Universidad Nacional de Rosario

Facultad de ciencias exactas, ingeniería y agrimensura

Análisis de Lenguajes de Programación

Trabajo práctico 2

Alumnos:

Petruskevicius Ignacio - Trincheri Lucio

Octubre 2021

ALP - Informe TP 2

Petruskevicius Ignacio - Trincheri Lucio, LCC UNR FCEIA 05/10/2021

1. Ejercicio 1

Usando recursión definimos la función `num`, que dado un entero devuelve su numeral de Church. Sabiendo que para todo número, su numeral de Church comienza con λ`sz.` solo falta completar con la cantidad correcta de aplicaciones de `s` sobre un `z` final.

Listing 1: Numeral de Church

```
1  ------------------------------------------------
2  -- Seccion 2 - Representaci n de Lambda Terminos
3  -- Ejercicio 1
4  ------------------------------------------------
5
6  -- Terminos con nombres
7  data LamTerm  =  LVar String
8                |  App LamTerm LamTerm
9                |  Abs String LamTerm
10               deriving (Show, Eq)
11
12 num :: Integer -> LamTerm
13 num n = Abs "s" (Abs "z" (numAux n))
14
15 numAux :: Integer -> LamTerm
16 numAux 0 = LVar "z"
17 numAux n = App (LVar "s") (numAux (n-1))
```

2. Ejercicio 2

Teniendo en cuenta que la función `conversion` debe tomar `LamTerm` que son el resultado del parser y devolver su representación sin nombres, primero analizamos el funcionamiento, entendiendo los ejemplos provistos y encontrando la solución a otros casos. Una vez clara, concluimos que necesitamos alguna estructura para almacenar el valor asociado a cada aparición de una variable ligada en la abstracción. Esta debe poder actualizar sus valores. Para ello usamos una lista de tuplas de la forma (variable, valor en la representación).

Luego de definido lo anterior se procede a construir la función `conversionAux` a partir de cada caso del tipo `LamTerm`. Los casos interesantes de nombrar son el de la `Abs` y `LVar`. En el primer caso se actualiza el estado de la variable ligada y se recursiona, y en el segundo se chequea que tipo de variable es y si es ligada se setea el numero que le correspone (almacenado en el estado) y si es global se la identifica como tal.

Listing 2: Función conversion

```haskell
-- Tipos de los nombres
data Name
    =  Global  String
    |  Quote   Int
    deriving (Show, Eq)

-- Terminos con nombres
data LamTerm  =  LVar String
              |  App LamTerm LamTerm
              |  Abs String LamTerm
              deriving (Show, Eq)

-- Terminos localmente sin nombres
data Term  = Bound Int
            | Free Name
            | Term :@: Term
            | Lam Term
            deriving (Show,Eq)


-----------------------------------------------
-- Seccion 2
-- Ejercicio 2: Conversion a terminos localmente sin nombres
-----------------------------------------------

stateUpdate :: String -> [(String, Int)] -> [(String, Int)]
stateUpdate abs state = (abs, 0) : (mapMaybe (\(x,y) -> if abs /= x then Just (x,y+1) else Nothing) state)

stateFind :: String -> [(String, Int)] -> Int
stateFind _ [] = -1
stateFind abs ((x,y):xs) | abs == x = y
                         | otherwise = stateFind abs xs
```

```
33
34 conversion :: LamTerm -> Term
35 conversion t = conversionAux t []
36
37 conversionAux :: LamTerm -> [(String, Int)] -> Term
38 conversionAux (App t1 t2) state = let e1 = conversionAux t1 state
39                                       e2 = conversionAux t2 state
40                                   in e1 :@: e2
41 conversionAux (Abs abs t) state = Lam (conversionAux t (stateUpdate abs state))
42 conversionAux (LVar abs) state = case (stateFind abs state) of
43                                     -1 -> (Free (Global abs))
44                                     n -> Bound n
```

Una vez finalizado esto, utilizamos el comando `print` para poder visualizar el correcto funcionamiento de la función desarrollada.

3. Ejercicio 3 y 4

Con el objetivo de evaluar las expresiones, y teniendo en cuenta que el proceso de sustitución puede ser complejo, desde el enunciado se propone utilizar el espacio de funciones de `Haskell` lo cual nos facilita la implementación de la sustitución. Entonces en estos 2 ejercicios se plantea una traducción de `Terms` a `Values` y una reducción de la expresión a su forma normal. Para conseguir esto ultimo, seguimos las reglas propuestas en el enunciado.

Sobre la implementación nos gustaría comentar que la función stateFindG es similar a la empleada en el ejercicio 2, solo que modificada su respuesta para variables libres. Además, nuevamente hacemos uso de una lista para almacenar las variables ligadas para luego reemplazar su valor a la hora de evaluar.

