Plantilla

Fernando Romero Cruz

Octubre 2024

Ejercicio 1

La expresión:

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

1. **Expresión:** Usa `let` para poder definir `sum`, una función recursiva que suma los números desde n hasta 0. La función utiliza `if0`, que va a verificar si n es igual a 0. Si sí lo es, entonces va a devolver 0 de lo contrario, va a devolver n más la suma de los números desde $n - 1$.

2. **Ejecución:** Se llama a `(sum 5)` pero esto genera un error de variable libre porque `sum` no está definido dentro del cuerpo de la función lambda. La función intenta llamar a `sum` recursivamente, pero no puede encontrar `sum` en su entorno, ya que no tiene acceso a su propia definición.

Modificación usando el Combinador de Punto Fijo Y
Ahora, modificamos la expresión utilizando el Combinador de Punto Fijo Y:

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

1. **Modificación:** Vamos a definir Y como el Combinador de Punto Fijo esto porque nos permite que las funciones se llamen a sí mismas. `sum` se define como la aplicación de Y a una función que va a tomar como argumento `sum`. Entonces ahora dentro de la definición de `sum`, podemos llamar a `sum` sin que generemos un error de variable libre.

2. **Ejecución:** Cuando llamamos a `(sum 5)`, ahora `sum` está definido correc-

tamente. La ejecución va a ser de la siguiente manera:

$$\begin{aligned}
\text{sum}(5) &= 5 + \text{sum}(4) \\
\text{sum}(4) &= 4 + \text{sum}(3) \\
\text{sum}(3) &= 3 + \text{sum}(2) \\
\text{sum}(2) &= 2 + \text{sum}(1) \\
\text{sum}(1) &= 1 + \text{sum}(0) \\
\text{sum}(0) &= 0
\end{aligned}$$

Esto se evalúa a $5 + 4 + 3 + 2 + 1 + 0 = 15$.

1. **Primera expresión:** Va a generar un error de variable libre porque `sum` no tiene acceso a su propia definición.
2. **Segunda expresión:** Funciona bien y el resultado de `(sum 5)` es 15, que es la suma de los números del 1 al 5.

1 Ejercicio 2

Evaluar la siguiente expresión en Racket, explicar su resultado y dar la continuación asociada a evaluar usando la notación $\lambda\uparrow$:

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

En la primera línea solamente definimos c como falso. Luego, en la segunda empezamos con la suma, donde la parte más anidada, con el `let/cc` captura la continuación actual en la variable k, la cual se almacena en c. Después de esto, se da el valor 4, así que (+ (`let/cc` k (`set!` c k) 4) 5) se evalúa como (+ 4 5) = 9. (+ 3 9) se evalúa como 12.
(+ 2 12) se evalúa como 14.
(+ 1 14) se evalúa como 15.
Por lo tanto, el valor de la segunda línea es 15.

Luego, en la última línea invocamos la continuación almacenada en c que vimos en el paso anterior. Aquí, (c 10) reemplaza la parte donde se evaluaba (+ 4 5) (donde está el `set!`) por (+ 10 5), que da 15. Después, la evaluación sigue como antes: (+ 3 15) = 18, (+ 2 18) = 20, y finalmente (+ 1 20) = **21**.

Notación $\lambda\uparrow$: $\lambda(v) =$`(+ 1 (+ 2 (+ 3 (+ v 5))))`

Ejercicio 3

Definición de la función recursiva `ocurrenciasElementos`:

```
-- Implementacion convencional
ocurrenciasElementos :: (Eq a) => [a] -> [a] -> [(a,Int)]
ocurrenciasElementos lista [] = []
ocurrenciasElementos lista (x:xs) = (x,ocurrencias) : rec
    where ocurrencias = length $ filter (== x) lista
          rec = ocurrenciasElementos lista (filter (/= x) xs)
```

Registros de activación

Sea la llamada: `ocurrenciasElementos [1,2,3] [1,2]`

Los registros de activación son:

Nombre: `ocurrenciasElementos`
Parámetro: `[1,2,3] [1,2]`
Nivel: 0
Valor de Retorno: `(1,1):ocurrenciasElementos [1,2,3] [2]`

Nombre: `ocurrenciasElementos`
Parámetro: `[1,2,3] [2]`
Nivel: 1
Valor de Retorno: `(2,1):ocurrenciasElementos [1,2,3] []`

Nombre: `ocurrenciasElementos`
Parametro: `[1,2,3] []`
Nivel:2
Valor de retorno: `[]`

Nueva Función

```
-- Funcion redefinida con recursion de cola
ocurRecurCola :: (Eq a) => [a] -> [a] -> [(a,Int)]
ocurRecurCola lista nums = aux lista (reverse nums) []
  where aux :: (Eq a) => [a] -> [a] -> [(a,Int)] -> [(a,Int)]
        aux lista [] acc = acc
        aux lista (x:xs) acc = aux lista filt (elem:acc)
          where filt = filter (/= x) xs
                elem = (x,length (filter (== x) lista))
```

Registros de activación

Sea la llamada: `ocurRecurCola [1,2,3] [1,2]`

Los registros de activación son:

Nombre: `ocurRecurCola`
Parámetro: `[1,2,3] [1,2]`
Nivel: 0
Valor de Retorno: `aux [1,2,3] [2,1] []`

Nombre: `aux`
Parámetro: `[1,2,3] [2,1] []`
Nivel: 0
Valor de Retorno: `aux [1,2,3] [1] [(2,1)]`

Nombre: `aux`
Parametro: `[1,2,3] [1] [(2,1)]`
Nivel: 0
Valor de retorno: `[aux [1,2,3] [] [(1,1),(2,1)]]`

Nombre: `aux`
Parámetro: `[1,2,3] [] [(1,1),(2,1)]`
Nivel: 0
Valor de retorno: `[(1,1),(2,1)]`