Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2019-20

Parte II Programación Funcional

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 5: Tipos algebraicos y orden superior

Índice

1. Las listas

Antes de resolver los ejercicios planteados en esta sección, se debería haber hecho una lectura comprensiva de la correspondiente sección, de mismo título, del documento de lectura previa de esta práctica.

Sirvan como recordatorio de algunas funcionalidades de las listas en Haskell los siguientes ejemplos:

- La función `length` puede emplearse sobre listas de cualquier tipo base:

```
> length [1,2,3]
3
> length ['a','b','c','d']
4
```

- El operador `(!!)` permite la indexación de listas, y puede usarse con listas de cualquier tipo base:

```
> [1,2,3] !! 2
3
>  ['a','b','c','d'] !! 0
'a'
```

- El operador `(++)` permite concatenar dos listas de cualquier tipo:

```
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
> ['a','b','c','d'] ++ ['e','f']
"abcdef"
```

Además, se facilitan los siguientes 2 ejercicios ya resueltos como ejemplos de funciones que manejan listas.

Ejercicio 1 (Resuelto) *Definir una función para calcular el valor binario correspondiente a un número entero no negativo:*

```
decBin :: Int -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> decBin 4
```

debe devolver la lista `[0,0,1]` *(empezando por el bit menos significativo).*

```
module DecBin where
 decBin :: Int -> [Int]
 decBin x = if x < 2 then [x]
            else (x `mod` 2) : decBin (x `div` 2)
```

Ejercicio 2 (Resuelto) *Definir una función para calcular el valor decimal correspondiente a un número en binario (representado como una lista de 1's y 0's):*

```
binDec :: [Int] -> Int
```

Por ejemplo, la evaluación de la expresión:

```
> binDec [0,1,1]
```

debe devolver el valor `6`.

```
module BinDec where
 binDec :: [Int] -> Int
 binDec (x:[]) = x
 binDec (x:y)  = x + binDec y * 2
```

Se pide resolver los siguientes ejercicios. Aunque en los anteriores ejemplos resueltos se ha escrito cada función en un módulo diferente, es recomendable escribir todas las funciones de esta sección en un mismo módulo.

Ejercicio 3 *Definir una función para calcular la lista de divisores de un número entero no negativo:*

```
divisors :: Int -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> divisors 24
```

debe devolver la lista `[1,2,3,4,6,8,12,24]`.

Ejercicio 4 *Definir una función para determinar si un entero pertenece a una lista de enteros:*

```
member :: Int -> [Int] -> Bool
```

Por ejemplo, la evaluación de la expresión:

```
> member 1 [1,2,3,4,8,9]
```

debe devolver el valor `True`. *Y la evaluación de la expresión:*

```
> member 0 [1,2,3,4,8,9]
```

debe devolver el valor `False`.

Ejercicio 5 *Definir una función para comprobar si un número es primo (sus divisores son 1 y el propio número) y una función para calcular la lista de los n primeros números primos:*

```
isPrime :: Int -> Bool
primes :: Int -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> isPrime 2
```

debe devolver el valor `True`*. Y la evaluación de la expresión:*

```
> primes 5
```

debe devolver la lista `[1,2,3,5,7]`*. Se recuerda que Haskell permite obtener fácilmente una lista con los elementos iniciales de otra lista infinita.*

Ejercicio 6 *Definir una función para seleccionar los elementos pares de una lista de enteros:*

```
selectEven :: [Int] -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> selectEven [1,2,4,5,8,9,10]
```

devuelve la lista `[2,4,8,10]`*.*

Ejercicio 7 *Definir ahora una función para seleccionar los elementos que ocupan las "posiciones pares" de una lista de enteros (recuerda que las posiciones en una lista empiezan por el índice cero, siguiendo el funcionamiento del operador* `!!`*):*

```
selectEvenPos :: [Int] -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> selectEvenPos [1,2,4,5,8,9,10]
```

devuelve la lista `[1,4,8,10]`*.*

Ejercicio 8 *Definir una función* `iSort` *para ordenar una lista en sentido ascendente. Para ello, definir antes una función* `ins` *que inserte correctamente un elemento en una lista ordenada (la ordenación se puede resolver recursivamente, considerando sucesivas operaciones de inserción de los elementos a ordenar en la parte de la lista ya ordenada):*

```
iSort :: [Int] -> [Int]
ins :: Int -> [Int] -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> iSort [4,9,1,3,6,8,7,0]
```

devuelve la lista `[0,1,3,4,6,7,8,9]`*. Y la evaluación de la expresión:*

```
> ins 5 [0,1,3,4,6,7,8,9]
```

devuelve la lista `[0,1,3,4,5,6,7,8,9]`*.*

2. Las funciones `map` y `filter`

Antes de resolver los ejercicios planteados en esta sección, se debería haber hecho una lectura comprensiva de la correspondiente sección, de mismo título, del documento de lectura previa de esta práctica.

