Programación declarativa. Tarea 3
Bringing you into the fold

Juan Alfonso Garduño Solís
Emiliano Galeana Araujo

Facultad de ciencias, UNAM

Fecha de entrega: Jueves 12 de marzo de 2020

1 Propiedades

(a)
```haskell
foldr f e . map g = foldr (f . g) e
```

Lo probaremos usando la ley de fusión.

Recordemos que

```haskell
map g = foldr ((:) . g) []
```

Usando inducción, vemos que cuando tenemos la lista vacía la función `foldr f e []` nos regresa `e` por definición. Por lo que para la lista no vacía tenemos lo siguiente.

```haskell
foldr f e (g (x:xs)) = h x (foldr f e xs)
-- La parte izquierda se reduce a...
f (g x) (foldr f a xs)
```

Por lo que podemos definir `h x y = f (g x) y` con la ley de fusión.

y `h` se puede ver como una composición (`h = f . g`), por lo que tenemos

```haskell
foldr f e . map g = foldr (f . g) e
```

Para cualquier lista finita.

(b)
```haskell
foldl f e xs = foldr (flip f) e (reverse xs)
```

Rescribiremos

```haskell
g = flip f
```

Por lo que provaremos:

```haskell
foldl f e xs = foldr g e (reverse xs)
```

Por inducción para listas finitas.

- Caso base ([ ]):

```
foldl f e [] = e
-- Por definicion de caso base de foldl
```

Por otro lado tenemos:

```
foldr g e (reverse []) = foldr g e []
-- Por definicion de reverse []
foldr g e [] = e
-- Por definicion de caso base de foldr
```

- Hipótesis

```
foldl f e xs = foldr g e (reverse xs)
```

- Paso inductivo
  - Por demostrar

```
foldl f e (x:xs) = foldr g e (reverse (x:xs))
```

```
foldl f e (x:xs) = foldl f (f e x) xs
-- Por definicion de foldl
foldl f (f e x) xs = foldr g (f e x) (reverse xs)
-- Por induccion
```

Por otro lado tenemos:

```
foldr g e (reverse (x:xs)) = foldr g e (reverse xs ++ [x])
-- Por definicion de reverse
foldr g e (reverse xs ++ [x]) = foldr g (foldr g e [x]) (
reverse xs)
-- Por c)
foldr g (foldr g e [x]) (reverse xs) = foldr g (f e x) (reverse
 xs)
```

Como llegamos a lo mismo, afirmamos que se cumple para cualquier lista. Solo recordemos cambiar cambiar a g.

(c)

```
foldr f e (xs ++ ys) = foldr f (foldr f e ys) xs
```

Tomemos en cuenta la siguiente igualdad.

```
foldr f e (xs ++ ys) = foldr f e (foldr (:) xs ys)
```

Recordemos la ley de fusión que nos dice que:

```
f . foldr g a = foldr h b
-- Donde
-- f a = b.
-- f (g x y) = h x (f y)
-- f es estricta.
```

Aplicamos la ley de la siguiente manera:

```
f = foldr f e
g = (:)
h = f -- f es la f del problema.
a = ys
b = foldr f e ys
```

Tomamos las siguientes cadenas de igualdades, basándonos en las definiciones anteiores:

```
f a = foldr f e a = foldr f e ys = b

h x (f y) = f x (foldr f e y) = foldr f e (x:y) = foldr f e ((:) x y)
= foldr f e (g x y) = f (g x y).
```

Con lo anterior, sabemos que la ley de fusión es aplicable. Por lo que al aplicarla, tenemos que

```
foldr f e (xs ++ ys) = foldr f e (foldr (:) ys xs) = foldr f (foldr f
e ys)xs
```

2 Árboles Binarios

Consideramos el siguiente tipo de dato algebráico en Haskell para definir árboles binarios.

```
data Tree a = Void | Node (Tree a) a (Tree a)
```

Y la función `foldT` que define el operador de plegado para la estructura `Tree`, definido como sigue:

```
foldT :: (b -> a -> b -> b) -> b -> Tree a -> b
foldT _ v Void = v
foldT f v (Node t1 r t2) = f t1' r t2'
where t1' = foldT f v t1
t2' = foldT f v t2
```

1. Da en términos de una función `h` el patrón de encapsulado por el operador `foldT`.

   ```
   h Void = v
   h (Node t1 r t2) = f (h t1) r (h t2)
   ```

2. Enuncia y demuestra la propiedad Universal del operador `foldT`, basándose en la Propiedad Universal vista en clase sobre el operador foldr de listas.

