} \\ & ((z \oplus x_1) \oplus x_2) \oplus x_3 \end{aligned}$$

y esta expresión es el resultado del plegado a la izquierda, por lo que ambos plegados computan el mismo valor.

5.2 Segundo Teorema de dualidad

Este teorema es una generalización del anterior. Si los operadores $(\oplus)$ y $(\otimes)$ y el valor z son tales que para todo a, b y c se cumple que

$$\begin{aligned} a \oplus (b \otimes c) &= (a \oplus b) \otimes c \\ a \oplus z &= z \otimes a \end{aligned}$$

es decir, si $(\oplus)$ y $(\otimes)$ asocian el uno con el otro, y z a la derecha de $(\oplus)$ es equivalente a z a la izquierda de $(\otimes)$, entonces

$$\mathit{foldr}\ (\oplus)\ z\ xs\ =\ \mathit{foldl}\ (\otimes)\ z\ xs$$

para toda lista finita xs. Es decir:

Segundo Teorema de dualidad

$$\left.\begin{array}{c}\forall a, b, c\ \centerdot\ a\ \oplus (b \otimes c) = (a \oplus b) \otimes c \\ \forall a\ \centerdot\ a\ \oplus z = z \otimes a\end{array}\right\} \Rightarrow \mathit{foldr}\ (\oplus)\ z\ xs\ =\ \mathit{foldl}\ (\otimes)\ z\ xs$$

Obsérvese que el primer teorema de dualidad es un caso concreto del segundo cuando $(\oplus) \equiv (\otimes)$.

Como ejemplo de aplicación del teorema, podemos demostrar que las dos definiciones de la función que calcula la longitud de una lista

```
length :: [a] → Int
length = foldr unoMás 0
  where
    unoMás x n = 1 + n
length' :: [a] → Int
length' = foldl másUno 0
  where
    másUno n x = n + 1
```

son equivalentes, ya que $\mathit{unoM\acute{a}s}$, $\mathit{m\acute{a}sUno}$ y 0 verifican las condiciones del teorema.

Ejercicio 3. Justificar el segundo teorema de un modo similar al primero. □

Ejercicio 4. ¿Puede utilizarse el segundo teorema de dualidad para comprobar que las siguientes definiciones computan el mismo resultado?

```
reverse :: [a] → [a]
reverse = foldr alFinal []
  where
    alFinal x xs = xs ++ [x]
reverse' :: [a] → [a]
reverse' = foldl alInicio []
  where
    alInicio xs x = x : xs
```

En caso afirmativo, ¿cuál de las dos definiciones es más eficiente? □

5.3 Tercer Teorema de dualidad

Este teorema establece la siguiente equivalencia para cualquier lista finita xs

Tercer Teorema de dualidad

$$\forall a, b\ \centerdot\ a \otimes b = b \oplus a\ \ \Rightarrow\ \ \mathit{foldr}\ (\oplus)\ z\ xs\ =\ \mathit{foldl}\ (\otimes)\ z\ (\mathit{reverse}\ xs)$$

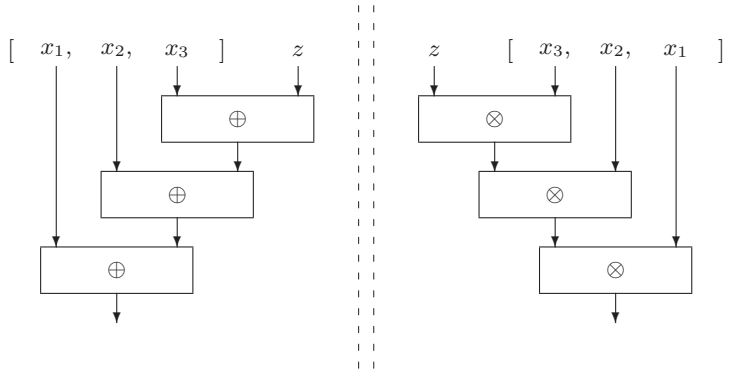

Figura 8. El tercer teorema de dualidad

donde la función *reverse* es aquella que invierte el orden de los elementos de una lista.

El teorema establece que se puede plegar en sentido contrario si previamente se invierte la lista y se utiliza el operador *simétrico* al original.

Como justificación de este teorema basta con ver el plegado a la izquierda como la imagen especular del plegado a la derecha (ver Figura 8).

Ejercicio 5. Utiliza el tercer teorema para demostrar la siguiente propiedad sobre listas finitas

$$\forall\, xs \;::\; [a] \;\boldsymbol{.}\; id\; xs \;=\; reverse\;(reverse\; xs)$$

Ayuda. La función identidad para listas puede ser definida del siguiente modo

$$id \;::\; [a] \;\to\; [a]$$
$$id \;= foldr\;(:)\;[]$$

□

Referencias

Bird, R. (1998). Introduction to Functional Programming using Haskell. Segunda Edición. Prentice Hall.

Dijkstra, E. W. y Sholten, C. (1989). Predicate Calculus and Program Semantics. Springer–Verlag.

Hutton, G. (1999) A tutorial on the universality and expressiveness of fold. Journal of Functional Programming, 9(4):355-372, Cambridge University Press, Julio 1999

Peyton Jones, S. y Hughes, J. (1999). Report on the programming Language Haskell 98. Disponible en `http://haskell.org/report`

Thompson, S. (1999). Haskell. The Craft of Functional Programming. Segunda Edición. Addison Wesley.