Simplytyped.hs

```
module Simplytyped (
      conversion,    -- conversion a terminos localmente sin nombre
      eval,          -- evaluador
      infer,         -- inferidor de tipos
      quote          -- valores -> terminos
      )
      where

import Data.List
import Data.Maybe
import Prelude hiding ((>>=))
import Text.PrettyPrint.HughesPJ (render)
import PrettyPrinter
import Common

-- conversion a términos localmente sin nombres
conversion :: LamTerm -> Term
conversion = conversion' []

conversion' :: [String] -> LamTerm -> Term
conversion' b (LVar n)     = maybe (Free (Global n)) Bound (n `elemIndex` b)
conversion' b (App t u)    = conversion' b t :@: conversion' b u
conversion' b (Abs n t u)  = Lam t (conversion' (n:b) u)

conversion' b (LetIn x t u) = Let (conversion' b t) (conversion' (x:b) u)
conversion' b (Asc t tt) = As (conversion' b t) tt
conversion' b LUnit = Unit
conversion' b (Par t u) = Pair (conversion' b t) (conversion' b u)
conversion' b (Pri t) = Fst (conversion' b t)
conversion' b (Seg t) = Snd (conversion' b t)
conversion' b Cero = Zero
conversion' b (Succ t) = Suc (conversion' b t)
conversion' b (Rec t u v) = R (conversion' b t) (conversion' b u) (conversion' b v)

----------------------
--- eval
----------------------

sub :: Int -> Term -> Term -> Term
sub i t (Bound j) | i == j     = t
sub _ _ (Bound j) | otherwise = Bound j
sub _ _ (Free n)             = Free n
sub i t (u :@: v)            = sub i t u :@: sub i t v
sub i t (Lam t' u)           = Lam t' (sub (i+1) t u)
sub i t (Let t' u)           = Let (sub i t t') (sub (i+1) t u)
sub _ _ Unit                 = Unit
sub i t (As t' tt)                 = As (sub i t t') tt
sub i t (Pair t1 t2)         = Pair (sub i t t1) (sub i t t2)
sub i t (Fst t')             = Fst (sub i t t')
sub i t (Snd t')             = Snd (sub i t t')
```

```
sub _ _ Zero                = Zero
sub i t (Suc t')            = Suc (sub i t t')
sub i t (R t1 t2 t3)        = R (sub i t t1) (sub i t t2) (sub i t t3)

-- evaluador de términos
eval :: NameEnv Value Type -> Term -> Value
eval _ (Bound _)            = error "variable ligada inesperada en eval"
eval e (Free n)             = fst $ fromJust $ lookup n e
eval _ (Lam t u)            = VLam t u
eval e (u :@: v)            = case (eval e u) of
              VLam t u' -> eval e (sub 0 tv u') where tv = quote (eval e v)
              _         -> error "Error de tipo en run-time, verificar type checker"

eval e (Let t u)            = eval e (sub 0 tv u ) where tv = quote (eval e t)
eval e (As t tt) = eval e t

eval e Unit = VUnit

eval e (Pair t1 t2) = VPar (eval e t1) (eval e t2)
eval e (Fst t) = case eval e t of
              VPar t1 t2 -> t1
              _          -> error "Error de tipo en run-time, verificar type checker"
eval e (Snd t) = case eval e t of
              VPar t1 t2 -> t2
              _          -> error "Error de tipo en run-time, verificar type checker"
eval e Zero = NV V0
eval e (Suc t) = case eval e t of
        NV V0       -> NV (VSuc V0)
        NV (VSuc nv) -> NV (VSuc (VSuc nv))
        _           -> error "Error de tipo en run-time, verificar type checker"

eval e (R t1 t2 t3) = case eval e t3 of
              NV V0       -> eval e t1
              NV (VSuc nv) -> eval e ((t2 :@: (R t1 t2 t)) :@: t) where t = quote (NV nv)
              _          -> error "Error de tipo en run-time, verificar type checker"


----------------------
--- quoting
----------------------

quote :: Value -> Term
quote (VLam t f) = Lam t f
quote VUnit = Unit
quote (VPar v1 v2) = Pair (quote v1) (quote v2)
quote (NV V0) = Zero
quote (NV (VSuc nv)) = Suc (quote (NV nv))

----------------------
--- type checker
----------------------
```

```
-- type checker
infer :: NameEnv Value Type -> Term -> Either String Type
infer = infer' []

-- definiciones auxiliares
ret :: Type -> Either String Type
ret = Right

err :: String -> Either String Type
err = Left

(>>=) :: Either String Type -> (Type -> Either String Type) -> Either String Type
(>>=) v f = either Left f v

-- fcs. de error

matchError :: Type -> Type -> Either String Type
matchError t1 t2 = err $ "se esperaba " ++
                   render (printType t1) ++
                   ", pero " ++
                   render (printType t2) ++
                   " fue inferido."

notfunError :: Type -> Either String Type
notfunError t1 = err $ render (printType t1) ++ " no puede ser aplicado."

notfoundError :: Name -> Either String Type
notfoundError n = err $ show n ++ " no está definida."


infer' :: Context -> NameEnv Value Type -> Term -> Either String Type
infer' c _ (Bound i) = ret (c !! i)
infer' _ e (Free n) = case lookup n e of
                     Nothing -> notfoundError n
                     Just (_,t) -> ret t
infer' c e (t :@: u) = infer' c e t >>= \tt ->
                   infer' c e u >>= \tu ->
                   case tt of
                     Fun t1 t2 -> if (tu == t1)
                                  then ret t2
                                  else matchError t1 tu
                     _         -> notfunError tt
infer' c e (Lam t u) = infer' (t:c) e u >>= \tu ->
                   ret $ Fun t tu

infer' c e (Let t1 t2) = infer' c e t1 >>= \tt ->
                         infer' (tt:c) e t2

infer' c e (As t tt) = infer' c e t >>= \tu -> if tu == tt then ret tt else matchError tt tu
infer' c e Unit = ret TUnit
infer' c e (Pair t t') = infer' c e t >>= \t1 ->
                   infer' c e t' >>= \t2 -> ret $ TPar t1 t2
```

```
infer' c e (Fst t) = infer' c e t >>= \tt ->
                case tt of
                TPar t1 t2 -> ret t1
                _ -> err $ "se esperaba un tipo TPar, pero "++ render(printType tt)++" fue inferido."

infer' c e (Snd t) = infer' c e t >>= \tt ->
                case tt of
                TPar t1 t2 -> ret t2
                _ -> err $ "se esperaba un tipo TPar, pero "++ render(printType tt)++" fue inferido."
infer' c e Zero = ret TNat
infer' c e (Suc t) = infer' c e t >>= \tt -> if tt == TNat then ret TNat else matchError TNat tt
infer' c e (R t1 t2 t3 ) =  infer' c e t1 >>= \tt ->
                    infer' c e t2 >>= \tu ->
                    case tu of
                      Fun t  (Fun TNat t') -> if t==tt && t==t'
                                      then infer' c e t3 >>= \tv -> if TNat==tv
                                                                    then ret t
                                                                    else matchError TNat tv
                                      else matchError (Fun  tt (Fun TNat tt)) tu
                      _                -> matchError (Fun tt (Fun TNat tt)) tu
```

---