Tarea 2

Programación Declarativa
Facultad de Ciencias, UNAM

Integrantes:

Ailyn Rebollar Pérez

Ángela Janín Ángeles Martínez

Parte teórica

1. Demuestra las siguientes propiedades:

```haskell
sum . map double = double . sum
sum . map sum = sum . concat
sum . sort = sum
```

en donde `double` se define de la siguiente manera:

```haskell
double :: Integer -> Integer
double x = 2 * x
```

y `sum`, `map`, `sort` y `concat` son definidas en el Prelude de Haskell.

La demostración de la propiedad se hará por inducción sobre la lista que se recibe
sum . map double = double . sum

Caso Base:
Consideramos [ ]:
P. D : sum (map double [ ]) = double (sum [ ])
Utilizamos la definición de sum y de double:
sum (map double [ ]) = sum [ ]
sum (map double [ ]) = 0
sum (map double [ ]) = 2 * 0
sum (map double [ ]) = double 0
sum (map double [ ]) = double (sum [ ])

$\therefore$ se cumple el caso base.

Hipótesis de Inducción : sum (map double xs)= double (sum xs)
Paso Inductivo:
P. D. sum . map double (x:xs) = double . sum (x:xs)
Desarrollamos el lado izquierdo a partir de las definiciones:

sum (map double (x:xs)) = sum (double x : map double xs)
= double x + sum (map double xs)
= 2 * x + 2 * (sum xs)
= 2 * (x + (sum xs))
= 2 * (sum (x:xs))
sum (map double (x:xs)) = double (sum (x:xs))

$\therefore$ se cumple el paso inductivo.
$\therefore$ se cumple la propiedad.

La demostración de la propiedad se realizará por inducción sobre la lista que se recibe.
sum . map sum = sum . concat
• Observación : la lista que se recibe debe ser una lista de listas.
Caso Base:
Consideremos [ ]
P.D: sum (map sum [[ ]]) = sum (concat [[ ]])
sum (map sum [[ ]]) = sum [0]
= 0

$= \text{sum } [\,]$
$= \text{sum (concat } [[\,]])$

$\therefore$ se cumple el caso base.

Hipótesis de Inducción: sum (map sum xs) = double (concat xs)
Paso Inductivo:
P.D: sum (map sum (x:xs)) = sum (concat (x:xs))

sum (map sum (x:xs)) = sum (sum x : (map sum xs))
= sum x + sum (map sum xs)
= sum x + sum (concat xs)
= sum (x ++ concat xs)
sum (map sum (x:xs)) = sum (concat (x:xs))

$\therefore$ se cumple el paso inductivo
$\therefore$ se cumple la propiedad.

La demostración de la propiedad se realizará por inducción sobre la lista que se recibe.

sum . sort = sum
Caso Base:
Consideremos [ ]
P.D : sum (sort [ ]) = sum [ ]

sum (sort [ ]) = sum [ ]
= 0
= sum [ ]
sum (sort [ ]) = sum [ ]

$\therefore$ se cumple el caso base.

Hipótesis de Inducción: sum (sort xs) = sum xs
Paso Inductivo:
PD: sum (sort (x:xs)) = sum (x:xs)

Sea sort = mergeSort y (xs ys) = parte (x:xs) funciones implementadas en Haskell y tomadas de la tarea 1.

sum (sort (x:xs)) = sum (mergeSort (x:xs))
= sum (mezcla xs ys)
= sum xs + sum ys
= sum (xs ++ ys)
sum (sort (x:xs)) = sum (x:xs)

$\therefore$ se cumple el paso inductivo,
$\therefore$ se cumple la propiedad.

2. En Haskell, la función `take` n toma los primeros n elementos de una lista, mientras que `drop` n regresa la lista sin los primeros n elementos de ésta. Demuestra o da un contraejemplo de cada una de las siguientes propiedades.

```
take n xs ++ drop n xs = xs
take m . take n = take ( min m n )
map f . take n = take n . map f
filter p . concat = concat . map ( filter p )
```

Para demostrar la propiedad se hará inducción sobre n.
take n xs ++ drop n xs = xs

Caso Base:
Consideramos $n = 0$
take 0 (x:xs) ++ drop 0 (x:xs) = [ ] ++ (x:xs) = (x:xs)

$\therefore$ se cumple el caso base.

