Tarea 3

Evaluación perezosa y eficiencia de programas

Ejercicio 1 (Listas infinitas) **9 Pt**

Sea $r \geq 0$. Si para alguna semilla $a_0 \geq 0$, la secuencia

$$a_n = \frac{1}{2} \cdot \left(a_{n-1} + \frac{r}{a_{n-1}} \right)$$

converge a algún límite $a < \infty$, se puede probar fácilmente que $a = \sqrt{r}$. Usando dicha propiedad, vamos a definir un algoritmo numérico para aproximar la raíz de un número. Para ello se pide que:

(a) [4 Pt] Defina la función `apps :: Double -> Double -> [Double]` que crea una lista de aproximaciones sucesivas de r a partir de una semilla a_0 dada. Concretamente, `apps a0 r` $= [a_0, a_1, a_2, \ldots]$.

(b) [4 Pt] Defina la función `approxLimit :: Double -> [Double] -> Double` que dado un ϵ y una lista infinita $[x_0, x_1, x_2, \ldots]$, devuelve el primer elemento de la lista que esté a una distancia menor o igual a ϵ del elemento anterior.

(c) [1 Pt] Utilizando las funciones definidas en los items (a) y (b), defina la función `approxSqrt :: Double -> Double -> Double -> Double` que aproxime la raíz cuadrada de un número a partir de una semilla dada, con una precisión dada. Por ejemplo, para aproximar la raíz de 2, a partir de la semilla 1, con una precisión 0.01, hacemos

```
> approxSqrt 2 1 0.01
> 1.4142156862745097
```

Ejercicio 2 (Árboles infinitos) **6 Pt**

El objetivo de este ejercicio es defir el árbol infinito de la Figura 1. Para ello vamos a considerar la siguiente declaración de árboles:

```
data Tree a = Node a [Tree a]
```

(a) [5 Pt] Defina la función `itTree :: (a -> [a]) -> a -> Tree a` tal que `itTree f root` genera un árbol de raíz `root`, donde los hijos de cada nodo se obtienen aplicando `f` al valor del nodo, como se ilustra en la Figura 2.

(b) [1 Pt] Utilizando la función `itTree`, defina el árbol infinito `infTree :: Tree Integer` de la Figura 1, de raíz 0, y donde los hijos de un nodo de valor n son $n+1, n+2, n+3, \ldots$.

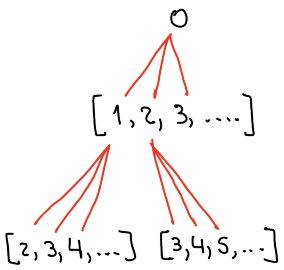

Figura 1: infTree

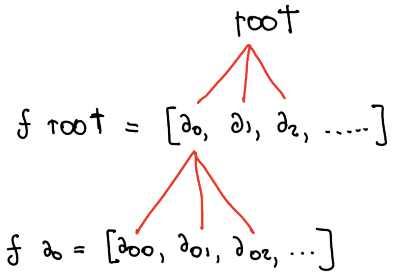

Figura 2: itTree f root

Ejercicio 3 (Estrategias de evaluación) **13 Pt**

Considere el siguiente par de funciones:

```
f :: [Int] -> (Int,Int) -> (Int,Int)
f []     c = c
f (x:xs) c = f xs (step x c)

step :: Int -> (Int,Int) -> (Int,Int)
step n (c0, c1) | even n    = (c0, c1 + 1)
                | otherwise = (c0 + 1, c1)
```

Dé la secuencia de reducción de la expresión `f [1,2,3,4] (0,1)`

(a) [6 Pt] usando la estrategia *call-by-value*, y

(b) [7 Pt] usando la estrategia *lazy* que ocupa Haskell.

Ejercicio 4 (Space leaks) **13 Pt**

(a) [1.5 Pt] De acuerdo a lo observado en el Ejercicio 3.b, ¿presenta la expresión `f [1,2,3,4] (0,1)` un *space leak*? Justifique su respuesta.

(b) [8 Pt] En caso que su respuesta haya sido afirmativa, adaptar la definición de `f` y/o `step` para que la reducción de una aplicación a `f` (como `f [1,2,3,4] (0,1)`) requiera el menor espacio posible.

(c) [1.5 Pt] Dadas las definiciones de abajo, ¿presenta la expresión `length [1..10]` un *space leak*? Justifique su respuesta.

```
length = length2 0

length2 n []     = n
length2 n (x:xs) = if n==0 then length2 1 xs else length2 (
   n+1) xs
```

(d) [2 Pt] En caso que su respuesta haya sido afirmativa, adaptar la definición de `length` y/o `length2` para evitar el *space leak*.

Ejercicio 5 (Síntesis de programas) **13 Pt**

Considere la siguiente función:

```
partMerge :: (a -> Bool) -> ([a], [a]) -> ([a], [a])
partMerge p (xs,ys) = (filter p xs ++ filter (not . p) ys,
                       filter (not . p) xs ++ filter p ys)
```

(a) [1.5 Pt] ¿Qué inconveniente le encuentra a dicha definición, en términos de eficiencia, concretamente en tiempo de reducción?

Para soluciones dicho inconveniente se propone redefinir la función en término de la función auxiliar `partcat` que satisface la siguiente especificación:

```
partcat p xs (us,vs) = (filter p xs ++ us, filter (not . p) xs ++
    vs)
```

y recorre `xs` una sola vez.

(b) [8 Pt] Derive formalmente (como se vio en clase) otra definición de `partMerge`, puramente en términos de `partcat`.

(c) [2 Pt] Proponga una definición de `partcat` en términos de `foldl` que recorra la lista `xs` una sola vez.

(d) [1.5 Pt] Si en la definición que propuso en el item anterior cambiamos `foldl` por `foldl'`, ¿obtenemos algún beneficio de eficiencia en términos de espacio o tiempo?

Ejercicio 6 (Controlando el tiempo) **6 Pt**

Considere la siguiente definición:

```
h :: Int -> Int
h 0 = 2
h 1 = 3
h 2 = 4
h n = 1 + 3 * h (n-1) + 3 * h(n-3)
```

(a) [1 Pt] ¿Qué inconveniente le encuentra a dicha definición, en términos de eficiencia, concretamente en tiempo de reducción?

(b) [5 Pt] Reemplace dicha definición por otra equivalente, que sea lo más optima posible en términos de tiempo de ejecución.