

Introducción a la Programación

Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2023

Recursión sobre listas

Motivación

Algunas operaciones

- ▶ `maximo :: Int -> Int -> Int`
- ▶ `maximo3 :: Int -> Int -> Int -> Int`
- ▶ `maximo4 :: Int -> Int -> Int -> Int -> Int`
- ▶ `⋮`
- ▶ `maximoN :: Int -> Int -> ... -> Int`

Pregunta

¿Hay alguna manera de definir funciones que nos permitan trabajar con cantidades arbitrarias de elementos?

Más concretamente, ¿podemos definir una función máximo que funcione por igual para 2, 10 o una cantidad N de elementos?

Respuesta: ¡Sí!, usando **listas**.

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

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]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`
- ▶ `[div 10 5, div 2 2] :: [Int]`
- ▶ `[[1], [2,3], [], [1,1000,2,0]] :: [[Int]]`
- ▶ `[1, True]`
- ▶ `[(1,2), (3,4), (5,2)]`

¿Cuál es el tipo de esta lista?

Operaciones

Algunas operaciones que nos brinda el Preludio de Haskell

- ▶ `head :: [a] -> a`
- ▶ `tail :: [a] -> [a]`
- ▶ `(:) :: a -> [a] -> [a]`

Tipar y evaluar las siguientes expresiones

- ▶ `head [(1,2), (3,4), (5,2)]`
- ▶ `tail [1,2,3,4,4,3,2,1]`
- ▶ `[1,2] : []`
- ▶ `head []`
- ▶ `head [1,2,3] : [4,5]`
- ▶ `head ([1,2,3] : [4,5])`
- ▶ `head ([1,2,3] : [4,5] : [])`

Creando listas

Formas rápidas para crear listas

Prueben las siguientes expresiones en GHCi

- ▶ `[1..100]`
- ▶ `[1,3..100]`
- ▶ `[100..1]`
- ▶ `[1..]`

Ejercicio

- ▶ Escribir una expresión que denote la lista estrictamente decreciente de enteros que comienza con el número 1 y termina con el número -100.
- ▶ Escribir una expresión que denote la lista estrictamente creciente de enteros entre -20 y 20 que son congruentes a 1 módulo 4.

Recursión sobre listas

¿Se puede pensar recursivamente en listas? ¿Cómo?

Implementar las siguientes funciones (en el pizarrón)

1. `longitud :: [Int] -> Int`
que indica cuántos elementos tiene una lista.
2. `sumatoria :: [Int] -> Int`
que indica la suma de los elementos de una lista.
3. `pertenece :: Int -> [Int] -> Bool`
que indica si un elemento aparece en la lista. Por ejemplo:
`pertenece 9 [] ~> False`
`pertenece 9 [1,2,3] ~> False`
`pertenece 9 [1,2,9,9,-1,0] ~> True`

Idea: Pensar cómo combinar el resultado de la función sobre la cola de la lista con el primer elemento. Recordar:

- ▶ `head [1, 2, 3] ~> 1`
- ▶ `tail [1, 2, 3] ~> [2, 3]`

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (Bool, Int, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ [] (lista vacía)
- ▶ algo : lista (lista no vacía)

