Paradigma Funcional

Módulo 5:
Orden superior

por Fernando Dodino
Franco Bulgarelli
Carlos Lombardi
Nicolás Passerini
Daniel Solmirano
Matías Freyre
Versión 2.0
Marzo 2017

Contenido

1 Introducción

Nos piden tres requerimientos:

- necesitamos saber qué clientes nos deben más de $ 10.000
- también qué clientes tienen un nombre palíndromo (capicúa)
- y qué clientes tienen alguna factura de exactamente $ 500.000

Representamos a un cliente con

- el nombre, un String
- la deuda, un número con decimales
- y la facturación, una lista con números (para simplificar el ejemplo)

```
data Cliente = Cliente {
    nombre :: String,
    deuda :: Float,
    facturas :: [Float]
} deriving (Show)
```

Modelamos los clientes en una lista

```
clientes = [
    Cliente "Biasutto" 6000 [4000, 5000],
    Cliente "Colombatti" 15000 [30000],
    Cliente "Marabotto" 200 [500000, 140000]
 ]
```

(atención con el espacio al final del corchete de la última línea, es obligatorio para que Haskell parsee correctamente la definición de clientes)

1.1 Clientes que nos deben más de 10.000

Del capítulo anterior, sabemos que un algoritmo recursivo puede resolver este problema:

```
clientesQueDeben plata [] = []
clientesQueDeben plata (cliente:clientes)
   | ((> plata) . deuda) cliente
                        = cliente:clientesQueDeben plata clientes
   | otherwise          = clientesQueDeben plata clientes
```

```
λ clientesQueDeben 10000 clientes
[Cliente {nombre = "Colombatti", deuda = 15000.0, facturas =
[30000.0]}]
```

1.2 Clientes con nombre palíndromo

Esta definición se parece bastante a la anterior...

```
clientesPalindromos [] = []
clientesPalindromos (cliente:clientes)
   | (palindromo . nombre) cliente
                          = cliente:clientesPalindromos clientes
   | otherwise            = clientesPalindromos clientes

palindromo nombre = nombre == (reverse nombre)
```

Codificarlo ya no es tan simpático.

1.3 Clientes con alguna factura de 500.000

Una vez más, el algoritmo recursivo nos obliga a escribir el caso base y el recursivo:

```
clientesConFacturaDe plata [] = []
clientesConFacturaDe plata (cliente:clientes)
   | ((elem plata) . facturas) cliente
                  = cliente:clientesConFacturaDe plata clientes
   | otherwise    = clientesConFacturaDe plata clientes
```

2 Introducción al orden superior

2.1 Cohesión de los componentes

Las tres funciones tienen más de un objetivo: por un lado recorren la lista separándola en cabeza y cola y por otro se encargan de definir el criterio por el cual me quedo con ciertos elementos. Esto se relaciona con el concepto de **cohesión de los componentes de un sistema,** en el caso del paradigma funcional los componentes se implementan mediante funciones:

- una función es más cohesiva que otra si se enfoca en menos objetivos a la vez.
- al disminuir la cohesión, no solo tengo más responsabilidades para cubrir, sino que es más probable cometer errores: puedo equivocarme en el criterio para filtrar, o puedo equivocarme en el algoritmo que recorre la lista. Por ejemplo, si me olvido el caso base, o lo cambio de esta manera

```
clientesConFacturaDe plata [ _ ] = []
```

ya hay un error en la función, no filtra correctamente el último elemento (y tampoco funciona para una lista vacía). También podría escribir mal el caso recursivo y entrar en loop infinito:

```
clientesConFacturaDe plata (cliente:clientes)
   | ((elem plata) . facturas) cliente
                = cliente:clientesConFacturaDe plata clientes
   | otherwise   = clientesConFacturaDe plata (cliente:clientes)
```

- podemos ver en consecuencia que un componente con menos cohesión es más difícil de testear, ya que requiere más puntos de control.

