Trabajo Práctico
Número 2

Análisis de Lenguajes de Programación

Realizado por

Gianfranco Paoloni
Sebastián Zimmermann

Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

FCEIA
FACULTAD DE
CIENCIAS EXACTAS,
INGENIERIA Y AGRIMENSURA

Ejercicio 1

Para el ejercicio 1, simplemente separamos el caso de un Abs de un caso de LVar y App, para no tener que considerar en la recursión el uso de 's' y 'z'.

```
-----------------------------------------------
-- Sección 2 - Representacón de Lambda Términos
-- Ejercicio 1
-----------------------------------------------

  num :: Integer -> LamTerm
  num n = Abs "s" $ Abs "z" (num' n)

  num' :: Integer -> LamTerm
  num' 0 = LVar "z"
num' m = App (LVar "s") (num' $ m-1)
```

Ejercicio 2

Para la conversión, utilizamos una función auxiliar conversion' para poder mantener un estado **state** cuyo índice representa el nivel de De Bruijn

```
------------------------------------------------
-- Sección 2 - Representación de Términos Lambda
-- Ejercicio 2: Conversión de Términos
------------------------------------------------

conversion  :: LamTerm -> Term
conversion t = conversion' t []

conversion' :: LamTerm -> [String] ->Term
conversion' (LVar s)    state = case (elemIndex s state) of
            Just n -> Bound (length state - n - 1)
            _      -> Free s
conversion' (App t1 t2) state = let v1 = conversion' t1 state
                                    v2 = conversion' t2 state
                                in (v1 :@: v2)
conversion' (Abs str t) state = let s = str : state
                                in (Lam $ conversion' t s)
```

Ejercicio 3 - 4 - 5

Para los ejercicios 3 y 4 simplemente pasamos la definición teórica al código.

Para el eval, decidimos analizar los casos de las variables libres y de las aplicaciones en funciones separadas.

```haskell
-------------------------------
-- Sección 3 - Evaluación
-------------------------------

shift :: Term -> Int -> Int -> Term
shift (Bound k) c d | k >= c    = Bound $ k+d
                    | otherwise = Bound k
shift (Free x)  c d = Free x
shift (t :@: u) c d = (shift t c d) :@: (shift u c d)
shift (Lam t)   c d = Lam (shift t c d)

subst :: Term -> Term -> Int -> Term
subst t1 t2 i = subst' t1 t2 i 0

subst' :: Term -> Term -> Int -> Int -> Term
subst' t1 t2 i j = case t1 of
    Free x  -> Free x
    Bound k -> substBound k t2 i j
    v :@: u -> (subst' v t2 i j) :@: (subst' u t2 i j)
    Lam t   -> Lam (subst' t t2 i (j+1))


substBound :: Int -> Term -> Int -> Int -> Term
substBound k t2 i j | k == i = shift t2 k j
                    | k >  i = Bound (k-1)
                    | k <  i = Bound k


eval :: NameEnv Term -> Term -> Term
eval nvs t = eval' nvs t 0

eval' :: NameEnv Term -> Term -> Int -> Term
eval' nvs t i = case t of
     Free x    -> evalFree nvs x i
     Bound k   -> Bound k
     t1 :@: t2 -> evalApp nvs t1 t2 i
     Lam t'    -> Lam $ eval' nvs t' (i+1)

evalFree :: NameEnv Term -> Name -> Int -> Term
evalFree nvs x i = case (lookup x nvs) of
     Just n    -> eval' nvs (shift n 0 i) i
     otherwise -> Free x


evalApp :: NameEnv Term -> Term -> Term -> Int -> Term
evalApp nvs (Lam t) t2 i = eval' nvs (subst t t2 i) i
evalApp nvs t1      t2 i = let t  = eval' nvs t1 i
                           in case t of
     Lam p     -> evalApp nvs t t2 i
otherwise -> t1 :@: eval' nvs t2 i
```

Ejercicio 6

Para implementar la raíz cuadrada de un número x, comparamos partiendo desde cero, que el producto del número actual por si mismo sea menor o igual a x. Mientras esta condición se cumple, hacemos recursión (con el operador de punto fijo). Cuando la condición se vuelve falsa, devolvemos el último número que la cumplió. Para esto utilizamos una función auxiliar que guarda el ”n”de la recursión (número actual) e implementamos la función menor o igual a partir de las dadas.

```
-- identidad
def id = \x . x

-- Booleanos
def true = \ t f . t
def false = \t f . f

def and = \a b. a b false
def or  = \a b. a true b

-- Pares
def pair = \x y p . p x y

def fst = \p . p true
def snd = \p . p false

-- Numerales de Church
def zero = \s z . z
def suc = \n s z . s (n s z)

def is0 = \n . n (\ x . false) true

def add = \n m s z . n s (m s z)
def mult = \n m s z . n (m s) z

def pred = \ n . fst (n (\p . pair (snd p) (suc (snd p))) (pair zero zero))

--Listas
def nil = \c n . n
def cons = \x xs c n . c x (xs c n)

def isnil = \xs . xs (\x ys . false) true

-- Combinador de Punto Fijo
def Y = \f . (\x . f (x x)) (\x . f (x x))

-- factorial
def fact = Y (\f n. (is0 n) (suc zero) (mult n (f (pred n))))

-- bottom
def bottom = (\x . x x) (\x . x x)

-- menor o igual
def le = Y (\f x y. (or (is0 x) (is0 y)) (or (is0 x) false) (f (pred x) (pred y)))

-- sqrt
```

```
def sqrt = \x. sqrtt x 0

-- sqrt helper
def sqrtt = Y (\f x n. (le (pow2 (suc n)) x) (f x (suc n)) (n))

-- potencia de dos
def pow2 = \x. mult x x
```