Escribir la función longitud :: [Int] -> Int usando *pattern matching*

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs
```

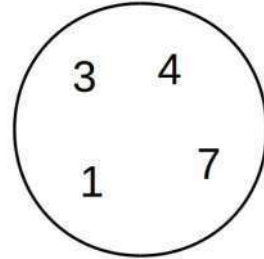
Escribir la función sumatoria :: [Int] -> Int usando *pattern matching*

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

Ejercicio: volver a implementar la función pertenece utilizando *pattern matching*.

Un nuevo tipo: Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.



¿Es buena idea usar una lista `[Int]`?

- ▶ Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
 - ▶ También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
 - ▶ Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
 - ▶ El **orden de los elementos** es relevante para las listas, pero no para conjuntos.
- ▶ ¿Y la lista `[1,3,4,7,7,7,1,4,7]`? ¿Sirve para representar a nuestro conjunto?
 - ▶ Las listas pueden tener **elementos repetidos**, pero eso no tiene sentido con conjuntos.

Vamos a usar `[Int]` para representar conjuntos de números, pero dejando claro que hablamos de conjuntos (sin orden ni repetidos). Para eso podemos hacer un **renombre de tipos**.

Conjuntos

Definición de tipo usando type

Definamos un renombre de tipos para conjuntos: `type Set a = [a]`

- ▶ Otra forma de escribir lo mismo, pero más descriptivo.
- ▶ `type` es la palabra reservada del lenguaje, `Set` es el nombre que le pusimos nosotros (podríamos haberlo llamado con otro nombre).
- ▶ Si bien internamente es una lista, la idea es tratar a `Set a` como si fuera conjunto (es un contrato entre programadores).
- ▶ Si nuestra función recibe un conjunto, **vamos a requerir** que no contenga elementos repetidos. (Haskell no hace nada para verificarlo.)
- ▶ Si nuestra función devuelve un conjunto, **debemos asegurar** que no contenga elementos repetidos. (Haskell tampoco hace nada automático.)
- ▶ Además, no hace falta preocuparse por el orden de los elementos. (Haskell no lo sabe.)

Conjuntos

Ejercicios entre todos

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

Primero pensemos la especificación del problema:

```
problema vacio() : seq<Z> {  
  requiere: { True }  
  asegura: { res = <> }  
}
```

Ahora escribamos la función en Haskell:

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

Conjuntos

- Definir la función `agregar :: Int -> Set Int -> Set Int` que dado un número y un conjunto agrega el primero al segundo.

Primero pensemos la especificación del problema:

```
problema agregar( $e : \mathbb{Z}$ ,  $s : seq\langle \mathbb{Z} \rangle$ ) :  $seq\langle \mathbb{Z} \rangle$  {  
  requiere: {sinRepetidos( $s$ )}  
  asegura: {( $\forall n : \mathbb{Z}$ )( $n \in s \rightarrow n \in res$ )  $\wedge$  ( $e \in res$ )}  
  asegura: {( $\forall n : \mathbb{Z}$ )( $n \in res \rightarrow ((n \in s) \vee (n = e))$ )}  
  asegura: {sinRepetidos( $res$ )}  
}
```

```
pred sinRepetidos( $s : seq\langle \mathbb{Z} \rangle$ ) {  
( $\forall i, j : \mathbb{Z}$ )( $0 \leq i < |s| \wedge 0 \leq j < |s| \wedge i \neq j \rightarrow s[i] \neq s[j]$ )  
}
```

Ahora escribamos la función en Haskell:

```
agregar x [] = [x]  
agregar x c | pertenece x c = c  
           | otherwise = (x:c)
```

¿Podríamos haber hecho otra implementación donde agreguemos el número al final de la lista? ¿Respetamos la especificación?

Conjuntos

- 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*($s1$) \wedge *sinRepetidos*($s2$)}
 asegura: { $res = true \leftrightarrow (\forall n : \mathbb{Z})(n \in s1 \rightarrow n \in s2)$ }
}

Ahora escribamos la función en Haskell:

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

Conjuntos

- Definir la función `iguales :: Set Int -> Set Int -> Bool` que determina dos conjuntos son iguales.

Primero pensemos la especificación del problema:

```
problema iguales(s1 : seq⟨ℤ⟩, s2 : seq⟨ℤ⟩) : Bool {  
  requiere: {sinRepetidos(s1) ∧ sinRepetidos(s2)}  
  asegura: {res = true ⇔ ((incluido(s1, s2) = True ∧  
    incluido(s2, s1) = True)}  
}
```

Ahora escribamos la función en Haskell:

```
iguales c1 c2 = incluido c1 c2 && incluido c2 c1
```

Introducción a la Programación

Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2023

Extensión del Lenguaje de Especificación: Variables de tipo