Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2020-21

Parte II Programación Funcional

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 5 - Material de lectura previa

Tipos algebraicos y orden superior

Índice

1. Tipos e Inferencia de Tipos

En programación funcional, los valores que pueden obtenerse al evaluar expresiones están organizados por *tipos*. Cada tipo representa un conjunto de valores. Por ejemplo, el tipo (primitivo o básico) `Int` representa los valores `0`, `1`, `2`, `-1`, `-2`, etc. El tipo `Bool` representa los valores booleanos `True` y `False`. El tipo `Char` los caracteres alfanuméricos `'a'`, `'b'`, `'A'`, `'B'`, `'0'`, ... En programación funcional, como en la mayoría de lenguajes de programación, *toda expresión tiene asociado un tipo*. Esto se puede expresar explícitamente mediante el operador de tipificación '`::`'

```
42 :: Int
6*7 :: Int
abs (2*(-2)) :: Int
```

Como ya se ha visto, se puede pedir al intérprete GHCi que informe acerca del tipo de cualquier expresión bien formada usando el comando de usuario "`:t`":

```
> :t 'a'
'a' :: Char


> :t "Hello"
"Hello" :: [Char]
```

En programación funcional, los identificadores de las funciones son expresiones válidas gracias a la currificación y, como tales, tienen un tipo. Además, no es necesario que el programador proporcione información explícita de tipificación de las funciones que defina o de las expresiones utilizadas, ya que el intérprete lo *infiere* automáticamente a partir de las definiciones.

Se puede probar, por ejemplo:

```
> :t show
> :t (+)
> :t (*)
> :t (3 +)
> :t (* 2.0)
```

Nota adicional: Obsérvese el caso particular de la función binaria `(*)` que espera dos argumentos de un tipo numérico y devuelve un valor del mismo tipo numérico.

```
Prelude> :t (*)
(*) :: (Num a) => a -> a -> a
```

Sin embargo, la función unaria `(* 2.0)` espera sólo un argumento, además restringido al tipo `Float` o alguno parecido, y devuelve su doble, es decir, el número que resulta de multiplicar el argumento por la constante real 2.0.

```
Prelude> :t (* 2.0)
(* 2.0) :: (Fractional a) => a -> a
```

Obsérvese que esta función se ha obtenido como una simple aplicación parcial del operador de multiplicación `(*)`, que es binario, al caso particular de haber fijado su segundo argumento para que tome el valor de la constante real 2.0, lo cual es posible gracias a que esta función está definida de forma currificada.

2. Estrategia de Reducción

La estrategia de reducción en Haskell es *lazy* (perezosa). Esta estrategia reduce una expresión (parcialmente) solo si realmente es necesario para calcular el resultado. Es decir, se reducen los argumentos solo lo suficiente para poder aplicar algún paso de reducción en el símbolo de función más externo.

Gracias a esto, es posible trabajar con estructuras de datos infinitas. En lenguajes que usan la estrategia *eager* (impaciente), como son la mayoría de los lenguajes imperativos y los lenguajes funcionales más antiguos, este tipo de estructuras no pueden manipularse.

La función `repeat'` devuelve una lista infinita (similar a la función `repeat` del `Prelude`):

```
repeat' :: a -> [a]
repeat' x = xs where xs = x:xs
```

La llamada `repeat' 3` devuelve la lista infinita `[3,3,3,3,3,3,3,3,3,...`

Nota adicional: Se debe tener cuidado porque la llamada `repeat' 3` no termina e imprime una lista infinita del número 3 por la pantalla, teniendo que parar la ejecución con `^C`.

Una lista infinita generada por `repeat'` puede ser usada como argumento parcial por una función que tiene un resultado finito. La siguiente función, por ejemplo, toma un número finito de elementos de una lista:

```
take' :: Int -> [a] -> [a]
take' _ [] = []
take' n (x:t)
   | n<=0 = []
   | otherwise = x : take' (n - 1) t
```

Por ejemplo, la llamada `take' 3 (repeat' 4)` devuelve la lista `[4,4,4]`.

3. Las listas

3.1. El tipo `lista`

En programación funcional es posible emplear tipos estructurados cuyos valores están compuestos por objetos de otros tipos. Por ejemplo, el tipo lista `[a]` puede servir para aglutinar en una única estructura objetos del *mismo* tipo (denotado, en este caso, por la variable de tipo `a`, que puede instanciarse a cualquier tipo). En Haskell, las listas pueden especificarse encerrando sus elementos entre corchetes y separándolos con comas:

```
[1,2,3] :: [Int]
['a','b','c','d'] :: [Char]
[cos,log,(1.0 +)] :: [Float -> Float]
         -- Lista de funciones de
         -- reales en reales: coseno, logaritmo en base 2,
         -- incrementar una unidad un número real.
```

Sin embargo, las listas

```
[1,'a',2]
['a',log,3]
[cos,2,(*)]
```

no son válidas (¿por qué? ¿qué responde GHCi al preguntar por el tipo?).