Ejercicio 9 *Definir, usando la función* `map`*, una función para duplicar todos los elementos de una lista de enteros:*

```
doubleAll :: [Int] -> [Int]
```

Por ejemplo, la evaluación de la expresión:

```
> doubleAll [1,2,4,5]
```

devuelve la lista `[2,4,8,10]`*.*

Ejercicio 10 *Expresar mediante listas intensionales las definiciones de las funciones* `map` *y* `filter`*. Nota: Se las puede llamar* `map'` *y* `filter'` *para evitar conflictos con las funciones predefinidas del* `Prelude`*.*

3. Tipos algebraicos

De nuevo es conveniente advertir de que, antes de resolver los ejercicios planteados en esta sección, se debería haber hecho una lectura comprensiva de la correspondiente sección, de mismo título, del documento de lectura previa de esta práctica.

3.1. Enumeraciones y Tipos Renombrados

Para el ejercicio que se plantea en esta sección, se han de definir, previamente, los siguientes tipos "sinónimos":

```
type Person = String
type Book = String
type Database = [(Person,Book)]
```

El tipo `Database` define una base de datos de una biblioteca como una lista de pares `(Person,Book)` donde `Person` es el nombre de la persona que tiene en préstamo el libro `Book`. Un ejemplo de base de datos es:

```
exampleBase :: Database
exampleBase = [("Alicia","El nombre de la rosa"),("Juan",
 "La hija del canibal"),("Pepe","Odesa"),("Alicia",
 "La ciudad de las bestias")]
```

A partir de esta base de datos ejemplo se pueden definir funciones para obtener los libros que tiene en préstamo una persona dada, `obtain`, para realizar un préstamo, `borrow`, y para realizar una devolución, `return'`.
Por ejemplo, la función `obtain` se puede definir así:

```
obtain :: Database -> Person -> [Book]
obtain dBase thisPerson
 = [book | (person,book) <- dBase, person == thisPerson]
```

que significa que la función devuelve la lista de todos los libros tales que hay un par `(person,book)` en la base de datos y `person` es igual a la persona cuyos libros se está buscando. Por ejemplo, la evaluación de la expresión:
`obtain exampleBase "Alicia"`
devuelve la lista: `["El nombre de la rosa","La ciudad de las bestias"]`

Ejercicio 11 *Crear un módulo con el anterior código y completar el programa con las definiciones para las funciones* `borrow` *y* `return'`*:*

```
borrow :: Database -> Book -> Person -> Database
return' :: Database -> (Person,Book) -> Database
```

3.2. Tipos recursivos. Árboles

A continuación se recuerdan las declaraciones de tipos de árboles (introducidas en el material de lectura previa) dado que se usarán en el resto de ejercicios de esta práctica:

```
data TreeInt = Leaf Int | Branch TreeInt TreeInt
```

- Ejemplo:
  `Branch (Leaf 0) (Branch (Leaf 0) (Leaf 1))`
  es un valor de tipo `TreeInt`.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- Ejemplo:

  ```
  numleaves (Leaf x)     = 1
  numleaves (Branch a b) = numleaves a + numleaves b
  ```

  es una función que calcula el número de hojas de un árbol del tipo `Tree a`.

```
data BinTreeInt  = Void | Node Int BinTreeInt BinTreeInt
```

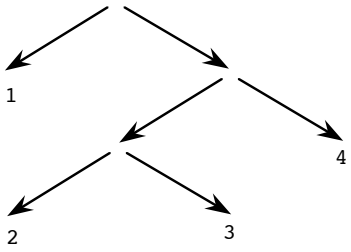
- Algunos ejemplos de valores de este tipo:

  ```
  treeB1 = Void
  treeB2 = (Node 5 Void Void)
  treeB3 = (Node 5
  ```

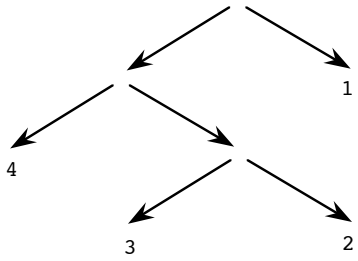
```
(Node 3 (Node 1 Void Void) (Node 4 Void Void))
(Node 6 Void (Node 8 Void Void)))
```

Ejercicio 12 *Definir una función que obtenga el árbol simétrico al que se le pasa como parámetro.*

```
symmetric :: Tree a -> Tree a
```



Un árbol de enteros...