Paso Inductivo:
Hipótesis de Inducción: take m (x:xs) ++ drop m (x:xs) = (x:xs)
Consideramos $n = m + 1$
take m+1 (x:xs) ++ drop m+1 (x:xs)
Usamos la definición de take. drop y append:

x:(take m xs) ++ drop m xs = x: (take m xs ++ drop m xs)
Por hipótesis de inducción, entonces:
x:(take m xs ++ drop m xs) = (x:xs)

$\therefore$ se cumple la propiedad en el paso inductivo.

take m . take n = take (min m n)

Para demostrar la propiedad, se hará inducción sobre la lista.
Caso Base:
Realizamos el análisis de la expresión (**min** m n)
Supongamos, entonces que (min m n) = n. Además, n = 0.
take m . take 0 xs = take 0 xs
Por definición de take, tenemos que:
take m . take 0 xs = [ ]
take m [ ] = [ ]
[ ] = [ ]

$\therefore$ se cumple el caso base.

Paso Inductivo:

Consideremos la siguiente definición:
take m xs = xs si y sólo si m >lenght xs

Hipótesis de Inducción:
take m . take n (x:xs) = take (min n m) (x:xs)
Realizando el análisis de casos para la expresión **min**, entonces:
Consideramos (**min** n m) = n
take m . take n (x:xs) = take n (x:xs)
Como n <m, y consideramos a n como la longitud de la lista dada como parámetro a take m, entonces por la definición dada arriba:
take m xs = xs
en específico
xs = take n (x:xs)

Por hipótesis de Inducción, entonces:
take m . take n (x:xs) = take n (x:xs)

$\therefore$ se cumple la propiedad en el paso inductivo.

Consideremos, ahora, la expresión (**min** m n) = m
take m . take n (x:xs) = take m (x:xs)

La propiedad se demostrará por inducción sobre la lista que se recibe.
map f . take n = take n . map f
Caso Base: Consideremos [ ]
P.D = map f (take n [ ]) = take n (map f [ ])
map f (take n [ ]) = map f [ ]
= [ ]
= map f [ ]
= take n (map f [ ])
Hipótesis de Inducción: map f (take m xs) = take m (map f xs)
Paso Inductivo:
PD: map f (take m+1 (x:xs)) = take m+1 (map f (x:xs))
Hacemos un análisis de casos de n:
- n = 0

map f (take 0 (x:xs)) = map f (x:xs)
= f x : map f xs
= take 0 (f x : map f xs)
= take 0 (map f (x:xs))
- n = m+1

P.D: map f (take m+1 (x:xs)) = take m+1 (map f (x:xs))
map f (take m+1 (x:xs)) = map f (x : take m xs)
= f x : map f (take m xs)
= f x : take m (map f xs)
= take m+1 (map f (x:xs))

$\therefore$ se cumple el paso inductivo.
$\therefore$ se cumple la propiedad.

La propiedad se demuestra por inducción sobre lista
filter p . concat = concat . map (filter p)

Caso Base:
Consideremos [ ]
PD: filter p (concat [[ ]]) = concat (map (filter p) [[ ]])
filter p (concat [[ ]]) = filter p [ ]
= [ ]
= filter p [ ]
= concat (map (filter p) [[ ]])

$\therefore$ se cumple el caso base.

Hipótesis de Inducción: filter p (concat xs) = concat (map (filter p) xs)
Paso Inductivo:
PD: filter p (concat (x:xs)) = concat (map (filter p) (x:xs))
filter p (concat (x:xs)) = filter p (x ++ concat xs)
= filter p x ++ filter p (concat xs)
= filter p x ++ concat (map (filter p) xs)
= concat(filter p x : map (filter p) xs)
= concat (map (filter p) (x:xs))

$\therefore$ se cumple el paso inductivo.
$\therefore$ se cumple la propiedad.

3. Consideremos la siguiente afirmación:

map (f . g) xs = **map** f $ **map** g xs

   a) ¿Se cumple para cualquier `xs`? Si es cierta, bosqueja la demostración. En caso contrario ¿qué condiciones se deben pedir sobre `xs` para que sea cierta?
   R = Sí, es cierto y un bosquejo de la demostración sería realizar inducción sobre xs desarrollando ambos lados de la igualdad.

b) Intuitivamente ¿qué lado de la igualdad resulta más eficiente? ¿Esto es cierto incluso en lenguajes con evaluación perezosa? Justifica ambas respuestas.

R= El lado izquierdo es el más eficiente de la igualdad debido a que la función map se realiza una vez sobre xs aplicando la función compuesta (f . g). También resulta ser el lado izquierdo más eficiente en lenguajes con evaluación perezosa porque a pesar de que no se evalúen al momento las funciones, la eficiencia sigue manteniendo por las veces que se ejecuta map o se recorre xs.