La lista vacía se denota como `[]`. Cuando no es vacía, se puede descomponer usando una notación que separa el elemento inicial, de la lista que contiene los elementos restantes. Por ejemplo:

```
1:[2,3]   ó   1:2:[3]   ó   1:2:3:[]
'a':['b','c','d']  ó  'a':'b':['c','d'] ó 'a':'b':'c':'d':[]
cos:[log]   ó   cos:log:[]
```

Los tipos que incluyen variables de tipo en su definición (como el tipo lista) son tipos genéricos o *polimórficos*. Se pueden definir funciones sobre tipos polimórficos, que pueden emplearse sobre objetos de cualquier tipo que sea una instancia de los tipos polimórficos involucrados. Por ejemplo, la función (predefinida) `length` calcula la longitud de una lista:

```
> :t length
length :: [a] -> Int
```

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

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

El operador `(!!)` permite la indexación de listas. Se usa para obtener el elemento en una posición dada de una lista:

```
> :t (!!)
(!!) :: [a] -> Int -> a
```

La indexación puede utilizarse con listas de cualquier tipo:

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

Otra función muy útil para listas es `(++)`, que se usa para 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"
```

Las cadenas de caracteres vistas en la práctica anterior (por ejemplo, `"hello"`) son simples listas de caracteres escritas con una sintaxis especial, que omite las comillas simples. Así, `"hello"` es tan solo una sintaxis conveniente para la lista `['h','e','l','l','o']`. Toda operación genérica sobre listas es, por tanto, aplicable también a las cadenas.

3.2. Los rangos (Listas aritméticas)

La forma básica de las listas aritméticas o rangos tiene la sintaxis `[first..last]` de modo que genera la lista de valores entre ambos (inclusive):

```
> [10..15]
[10,11,12,13,14,15]
```

La sintaxis de los rangos en Haskell permite las siguientes opciones:

- `[first..]`
- `[first,second..]`
- `[first..last]`
- `[first,second..last]`

Cuyo comportamiento se puede deducir de los siguientes ejemplos:

```
[0..] -> 0, 1, 2, 3, 4, ...
[0,10..] -> 0, 10, 20, 30, ...
[0,10..50] -> 0, 10, 20, 30, 40, 50
[10,10..] -> 10, 10, 10, 10, ...
[10,9..1] -> 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
[1,0..] -> 1, 0, -1, -2, -3, ...
[2,0..(-10)] -> 2, 0, -2, -4, -6, -8, -10
```

3.3. Las listas intensionales

Haskell proporciona una notación alternativa para las listas, las llamadas *listas intensionales* (notación que es útil también para describir cálculos que necesitan `map` y `filter`, tal como se verá en la siguiente sección). He aquí un ejemplo:

```
> [x * x | x <- [1..5], odd x]
[1,9,25]
```

La expresión se lee *la lista de los cuadrados de los números impares en el rango de 1 a 5*.

Formalmente, una lista intensional es de la forma `[e|Q]`, donde `e` es una expresión y `Q` es un *cualificador*. Un cualificador es una secuencia, potencialmente vacía, de *generadores* y *guardas* separados por comas. Un generador toma la forma `x <- xs`, donde `x` es una variable o tupla de variables, y `xs` es una expresión de tipo lista. Una guarda es una expresión booleana. He aquí algunos ejemplos:

```
> [(a,b) | a <- [1..3], b <- [1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
>
> [(a,b) | a <- [1..2], b <- [1..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

Los generadores posteriores pueden depender de variables introducidas por los precedentes, como en:

```
> [(i,j) | i <- [1..4], j <- [i+1..4]]
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Se pueden intercalar libremente generadores y guardas:

```
> [(i,j) | i <- [1..4], even i, j <- [i+1..4], odd j]
[(2,3)]
```

4. Las funciones `map` y `filter`

Se trata de dos funciones predefinidas útiles para operar con listas. Son funciónes muy comunes de las denominadas *de orden superior* al aceptar (en este caso como primer argumento) *funciones*.

4.1. La función `map`

La función `map` aplica una función a cada elemento de una lista. Por ejemplo:

```
> map square [9,3]
[81,9]
> map (<3) [1,2,3]
```

```
[True,True,False]
```

donde `square` es la función que calcula el cuadrado de un número entero (la cual podría definirse, por ejemplo, mediante `let square = (^2)`). La definición de `map` es:

```
map          :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Existen bastante leyes algebraicas útiles relacionadas con `map`. Dos hechos básicos son:

$$\texttt{map}\ id = id$$
$$\texttt{map}\ (f\ .\ g) = (\texttt{map}\ f)\ .\ (\texttt{map}\ g)$$

donde id es la función identidad y "." [1] es la composición de funciones. La primera ecuación expresa que al aplicar la función identidad a todos los elementos de una lista, la lista queda sin modificar. Las dos apariciones de `id` en la primera ecuación tienen tipos diferentes: el tipo del de la parte izquierda es id `:: a -> a` mientras que el de la parte derecha es id `:: [a] -> [a]`. La segunda ecuación expresa que aplicar primero g a todo elemento de una lista y después aplicar f a todo elemento de la lista resultado, da el mismo resultado que aplicar $f\ .\ g$ a la lista original. Leída de derecha a izquierda, esta ley permite reemplazar dos recorridos de una lista por uno solo, con la correspondiente ganancia en eficiencia.