2.2 Separando lo que se repite

En las tres funciones anteriores se repite el algoritmo: no es exactamente el mismo código pero sí la idea de *filtrar elementos de una lista*. Y lo que es diferente lo marcamos con otro color:

```
clientesQueDeben plata [] = []
clientesQueDeben plata (cliente:clientes)
   | ((> plata) . deuda) cliente
                      = cliente:clientesQueDeben plata clientes
   | otherwise        = clientesQueDeben plata clientes

clientesPalindromos [] = []
clientesPalindromos (cliente:clientes)
   | (palindromo . nombre) cliente
                        = cliente:clientesPalindromos clientes
   | otherwise          = clientesPalindromos clientes

clientesConFacturaDe plata [] = []
clientesConFacturaDe plata (cliente:clientes)
   | (elem plata . facturas) cliente
                     = cliente:clientesConFacturaDe plata clientes
   | otherwise       = clientesConFacturaDe plata clientes
```

Las tres abstracciones que resaltamos con otro color comparten algo en común: son funciones que reciben un Cliente y devuelven un Bool. Como todo valor, la función también podemos pasarla por parámetro, entonces construimos una nueva función filter, más general.

3 Filter: seleccionar

3.1 Definición

```
filter f [] = []
filter f (x:xs) | f x       = x : filter f xs
                | otherwise = filter f xs
```

Para filtrar los que deben más de 10.000 hacemos la consulta desde la consola:

```
λ filter ((> 10000) . deuda) clientes
```

Lo mismo para buscar los clientes con nombres palíndromos:

```
λ filter (palindromo . nombre) clientes
```

Y por último, encontramos los clientes con una factura de 500.000:

```
λ filter (elem 500000 . facturas) clientes
```

¿De qué tipo es filter?

- Recibimos una función que se verifica contra cada uno de los elementos de la lista y devuelve un Bool
- Una lista con elementos de cualquier tipo
- Y devolvemos la lista con los elementos que cumplen el criterio, por lo tanto debe respetar ser del mismo tipo que la lista original

```
filter :: (a -> Bool) -> [a] -> [a]
```

¿Y el Cliente?
Si miramos de nuevo la definición de filter, vemos que no hay nada que indique que la lista es de clientes, por eso decimos que la lista tiene “elementos de cualquier tipo”. Lo que debemos tener en cuenta es que los tipos de la lista (lista de **a**) y la función f (recibe un **a**, devuelve un booleano) coincidan.

3.2 Las funciones como expresiones (o bloques de código)

En capítulos anteriores vimos que las funciones son valores, por lo tanto las podemos usar para modelar estructuras, para aplicarlas total o parcialmente, y **ahora además podemos pasar funciones como parámetro**.

Mientras que las funciones que vimos hasta ahora eran de primer orden, las funciones que reciben o devuelven funciones se llaman de **orden superior**: la ventaja es que permiten construir funciones más generales, recibiendo

funciones que abstraen *porciones de código*. Quien desarrolló el filter no sabía si lo iba a hacer

- para saber qué palabras de un discurso comienzan con 'a'
- para saber qué números de una lista son primos
- para saber qué clientes nos deben más de 10.000 pesos

pero sí codificó filter de una manera lo suficientemente genérica para poder resolver todos esos casos y más, sabiendo que así como paso 4, True ó "ernesto" como valores, también puedo pasar min 4, (== 3), even o (palindromo . nombre). La única diferencia entre ambos grupos de valores es que a min le puedo pasar parámetros para aplicarlo mientras que al número 4 no.

