Trabajo Práctico 2 - ALP

Camilo Garcia Gonzalez

1 de Octubre del 2019

1 Numerales de Church.

```
appN :: Integer -> LamTerm -> LamTerm ->LamTerm
appN 0 t1 t2 = t2
appN n t1 t2 = App t1 $ appN (n-1) t1 t2

num :: Integer -> LamTerm
num n = Abs "s" $ Abs "z" $ appN n (LVar "s") (LVar "z")
```

2 Conversión de λ-términos a su representación local sin nombres.

```
getLevel :: NameEnv Int -> String -> Maybe Int
getLevel s v = case ns of
                 [] -> Nothing
                 _  -> Just (foldr1 max ns)
  where ns = [m | (u, m) <- s, u == v]

conversion' :: Int -> NameEnv Int -> LamTerm -> Term
conversion' n s (LVar v)     = case (getLevel s v) of
                                 Just m  -> Bound m
                                 Nothing -> Free v
conversion' n s (App t1 t2) = (conversion' n s t1) :@:
                               (conversion' n s t2)
conversion' n s (Abs v t)    = Lam (conversion' n' s' t)
                               where n' = n+1
                                     s' = ((v,n'):s)

conversion   :: LamTerm -> Term
conversion = conversion' (-1) []
```

3 Función shift para λ-términos en su representación sin nombres.

```
shift :: Term -> Int -> Int -> Term
shift (Free v)     _ _ = Free v
shift (Bound k)    c d = if k < c then (Bound k)
                                  else (Bound $ k+d)
shift (Lam t)      c d = Lam $ shift t c d
shift (t1 :@: t2) c d = (shift t1 c d) :@: (shift t2 c d)
```

4 Función sustitución para λ-términos en su representación sin nombres.

```
subst' :: Term -> Term -> Int -> Int-> Term
subst' (Free v)     _  _ _ = Free v
subst' (Bound k)    t' i j = if k == i
                                then (shift t' i j) else
                                if k > i
                                  then (Bound $ k-1)
                                  else (Bound k)
subst' (Lam t)      t' i j = Lam $ subst' t t' i (j+1)
subst' (t1 :@: t2) t' i j = (subst' t1 t' i j) :@:
                             (subst' t2 t' i j)

subst :: Term -> Term -> Int -> Term
subst t t' i = subst' t t' i 0
```

5 Evaluador de λ-términos en su representación sin nombres.

```
eval' :: NameEnv Term -> Int -> Term -> Term
eval' s n (Free v)     = case lookup v s of
                            Nothing -> Free v
                            Just t  -> shift t 0 (n+1)
eval' _ _ (Bound k)    = Bound k
eval' s n ((Lam t1) :@: t2) = eval' s n t
  where t = subst t1 t2 (n+1)
eval' s n (Lam t)      = Lam $ eval' s (n+1) t
eval' s n (t1 :@: t2) = case eval' s n t1 of
                            Lam t -> eval' s n $ (Lam t) :@: t2
                            t     -> t :@: eval' s n t2
```

```
eval :: NameEnv Term -> Term -> Term
eval s t = eval' s (-1) t
```

6 Representación en λ-términos de la función raíz cuadrada entera.

Una definición de la raíz cuadrada podría ser:

$$raiz(n) = min\{i \in N | (i+1)^2 > n\}$$

En Haskell:

```
raiz' :: Int -> Int -> Int
raiz' n i = if (i+1)^2 > n then i
            else raiz' n (i+1)

raiz :: Int -> Int
raiz n = raiz' n 1
```

Luego para representar esta función en λ-cálculo, usaremos algunos términos definidos en Prelude.lam más unos términos que definimos a continuación:

```
-- not logico
def not = \p. if p false true

-- if then else
def if = \p x y. p x y

-- mayor estricto
def grater = \x y. and (is0 (dif y x)) (not (is0 (dif x y)))

-- resta
def dif = \x y. y pred x
```

Con lo que estamos en condiciones de definir la función raíz en λ-cálculo como:

```
-- raiz cuadrada
def raiz' = Y (\f n i. if (grater (mult (suc i) (suc i)) n)
                          i
                          (f n (suc i)))
def raiz  = \n. raiz' n (suc zero)
```