Programación Funcional - Práctico 5

Curso 2017

1. Dada la siguiente definición:

   $$\begin{aligned}&\mathit{replicate}\ 0\ \_ = [\,]\\&\mathit{replicate}\ n\ x = x : \mathit{replicate}\ (n-1)\ x\end{aligned}$$

   Describir las secuencias de reducción en los casos de evaluación perezosa y por valor para las siguientes expresiones:

   (a) $\mathit{head}\ (\mathit{replicate}\ 5\ 1)$

   (b) $\mathit{map}\ (*2)\ (\mathit{replicate}\ 2\ 2)$

   (c) $\mathit{length}\ (\mathit{map}\ (*2)\ (\mathit{replicate}\ 2\ 2))$

2. Hacer lo mismo que en el ejercicio 1, pero considerando la siguiente definición de *replicate*:

   $$\mathit{replicate}\ n\ x = \mathit{take}\ n\ (\mathit{repeat}\ x)$$

3. Dadas las siguientes definiciones:

   $$\begin{aligned}&\mathit{minlist} :: [\mathit{Int}] \to \mathit{Int}\\&\mathit{minlist} = \mathit{head} \circ \mathit{sort}\\&\mathit{sort} \quad = \mathit{foldr}\ \mathit{insert}\ [\,]\\&\mathit{insert}\ x\ [\,] \qquad = [x]\\&\mathit{insert}\ x\ (y : \mathit{ys}) = \mathbf{if}\ x \leqslant y\ \mathbf{then}\ x : y : \mathit{ys}\ \mathbf{else}\ y : \mathit{insert}\ x\ \mathit{ys}\end{aligned}$$

   Describir las secuencias de reducción de la expresión $\mathit{minlist}\ [3,1,2]$ en el caso de evaluación perezosa.

4. Dadas las siguientes definiciones

   $$\begin{aligned}&\mathit{from}\ n = n : \mathit{from}\ (n+1)\\&\mathit{nats} = \mathit{from}\ 0\end{aligned}$$

   ¿Qué producen las siguientes definiciones usando evaluación perezosa?

   (a) $\mathit{lista1} = \mathit{map}\ \mathit{fst}\ (\mathit{zip}\ \mathit{nats}\ (1 : \mathit{lista1}))$

(b) $lista2 = 1 : (map\ fst\ (zip\ nats\ lista2))$

5. Dadas las siguientes definiciones

$$loop = tail\ loop$$
$$a = 1 : b$$
$$b = 2 : a$$

¿Qué resultado tienen las siguientes expresiones usando evaluación perezosa?

(a) $take\ 4\ (zipWith\ (+)\ a\ b)$

(b) **if** $loop \equiv \perp$ **then** 4 **else** 5

(c) $4 * length\ (replicate\ 3\ (head\ loop))$

(d) $6 + head\ (tail\ a)$

(e) $6 + head\ (tail\ loop)$

6. Los números de Hamming forman una sucesión estrictamente creciente (sin repetidos) de números que cumplen las siguientes condiciones:

- El número 1 está en la sucesión
- Si x está en la sucesión, entonces también están $2\ x$, $3\ x$ y $5\ x$
- Ningún otro número está en la sucesión.

(a) Defina la lista infinita $hamming :: [Integer]$ de los números de Hamming.

(b) Defina una función $hammingTo :: Integer \to [Integer]$ que retorne los números de Hamming menores a un número dado.

7. Mantener una casa en orden es un trabajo que nunca tiene fin. Algunas de las tareas que se deben realizar son: limpiar, cocinar, fregar, lavar ropa, ir de compras... y cuando se termina de hacer todo, hay que volver a empezar!

(a) Defina un tipo de datos *Tarea* que modele las posibles tareas de una casa

(b) Defina la lista infinita de *tareas*, que consiste en primero limpiar, después cocinar, después fregar, después lavar ropa, después ir de compras, después volver a limpiar, etc.