3.3 Composición y orden superior

¿Qué diferencia hay entre composición de funciones y orden superior?
`(palindromo . nombre)` vs. `filter even [1..10]`

En el primer caso **se construye una función adhoc a partir de la composición de dos funciones existentes**: esa función recibe un cliente y permite determinar si su nombre es un palíndromo. Ambas son funciones de primer orden, si coinciden el dominio e imagen de ambas funciones puedo componerlas en uno u otro sentido:

```
λ ((* 2) . (+ 1)) 5
12

λ ((+ 1) . (* 2)) 5
11
```

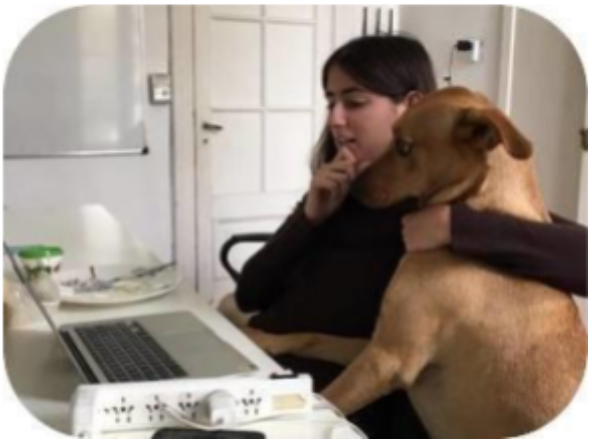
En el segundo caso tenemos una función como valor de primer orden. Pasamos *even* a *filter*, y luego *filter* usa esa función que le paso como parámetro. *Filter* es una función de orden superior (mientras que *even* es de orden simple). Lo potente es que **filter recibe una función que ni siquiera conoce**. Simplemente la usa (delega la responsabilidad a la otra función).

3.4 Ventajas de usar funciones de orden superior

A nivel responsabilidades: si tuviera una consultora, podría decirle a diferentes programadores:

Necesitamos una función que filtre elementos de una lista en base a un criterio que pasen por parámetro

Necesitamos una función para saber si un cliente tiene un nombre que sea palíndromo y otro para saber si un cliente debe más de x $

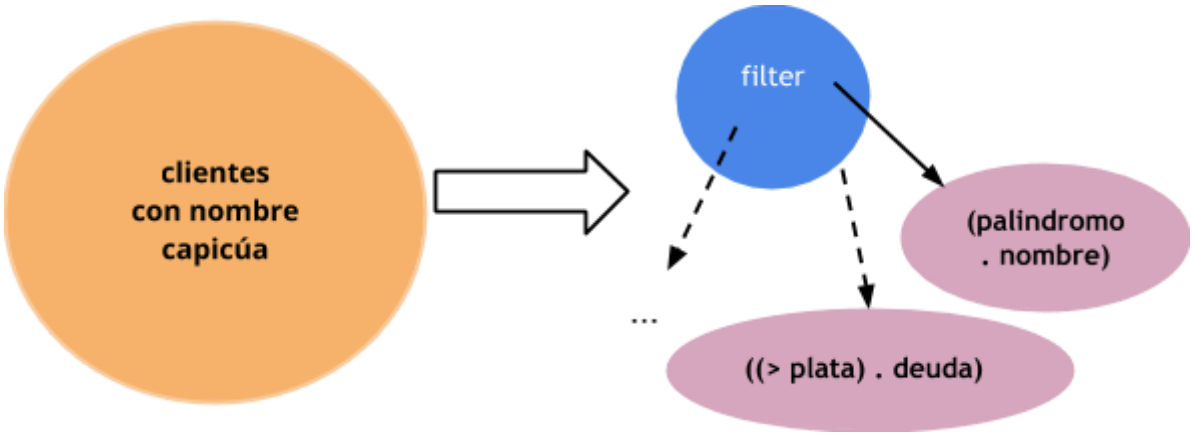


Ésta es la base para dividir trabajo. ¿En qué necesitan ponerse de acuerdo ambas personas? En la interfaz de la función que determina el criterio,
`(a -> Bool)`
y entonces cada una puede trabajar por separado. O incluso puedo pedirle a otros programadores que desarrollen nuevas funciones que filter puede aceptar:

Santi, necesitamos una función que me diga si un jugador de fútbol es un mercenario (cuando jugó en más de 10 clubes), y otra que determine si un actor hizo más de dos películas en un mismo año.

