Trabajo Práctico III

Análisis de Lenguajes de Programación

Borrero, Paula (P-4415/6) Herranz, Cecilia (H-0471/5)

15 de octubre de 2015

Ejercicio 1

Árbol de derivación de tipos de $S = \lambda x : B \to B \to B.\lambda y : B \to B.\lambda z : B.(xz)(yz)$ en página 4.

El resultado del árbol queda comprobado usando el intérprete:

```
ST> :type S
(B -> B -> B) -> (B -> B) -> B -> B
```

Ejercicio 2

infer retorna un valor de tipo Either para poder devolver mensajes de error cuando no se pueda inferir el tipo de la expresión.

$\gg=$ toma un valor v de tipo Either String Type y una función f de tipo Type $\to$ Either String Type y retorna Left v si es una cadena (Left) o la función f aplicada a v si es un tipo (Right). Sirve para propagar errores.

Ejercicio 5

Árbol de tipado para $(let\ z = ((\lambda x : B.x)\ as\ B \to B)\ in\ z)\ as\ B \to B$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : B \in x : B}{x : B \vdash x : B}\ T\text{-}VAR}{\vdash\ (\lambda x : B.x) : B \to B}\ T\text{-}ABS}{\vdash\ (\lambda x : B.x)\ as\ B \to B : B \to B}\ T\text{-}ASCRIBE \quad \dfrac{z : B \to B \in z : B \to B}{z : B \to B \vdash z : B \to B}\ T\text{-}VAR}{\vdash\ (let\, z = ((\lambda x : B.x)\ as\ B \to B)\ in\ z) : B \to B}\ T\text{-}LET}{\vdash\ (let\, z = ((\lambda x : B.x)\ as\ B \to B)\ in\ z)\ as\ B \to B : B \to B}\ T\text{-}ASCRIBE$$

El resultado del árbol queda comprobado usando el intérprete:

```
ST> :type (let z = ((\x:B.x) as B -> B) in z) as B->B
B -> B
```

Ejercicio 7

Reglas de evaluación para pares:

$$\frac{t_1 \to t_1'}{(t_1, t_2) \to (t_1', t_2)}\ (E{-}PAIR1) \qquad \frac{t_2 \to t_2'}{(v, t_2) \to (v, t_2')}\ (E{-}PAIR2)$$

$$\frac{t \to t'}{FST\ t \to FST\ t'}\ (E{-}FST0) \qquad \frac{t \to t'}{SND\ t \to SND\ t'}\ (E{-}SND0)$$

$$\frac{}{FST\ (v_1, v_2) \to v_1}\ (E{-}FST1) \qquad \frac{}{SND\ (v_1, v_2) \to v_2}\ (E{-}SND1)$$

Ejercicio 9

Árbol de tipado para $fst\ (unit\ as\ Unit,\ \lambda x : (B, B).snd\ x)$:

$$\dfrac{\dfrac{\dfrac{}{\vdash unit : Unit}\ T{-}UNIT}{\vdash unit\ as\ Unit : Unit}\ T{-}ASCRIBE \quad \dfrac{\dfrac{\dfrac{x : (B,B) \in x : (B,B)}{x : (B,B) \vdash x : (B,B)}\ T{-}VAR}{x : (B,B) \vdash snd\ x : B}\ T{-}SND}{\vdash \lambda x : (B,B).snd\ x : (B \to B) \to B}\ T{-}ABS}{\dfrac{\vdash (unit\ as\ Unit, \lambda x : (B,B).snd\ x) : (Unit, (B \to B) \to B)}{\vdash fst\ (unit\ as\ Unit, \lambda x : (B,B).snd\ x) : Unit}\ T{-}FST}\ T{-}PAIR$$

El resultado del árbol queda comprobado usando el intérprete:

```
ST> :type fst (unit as Unit, \x:(B,B). snd x)
Unit
```