Listing 3: Evaluación

```haskell
-- Terminos localmente sin nombres
data Term = Bound Int
          | Free Name
          | Term :@: Term
          | Lam Term
          deriving (Show,Eq)

-- Tipos de los nombres
data Name
    =  Global  String
    |  Quote   Int
    deriving (Show, Eq)

type NameEnv v = [(Name, v)]

-- Valores
data Value
    =  VLam      (Value -> Value)
    |  VNeutral  Neutral

data Neutral
    =  NFree  Name
    |  NApp   Neutral Value

--------------------------------
-- Seccion 3
--------------------------------

vapp :: Value -> Value -> Value
vapp (VLam x) z = x z
vapp (VNeutral x) z = VNeutral (NApp x z)

eval :: NameEnv Value -> Term -> Value
eval e t = eval' t (e, [])

```

```
36 eval' :: Term -> (NameEnv Value, [Value]) -> Value
37 eval' (Bound ii) (_, lEnv) = lEnv !! ii
38 eval' (Free name) (nEnv, _) = stateFindG name nEnv
39 eval' (a :@: b) e@(nEnv, lEnv) = vapp (eval' a e) (eval' b e)
40 eval' (Lam f) (nEnv, lEnv) = VLam (\x-> eval' f (nEnv, x:lEnv))
41
42 stateFindG :: Name -> NameEnv Value -> Value
43 stateFindG abs [] = VNeutral (NFree abs)
44 stateFindG abs ((x,y):xs) | abs == x = y
45                           | otherwise = stateFindG abs xs
```

4. Ejercicio 5

El problema ahora es que, una vez aprovechado el espacio de funciones de Haskell para ayudarnos en la sustitución (más específicamente para representar valores que sean abstracciones), no es posible examinar las funciones resultantes. Para ello es necesario pasar las funciones nuevamente a términos, lo que se realizó de la siguiente manera.

Listing 4: Mostrando Valores

```
1  --------------------------------
2  -- Seccion 4 - Mostrando Valores
3  --------------------------------
4  -- Terminos localmente sin nombres
5  data Term = Bound Int
6              | Free Name
7              | Term :@: Term
8              | Lam Term
9              deriving (Show,Eq)
10
11 -- Valores
12 data Value
13     =  VLam      (Value -> Value)
14     |  VNeutral  Neutral
15
16 data Neutral
17     =  NFree  Name
18     |  NApp   Neutral Value
19
20
21 quote :: Value -> Term
22 quote v = quoteAux v 0
23
24 -- Int: cantidad de variables frescas aplicadas hasta el momento.
25 quoteAux :: Value -> Int -> Term
26 -- como el termino del lam puede ser otro lam debemos recurcionar aumentando el ↩
       numero de variables frescas pues acabamos de usar la numero i.
27 quoteAux (VLam f) i = Lam (quoteAux (f (VNeutral (NFree (Quote i)))) (i+1))
28 quoteAux (VNeutral neu) i = (quoteAux' neu i)
29
30 quoteAux' :: Neutral -> Int -> Term
31 quoteAux' (NFree (Quote k)) n = Bound (n-k-1)
32 quoteAux' (NFree  name) n = Free name
33 quoteAux' (NApp neu v) n = (quoteAux' neu n) :@: (quoteAux v n)
```

5. Ejercicio 6

La idea detrás de realizar la función `esPrimo` fue comprobar como primera instancia si el número es divisible por 2 (es par) y en caso contrario dividir el número por los valores impares desde 3 hasta la mitad del valor buscado (sin incluir).

La función esPrimo y sus auxiliares se encuentran en su propio archivo nombrado Ejercicio6.lam.

Listing 5: Parser Haskell

```
-- if
def if = \c t e . c t e

-- not
def not = \x . if x false true

def is1 = \n . (and (not (is0 n)) (is0 (pred n)))

-- Funciones de comparaci n:
-- equals:
def equals' = \f x y . if (or (and (not (is1 x)) (is1 y)) (and (is1  x) (not(is1  y)))) false (f (pred x) (pred y))
def equals = Y (\f x y . if (and (is0 x) (is0 y)) true (equals' f x y))

-- gr: Dado dos numeros, indica si el primero es mayor estricto que el segundo.
def gr = Y (\f x y . if (is0 x) false (if (is0 y) true (f (pred x) (pred y))))

-- Operaciones entre naturales:
-- Resta: Dados dos n meros devuelve la resta parcial.
def sub = Y (\f n x . if (is0 x) n (if (is0 n) (f 0 (pred x)) (f (pred n) (pred x))))

-- Divisible: Dados dos n meros indica si el primero es divisible por el segundo.
def divisible =  Y (\f n x . if (gr n x) (f (sub n x) x) (equals n x))

-- Divisor: Dados dos n meros divide el primero por el segundo.
def dividir = Y (\f n x . if (gr x n) 0 (suc (f (sub n x) x)))

-- esPrimo': Dado un n mero, su divisor final y un valor inicial, devuelve si el n mero tiene alg n divisor desde el el valor incial hasta su ultimo divisor.
def esPrimo' = Y (\f n d x. if (not (gr d x)) false (or (divisible n x) (f n d (suc (suc x)))))
-- esPrimo: Dado un n mero se fija si es divisible por 2 o si es divisible por los impares hasta el mismo.
def esPrimo = \n . if (divisible n 2) (equals n 2) (not (esPrimo' n (dividir n 2) 3))
```