3.5 Volviendo sobre la cohesión y el acoplamiento

Cuando construí la función clientesQueDeben tenía una función que hacía 2 cosas, ahora tengo dos funciones que cumplen un solo objetivo cada una, y son componentes más cohesivos:

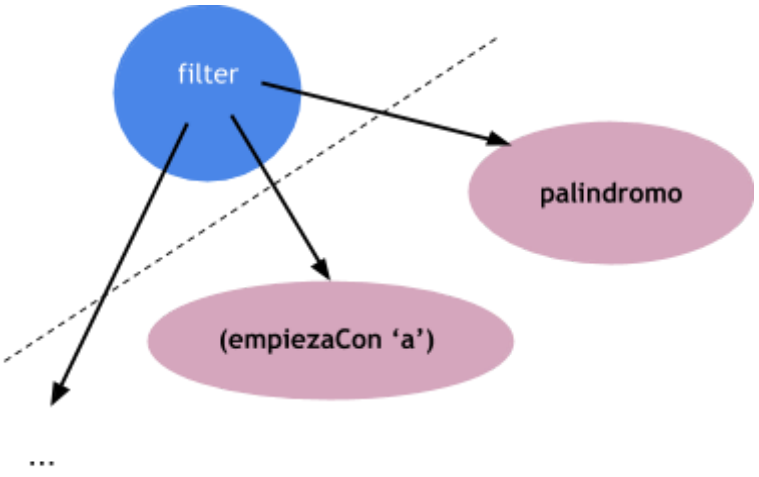


Eso facilita la prueba unitaria: puedo testear *((> plata) . deuda)*[1] en forma independiente de *filter*. Si todo está en la misma función, solo puedo probar la función *clientesQueDeben* en su totalidad.

Por otra parte `(a -> Bool)` implica que hay un **contrato** entre ambos programadores, es el punto de encuentro para coordinar ambas tareas. Supongamos que al hacer esta consulta

```
λ filter palindromo ["neuquen", "salta", "anana"]
["salta"]
```

nos damos cuenta de que hay un error en la función palindromo y hay que corregirla. ¿También debemos modificar *filter*? No, porque filter sólo conoce el tipo de f y la usa, no le interesa cómo está implementada. Hay **acoplamiento** entre ambas funciones, pero sólo el que necesitan ambas funciones para lograr el objetivo.



[1] E incluso deuda y (> plata) se pueden testear por separado

3.6 Filter y la declaratividad

Si repasamos las soluciones posibles para filtrar los palíndromos de un conjunto de palabras:

A) Filter por consola

```
λ filter palindromo ["neuquen", "salta", "anana"]
```

B) Con recursividad

```
palindromos [] = []
palindromos (palabra:palabras)
   | palindromo palabra = palabra : palindromos palabras
   | otherwise          = palindromos palabras
```

C) En pseudocódigo

```
type
   LISTA = array[1..longitudMaxima] of integer
function palindromos(palabras: LISTA) : LISTA
begin
   var
      result : String[];
      i, j : integer;

   result ← [];
   for i ← de 1 a LONGITUD(palabra)
   begin
      if (palabras[i] es palíndromo)
      begin
         result[j] ← palabras[i]
         j ← j + 1
      end
   end
   ↑ result
end
```

La solución con filter es más declarativa que la recursiva:

- en la solución recursiva se que filtro los elementos separando en cabeza y cola
- debo pensar en los casos base y recursivo
- eso no ocurre en la solución con filter, tengo menor control sobre el algoritmo: no sé en qué orden se filtran los elementos... pero por otro lado, no me interesa

A su vez, la solución recursiva es más declarativa que la solución en pseudocódigo, donde el algoritmo:

- debe inicializar la lista original y la resultante
- utiliza variables de iteración por separado (*i*, *j*), recordando sumar en el cada caso según corresponda
- sabe que la lista original tiene longitud fija y se accede por índice numérico
- filtra los palíndromos
- y por último devuelve la lista resultante al final del for: podríamos devolver result mientras recorremos la lista, o incluso antes, por eso claramente la solución en pseudocódigo es más procedimental que la solución recursiva.