   - Propiedad Universal:

   $$\begin{aligned} h\ Void &= v \\ h\ Node\ t_1\ r\ t_2 &= f\ (h\ t_1)\ r\ (h\ t_2) \ \Leftrightarrow\ h\ t = foldT\ f\ F \end{aligned}$$

   - Demostración:

   $\Rightarrow$)

   $$\begin{aligned} h\ Void &= v \\ h\ Node\ t_1\ r\ t_2 &= f\ (h\ t_1)\ r\ (h\ t_2) \ \Rightarrow\ h\ t = foldT\ f\ F \end{aligned}$$

* Caso base:

$$\begin{aligned} h\ Void &= v \\ h\ Void &= foldT\ f\ v\ Void \end{aligned}$$

* Hipótesis de inducción:

$$h\ t = foldT\ f\ v\ t$$

* Paso inductivo:

$$\begin{aligned} h\ Node\ t_1\ r\ t_2 &= foldT\ f\ v\ Node\ t_1\ r\ t_2 \\ f\ (h\ t_1)\ r\ (h\ t_2) &= foldT\ f\ v\ Node\ t_1\ r\ t_2 \end{aligned}$$

Aplicamos la hipotesis de inducción

$$f\ (foldT\ f\ v\ t_1)\ r\ (foldT\ f\ v\ t_2) = foldT\ f\ v\ Node\ t_1\ r\ t_2$$

Por definición de $foldT$

$$foldT\ f\ v\ Node\ t_1\ r\ t_2 = foldT\ f\ v\ Node\ t_1\ r\ t_2$$

3 Función `scanr`

Calcula una definición eficiente para scanr partiendo de la siguiente:

```
scanr f e = map (foldr f e) . tails
```

Recordemos la definición de tails.

```
tails :: [a] -> [[a]]
tails []     = [[]]
tails (x:xs) = (x:xs):tails xs
```

El caso base sería como sigue:

```
-- Definicion de scanr
scanr f e [] = [e]
```

El caso inductivo sería para la lista `(x:xs)`.

```
-- Por la especificacion del problema
scanr f e (x:xs) = map (foldr f e) . (tails (x:xs))
-- Definicion de tails (x:xs)
map (foldr f e) . (tails (x:xs)) = map (foldr f e) . ((x:xs): tails xs)
-- Definicion de map
map (foldr f e) . ((x:xs): tails xs) = foldr f e (x:xs): map (foldr f e) (
 tails xs)
-- Por la especificacion del problema
foldr f e (x:xs): map (foldr f e) (tails xs) = foldr f e (x:xs): scanr f e
  (xs)
-- Definicion de foldr
foldr f e (x:xs): scanr f e (xs) = f x (foldr f e xs): scanr f e xs
-- Sustituimos lo siguiente: foldr f e xs por head (scanr f e xs)
f x (foldr f e xs): scanr f e xs = f x (head ys): ys where ys = scanr f e
 xs
```

Con lo anterior, podemos dar la siguiente definición eficiente para scanr.

```haskell
scanr f e []     = [e]
scanr f e (x:xs) = f x (head ys): ys
                   where ys = scanr f e xs
```

Faltaría ver que la afirmación que hicimos es cierta (foldr f e xs = head (scanr f e xs)). Bosquejaremos la demostración nada más, pues el punto del ejercicio era demostrar lo anterior.

Recordemos lo que hace cada función:

scanr Toma el segundo argumento y el último item de la lista y aplica la función, luego toma el penúltimo item de la lista y el resultado y continúa aplicando. Regresa la lista de resultados finales.

foldr Toma el segundo argumento y el último item de la lsita y aplica la función, luego toma el penúltimo item de la lista y continúa aplicando. Regresa el resultado final.

Sabiendo lo que hacen las funciones podemos ver que mientras `foldr` regresa la aplicación a toda la lista, `scanr` regresa la lista con la función aplicada a cada elemento. Es importante notar que `scanr` crece la lista y aplica de atrás para adelante y `foldr` igual aplica de atrás para adelante. Ambas usan el resultado de la aplicación n en la aplicación $n+1$. Por lo que podemos ver que hacen prácticamente lo mismo. Pero no regresan lo mismo. Entonces aquí entra la función `head` aplicada a `scanr`, la cual saca la cabeza de la lista a la que se le aplicó la función en `scanr`. Este elemento es el resultado de la aplicación a toda la lista, que es lo mismo que hace `foldr`.

Por lo que podemos decir que foldr f e xs = head (scanr f e xs) se cumple.