Ejercicio 11

La función Ack queda definida como:

```
def ack = \m:Nat. R (\n:Nat. suc n) (\x:Nat->Nat. \y:Nat. \n:Nat. R (x (suc 0))
                                                    (\p:Nat. \q:Nat. x p) n) m
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}B{\to}B{\to}B,y{:}B{\to}B,z{:}B \vdash x{:}B{\to}B{\to}B}\,T{-}VAR \quad \dfrac{}{x{:}B{\to}B{\to}B,y{:}B{\to}B,z{:}B \vdash z{:}B}\,T{-}VAR}{x{:}B{\to}B{\to}B,y{:}B{\to}B,z{:}B \vdash (xz){:}B{\to}B}\,T{-}APP \quad \dfrac{\dfrac{}{x{:}B{\to}B{\to}B,y{:}B{\to}B,z{:}B \vdash y{:}B{\to}B}\,T{-}VAR \quad \dfrac{}{x{:}B{\to}B{\to}B,y{:}B{\to}B,z{:}B \vdash z{:}B}\,T{-}VAR}{x{:}B{\to}B{\to}B,y{:}B{\to}B,z{:}B \vdash (yz){:}B}\,T{-}APP}{x{:}B{\to}B{\to}B,y{:}B{\to}B,z{:}B \vdash (xz)(yz){:}B}\,T{-}APP}{x{:}B{\to}B{\to}B,y{:}B{\to}B \vdash \backslash z{:}B.(xz)(yz){:}B{\to}B}\,T{-}ABS}{x{:}B{\to}B{\to}B \vdash \backslash y{:}B{\to}B.\backslash z{:}B.(xz)(yz){:}(B{\to}B){\to}B{\to}B}\,T{-}ABS}{\vdash \backslash x{:}B{\to}B{\to}B.\backslash y{:}B{\to}B.\backslash z{:}B.(xz)(yz){:}(B{\to}B{\to}B){\to}(B{\to}B){\to}B{\to}B}\,T{-}ABS$$

Los ejercicios 3, 4, 6, 8 y 10 se encuentran resueltos en el código escrito a continuación:

Código de Common.hs:

```
module Common where

  -- Comandos interactivos o de archivos
  data Stmt i = Def String i   --  Declarar un nuevo identificador x, let x = t
              | Eval i         --  Evaluar el término
    deriving (Show)

  instance Functor Stmt where
    fmap f (Def s i) = Def s (f i)
    fmap f (Eval i)  = Eval (f i)

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

  -- Entornos
  type NameEnv v t = [(Name, (v, t))]

  -- Tipo de los tipos
  data Type = Base
            | Unit
            | Nat
            | Fun Type Type
            | Tup Type Type
            deriving (Show, Eq)

  -- Términos con nombres
  data LamTerm  =  LUnit
                |  LZero
                |  LSuc LamTerm
                |  LR LamTerm LamTerm LamTerm
                |  LVar String
                |  Abs String Type LamTerm
                |  App LamTerm LamTerm
```

```
                | LLet String LamTerm LamTerm
                | LAs LamTerm Type
                | LTup LamTerm LamTerm
                | LFst LamTerm
                | LSnd LamTerm
                deriving (Show, Eq)


-- Términos localmente sin nombres
data Term  = TUnit
           | Zero
           | Bound Int
           | Free Name
           | Suc Term
           | R Term Term Term
           | Term :@: Term
           | Lam Type Term
           | Let Term Term
           | As Term Type
           | TTup Term Term
           | Fst Term
           | Snd Term
        deriving (Show, Eq)

-- Valores
data Value = VUnit
           | VZero
           | VSuc Value
           | VLam Type Term
           | VTup Value Value
          deriving Show