A continuación veremos las funciones de orden superior más comunes

4 Map: transformación

Transforma una lista en otra aplicando una función a todos sus elementos.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

¿De qué tipo es map?

- Recibe una función que transforma un elemento en otro que puede ser del mismo tipo o no
    - por eso utilizamos una nueva variable de tipo, *b* puede coincidir con *a* o ser un tipo distinto
- la lista original de *a*s
- y la lista resultante es de *b*s

```
map :: (a -> b) -> [a] -> [b]
```

Algunos usos: duplicar el valor de una lista de números

```
λ map (* 2) [1..5]
[2,4,6,8,10]
```

Convertir una cadena de caracteres a mayúsculas:

```
upperCase palabra = map toUpper palabra
```

4.1 Ejemplos de map heterogéneos

Devolver la longitud (en caracteres) de una lista de palabras

```
sumarPalabras ["paradigmas", "rules", "the", "world"]
```

Lo podemos pensar como

- contar las letras de cada palabra (y utilizarlo en el contexto de map)
- sumar las longitudes

```
sumarPalabras palabras = (sum . map length) palabras
sumarPalabras palabras = (sum . map length) palabras
sumarPalabras = sum . map length
```

4.2 Construyendo funciones de orden superior

4.2.1 con map

¿Qué devuelve?

```
λ map (+) [1, 2, 3]
```

Una lista de funciones

```
[ (1 +), (2 +), (3 +)]
```

Solo que debemos tener importada la definición de show para funciones mediante
import Text.Show.Functions para ver

```
[ <function>, <function>, <function>]
```

4.2.2 sin map

Otra forma de construir una función puede ser:

```
sumarDesde n = (n +):sumarDesde (n + 1)
```

¿De qué tipo es?

- Recibo un número
- Devuelvo una lista de funciones que van de un número a otro

```
sumarDesde :: Int -> [Int -> Int]
```

Veamos un ejemplo:

```
λ (head . sumarDesde 3) 2
5
```

Intervienen aquí todos estos conceptos

- sumarDesde es una función de orden superior
- que devuelve una lista infinita, que se evalúa en forma diferida
- se compone con head
- y así logra converger a un valor (+ 3) que es una función aplicada parcialmente

5 all / any: algunos o todos...

Resolver una función que indique si todos los elementos de una lista cumplen una determinada condición:

```
λ all even [1..3]
False
```

Resolver la función any, que indique si alguno de los elementos de una lista cumplen una determinada condición.

```
λ any (elem 3) [ [6..9], [2..4] ]
True
```

Ayuda: hay una función

```
and :: [Bool] -> Bool
```

que opera una lista de booleanos aplicándole un and entre sí.

```
λ and [True, True]
True

λ and [True, False, True]
False
```

Y también existe una función or, que aplica un or entre una lista de booleanos (devuelve True si existe algún valor verdadero en esa lista).

Tanto any como all reciben

- una función que evalúan sobre cada elemento de una lista y devuelve un booleano
- una lista de elementos
- y devuelven un valor booleano

```haskell
any  :: (a -> Bool) -> [a] -> Bool
any f xs = (or . map f) xs
any f ~~xs~~ = (or . map f) ~~xs~~
any f    = or . map f
```

any transforma los elementos en una lista de booleanos (que indica si cumplen o no el criterio pasado como parámetro), y luego se compone con or para reducirlo a un único valor booleano.

all funciona en forma análoga con la función and:

```haskell
all  :: (a -> Bool) -> [a] -> Bool
all f = and . map f
```

6 fold: reducción[2]

6.1 Introducción

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[3], agregar.

6.2 Intercalando operaciones binarias

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]

[2] La siguiente es una explicación extractada del apunte La familia fold, de Franco Bulgarelli que también está linkeado en los apuntes de la cátedra

[3] 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".

