La familia Fold

Franco Bulgarelli
0.2

- Sobre este apunte
- Foldl, foldr, foldl1, foldr1, foldl'.... mamaaaaaa!!!
- La familia Fold
- Una primera aproximación
- Asociatividad
- Foldeando la lista vacía
- Algunos foldeos populares
- La metáfora del papel plegado
- Implementando fold recursivamente
- Al infinito y mas allá

Sobre este apunte

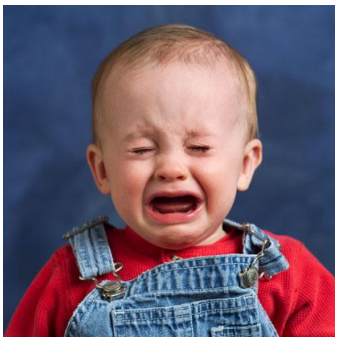
Foldl y sus variantes, son, de entre todas las funciones típicas de orden superior sobre listas, algunas de las más esotéricas. A diferencia de funciones más intuitivas como filter o any, los folds (o “plegados”) nos presentan abstracciones más difíciles de visualizar.

Sin embargo, los folds constituyen una herramienta muy poderosa porque nos propone formas generales de resolver problemas de recorrido de estructuras (por ejemplo, listas) en las que se requiere de un acumulador. Estos básicamente permiten implementar una gran mayoría de las funciones sobre listas del Prelude que el alumno usa en la materia.

Pero aún cuando normalmente preferimos usar en nuestro día a día aquellas funciones más específicas y de más alto nivel, aprender folds nos sirve para desarrollar nuevas formas de pensar soluciones a problemas, que, de otra forma, deberían ser resueltos recursiva o iterativamente.

Este texto pretende ser una presentación detallada de los principales folds sobre listas y abrir el juego para que el lector curioso pueda ir más allá de lo requerido en la materia.

Foldl, foldr, foldl1, foldr1, foldl'.... mamaaaaaa!!!



Todo alumno de paradigmas se topa antes o después con una peculiar función llamada foldl.

Este encuentro puede ser a veces una experiencia traumática. Sobre todo, si el alumno, aunque deseoso de entender qué hacen, faltó a la clase tan importante donde, le contaron, se explicó "fold" ("no es foldl, pero bueno, se parece", piensa él). O peor: quizás estuvo, no entendió y no se animó a preguntar.

El alumno va entonces al intérprete y le pide el tipo a la función: le han dicho sus docentes que los tipos de las funciones hablan y dan pistas. Tipea en GHCi (o Hugs) y, este, indiferente, le responde:

```
> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Observa que hay relaciones entre las variables de tipo que aparecen, pero determina que probablemente ya lo volverán a explicar más adelante. No vale la pena preocuparse hoy.

Las clases avanzan, y el alumno empieza a ver que los docentes y sus compañeros de curso hacen cosas muy extrañas con foldl: calculan factoriales, aplanan listas, producen helados, alimentan pollos.

Para sumar a la confusión, a veces a los docentes se les escapa algún oscuro foldr (o al revés), y (oh!) un foldl1 (y foldr1, respectivamente).

Su compañero de banco para colmo se cebó y se pasa las tardes implementando todas las funciones de listas con folds (hasta implementa foldr con fold!). Hay rumores sobre que un ayudante le dijo que todo se puede implementar con fold, pero ese ayudante ahora estaría preocupado porque ha creado un monstruo y no se anima a merodear por el aula.

Ya desesperado, el alumno busca ayuda en Google, Wikipedia, Stack Overflow, y la Wiki de Haskell.

Allí, extraños internautas que son parte matemáticos (especializados en teoría de categorías, cálculo lambda, y otras cosas así que asustan) y programadores trasnochados Rumanos le dicen que use cosas tan bizarras en internet como foldr' y fold1', porque, afirman, tarde o temprano hay que ser "estricto" si no quiere quedarse sin memoria y el rendimiento importa. (wow, wow woooow..., ¿pero no le había dicho su docente que eso no importaba?)

El alumno, entonces, recordará cuando sus docentes les hablaban de los nombres expresivos y la no importancia de la eficiencia, y murmurará, sumido en la profunda desesperación, una amarga frase “me mintieron.....”

¿Qué puede hacer ahora? Llorar siempre es una opción. Otra mejor, es seguir leyendo :D

La familia Fold

(Una introducción, ahora en serio)

Fold es una familia de funciones de orden superior, que tienen todas un objetivo similar: *combinar* los elementos de ciertas *estructuras* (como por ejemplo, las listas), usando una *operación binaria* (una función de dos parámetros). Como veremos más adelante, algunas variantes toman además un valor inicial de acumulación.

Es una noción muy general, presente en muchos paradigmas y lenguajes, a veces con distintos nombres: acumular, combinar, reducir[1], agregar.

Una primera aproximación

Una primera aproximación gráfica e informal a los folds es la noción de “intercalar” sintácticamente una operación binaria entre los elementos de la lista.

Por ejemplo, supongamos que tenemos la lista [4,8,1]

[4 , 8 , 1]

Si “intercalamos” la operación (+) entre sus elementos, eliminando corchetes y reemplazando las comas por la operación, lo que obtenemos es la siguiente expresión:

4 + 8 + 1

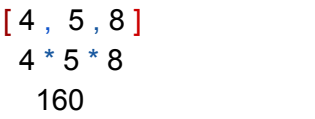
Que reduce a 13.

Con lo cual vemos que “intercalar” la función (+) entre los elementos de una lista es equivalente a calcular su sumatoria.

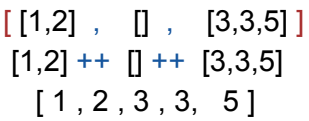
De igual forma, combinar utilizando la función multiplicación nos da la productoria.

[1] Si bien “plegar” es la traducción literal para fold, probablemente reducir sea una traducción más apropiada. Sin embargo, dado que en el paradigma funcional también se dice (beta-) reducir al proceso de obtener el valor de una expresión, en este documento preferiremos el término “combinar”.

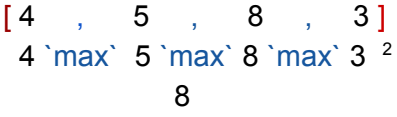
Gráficamente



Y si “intercalamos” el operador (++) entre los elementos de una lista de listas, estamos aplanando la misma:



Como último ejemplo, si tenemos una función que nos da el máximo entre dos elementos (max) y la usamos para los elementos de la lista, obtendremos el máximo de la lista:



El fold más simple que implementa justamente esta idea es foldl1, el cual combina los elementos de una lista dada, usando una función dada:

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

Sabiendo esto, podemos definir varias funciones, por ejemplo:

```
sum = foldl1 (+)
product = foldl1 (*)
concat = foldl1 (++)
maximum = foldl1 max
```

Algunas de estas funciones aún no están terminadas; como veremos más adelante.

Asociatividad

En los ejemplos anteriores se utilizaron siempre funciones que tienen una propiedad muy particular: la suma, la multiplicación, la concatenación y el máximo son operaciones *asociativas*

[2] Aquí estamos utilizando la notación infija de funciones simplemente por cuestiones visuales

tanto a izquierda como a derecha. Por ejemplo:

```
4 + 5 + 6 == ( 4 + 5 ) + 6 == 4 + ( 5 + 6 )
-->>[3]  15   ==   9 + 6   == 4 + 11
```

```
4 `max` 5 `max` 6  ==  (4 `max` 5) `max` 6 ==  4 `max` (5 `max` 6)
-->>              6   ==    5 `max` 6   ==  4 `max` 6
```

Pero, ¿qué ocurre cuando la operación no posee esta propiedad? La división es un buen ejemplo de este tipo de operaciones:

```
 (4 / 2 ) / 2   /=  4  /  (2  /  2)
-->>  2 / 2   /=   4 / 1
```

¿Como asocia foldl1, entonces? De **izquierda** a derecha (de allí la **L de left** de fold**l**1)
¿Y si queremos asociar de **derecha** a izquierda? Allí tenemos fold**r**1, con **R de right**.

Ejemplos:

`foldl1 (+) [10, 2, 3, 4]`
`-->>  ( ( 10 + 2 ) + 3 ) + 4`
`-->>  19`
	`foldr1 (+) [10, 2, 3, 4]`
	`-->>  10 + (2 + (3 + 4))`
	`-->>  19`

`foldl1 (/) [10, 2, 3, 4]`
`-->>  ( ( 10 / 2 ) / 3 ) / 4`
`-->>  0.4166666666666667`
	`foldr1 (/) [10, 2, 3, 4]`
	`-->>  10 / ( 2 / ( 3 / 4 ) )`
	`-->> 3.75`

La primera moraleja es que usar foldr1 y foldl1 no será indistinto cuando la asociatividad importe.

[3] -->> se lee “reduce en uno o más pasos a”

Foldeando la lista vacía

Retomando los ejemplos anteriores, surgen algunas preguntas: siendo que la operación utilizada en la combinación toma dos argumentos, ¿qué sucede cuando la lista tiene un único elemento? ¿Y cuando no tiene ninguno?

Hagamos algunas consultas en el intérprete:

```
Prelude> foldl1 (+) [ 1 ]
1
Prelude> foldl1 max [ 4 ]
4
```

Vemos que la primera pregunta es fácil de contestar: si la lista tiene un solo elemento devuelve el primero de la secuencia.

Ahora, si la lista está vacía, tenemos un problema: `foldl1`/`foldr1` fallan.