4.2. La función `filter`

La función `filter` toma una función booleana `p` y una lista `xs` y devuelve la sublista de `xs` cuyos elementos satisfacen `p`. Por ejemplo:

```
> filter even [1,2,4,5,32]
[2,4,32]
```

La definición de `filter` es:

```
filter          :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = if p x then x:filter p xs else filter p xs
```

5. Tipos algebraicos

5.1. Enumeraciones

En un lenguaje funcional como Haskell, el programador puede definir nuevos tipos con sus valores asociados, empleando los llamados tipos algebraicos. Por ejemplo, con la declaración:

[1] También escrito " · " en notación matemática.

```
data Color = Red | Green | Blue
```

se establece un nuevo tipo de datos `Color`. El tipo `Color` contiene solo tres valores denotados por los correspondientes constructores de datos constantes `Red`, `Green` y `Blue`. Recuérdese que, en Haskell, los identificadores que se refieren a tipos de datos y a constructores de datos siempre comienzan por una *letra mayúscula*.

5.2. Tipos Renombrados

También se pueden declarar renombramientos de tipos ya definidos, denominados tipos "sinónimos". Por ejemplo:

```
type Distance = Float
type Angle = Float
type Position = (Distance,Angle)
type Pairs a = (a,a)
type Coordinates = Pairs Distance
```

De hecho, el tipo `String` es un renombramiento, como ya se había mencionado antes:

```
type String = [Char]
```

y se le puede aplicar todas las operaciones sobre listas, por ejemplo las comparaciones de orden (orden lexicográfico):

```
> "hola" < "halo"
False
> "ho" < "hola"
True
```

A menudo se prefiere mostrar las cadenas sin que aparezcan las dobles comillas en la salida y donde los caracteres especiales, como salto de línea, etc., se impriman como el caracter real que representan. Haskell dispone de *órdenes* primitivas para imprimir cadenas, leer de ficheros, etc. Por ejemplo, usando la función `chr` que convierte un entero al carácter que éste representa:

```
> putStr ("Esta frase tiene un" ++ [chr 10] ++ "fin de linea")
Esta frase tiene un
fin de linea
```

Se recuerda la necesidad de importar el módulo `Data.Char` para usar la función `chr`.

5.3. Tipos recursivos. Árboles

Se pueden dar diferentes definiciones para este tipo de datos, llamados árboles. Consideraremos dos clases de árboles en los siguientes apartados.

5.3.1. Árboles que almacenan valores en las hojas: `TreeInt` y `Tree a`

La declaración:

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

establece un nuevo tipo de datos `TreeInt` (donde `TreeInt` es, de nuevo, un constructor de tipo constante) que consta de un número infinito de valores definidos recursivamente con la ayuda de los símbolos constructores de datos `Leaf` (unario) y `Branch` (binario), que toman como argumentos un número entero y dos árboles `TreeInt`, respectivamente.

Considérense algunos ejemplos de valores o datos de los tipos anteriores:

- `[Red,Green,Red]` es un valor de tipo `[Color]`
- `Branch (Leaf 0) (Branch (Leaf 0) (Leaf 1))` es un valor de tipo `TreeInt`.

Nada impide definir tipos genéricos empleando estos recursos expresivos. El tipo:

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

es un tipo algebraico polimórfico para definir árboles de cualquier tipo de datos, de forma similar a como las listas admiten cualquier tipo de datos.

Por supuesto, también se pueden definir funciones sobre estos nuevos tipos de datos algebraicos definidos por el usuario. Por ejemplo, la función `numleaves` definida como sigue:

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

calcula el número de hojas de un árbol del tipo `Tree a`.

5.3.2. Árboles que almacenan valores en los nodos: `BinTreeInt`

Considérese ahora la siguiente definición alternativa para el tipo de datos de los árboles binarios de enteros:

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

en el que los valores enteros se almacenan en los nodos y donde las hojas son subárboles que tienen tan solo raíz (denotados por el símbolo constructor de datos `Void`). A continuación, se muestran algunos ejemplos de árboles binarios de enteros:

```
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)))
```

donde `treeB1` es un árbol vacío, `treeB2` es un árbol con un solo elemento, y `treeB3` es un árbol con el valor 5 en su nodo raíz, los valores 3, 1, 4 en los nodos de su rama izquierda, y los valores 6, 8 en los nodos de su rama derecha.

Se dice que un árbol binario está ordenado si los valores almacenados en el subárbol izquierdo de un nodo dado son siempre menores o iguales al valor en dicho nodo, mientras que los valores almacenados en su subárbol derecho son siempre mayores. Este tipo de árboles también recibe el nombre de árbol binario de búsqueda (*binary search tree*). Los anteriores ejemplos corresponden a árboles ordenados.