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. Gráficamente

```
[ 4 , 5 , 8 ]
  4 * 5 * 8
     160
```

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

```
[[1,2] ,   [] ,   [3,3,5] ]
 [1,2] ++  [] ++  [3,3,5]
    [ 1, 2, 3, 3, 5 ]
```

Y por último, 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:

```
[ 4   ,   5   ,   8   ,   3 ]
  4 `max`  5 `max` 8 `max` 3
             8
```

[4]

El fold más simple que implementa justamente esta idea es foldl1, que combina los elementos de una lista en base a una función:

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

Sabiendo esto, podemos definir varias funciones, por ejemplo:

```
sum = foldl1 (+)
product = foldl1 (*)
```

[4] Estamos utilizando aquí la notación infija para funciones, donde

```
λ max 4 5
```

en notación prefija también puede escribirse como

```
λ 4 `max` 5
```

en notación infija (con el apóstrofe o Alt 96)

```
concat = foldl1 (++)
maximum = foldl1 max
```

6.3 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* tanto a izquierda como a derecha. Por ejemplo:

```
  4 + 5 + 6  == ( 4 + 5 ) + 6 == 4 + ( 5 + 6 )
-->>[5]     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 **foldl1**)
¿Y si queremos asociar de derecha a izquierda? Allí tenemos **foldr1**, 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 operación no soporte la asociatividad.

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

6.4 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:

```
λ foldl1 (+) [ 1 ]
1
λ 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.

```
λ 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, también llamado semilla (*seed*).

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[6])
   (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.

6.5 Map en base a foldr

Sin embargo, nótese que mientras que foldl1/foldr1 tomaban funciones binarias internas, foldl/foldr aceptan también operaciones binarias externas, de las formas

[6] Por cierto, ¿qué devuelve `foldr (/) 1 [2]` vs. `foldl (/) 1 [2]`?

`a -> b -> a`
y

`a -> b -> b`,
respectivamente. Como consecuencia, ahora el acumulador no tiene por qué 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 que ser del tipo del elemento de la lista. Es decir, podemos mapear mientras plegamos. Para mostrarlo, primero hagamos

```
λ foldr (:) [] [4, 5, 8]
```

que a la larga no termina haciendo nada:

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

Recordemos que

```
λ 4:[]  -- notación infija
[4]
λ 4:[8]
[4,8]
```

```
λ (:) 4 []  -- notación prefija
[4]
λ (:) 4 [8]
[4,8]
```

Y si repasamos la definición de la *operación binaria externa* que necesita foldr, es `a -> b -> b` , donde en este caso
`a ==> Int`
`b ==> [Int]`
lo que resulta en `[Int]`

Ahora compongamos (:) con una f genérica

```
λ foldr ((:) . f) [] [4, 3, 7]
 [ 4 ,  5 , 7 ]   []
   4 ((:) . f) 3 ((:) . f) 7 ((:) . f) []
   f 4 : f 3 : f 7 :             []
  [ f 4, f 3 , f 7 ]
```

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

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

Veamos un ejemplo concreto

```
λ foldr ((:) . (2 *)) [] [4, 3, 7]
[8, 6, 14]
```

```
 [ 4 ,  3 , 7 ]   []
   4 ((:) . (2 *)) 3 ((:) . (2 *)) 8 ((:) . (2 *)) []
   (2 *) 4 : (2 *) 3 : (2 *) 7 :           []
  [ 8      , 6       , 14 ]
```

Aquí vemos que

```
7 `((:) . (2 *))` []  ==> [14]
```

es equivalente a

```
((:) . (2 *)) 7 []    ==> [14]
```

De nuevo, si foldr espera una operación binaria externa (`a -> b -> b`) la función que construimos con composición repite los tipos:

`a ==> Int`

`b ==> [Int]`

lo que resulta en `[Int]`

Consultamos los tipos a Haskell para mayor claridad:

```
λ :t ((:) . (2 *))
((:) . (2 *)) :: Num a => a -> [a] -> [a]
λ :t ((:) . (2 *)) 8
((:) . (2 *)) 7 :: Num a => [a] -> [a]
```

Otra forma de definir map en base a foldr podría ser:

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

6.6 Foldeando funciones

Un guerrero se representa con un nombre y la fuerza destructora que tiene. Un entrenador mejora las capacidades de un guerrero, lo que podríamos modelar como una función