```
Prelude> foldr1 (+) []
*** Exception: Prelude.foldr1: empty list
```

Es decir, la lista vacía no es parte del dominio de estas funciones.

Quizás esto no sea problema al definir `maximum`, dado que no tiene sentido hablar del máximo de una lista vacía. Pero sí tiene sentido hablar de la sumatoria de una lista vacía: debería ser 0. Y la productoria de una lista vacía debería ser 1.

Allí entran en juego las funciones `foldl` y `foldr`:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Estas variantes son análogas a `foldl1` y `foldr1`, con la diferencia de que toman un elemento inicial de la acumulación.

Retomando nuestra metáfora gráfica de la intercalación, estos elementos iniciales se colocan, respectivamente, a izquierda o derecha de la lista a intercalar con la operación binaria.

Supongamos que queremos combinar la lista [4, 5, 8] con la multiplicación usando el elemento inicial 1. Si lo hacemos con `foldl`, nos queda:

```
   [ 4 , 5 , 8 ]
 1   [   4 , 5  , 8 ] (introduzco el valor inicial a izquierda)
(((1  *  4) * 5) * 8)
  1  *  4  * 5 * 8
     160
```

Y con `foldr`:

```
   [ 4 , 5 , 8 ]
 [ 4 , 5 , 8 ]    1 (introduzco el valor inicial a derecha)
  (4 * (5 * (8 * 1)))
  4 * 5 * 8 * 1
     160
```

En ambos casos, si intento combinar una lista vacía con cualquier operación binaria y un elemento inicial de acumulación, el resultado será dicho valor inicial.

Por ejemplo, combinando [], con la multiplicación y 1 como valor inicial:

foldl	foldr

`  [ ]`
`1  [ ]`
`  1`
	`  [ ]`
	`[ ]  1`
	`  1`

Con esta nueva abstracción podemos definir sum, product y concat, que ahora sí soportan la lista vacía:

```
sum = foldl (+) 0
product = foldl (*) 1
concat = foldl (++) []
```

Como vemos en estos ejemplos, cuando la operación binaria es cerrada (interna), es decir, es de la forma a -> a -> a, el acumulador será **típicamente** el elemento neutro de la misma.

Sin embargo, nótese que mientras que foldl1/foldr1 tomaban funciones binarias internas, foldl/foldr aceptan también operaciones binarias externas, de las formas `a -> b -> a` y `a -> b -> b`, respectivamente. Como consecuencia, ahora el acumulador no tiene porqué ser del mismo tipo que los elementos de la lista.

Esto nos da mucho más poder: no solamente podemos reducir una lista a un valor final, sino que este valor no tiene porqué ser del tipo del elemento de la lista. Es decir, podemos mapear mientras plegamos. Para mostrarlo, primero hagamos foldr (:) [] [4, 5, 8], que a la larga no termina haciendo nada:

```
[ 4 ,  5 , 8 ]   []
   4 : 5 : 8 : []
   [ 4, 5 , 8 ]
```

Ahora compongamos (:) con una f genérica foldr ((:).f) [] [4, 5, 8]:

```
      [ 4 ,  5 , 8 ]   []
 4 ((:).(f)) 5 ((:).(f)) 8 ((:).(f)) []
      f 4 : f 5 : f 8 : []
      [ f 4, f 5 , f 8 ]
```

[4]

```
map f = foldr consf []
                where consf = (:).f
```

(“Chicken Norris”, 1000) [marcelito, miyagui, arguiñano]
(“Chicken Norris”, 1000) `entrenar` marcelito `entrenar` miyagui `entrenar` arguiñano

Algunos foldeos populares

Dado el poder de los folds, prácticamente todas las funciones de listas del Prelude pueden ser implementadas en términos de estas funciones. Ejemplo:

```
length = foldr incr 0
      where incr _ a =  a + 1
```

Algunos desafíos; desarrollar:

- take (de Data.List)
- csv: toma una lista de strings y devuelve un string con los elementos separados por comas. Por ejemplo csv [“hola”, “hello”] == “hola,hello”
- camelCaseSplit: convierte un string camelCase en un string donde las palabras están separadas por espacios. Por ejemplo camelCaseSplit “camelCaseSplit” = “camel Case Split”
- zip generico: zipear listas de listas
- reverse (de Data.List)
- filter (de Data.List)

[4] Esto no es Haskell válido, a fines ilustrativos. La forma correcta sería: 4 `g` 5 `g` 8 `g` [] where g = (:).f

- zip (de Data.List)

Implementando fold recursivamente

```
foldr f i [] = i
foldr f i (x:xs) = f x (foldr f i xs)


foldl f i [] = i
foldl f i (x:xs) = foldl f (f i x) xs
```

Una característica interesante de esta definición es que es recursiva a la cola: esto significa que la aplicación más exterior que realiza es la llamada recursiva. La definición de foldr, entre tanto, no es recursiva a la cola, dado que la aplicación más exterior es la aplicación de f.

La ventaja típica de las funciones recursivas a la cola es que son fáciles de traducir a iterativas. En el caso particular de Haskell, sin embargo, esto no constituye una ventaja significativa, dado que el motor es lo suficientemente inteligente como para evaluar funciones no recursivas a la cola de forma eficiente.

Al infinito y más allá

(Para qué quiero foldr)

Advertencia: esto va más allá de PdeP.

Foldl no se lleva bien con listas demasiado grandes, potencialmente infinitas. Cuando empleemos una operación que podría terminar o arrojar resultados parciales antes de evaluar toda la lista, foldr es una mejor opción.

Dicho de otra forma, si la operación es lazy a derecha, foldr puede terminar donde foldl no.