...y su simétrico

Nota: Si se prueba la función `symmetric` con `ghci` se verá que da un error:

```
> symmetric (Branch (Leaf 5) (Leaf 7))
<interactive>:5:1:
    No instance for (Show (Tree a0))
      arising from a use of 'print'
    Possible fix: add an instance declaration for (Show (Tree a0))
    In a stmt of an interactive GHCi command: print it
```

esto es debido a que no sabe cómo mostrar el resultado. El resultado se muestra con la función `show` (en cierta manera, como el `toString` de Java). Se va a utilizar un mecanismo muy sencillo para proporcionar un comportamiento por defecto para mostrar, con `show`, un tipo de datos algebraico. Basta con añadir `deriving Show` al finalizar la declaración de un tipo de datos algebraico:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
```

Ejercicio 13 *Definir las funciones*

```
listToTree :: [a] -> Tree a
treeToList :: Tree a -> [a]
```

la primera de las cuales convierte una lista no vacía en un árbol, realizando la segunda de ellas la transformación contraria.

Sobre el tipo de datos `BinTreeInt` resuélvanse los siguientes ejercicios. Es también muy recomendable añadir `deriving Show` al finalizar la declaración de este tipo, a fin de poder mostrar el resultado de las funciones.

Ejercicio 14 *Definir una función*

```
insTree :: Int ->  BinTreeInt -> BinTreeInt
```

para insertar un valor entero en su lugar en un árbol binario ordenado.

Ejercicio 15 *Dada una lista no ordenada de enteros, definir una función*

```
creaTree :: [Int] ->  BinTreeInt
```

que construya un árbol binario ordenado a partir de la misma.

Ejercicio 16 *Definir una función*

```
treeElem :: Int ->  BinTreeInt -> Bool
```

que determine "de forma eficiente" si un valor entero pertenece o no a un árbol binario ordenado.

Ampliación

Se os proponen algunos ejercicios como ampliación de esta práctica. Resolverlos se considera ampliación por cuanto podrían exceder el tiempo de las 2 sesiones de laboratorio. Abordar su resolución, sin embargo, es muy recomendable como trabajo preparatorio de la evaluación.

Ejercicio 17 *Definir una función para determinar cuántas veces aparece repetido un elemento en una lista de enteros:*

```
repeated :: Int-> [Int] -> Int
```

Por ejemplo, la evaluación de la expresión:

```
> repeated 2 [1,2,3,2,4,2]
```

debe devolver el valor `3`.

Ejercicio 18 *Definir una función para* concatenar *listas de listas, que tome como argumento una lista de listas y devuelva la concatenación de las listas consideradas:*

```
concat' :: [[a]] -> [a]
```

Por ejemplo, la evaluación de la expresión:

```
> concat' [[1,2],[3,4],[8,9]]
```

debe devolver la lista `[1,2,3,4,8,9]`.

Ejercicio 19 *¿Qué computa la expresión misteriosa que aparece a continuación?:*

```
> (sum . map square . filter even) [1..10]
```

donde `sum` *es la función predefinida que suma los elementos de una lista de enteros, y* `square` *es una función, no predefinida, que calcula el cuadrado de un número entero:*

```
square :: Int -> Int
square x = x * x
```

Ejercicio 20 *Escribir la definición de la función* `numleaves`*, presentada anteriormente como ejemplo del tipo* `Tree a`*, en un fichero y cargarlo en el intérprete* GHCi*. Piensa cuál es el tipo de la función* `numleaves` *y comprueba el proporcionado por el intérprete mediante el comando* `:t`*.*

Ejercicio 21 *Considérese la declaración de árbol binario de enteros vista anteriormente:*

```
data BinTreeInt  = Void | Node Int BinTreeInt BinTreeInt
                   deriving Show
```

Se quiere implementar una función `dupElem` *que devuelva un árbol con la misma estructura pero con todos sus valores multiplicados por dos.*

Considérese que se aplicara `dupElem` *a los árboles* `treeB1`*,* `treeB2` *y* `treeB3` *declarados anteriormente.*
La evaluación de la expresión:

```
> dupElem treeB1
```

devuelve `Void`*. La evaluación de la expresión:*

```
> dupElem treeB2
```

devuelve `Node 10 Void Void`*. Y la evaluación de la expresión:*

```
> dupElem treeB3
```

devuelve:

```
Node 10
(Node 6 (Node 2 Void Void) (Node 8 Void Void))
(Node 12 Void (Node 16 Void Void)).
```

Ejercicio 22 *Considérese la siguiente declaración:*

```
data Tree a = Branch a (Tree a) (Tree a) | Void deriving Show
```

Se quiere implementar una función `countProperty` *con la siguiente signatura:*

```
countProperty :: (a -> Bool) -> (Tree a) -> Int
```

que devuelva el número de elementos del árbol que cumplen la propiedad. Por ejemplo, la evaluación de la expresión:

```
> countProperty (>9) (Branch 5 (Branch 12 Void Void) Void)
```

devuelve `1`*, y la evaluación de la expresión:*

```
> countProperty (>0) (Branch 5 (Branch 12 Void Void) Void)
```

devuelve `2`*.*