```
Guerrero -> Guerrero
```

devolviendo el guerrero original pero “entrenado”. Tenemos al guerrero Chicken Norris, y tres entrenadores: Marcelito, Miyagui y Arguiñano. Entrenamos a Chicken Norris con todos los entrenadores posibles:

```haskell
type Guerrero = (String, Int)
type Entrenador = Guerrero -> Guerrero

entrenar :: Entrenador -> Guerrero -> Guerrero
entrenar entrenador guerrero = entrenador guerrero

chickenNorris = ("Chicken Norris", 1000)
marcelito = ... función que entrena a un guerrero ...
miyagui = ... función que entrena a un guerrero ...
arguiñano = ... función que entrena a un guerrero ...
```

Y probamos en la consola

```
λ foldr entrenar chickenNorris [marcelito, miyagui, arguiñano]
```

Lo que termina resultando en...

```
marcelito `entrenar` miyagui `entrenar` arguiñano `entrenar`
("Chicken Norris", 1500)
```

6.7 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:

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

en la función incr el primer parámetro -el elemento de la lista- no nos importa porque solo queremos sumar un elemento más.

Por otra parte, también podemos componer una lista de funciones con el operador `(.)`. En ese caso, fold termina devolviendo una función que puede aplicarse contra un valor:

```
λ foldr (.) (1 +) [(^ 2), (4 *)]
<function>

λ (foldr (.) (1 +) [(^ 2), (4 *)]) 0
16
```

Algunos desafíos para 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 (de Data.List)
- zip generico: zipear listas de listas
- reverse (de Data.List)
- filter (de Data.List)

6.8 foldl vs. foldr[8]

```
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 la definición que hace foldl es que es *recursiva a la cola*: si podemos resolver el valor de la expresión (f i x) y no hay otras operaciones pendientes con posterioridad a la llamada recursiva, Haskell no necesita mantener el estado de la pila.

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

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. En otras tecnologías, tener un algoritmo con recursividad a la cola implica poder procesar una lista de valores grande sin que el tamaño de la pila se desborde (el famoso "Stack Overflow").

La definición de foldr, entre tanto, no es recursiva a la cola, dado que la aplicación más externa es la aplicación de f en donde el segundo parámetro es la llamada recursiva.

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

Necesitamos mantener el estado del juego de variables para que al calcular (foldr f i xs) podamos aplicarla con f x. Pero contamos con la ayuda de Haskell y su mecanismo de evaluación diferida, entonces solo va a reducir las expresiones que necesite.

[8] Aclaración: esta comparación excede el alcance de la materia y es absolutamente BONUS

Por otra parte, 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. Para ilustrar esto que hemos dicho veamos un ejemplo concreto: la función repeat permite construir una lista infinita de un elemento:

```
repeat :: a -> [a]
repeat x = x:repeat x
```

Vemos qué sucede al evaluar en consola cada una de estas expresiones:

```
λ foldr (&&) False (repeat False)
False

λ foldl (&&) False (repeat False)
... nunca termina de evaluar ...
```

El motivo es simple

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

se reemplaza como

```
foldr (&&) False (False:[False, False...]) = (&&) False (foldr f i xs)
```

y Haskell no necesita evaluar el segundo argumento, es claro que la operación and con un valor booleano False devuelve False; no importa el segundo argumento:

```
(&&) :: Bool -> Bool -> Bool
(&&) False _ = False
```

En el caso de foldl...

```
foldl (&&) False (repeat False) =
      foldl (&&) ((&&) False False) [False, False ...] =
      foldl (&&) False [False, False ...] =
      foldl (&&) ((&&) False False) [False, False ...] =
      foldl (&&) False [False, False ...] =
```

Como vemos, resolver la expresión `((&&) False False)` no nos sirve de mucho, ya que el algoritmo de foldl necesita llegar a la lista vacía para resolver el caso base. Por eso la solución no puede converger a ningún valor posible.

