Programación Funcional

Trabajo Práctico Nro. 11

Temas: Derivación y síntesis de programas. Combinadores.

Bibliografía relacionada:

- Richard Bird and Oege de Moor. Algebra of Programming. Prentice-Hall, 1997.
- Hughes, J. 1995. The Design of a Pretty-printing Library. In Advanced Functional Programming, First international Spring School on Advanced Functional Programming Techniques-Tutorial Text (May 24 - 30, 1995). J. Jeuring and E. Meijer, Eds. Lecture Notes In Computer Science, vol. 925. Springer-Verlag, London, 53-96.

1. Derivar, para cada una de las siguientes funciones, definiciones recursivas en términos de sí mismas:

   a) `inv :: (a -> Bool) -> [(a,b)] -> [(b,a)]`
   `inv p xs = map swap (filter (p . fst) xs)`
   `where swap (x,y) = (y,x)`,
   que toma un predicado `p` y una lista de pares `ys` y devuelve la lista de los pares `(y,x)` para aquellos pares `(x,y)` de `ys` tales que `p x` es verdadero.

   b) `sqlist :: [Int] -> [Int]`
   `sqlist xs = map (^2) xs`,
   que devuelve la lista de los cuadrados de los elementos de la lista dada.

   c) `pc :: [a] -> [b] -> [(a,b)]`
   `pc xs ys = concat (map (\x -> map (pair x) ys) xs)`
   `where pair x y = (x,y)`,
   que devuelve el producto cartesiano de las listas dadas.

   d) `distance :: [(Int,Int)] -> [Int]`
   `distance xs = map (\(x,y) -> sqrt (x^2 + y^2)) xs`,
   que dada una lista de puntos del plano devuelve la lista de las distancias al origen.

2. Considerando definidas las siguientes funciones:

```
uncurry f (x,y) = f x y
apply = id
prod = foldr (*) 1
sum = foldr (+) 0
```

   Derivar definiciones recursivas en términos de sí mismas para cada una de las siguientes funciones:

a) `zipW :: (a -> b -> c) -> [a] -> [b] -> [c]`
`zipW f xs ys = map (uncurry apply) (zip (map f xs) ys)`

b) `prodsum :: [Int] -> (Int,Int)`
`prodsum ls = (prod ls, sum ls)`

c) `insert :: a -> [a] -> [a]`
`insert x ys = takeWhile (<= x) ys ++ [x] ++ dropWhile (<= x) ys`

d) `is_sorted :: [Int] -> Bool`
`is_sorted ys = foldl (&&) True (map (uncurry (<=)) (zip ys (tail ys)))`

3. Dada la siguiente especificación: `lookup :: [(a,b)] -> a -> b`
`lookup zs k = head (map snd (filter ((==k) . fst) zs))`

   a) Decir qué hace. Dar un ejemplo de una aplicación en la que `lookup` sea útil.

   b) Derivar una definición recursiva en términos de sí misma.

4. Dada una definición ineficiente de una función, tomada como especificación: ¿qué se puede decir de la eficiencia de nuevas definiciones derivadas a partir de ella?

5. Explicar las diferencias y similitudes entre sintetizar, derivar y transformar.

6. Escribir una función de pretty-printing para los siguientes tipos, usando la librería de pretty-printing hecha por John Huges.

   a) El tipo `TipTree` de la práctica 6.

   b) El tipo que definió para los conjuntos definidos por extensión en la práctica 6, ejercicio 1 a).

   c) El tipo `Form` de la práctica 6.

   d) El tipo `BinTree` de la práctica 9.

   e) El tipo `GenTree` de la práctica 9.

Ejercicios complementarios

7. Derivar una definición recursiva del predicado `incl` que determina si una lista está incluída en otra (en el sentido de que todos los elementos de la primera están también en la segunda) a partir de la siguiente definición:

   `incl xs ys = foldl (&&) True (map (‘elem‘ ys) xs)`

8. Derivar una definición recursiva en terminos de sí misma, para la función `combine`.

```
one,two :: TipTree Bool -> Bool
one (Tip x) = False
one (Join t1 t2) = two t1 || two t2

two (Tip x) = not x
two (Join t1 t2) = one t1 && one t2

combine :: TipTree Bool -> (Bool,Bool)
combine t = (one t, two t)
```