4 Función `cp`

Considera la siguiente definición de la función `cp` que calcula el producto cartesiano.

```haskell
cp :: [[a]] -> [[a]]
cp = foldr f e
```

1. En la definición anterior, ¿Quiénes son `f` y `e`?

   - `f`:

     ```haskell
     f :: [a] -> [[a]] -> [[a]]
     f [] _ = []
     f (x:xs) l = (map (x:) l) ++ (f xs l)
     ```

   - `e`: `[[]]`

2. Dada la siguiente ecuación

   ```haskell
   length . cp = product . map length
   ```

   en donde length calcula la longitud de una lista y product regresa el resultado de la multiplicación de todos los elementos de una lista. Demuestra que la ecuación es cierta, para esto es necesario reescribir ambos lados de la ecuación como instancias de `orange`foldr y ver que son idénticas.

5 Parte extra

En una granja con mucho folklore se discute acerca del siguiente razonamiento. El día que nace un becerro, cualquiera lo puede cargar con facilidad. Y los becerros no crecen demasiado en un día, entonces si puedes cargar a un becerro un día, lo puedes cargar también al día siguiente, siguiendo con este razonamiento entonces también debería serte posible cargar al becerro el día siguiente y el siguiente y así sucesivamente. Pero después de un año, el becerro se va a convertir en una vaca adulta de 1000kg algo que claramente ya no puedes cargar.

Este es un razonamiento inductivo, la base es el día que el becerro nace, suponemos cierto que se puede cargar en el día n de vida del becerro y si se puede cargar ese día, como no crece mucho en un día entonces también se puede cargar en el día *n+ 1*. Pero claramente la conclusión es falsa.

Para este ejercicio hay dos posibles soluciones, la primera es indicar en donde está el error en el razonamiento inductivo o la segunda es cargar una vaca adulta así demostrando que el argumento es correcto.

En este caso queremos hacer inducción sobre el número de días transcurridos tras el día de nacimiento del becerro con el objetivo de demostrar una propiedad sobre el peso de los becerros: Su peso no es el suficiente para evitar cargarlo. Vamos a ver los números como conjuntos de becerros, de manera que el primer conjunto sería el conjunto que tiene al becerro con un día de nacido, el segundo conjunto sería el conjunto que tiene al becerro con un día de nacido y al becerro con dos días de nacido y así sucesivamente.
De una manera más formal la demostración enunciada sería de la siguiente manera:

- **Caso base:**
  Es posible cargar a todos los becerros que están en el conjunto que solo tiene al becerro con un día de nacido. Se cumple.

- **Hipótesis de inducción:**
  Es posible cargar a todos los becerros que están en el conjunto n-esímo, el cual tiene al becerro con un día de nacido, al que tiene dos, al que tiene tres y así sucesivamente hasta n.

- **Paso inductivo:**
  Sabemos por hipótesis que es posible cargar a todos los becerros del conjunto con n becerros. Para demostrar que se cumple para el conjunto con $n+1$ becerros hacemos lo siguiente:
  Formamos a todos los becerros de menor a mayor respecto al número de días que tienen de nacidos, la fila tiene $n+1$ becerros, si quitamos al que tiene $n+1$ días de nacido nos queda una fila de n becerros y por lo tanto el conjunto que tiene a los becerro de la fila cumple la hipótesis, pero ¿qué hay del becerro que quitamos?, lo podemos devolver a la fila y ahora quitamos por ejemplo el primero, entonces de nuevo tenemos una fila con n a la que podemos aplicar la hipótesis de inducción y concluir que si podemos cargar a todos los becerros del conjunto con n becerros, entonces también podemos cargar a todos los becerros del conjunto con $n+1$ becerros.

La demostración parece correcta, sin embargo hay un agujero en el paso inductivo; de un conjunto con $n+1$ becerros hacemos dos conjuntos, uno sin el último: $\{1, 2, 3, 4, \ldots, n\}$ y otro sin el primero: $\{2, 3, 4, \ldots, n, n+1\}$, el argumento funciona porque aplicamos la hipótesis a los primeros n becerros y a los últimos n becerros, si podemos cargar a los primeros n y también podemos cargar a los n últimos, entonces podemos cargar a los $n+1$ becerros, ¿no?. Pues

no, supongamos $n = 1$, entonces $n + 1 = 2$, por tanto el conjunto sobre el que queremos demostrar es: $\{1, 2\}$, el conjunto con los n primeros becerros es: $\{1\}$ y el de los últimos es $\{2\}$, el argumento ahora sería: si podemos cargar al primer becerro y podemos cargar al último, podemos cargar a todos los becerros del conjunto. Ya no suena tan bien, el problema es que ya no tenemos becerros en la intersección de los primeros n y los últimos n, y aunque la intuición nos dice que sí deberíamos ser capaces de cargar un becerro de dos días de nacido, la inducción no funcióna por intuición, y como este caso rompe la cadena de implicaciones de la inducción, no podemos afirmar nada de los siguientes becerros.
El problema entonces es que no se puede demostrar para $n + 1 = 2$.