Recursión sobre listas T7

Listas

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir. El tipo de una lista se escribe como `[tipo]`.

Por ejemplo:
`[True, False, False, True, False] :: [Bool]`
`[div 10 5, div 2 2] :: [Int]`
`[[1], [2,3], [], [1, 100, 2]] :: [[Int]]`

Una lista vacía se denota `[]` y puede ser de cualquier tipo.

Algunas operaciones con listas que vienen en el Prelude
`head :: [a] -> a`
`tail :: [a] -> [a]`
`(:) :: a -> [a] -> [a]`

Atajos
`[1..100]` :lista de enteros del 1 al 100.
`[3,1..100]` lista de enteros del 1 al 100, con una distancia de 2 entre ellos (ej: [3,1..20] = [1,3,5,7,9,11,13,15,17,19])
`[1..]`:lista infinita que empieza en 1

Expresión que denota la lista estrictamente decreciente de enteros que comienza con el número 1 y termina con el número -100 : `[1,0..(-100)]`
lista estrictamente creciente de enteros entre −20 y 20 que son congruentes a 1 módulo 4: `[x | x <- [-20..20], x ´mod´ 4 == 1]`

Recursión sobre listas

Función longitud
La función se llama recursivamente sobre la cola de la lista hasta que se llega al caso base de una lista vacía, momento en el que se devuelve el valor 0.

```
longitud :: [Int] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

Función sumatoria
En el caso base, cuando la lista está vacía, la función retorna 0. En el caso recursivo, se extrae el

primer elemento de la lista (x) y se suma con el resultado de llamar a sumatoria con el resto de la lista (xs). Así, se van sumando todos los elementos de la lista.

```
sumatoria :: [Int] -> Int
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs
```

Función pertenece
En la primera ecuación, se define el caso base, si la lista está vacía, no se encuentra el elemento, por lo que se devuelve False.

En la segunda ecuación, se define el caso recursivo, donde se compara el elemento x con el primer elemento de la lista y. Si son iguales, se devuelve True. Si no son iguales, se llama recursivamente a la función pertenece con el elemento x y el resto de la lista ys.

```
pertenece :: Int -> [Int] -> Bool
pertenece _ [] = False
pertenece x (y:ys) = x == y || pertenece x ys
```

Conjuntos

Podríamos representar un conjunto con una lista. Sin embargo, el orden de los elementos es relevante para las listas, pero no para
conjuntos. También, las listas pueden tener elementos repetidos, pero eso no tiene sentido con conjunto.

Para representar conjuntos con listas, dejando en claro que hablamos de conjuntos (sin orden ni repetidos), podemos usar el **renombre de tipos**.

- **Definición de tipo usando Type**
  `type Set a = [a]`
  type es la palabra reservada del lenguaje.
  Set es el nombre que le pusimos nosotros.

Internamente Set es una lista, pero estamos haciendo un contrato entre programadores para tratar a Set como si fuera un conjunto:

- Si nuestra función recibe un conjunto, vamos a *requerir* que no contenga elementos repetidos.
- Si nuestra función devuelve un conjunto, debemos *asegurar* que no contenga elementos repetidos

El lenguaje no respetará estas pautas, se trata de un contrato exclusivamente con los programadores, por lo que ellos deben asegurarse de cumplirlas, no Haskell

Ejemplos:

- Definir `vacio :: Set Int` que devuelve el conjunto vacío

  Primero pensemos la especificación del problema:
  problema $vacio() : seq\langle\mathbb{Z}\rangle$ {
    requiere: $\{True\}$
    asegura: $\{res = \langle\rangle\}$
  }

  Ahora escribamos la función en Haskell:

```haskell
type Set a = [a]

vacio :: Set Int
vacio = []
```

- Definir la función `incluido :: Set Int -> Set Int -> Bool` que determina si el primer conjunto está incluido en el segundo.

  Primero pensemos la especificación del problema:
  problema $incluido(s1 : seq\langle\mathbb{Z}\rangle, s2 : seq\langle\mathbb{Z}\rangle) : Bool$ {
    requiere: $\{sinRepetidos(s1) \land sinRepetidos(s2)\}$
    asegura: $\{res = true \leftrightarrow (\forall n : \mathbb{Z})(n \in s1 \rightarrow n \in s2)\}$
  }

  Ahora escribamos la función en Haskell:

```haskell
incluido [] _  = True
incluido (x:xs) c = pertenece x c && incluido xs c
```

- Definir `vacio :: Set Int` que devuelve el conjunto vacío

  Primero pensemos la especificación del problema:
  problema $vacio() : seq\langle\mathbb{Z}\rangle$ {
    requiere: $\{True\}$
    asegura: $\{res = \langle\rangle\}$
  }

  Ahora escribamos la función en Haskell:

```haskell
type Set a = [a]

vacio :: Set Int
vacio = []
```

- Definir la función `incluido :: Set Int -> Set Int -> Bool` que determina si el primer conjunto está incluido en el segundo.

  Primero pensemos la especificación del problema:
  problema $incluido(s1 : seq\langle\mathbb{Z}\rangle, s2 : seq\langle\mathbb{Z}\rangle) : Bool$ {
    requiere: $\{sinRepetidos(s1) \land sinRepetidos(s2)\}$
    asegura: $\{res = true \leftrightarrow (\forall n : \mathbb{Z})(n \in s1 \rightarrow n \in s2)\}$
  }

  Ahora escribamos la función en Haskell:

```haskell
incluido [] _  = True
incluido (x:xs) c = pertenece x c && incluido xs c
```