Universidad Nacional Autónoma de México

Facultad de Ciencias

Lenguajes de Programación | 7098

Semanal 7 : | Combinador de punto fijo, continuaciones y recurrsión de cola

Sosa Romo Juan Mario | 320051926

Legorreta Esparragoza Juan Luis | 319317532

Erik Eduardo Gómez López | 320258211

15/10/24

1. Dada la siguiente expresión en MiniLisp:

```
(let (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1))))))
    (sum 5))
```

   (a) Ejecutarla y explicar el resultado.

   (b) Modificarla usando el combinador de punto fijo Y, volver a ejecutarla y explicar el resultado.

2. Evaluar la siguiente expresión en Racket.

```
> (define c #f)
> (+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))
> (c 10)
```

   (a) Explicar su resultado:

   (b) Dar la continuación asociada a evaluar usando la notación $\lambda \uparrow$.

3. Realizar los siguientes ejercicios en Haskell:

   (a) Definir la función recurisva ocurrenciasElementos que toma como argumentos dos listas y devuelve una lista de parejas, en donde cada pareja contiene en su parte izquierda un elemento de la segunda lista y en su parte derechael número de veces que aparece dicho elemento en la primera lista. Por ejemplo:

```
> ocurrenciasElementos [1,3,6,2,4,7,3,9,7] [5,2,3]
[(5,0),(2,1),(3,2)]
```

   Yo llegue a la siguiente solución, que usa una función auxiliar para contar la cantidad de veces que aparece un elemento en una lista:

```
-- Ocurrencias.hs
cuentaElemento :: Int -> [Int] -> Int
cuentaElemento _ [] = 0
cuentaElemento x [y]
    | x == y    = 1
    | otherwise = 0
cuentaElemento x (y:ys)
    | x == y    = 1 + cuentaElemento x ys
    | otherwise = cuentaElemento x ys
```

```
ocurrenciasElementos :: [Int] -> [Int] -> [(Int, Int)]
ocurrenciasElementos _ [] = []
ocurrenciasElementos xs (y:ys) = (y, cuentaElemento y xs) :
                                                    ocurrenciasElementos xs ys
```

La idea es que la función ocurrenciasElementos recorre la lista de elementos a contar y por cada elemento llama a la función cuentaElemento que cuenta la cantidad de veces que aparece el elemento en la lista. La función cuentaElemento recorre la lista de elementos y por cada elemento compara si es igual al elemento a contar, si es así suma 1 al contador y sigue con el resto de la lista, si no es igual sigue con el resto de la lista. Cuando la lista esta vacía devuelve el contador.

(b) Mostrar los registros de activación generados por la función definida en el ejercicio anterior con la llamada ocurrenciasElementos [1,2,3] [1,2].

Ahora mismo nuestras llamadas dejan pendiente el 'cons' de la lista de salida, por lo que el registro de activación se queda pendiente de completa, lo mismo pasa para la llamada de la función cuentaElemento, por el mas, asi es que la pila, se veria algo como:

```
ocurrenciasElementos [1,2,3] [1,2]
(1, cuentaElemento 1 [1,2,3]) : ocurrenciasElementos [1,2,3] [2]
cuentaElemento 1 [1,2,3] = 1 + cuentaElemento 1 [2,3]
cuentaElemento 1 [2,3] = 0 + cuentaElemento 1 [3]
cuentaElemento 1 [3] = 0 + cuentaElemento 1 []
cuentaElemento 1 [] = 0

(1, 1) : ocurrenciasElementos [1,2,3] [2]
(1, 1) : (2, cuentaElemento 2 [1,2,3]) : ocurrenciasElementos [1,2,3] []
cuentaElemento 2 [1,2,3] = 0 + cuentaElemento 2 [2,3]
cuentaElemento 2 [2,3] = 1 + cuentaElemento 2 [3]
cuentaElemento 2 [3] = 0 + cuentaElemento 2 []
cuentaElemento 2 [] = 0

(1, 1) : ocurrenciasElementos [1,2,3] [2]
(1, 1) : (2, 1) : ocurrenciasElementos [1,2,3] []
(1, 1) : (2, 1) : []
[(1, 1), (2, 1)]
```

Lo escribi de una manera que se entendiera que esta pasando pero en realidad se mete a la pila solo la función, entonces el problema es que la pila tiene que llamar a la función cuentaElemento, para ir contando y tambien la función ocurrenciasElementos para ir gurdando los resultados, entonces la pila se va llenando de llamadas pendientes, hasta que se llega a la base de la recursión y se comienza a desapilar, para completar las llamadas pendientes.

(c) Optimizar la función definida usando recursión de cola. Deben transformar todas las funciones auxiliares que utilicen.

```
-- Funcion que usa recursion de cola
cuentaElemento2 :: Int -> [Int] -> Int
cuentaElemento2 n xs = cuentaElemento' n xs 0
    where
        cuentaElemento' :: Int -> [Int] -> Int -> Int
```

```
        cuentaElemento' _ [] acc = acc
        cuentaElemento' n [x] acc
            | n == x    = acc + 1
            | otherwise = acc
        cuentaElemento' n (x:xs) acc
            | n == x    = cuentaElemento' n xs (acc + 1)
            | otherwise = cuentaElemento' n xs acc

ocurrenciasElementos2 :: [Int] -> [Int] -> [(Int, Int)]
ocurrenciasElementos2 xs ys = ocurrenciasElementos' xs ys []
    where
        ocurrenciasElementos' :: [Int] -> [Int] ->
                [(Int, Int)] -> [(Int, Int)]

        ocurrenciasElementos' _ [] acc = acc

        ocurrenciasElementos' xs (y:ys) acc = ocurrenciasElementos' xs ys
                                    (acc ++ [(y, cuentaElemento2 y xs)])
```

Disculpen por la manera de escribirlo pero era para que cupiera, la idea es que la función cuentaElemento2 es la versión de la función cuentaElemento que usa recursión de cola, y la función ocurrenciasElementos2 es la versión de la función ocurrenciasElementos que usa recursión de cola. La funcionalidad de ambas es la misma pero se va a mantener un solo registro en la pila de llamadas, y se va a usar una lista o un int para acumular los resultados.

(d) Mostrar los registros de activación generados por la versión de cola con la misma llamada.