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

3 Función `scanr`

Calcula una definición eficiente para scanr partiendo de la siguiente:

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

4 Función `cp`

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

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

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

2. Dada la siguiente ecuación

```
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.