Auxiliar 6

Estrategias de evaluación

Profesor: Federico Olmedo
Auxiliar: Damián Árquez

Estrategias de evaluación

1. La siguiente es una solución al problema visto en la Tarea 1, en que se debía implementar una versión eficiente de una función exponente.

Código 1: effPow

```
effPow :: Num a => a -> Int -> a
effPow b 0 = 1
effPow b 1 = b
effPow b n | n < 0      = error "exponente negativo"
           | even n     = m * m
           | otherwise = b * m * m
  where
    m = effPow b (n `div` 2)

```

Escriba, paso a paso, la reducción de la expresión `effPow 2 5` utilizando las estrategias `call-by-value` y *lazy evaluation* de haskell (con grafos). **Nota**: En este ejercicio optimizamos explícitamente el calculo de m. Para la versión call-by-value haga de cuenta que dicha optimización no existe, es decir, que m se calcula dos veces en cada evaluación.

2. Las siguientes son las definiciones de las funciones `zip` e `iterate` disponibles en el preludio de Haskell.

Código 2: effPow

```
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _      _      = []

iterate :: (a -> a) -> a -> [a]
iterate f a = a : iterate f (f a)

```

Escriba, paso a paso, la reducción de la siguiente expresión: `zip [1..3] (iterate (+1) 0)`. Utilice las reglas de evaluación de Haskell.

Criba de Eratóstenes

La criba de Eratóstenes es un algoritmo que permite encontrar todos los números primos menores a un n determinado. El algoritmo funciona de la siguiente manera:

1. Se escribe una lista con todos lo naturales entre 2 y n.
2. Se marca el primer número que no esté marcado ni tachado.
3. Se tachan todos los números multiplos del número que se acaba de marcar.
4. Se vuelve al paso 2 hasta que no queden números sin marcar o tachar. [1]

Puede ver un GIF del algoritmo en el siguiente link[2].

Utilice el comportamiento lazy de Haskell para definir una lista infinita que contenga todos los primos.

Puzzle

Se le entregan dos números naturales: `x0` y `xf`. Tiene a su disposicisión dos funciones:

- `f(x) = 2x + 1`
- `g(x) = 3x + 1`

Queremos encontrar el mínimo número de aplicaciones de `f` y `g` a x0 para llegar a xf, o en su defecto, determinar que no es posible llegar a xf.
Por ejemplo,

- Se puede llegar de 2 a 7 con una aplicación: `xf = 7 = 3 * 2 + 1 = g(2) = g(x0)`.
- Se puede llegar de 2 a 15 con dos aplicaciones: `xf = 7 = 2 * (7) + 1 = 2 * (3 * 2 + 1) + 1 = 2 * g(2) + 1 = f(g(2)) = f(g(x0))`.
- No se puede llegar de 2 a 8.

Para resolver este problema pensaremos en el puzzle como un árbol binario *infinito*. La raiz es x0, cada vez que bajamos por la izquierda aplicamos `f()` y cada vez que bajamos por la derecha aplicamos `g()`. Note que al razonar de esta manera, al encontrar xf, la cantidad de aplicaciones equivale a la profundidad del nodo en el que se encuentra, es decir, si la profundidad del nodo con valor xf es d, entonces hay que realizar d aplicaciones de f o g para llegar a xf.

[1] En realidad se puede optimizar para parar antes
[2] https://commons.wikimedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif

1. Note que el razonamiento anterior funciona cuando xf se encuentra en un nodo. Sin embargo, dado que estamos tratando con una estructura infinita, debemos saber cuando parar. Encuentre una condición sobre la cual podemos estar seguros de que xf no se encuentra en el árbol.

2. Escriba la definición de un nuevo tipo `Tree`, tal que cada nodo almacene la siguiente información: `value`, `depth`, `lChild`, `rChild`.

3. Escriba una función `makeTree` que tome x0 y retorne un árbol infinito como el descrito anteriormente. Note que ahora es el momento de asignar `depth` a cada nodo.

4. Dado que la estructura del arbol tiene una profundidad infinita no podemos utilizar algoritmos como DFS. Por el contrario, en este caso nos sirve más un razonamiente como BFS. Escriba una función `traverseNodes` que tome un árbol y retorne una lista de árboles (o nodos) en el orden en que se recorrería el árbol con BFS. Note que esta lista también será infinita.

5. Finalmente, escriba la función `distance` que dado x0 y xf, retorna algo de tipo `Maybe Int`, es decir, retorna la cantidad de aplicaciones de f y g, o `Nothing` si es que no es posible llegar a xf.