-- Contextos del tipado
type Context = [Type]
```

Código de Symplytyped.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 LUnit          = TUnit
conversion' b LZero          = Zero
conversion' b (LVar n)       = maybe (Free (Global n)) Bound (n `elemIndex` b)
conversion' b (LFst t)       = Fst (conversion' b t)
conversion' b (LSnd t)       = Snd (conversion' b t)
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 (LLet n t1 t2) = Let (conversion' b t1) (conversion' (n:b) t2)
conversion' b (LAs u t)      = As (conversion' b u) t
conversion' b (LTup t1 t2)   = TTup (conversion' b t1) (conversion' b t2)
conversion' b (LSuc t)       = Suc (conversion' b t)
conversion' b (LR t0 t1 t2)  = R (conversion' b t0) (conversion' b t1)
                                                    (conversion' b t2)


--- 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 _ _ TUnit                 = TUnit
sub _ _ Zero                  = Zero
sub i t (Fst t')              = Fst (sub i t t')
sub i t (Snd t')              = Snd (sub i t t')
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 t0 t1)           = Let (sub i t t0) (sub (i+1) t t1)
sub i t (As u t')             = As (sub i t u) t'
sub i t (Suc t')              = Suc (sub i t t')
sub i t (TTup t0 t1)          = TTup (sub i t t0) (sub i t t1)
sub i t (R t0 t1 t2)          = R (sub i t t0) (sub i t t1) (sub i t t2)
```

```
-- evaluador de términos
eval :: NameEnv Value Type -> Term -> Value
eval _ TUnit                 = VUnit
eval _ Zero                  = VZero
eval e (Suc t)               = VSuc (eval e t)
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 (Lam _ u :@: Lam s v) = eval e (sub 0 (Lam s v) u)
eval e (Lam t u :@: v)       = case eval e v of
                  VLam t' u' -> eval e (Lam t u :@: Lam t' u')
                  VUnit      -> eval e (sub 0 TUnit u)
                  VZero      -> eval e (sub 0 Zero u)
                  VSuc x     -> eval e (sub 0 (quote (VSuc x)) u)
                  _          -> error "Error de tipo en run-time,
                                          verificar type checker"
eval e (u :@: v)             = case eval e u of
                  VLam t u' -> eval e (Lam t u' :@: v)
                  _         -> error "Error de tipo en run-time,
                                          verificar type checker"
eval e (Let t0 t1)           = eval e (sub 0 (quote (eval e t0)) t1)
eval e (As u t)              = eval e u
eval e (Fst t)               = case eval e t of
                                     VTup t0 t1 -> t0
                                     _          -> error "Error de tipo en run-time,
                                                    el argumento debe ser una tupla"
eval e (Snd t)               = case eval e t of
                                     VTup t0 t1 -> t1
                                     _          -> error "Error de tipo en run-time,
                                                    el argumento debe ser una tupla"
eval e (TTup t0 t1)          = VTup (eval e t0) (eval e t1)
eval e (R t0 t1 t2) = case eval e t2 of
                       VZero  -> eval e t0
                       VSuc t -> eval e ((t1 :@: (R t0 t1 (quote t))) :@: (quote t))
                      _       -> error "Error de tipo en run-time, el
                                          tercer argumento de R tiene que ser Nat"


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

quote :: Value -> Term
quote VUnit      = TUnit
quote VZero = Zero
```

```
quote (VSuc t) = Suc (quote t)
quote (VLam t f) = Lam t f
quote (VTup t0 t1) = TTup (quote t0) (quote t1)

-----------------------
--- 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' _ _ TUnit     = ret Unit
infer' _ _ Zero      = ret Nat
infer' c e (Suc t)  = infer' c e t >>= \tt->
                            case tt of
                                Nat -> ret Nat
                                _   -> matchError Nat tt
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 t0 t1) = infer' c e t0 >>= \tt0 ->
                                infer' (tt0:c) e t1
infer' c e (As u t) = infer' c e u >>= \tu ->
                           if tu == t then ret t else matchError t tu
infer' c e (TTup t0 t1) = infer' c e t0 >>= \tt0 ->
                          infer' c e t1 >>= \tt1 ->
                              ret (Tup tt0 tt1)
infer' c e (Fst t) = infer' c e t >>= \tt ->
                       case tt of
                           Tup tt0 tt1 -> ret tt0
                           _           -> notfunError tt
infer' c e (Snd t) = infer' c e t >>= \tt ->
                       case tt of
                           Tup tt0 tt1 -> ret tt1
                           _           -> notfunError tt
infer' c e (R t0 t1 t2) = infer' c e t0 >>= \tt0 ->
                          infer' c e t1 >>= \tt1 ->
                          infer' c e t2 >>= \tt2 ->
                          case tt1 of
                            Fun t t' -> if t == tt0 && t' == Fun Nat t && tt2 == Nat
                                        then ret t
                                        else matchError (Fun t t') tt1
                            _        -> notfunError tt1
```