7 flip

Otra función de orden superior bastante útil es flip, que permite recibir una función y aplicarla con los parámetros invertidos:

```
flip f y x = f x y
```

El tipo de la función flip es:

```
flip :: (a -> b -> c) -> b -> a -> c
```

Volviendo al ejemplo de los clientes, tenemos una función que nos dice si el total de facturación supera un determinado monto:

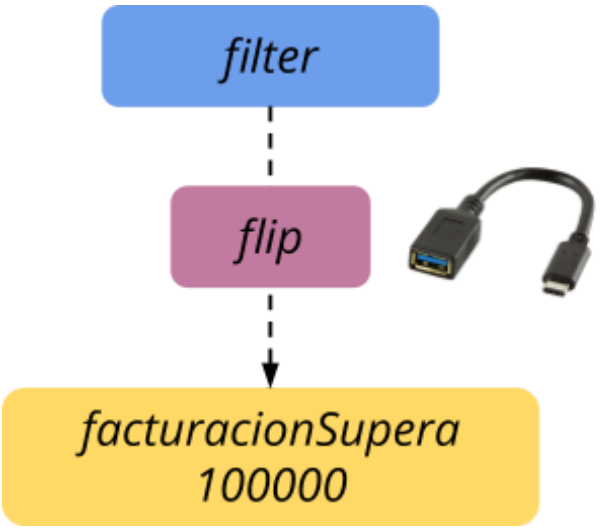
```
facturacionSupera cliente monto =
     ((> monto) . sum . facturas) cliente
```

Esto es inconveniente para poder filtrar los clientes que superan $ 100.000 de facturación:

```
filter (facturacionSupera ???? 100000) clientes
```

No obstante, flip sirve como ***adapter*** para acoplar ambas funciones sin tener que modificar la definición original de facturacionSupera:

```
filter (flip facturacionSupera 100000) clientes
```



8 $ o función aplicación

La función $ es otra función de orden superior que permite aplicar un conjunto de parámetros a una función:

```
($) :: (a -> b) -> (a -> b)
($) f x = f x
```

o lo que es lo mismo

```
f $ x = f x
```

Esta función permite evitar paréntesis, reduciendo

- una función a izquierda
- un valor a derecha (claro, incluida una función)

Entonces esta notación

```
λ (take 1 . filter even) [1..10]
```

se simplifica a

```
λ take 1 . filter even $ [1..10]
```

Y esta expresión

```
λ show (head [1, 2])
```

se puede simplificar a

```
show $ head [1, 2]
```

8.1 Foldr + $

Si la función $ aplica los parámetros a una función, podemos pensar en la composición como un fold que aplica sucesivamente un conjunto de funciones a un valor

```
λ foldr ($) 0 [(^ 6), (2 *), (1 +)]
(^ 6) $ (2 *) $ (1 +) $ 0
(^ 6) $ (2 *) 1
(^ 6) 2 = 2 ^ 6
64
```

Pero la composición de funciones es más general, permite aplicar valores de distinto tipo:

```
λ (even . (1 +)) 0
False
```

mientras que en la solución con fold(*l*/*r*) no podemos tener en una lista funciones que trabajen sobre distintos tipos, porque even devuelve un booleano y no se ajusta al valor 0 que define el tipo del segundo parámetro de la función binaria de reducción:

```
λ foldr ($) 0 [even, (1 +)]

<interactive>:9:11:
    No instance for (Num Bool) arising from the literal ‘0’
    In the second argument of ‘foldr’, namely ‘0’
    In the expression: foldr ($) 0 [even, (+ 1)]
    In an equation for ‘it’: it = foldr ($) 0 [even, (+ 1)]
```

9 Resumen

En el presente capítulo hemos conocido las funciones de orden superior, una herramienta que reafirma el tratamiento de las funciones como valores, al igual que cualquier otro tipo de dato. La consecuencia es que podemos recibir un bloque de código para evaluar una expresión en el momento en que lo necesitemos, delegando el comportamiento a la función que recibimos como parámetro y logrando de esa manera tener abstracciones mucho más generales.