(c) Una pareja moderna divide las *tareas* en partes iguales. Defina la función $tareasPareja :: Int \to [Tareas] \to ([Tareas], [Tareas], [Tareas])$ que dado un número n de tareas a realizar y la lista (infinita) de tareas, retorne una tripla donde el primer y segundo componente contienen la división de las n tareas y el tercer componente las tareas que restan por hacer.

(d) Defina la función $planificar :: Int \to Int \to ([\,Tareas\,], [\,Tareas\,])$, que dadas la cantidad de tareas que se deben realizar por día y la cantidad de días, retorna las tareas que debe realizar cada miembro de la pareja en el período.

8. Un juego consiste en largar a una persona con los ojos vendados a caminar por una habitación llena de obstáculos en búsqueda de un objeto. Si se da contra un obstáculo pierde, si llega a encontrar el objeto gana y puede sacarse la venda.

   La configuración en un momento dado del juego se define por el siguiente tipo:

```
data Juego = Juego Int     -- eje x
                   Int     -- eje y
                   Pos     -- jugador
                   [Pos]   -- obstaculos
                   Pos     -- objeto
type Pos = (Int, Int)
```

   Donde un valor de la forma ($Juego\ tamX\ tamY\ jug\ obs\ obj$) indica los tamaños $tamX$ y $tamY$ de los ejes x e y respectivamente, la posición pos del jugador, la lista de posiciones obs de los obstáculos y la posición obj del objeto. Las posiciones se representan como duplas (x, y) que van de 0 a $tamX - 1$ en el eje de las x y de 0 a $tamY - 1$ en el eje de las y.

   (a) Implementar la función *iniciar*, que dados los componentes de la configuración de un juego, retorna un valor de tipo juego si todas las posiciones son correctos con respecto al tamaño de la habitación:

$$iniciar :: Int \to Int \to Pos \to [\,Pos\,] \to Pos \to Maybe\ Juego$$

   (b) Un árbol de juego representa un juego, conteniendo todos los posibles movimientos desde su posición inicial.

```
data ArbolJuego = Fin   Resultado
                | Sigue ArbolJuego    -- adelante
                        ArbolJuego    -- atras
                        ArbolJuego    -- izquierda
                        ArbolJuego    -- derecha
data Resultado = Gana | Pierde
  deriving Show
```

   Los caminos de la raíz hacia las hojas (*Fin*) son posibles jugadas completas con un resultado dado (*Gana* o *Pierde*).

   Los movimientos posibles son:

   - adelante: avanzar en el eje de las x

- atrás: retroceder en el eje de las x
- izquierda: retroceder en el eje de las y
- derecha: avanzar en el eje de las y

Los movimientos que hacen que la persona "se choque contra la pared" lo dejan en el mismo lugar donde estaba (pero se cuentan como un movimiento).

$$\textbf{data}\ Mov = Ade \mid Atr \mid Izq \mid Der$$
$$\quad \textbf{deriving}\ Show$$

Implementar la función *arbol*, que dada una configuración inicial retorna el árbol de juego:

$$arbol :: Juego \rightarrow ArbolJuego$$

(c) Implementar la función *mover*, que dado un movimiento y un árbol de juego, retorna el sub árbol resultante de realizar dicho movimiento (si es posible):

$$mover :: Mov \rightarrow ArbolJuego \rightarrow ArbolJuego$$

(d) Implementar la función *jugar*, que dada una lista que representa a una secuencia de movimientos y un árbol de juego, retorna el resultado de realizar (algún prefijo de) dichos movimientos o *Nothing* si se queda en un estado en el que el juego no ha terminado (no ha ganado ni perdido):

$$jugar :: [Mov] \rightarrow ArbolJuego \rightarrow Maybe\ Resultado$$

(e) Implementar la función *mejorJugada*, que dado un árbol de juego retorna la secuencia de movimientos más corta que resulte en ganar el juego.

$$mejorJugada :: ArbolJuego \rightarrow [Mov]$$