Código de Parse.y:

```
{
module Parse where
import Common
import Data.Maybe
import Data.Char

}
```

```
%monad { P } { thenP } { returnP }
%name parseStmt Def
%name parseStmts Defs
%name term Exp

%tokentype { Token }
%lexer {lexer} {TEOF}

%token
    '='     { TEquals }
    ':'     { TColon }
    '\\'    { TAbs }
    '.'     { TDot }
    '('     { TOpen }
    ')'     { TClose }
    '->'    { TArrow }
    ','     { TComma }
    LET     { TLet }
    IN      { TIn }
    VAR     { TVar $$ }
    TYPE    { TType }
    DEF     { TDef }
    TYPEUNIT { TTypeUnit }
    NAT     { TNat }
    ZERO    { TZero }
    SUC     { TSuc }
    REC     { TR }
    AS      { TAs }
    UNIT    { TokenUnit }
    FST     { TFst }
    SND     { TSnd }


%right VAR
%left '='
%right '->'
%right '\\' '.' LET IN
%left AS
%right REC
%right SUC
%right SND FST


%%
```

```
Def     :  Defexp                       { $1 }
        |  Exp                                { Eval $1 }
Defexp  : DEF VAR '=' Exp               { Def $2 $4 }

Exp     :: { LamTerm }
        : '\\' VAR ':' Type '.' Exp     { Abs $2 $4 $6 }
        | LET VAR '=' Exp IN Exp        { LLet $2 $4 $6 }
        | Exp AS Type                   { LAs $1 $3 }
        | REC Atom Atom Exp             { LR $2 $3 $4 }
        | FST Atom                      { LFst $2 }
        | SND Atom                      { LSnd $2 }
        | NAbs                          { $1 }

NAbs    :: { LamTerm }
        : NAbs Atom                     { App $1 $2 }
        | Atom                          { $1 }

Atom    :: { LamTerm }
        : VAR                           { LVar $1 }
        | SUC Atom                      { LSuc $2 }
        | '(' Exp ',' Exp ')'           { LTup $2 $4 }
        | '(' Exp ')'                   { $2 }
        | UNIT                          { LUnit }
        | ZERO                          { LZero }

Type    : TYPE                          { Base }
        | TYPEUNIT                      { Unit }
        | NAT                           { Nat }
        | '(' Type ',' Type ')'         { Tup $2 $4 }
        | Type '->' Type                { Fun $1 $3 }
        | '(' Type ')'                  { $2 }

Defs    : Defexp Defs                   { $1 : $2 }
        |                               { [] }

{

data ParseResult a = Ok a | Failed String
                     deriving Show
type LineNumber = Int
type P a = String -> LineNumber -> ParseResult a

getLineNo :: P LineNumber
getLineNo = \s l -> Ok l
```

```
thenP :: P a -> (a -> P b) -> P b
m `thenP` k = \s l-> case m s l of
                        Ok a     -> k a s l
                        Failed e -> Failed e

returnP :: a -> P a
returnP a = \s l-> Ok a

failP :: String -> P a
failP err = \s l -> Failed err

catchP :: P a -> (String -> P a) -> P a
catchP m k = \s l -> case m s l of
                       Ok a     -> Ok a
                       Failed e -> k e s l

happyError :: P a
happyError = \ s i -> Failed $ "Línea "++(show (i::LineNumber))++":
                                                        Error de parseo\n"++(s)

data Token = TVar String
               | TType
               | TDef
               | TTypeUnit
               | TNat
               | TAbs
               | TDot
               | TOpen
               | TClose
               | TColon
               | TArrow
               | TComma
               | TEquals
               | TLet
               | TAs
               | TIn
               | TokenUnit
               | TFst
               | TSnd
               | TZero
               | TSuc
               | TR
               | TEOF
              deriving Show
```

```
lexer cont s = case s of
                    [] -> cont TEOF []
                    ('\n':s)  ->  \line -> lexer cont s (line + 1)
                    (c:cs)
                          | isSpace c -> lexer cont cs
                          | isAlpha c -> lexVar (c:cs)
                    ('-':('-':cs)) -> lexer cont $ dropWhile ((/=) '\n') cs
                    ('{':('-':cs)) -> consumirBK 0 0 cont cs
                    ('-':('}':cs)) -> \ line -> Failed $ "Línea "++(show line)++":
                                                    Comentario no abierto"
                    ('-':('>':cs)) -> cont TArrow cs
                    ('\\':cs)-> cont TAbs cs
                    ('.':cs) -> cont TDot cs
                    ('(':cs) -> cont TOpen cs
                    ('-':('>':cs)) -> cont TArrow cs
                    (')':cs) -> cont TClose cs
                    (':':cs) -> cont TColon cs
                    ('=':cs) -> cont TEquals cs
                    (',':cs) -> cont TComma cs
                    ('0':cs) -> cont TZero cs
                    unknown -> \line -> Failed $ "Línea "++(show line)++":
                          No se puede reconocer "++(show $ take 10 unknown)++ "..."
                    where lexVar cs = case span isAlpha cs of
                                          ("B",rest)   -> cont TType rest
                                          ("Unit", rest) -> cont TTypeUnit rest
                                          ("Nat", rest) -> cont TNat rest
                                          ("def",rest) -> cont TDef rest
                                          ("let",rest) -> cont TLet rest
                                          ("in",rest) -> cont TIn rest
                                          ("as", rest) -> cont TAs rest
                                          ("unit", rest) -> cont TokenUnit rest
                                          ("fst", rest) -> cont TFst rest
                                          ("snd", rest) -> cont TSnd rest
                                          ("suc", rest) -> cont TSuc rest
                                          ("R", rest) -> cont TR rest
                                          (var,rest)   -> cont (TVar var) rest
                          consumirBK anidado cl cont s = case s of
                                                           ('-':('-':cs)) ->
                                                              consumirBK anidado cl
                                                              cont $ dropWhile
                                                              ((/=) '\n') cs
                                                            ('{':('-':cs)) ->
                                                              consumirBK (anidado+1)
                                                              cl cont cs
```

```
                                                    ('-':('}':cs)) ->
                                                     case anidado of
                                                      0 -> \line -> lexer
                                                           cont cs (line+cl)
                                                      _ -> consumirBK
                                                           (anidado-1) cl
                                                           cont cs
                                                     ('\n':cs) ->
                                                       consumirBK anidado
                                                       (cl+1) cont cs
                                                     (_:cs) -> consumirBK
                                                        anidado cl cont cs

stmts_parse s = parseStmts s 1
stmt_parse s = parseStmt s 1
term_parse s = term s 1
}
```

