Trabajo Practico 2

Facundo Emmanuel Messulam
Franco Ignacio Vallejos Vigier

11 de octubre de 2021

1. Ejercicio 1

Calculo de numeral de Church.

```
num :: Integer -> LamTerm
num x = Abs "s z" (foldr (App) (LVar "z") [LVar "s" | i <- [1..x]])
```

2. Ejercicio 2

Conversion lambda termino a términos sin nombres con índices De Bruijn.

```
conversion :: LamTerm -> Term
conversion x = conversionAux [] x

conversionAux :: [String] -> LamTerm -> Term
conversionAux xs x@(LVar _) = convertir xs x
conversionAux xs (App x y)  = (conversionAux xs x) :@: (conversionAux xs y)
conversionAux xs (Abs x y)  = let
                                (first, cont) = desglosar x
                                adentro | cont == "" = y
                                        | otherwise  = Abs cont y
                              in Lam (conversionAuPygmentsx (first:xs) adentro)

convertir :: [String] -> LamTerm -> Term
convertir xs (LVar x) = case (elemIndex x xs) of
                        Just i -> Bound i
                        Nothing -> Free (Global x)

desglosar :: String -> (String, String)
desglosar x = let
                f [] = []
                f (x:s) | x == ' ' = []
                        | otherwise = x:(f s)
                g [] = []
                g (x:s) | x == ' ' = s
                        | otherwise = g s
              in (f x, g x)
```

3. Ejercicio 3

Aplicación de abstracción utilizando Haskell cuando la expresión los permite.

```
vapp :: Value -> Value -> Value
vapp (VLam f) x = f x
vapp (VNeutral n) x = VNeutral (NApp n x)
```

4. Ejercicio 4

Evaluación de los términos teniendo en cuenta el espacio global de nombres.

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

eval' :: Term -> (NameEnv Value, [Value]) -> Value
eval' (Bound ii) (_, lEnv)     = lEnv !! ii
eval' (Free n)   (gEnv, _)     = let
                                    rs = (filter (\(n', v) -> n' == n) gEnv)
                                 in case rs of
                                    []         -> error "Non existent function used!"
                                    ((_, r):_) -> r
eval' (x :@: y)   e            = vapp (eval' x e) (eval' y e)
eval' (Lam x)     (gEnv, lEnv) = VLam (\v -> eval' x (gEnv, v:lEnv))
```

5. Ejercicio 5

Conversión de los términos a una forma que permite la impresión (es decir, sin usar funciones).

```
quote :: Value -> Term
quote v = quote' 0 v

quote' :: Int -> Value -> Term
quote' i (VLam f)                      = Lam (quote' (i+1) (f (VNeutral (NFree (Quote i)))))
quote' i (VNeutral (NFree (Global s))) = Free (Global s)
quote' i (VNeutral (NFree (Quote k)))  = Bound (i - k - 1)
quote' i (VNeutral (NApp  n v))        = ((quote' (i+1) (VNeutral n)) :@: (quote' (i+1) v))
```

6. Ejercicio 6

Implementación de un programa que trivialmente calcula si un numero es primo. Chequea si el numero es 1, si lo es, confirma que es primo. Si el numero n no es 1 calcula recursivamente si el numero es divisible por todos los números entre 2 y n (sin incluir este ultimo). Si el numero es divisible por alguno de estos afirma que no es primo y si no es divisible por ninguno de estos afirma que es primo.

El calculo de divisibilidad entre x e y se hace encontrando i tal que $i*y = x$, esto es una recursión sobre todos los enteros desde $i = 1$ hasta que la condición que $i*x <= y$ sea falsa o $i * y = x$, lo que ocurra primero.

Para calcular igualdad y mayor o igual se usan recursiones simples que restan 1 y chequean si alguno de los valores es 0. Dependiendo de que valor es cero se sabe cual valor es igual, o, mayor o igual: $0 = 0$ y $x >= 0 \forall x$ donde x es un numeral de Church.

```
-- not: (!)
def not = (\x . x false true)
-- isEqual: (==)
def isEqual =
(Y (\isEqual x y .
  (and (is0 x) (is0 y))
  true
  (
    (
      or
      (and (is0 x) (not (is0 y)))
      (and (not (is0 x)) (is0 y))
    )
    false
    (
      isEqual
      (pred x)
      (pred y)
    )
  )
))
-- isGreaterEqual: (>=)
def isGreaterEqual =
(Y (\isGreaterEqual x y .
  (isEqual x y)
  true
  (
    (is0 x)
    false
    (isGreaterEqual (pred x) (pred y))
  )
))
-- isDivisor: (x / y) es entero?
def isDivisor =
(\x y.
  (Y (\isDivisorAux i x y.
    (
      (isGreaterEqual (mult i y) x)
      (isEqual (mult i y) x)
      (isDivisorAux (suc i) x y)
    )
  ))
  1 -- Empiezo a chequear si es (i*y == x) con i = 1
  x
  y
```

```
)
-- isPrimo: no tiene divisores que no son el numero y uno?
def isPrimo =
(\x.
  (isEqual x 1)
  true
  (
    (Y (\isPrimoAux i x.
      (isDivisor x i)
      (isEqual x i)
      (isPrimoAux (suc i) x)
    ))
    2 -- Empiezo a chequar si es divisor con 2
    x
  )
)
```