Código de PrettyPrinter.hs:

```
module PrettyPrinter (
       printTerm,     -- pretty printer para terminos
       printType,     -- pretty printer para tipos
       )
       where

import Common
import Text.PrettyPrint.HughesPJ

-- lista de posibles nombres para variables
vars :: [String]
vars = [ c : n | n <- "" : map show [1..], c <- ['x','y','z'] ++ ['a'..'w'] ]

parensIf :: Bool -> Doc -> Doc
parensIf True  = parens
parensIf False = id

-- pretty-printer de términos

pp :: Int -> [String] -> Term -> Doc
pp ii vs TUnit             = text "unit"
pp ii vs Zero              = text "0"
pp ii vs (Suc t)           = text "suc " <>
                             parensIf (not (isAtm t)) (pp ii vs t)
pp ii vs (Bound k)         = text (vs !! (ii - k - 1))
```

```
pp _  vs (Free (Global s)) = text s
pp ii vs (i :@: c)         = sep [parensIf (isLam i) (pp ii vs i),
                                  nest 1 (parensIf (isLam c || isApp c) (pp ii vs c))]
pp ii vs (Lam t c)         = text "\\" <>
                             text (vs !! ii) <>
                             text ":" <>
                             printType t <>
                             text ". " <>
                             pp (ii+1) vs c
pp ii vs (Let t0 t1)       = text "let " <>
                             text (vs !! ii) <>
                             text " = " <>
                             pp ii vs t0 <>
                             text " in " <>
                             pp (ii+1) vs t1
pp ii vs (As u t)          = pp ii vs u <>
                             text " as " <>
                             printType t
pp ii vs (TTup t0 t1)      = text "(" <>
                             pp ii vs t0 <>
                             text "," <>
                             pp ii vs t1 <>
                             text ")"
pp ii vs (Fst t)           = text "fst " <>
                             pp ii vs t
pp ii vs (Snd t)           = text "snd" <>
                             pp ii vs t
pp ii vs (R t0 t1 t2)      = text "R" <>
                             pp ii vs t0 <>
                             pp ii vs t1 <>
                             pp ii vs t2

isLam (Lam _ _) = True
isLam  _        = False

isApp (_ :@: _) = True
isApp _         = False

isAtm TUnit        = True
isAtm Zero         = True
isAtm (TTup t0 t1) = True
isAtm _            = False
```

```
-- pretty-printer de tipos
printType :: Type -> Doc
printType Base         = text "B"
printType Unit         = text "Unit"
printType Nat          = text "Nat"
printType (Fun t1 t2)  = sep [ parensIf (isFun t1) (printType t1),
                               text "->",
                               printType t2]
printType (Tup t1 t2)  = text "(" <>
                         printType t1 <>
                         text "," <>
                         printType t2 <>
                         text ")"


isFun (Fun _ _)        = True
isFun _                = False

fv :: Term -> [String]
fv TUnit             = []
fv Zero              = []
fv (Suc t)           = fv t
fv (Bound _)         = []
fv (Free (Global n)) = [n]
fv (Free _)          = []
fv (t :@: u)         = fv t ++ fv u
fv (Lam _ u)         = fv u
fv (Let t0 t1)       = fv t0 ++ fv t1
fv (As u t)          = fv u
fv (Fst t)           = fv t
fv (Snd t)           = fv t
fv (TTup t0 t1)      = fv t0 ++ fv t1
fv (R t0 t1 t2)      = fv t0 ++ fv t1 ++ fv t2

---
printTerm :: Term -> Doc
printTerm t = pp 0 (filter (\v -> not $ elem v (fv t)) vars) t
```